FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Využití jazyka P4 pro generování síťových bezpečnostních aplikací
Student:	Bc. Jiří Havránek
Vedoucí:	Ing. Tomáš Čejka, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Nastudujte vysokoúrovňový jazyk P4 pro popis funkcionality a architektury síťových aplikací/zařízení na zpracování síťových paketů.
Proveďte analýzu architektury existujícího exportéru síťových toků jako příklad aplikace, která zpracovává síťové pakety a používá se pro bezpečnostní analýzu síťového provozu.
Navrhněte způsob využití, případně navrhněte rozšíření jazyka P4 tak, aby bylo možné popsat a následně vygenerovat exportér síťových toků schopný exportovat informace z aplikačních vrstev (L7).
Navrhněte a implementujte rozšířenou část překladače P4 (tzv. backend) pro automatické generování flow exportéru dle popisu v P4.
Vytvořený backend otestujte a porovnejte výkonnost generovaného flow exportéru a existujícího open source exportéru flow_meter.

Seznam odborné literatury

[1] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," Microprocessors and Microsystems, vol. 56, pp. 22–33, 2018.
[2] J. Havránek, P. Velan, T. Cejka, and P. Benáček, "Enhanced Flow Monitoring with P4 Generated Flexible Packet Parser," in AIMS 2018, 2018.
[3] https://p4.org
[4] https://github.com/CESNET/Nemea-Modules/tree/master/flow_meter

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 31. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Využití jazyka P4 pro generování síťových bezpečnostních aplikací

Bc. Jiří Havránek

Katedra informační bezpečnosti
Vedoucí práce: Ing. Tomáš Čejka, Ph.D.

7. května 2019

Poděkování

Rád bych tímto poděkoval svému vedoucímu Ing. Tomáši Čejkovi, Ph.D. za vedení a podporu během celé doby vzniku této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Prague dne 7. května 2019 .....

Odkaz na tuto práci

Havránek, Jiří. *Využití jazyka P4 pro generování síťových bezpečnostních aplikací*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se věnuje využití vysokoúrovňového jazyka P4 pro generování bezpečnostních síťových aplikací. Využití je demonstrováno na popisu nového exportéru síťových toků, který vychází z existujícího exportéru `flow_meter`. Existující exportér a jazyk P4 jsou analyzovány a jsou identifikovány klíčové komponenty architektury. Mezi tyto komponenty patří parser paketů, flow cache, komponenta pro export toků ve formátu IPFIX a parsovací pluginy aplikačních protokolů. V práci je navržen vhodný popis těchto komponent pomocí programových konstrukcí jazyka P4. P4 program je následně možné přeložit na zdrojové kódy exportéru v C pomocí překladače jazyka P4, do kterého byl touto prací vytvořen backend.

Měření a porovnání ukázalo, že nově vytvořený exportér je rychlejší než původní. Navíc je více flexibilní a podporuje mnohem více protokolů, jelikož je automaticky generovaný z P4 popisu. Je možné jednoduše přidávat nové protokoly, upravovat algoritmus vytváření toků v cache, upravovat záznam o toku a mnohem jednodušeji vytvářet nové parsovací pluginy aplikačních protokolů. Jednotlivé generované komponenty jsou měřeny zvlášť a byly obecně pomalejší než jejich ručně optimalizovaná verze.

Klíčová slova P4, kompilátor, backend, generování kódu, exportér toků, aplikační protokoly

Abstract

This work focuses on usage of P4 high-level language for generation of network security applications. Usage is demonstrated by writing new flow exporter that is based on existing exporter `flow_meter`. Existing exporter and P4 language are analyzed and key architecture components are identified. These components are packet parser, flow cache, IPFIX flow export component and application protocols parsing plugins. Work proposes proper design of these components by P4 programming constructs. P4 program can be afterwards compiled into C source codes of flow exporter by P4 compiler that contains backend created by this work.

Measurements and evaluation show that the created new version of flow exporter is faster than the original one. In addition, it is more flexible and can support much more protocols, since it is automatically generated from P4 language. It is possible to simply add new protocols, change algorithm of flow creation in flow cache, modify flow record and create new application protocols parsing plugins. Generated components were measured separately and were generally slower than their hand optimized versions in existing exporter.

Keywords P4, compiler, backend, code generation, flow exporter, application protocols

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Monitorovaní síťového provozu je klíčovou součástí každé síťové infrastruktury. Přináší operátorům přehled o provozu, aktivitě a síťových zařízení. To umožňuje údržbu, rozvoj a bezpečnostní dohled nad sítí.

Moderní monitorovací systémy jsou založeny na sledování takzvaných síťových toků (flow). Síťové toky reprezentují jednosměrnou komunikaci mezi dvěma síťovými entitami. Tato komunikace se skládá z posloupnosti síťových paketů, které obsahují shodné hodnoty klíčových políček z hlaviček protokolů. Obvyklá sada klíčových políček obsahuje IP adresy, protokol a porty z transportní vrstvy ISO/OSI modelu. Záznamy obsahují i statistické informace navíc jako jsou čítače přenesených paketů a bajtů. Díky tomu zůstává informace o celkovém provozu na síti i v agregované podobě síťového toku.

Zařízení pro vytváření síťových toků se běžně označuje jako exportér síťových toků (flow exporter). Flow exportér analyzuje provoz ve formě paketu a agreguje je v takzvané flow cache. Příkladem exportéru síťových toků je `flow_meter` [1] napsaný v jazyce C++, na kterém jsem pracoval v rámci své bakalářské práce [2]. Umí zpracovávat i L7 protokoly pomocí pluginů, které se dají napsat rovněž v C++.

V roce 2013 byl vyvinut vysokoúrovňový aplikačně specifický jazyk P4 [3], který slouží pro popis síťových zařízení a jejich funkcionality. Motivací jazyka P4 bylo zvýšit flexibilitu a rozšiřitelnost zejména v oblasti software defined networking (SDN) [4]. P4 obsahuje konstrukce pro popis parseru síťových protokolů, vztahu mezi nimi a `match action` tabulky (specifikující operace prováděné nad vybrannými pakety). Dále obsahuje konstrukty pro zavedení vlastních operací pomocí takzvaných `extern` bloků.

Jazyk P4 byl navržen primárně pro podporu protokolů nižších vrstev do L4 ISO/OSI modelu. Hlavičky protokolů mají ve většině případů fixní délku, případně ji lze snadno vypočítat (například políčka options protokolu IPv4). Se zpracováním aplikačních vrstev se v návrhu jazyka P4 vůbec nepočítá.

Mnohé aplikace jako je flow exportér a aplikační firewall mohou využívat data z vyšších vrstev ISO/OSI modelu. Vývoj a rozšiřitelnost takových aplikací

je náročný. Proto jsem začal pracovat na této závěrečné práci, která hledá způsob, jak využít flexibility P4 pro tyto aplikace.

Kapitola 1

Cíl práce

Cílem této diplomové práce je použití vysokoúrovňového jazyka P4 pro snadný popis síťových bezpečnostních aplikací. Popis aplikace v P4 bude demonstrován na generování kódu exportéru síťových toků. Vygenerovaný exportér bude funkcionalitou co nejvíce odpovídat exportéru `flow_meter` z mé bakalářské práce.

Pro účely generování kódu bude využit existující P4 kompilátor [5], do kterého je možné vytvořit vlastní backend. Přidaný backend bude schopný generovat zdrojové kódy klíčových částí exportéru pomocí popisu jednotlivých stavebních bloků v jazyce P4. Vygenerovaný zdrojový kód bude poté možné jednoduchým způsobem zkompilovat do podoby použitelné spustitelné aplikace.

Součástí práce bude také analýza architektury existujícího exportéru toků `flow_meter`. Analýza poslouží pro stanovení požadavků na generování jednotlivých částí exportéru. U vygenerovaného i existujícího exportéru bude změřena výkonnost a stanoveno porovnání z hlediska jejich výhod a nevýhod.

Hlavním přínosem popsaného exportéru v P4 bude snadnější popis procesu parsování, procesu vytváření a exportování toků. Dále snadnější popis parsovacích pluginů pro zpracování aplikačních protokolů. Přístup použitý v této práci bude možné využít i v jiných aplikací jako je například aplikační firewall.

Kapitola 2

Analýza

Kapitola je věnována analýze použitých technologií v této práci a existujících řešení. Popsány jsou zejména základy jazyka P4, z jakých konstruktů se skládá a k čemu se využívají. Tyto poznatky poslouží k pozdějšímu návrhu částí exportéru ke generování.

Pro lepší pochopení požadavků na nový generovaný exportér je popsána analýza existujícího exportéru `flow_meter`. Rozebírány jsou i základní charakteristiky aplikačních protokolů, které exportér zpracovává. Analýza těchto protokolů odhalí, které se dají snadno zpracovávat pomocí P4 programu a které naopak ne.

2.1 Jazyk P4

Jazyk P4 (Programming Protocol-independent Packet Processors) slouží pro popis síťových zařízení. Mezi jeho hlavní výhody patří snadná programovatelnost, konfigurovatelnost, platformní a protokolová nezávislost. Velký důraz je kladen na jednoduchou manipulaci s pakety a možnost flexibilně reagovat na měnící se prostředí sítě.

P4 rozlišuje dva druhy režimu práce zařízení. Jedná se o konfiguraci a běžný provoz. Během konfigurace se do zařízení nahrává program v jazyce P4 a je kompilován do reprezentace, která je pro dané zařízení nejvhodnější. Po naprogramování zařízení se přejde do normálního provozu, kde existuje jeden centrální kontrolér, který řídí všechna zařízení prostřednictvím plnění tabulek [6].

Snadnou programovatelnost zajišťuje vysokoúrovňový pohled na problematiku sítí. Velkou výhodou P4 je možnost snadno definovat nové hlavičky protokolů a způsob, jak je zpracovávat. Díky tomu je možné si dodefinovat podporu pro vlastní protokoly a nebo jen reagovat na nové a měnící se standardy.

Psaní P4 programů bylo navrženo tak, aby byly platformě nezávislé. To znamená, že libovolný P4 program může být zkompilován pro nemalé množství

výpočetních jednotek jako jsou CPU, FPGA, ASIC, SoC nebo třeba síťový procesor. Pro každý P4 cíl (target), jak jsou označovány cílové platformy, je ale nutné mít napsaný příslušný backend pro generování platformě závislého kódu.

V současné době existují dvě verze jazyka P4. Starší $P4_{14}$ a novější $P4_{16}$, která je sice stále ve vývoji, ale umožňuje mnohem širší využití. V celé práci se pracuje s novější verzí jazyka $P4_{16}$.

2.1.1 Objekty a typy

$P4_{16}$ je staticky typovaný jazyk. Programy při kompilaci, které neprojdou typovou kontrolou, se považují za chybné a kompilace skončí s chybou. Výhoda tohoto přístupu je v tom, že se tím eliminuje chybné chování, které nemusí být na první pohled vidět.

P4 podporuje pro některé ze základních typů možnost přetypování. Nejčastěji se bude jednat o typy `bit` a `int`. Nevýhodou statického typování je zdlouhavé psaní při přetypování oproti například jazyku C. Na druhou stranu má vývojář přesnou kontrolu nad jednotlivými operacemi.

Například pro získání velikosti TCP options v bitech se v C použije výraz `(tcp.data_offset - 5) * 32`, kde offset z TCP hlavičky má velikost 4 bity a jedná se o neznaménkové číslo. V P4 je nejprve nutné zvýšit počet bitů `tcp.data_offset` na 10 a pak přetypovat na znaménkový integer, aby bylo možné provést odečtení. Poté je potřeba přetypovat `5` na `int<10>` a celý výraz v závorce přetypovat zpět na neznaménkový integer. Celý výraz bude vypadat jako `(bit<10>)(((int<10>)(bit<10>)tcp.data_offset - (int<10>)5) * 32)`

Z předchozího příkladu je vidět, že jednoduchá operace v C se může v P4 přetypováním znepřehlednit. V odstavcích byly rovněž zmíněny dva datové typy, které reprezentují čísla a lze je mezi sebou převádět, pokud mají stejnou bitovou délku. Následující výčet obsahuje i ostatní typy a jejich stručný popis. Informace jsem čerpal jsem z dokumentace [7] pro aktuální verzi jazyka $P4_{16}$ 1.0.0.

void se používá ve speciálních případech, zejména jako návratový typ funkcí (které nic nevrací)

bool klasický datový typ reprezentující pravdivostní hodnoty 0 nebo 1

bit reprezentuje 1 bit neznaménkového integeru. V hranatých závorkách lze specifikovat libovolnou bitovou délku, `bit<8>` například

int reprezentuje znaménkový integer, počet bitů musí být větší než 1

varbit je datový typ obsahující proměnlivý počet bitů. Použitím `varbit<240>` lze uložit 0 až 240 bitů

enum je klasický výčtový typ

error reprezentuje chybu

match_kind v match action tabulkách značí typ porovnání, například exact, ternary nebo lpm

header je typ podobný struktuře a reprezentuje hlavičku síťového protokolu. Může obsahovat typy `bit`, `int`, `varbit` a `struct`

header_union datový typ jako `union` v C. Jako členské proměnné lze volit pouze typ `header`

struct je ekvivalent `struct` v C

tuple je podobný struktuře, neobsahuje pojmenovaná políčka

Mezi další typy patří `extern`, `control`, `parser` bloky a `package`. Použití typů `control` a `parser` je podobný definici funkce v C, s tím rozdílem, že nemají návratovou hodnotu a lze nastavovat parametry jako vstupní nebo výstupní pomocí klíčových slov `in`, `out` nebo `inout`. Tyto typy mohou navíc obsahovat konstruktor. Spolu s `struct`, `header`, `header_union` a `enum` se tato klíčová slova mohou vyskytovat pouze při deklaraci typu. Následující výčet obsahuje stručný popis jednotlivých typů.

extern slouží k přidání nových funkcionalit, například lze přidat objekt čítač, který bude umět v konstruktoru nastavit velikost a typ čítání a bude obsahovat funkci pro čítání

control lze použít pro kontrolní logiku, například spouštění různých akcí

parser se používá pro popis konečného automatu sloužící k parsování paketů

package umí shlukovat ostatní typy, například v každém P4 programu musí existovat `package` zvaný `main`, který určuje komponenty cílové architektury

2.1.2 Deklarace hlaviček

Součástí každého P4 programu je deklarace hlaviček síťových protokolů, se kterými se v programu pracuje. Deklarovat je lze pomocí klíčového slova `header` obdobně jako struktury v jazyce C. Poté následuje název hlavičky a blok s členskými proměnnými ze závorek.

Členské proměnné se zapisují ve tvaru typ proměnné, její název a ukončení středníkem. Jako typ nelze zvolit `header`, ale fungují `struct`, `bit`, `int` atd. Nejčastěji se setkáme s beznaménkovým typem `bit`.

Příklad kódu popisuje standardní ethernetovou hlavičku. V této hlavičce se nevyskytují žádné proměnné typu `varbit`, jedná se o hlavičku fixní délky. V tomto případě není potřeba specifikovat délku hlavičky pro parsování, je dána součtem velikostí jednotlivých typů.

```
header ethernet_h {
    bit<48> dst_addr;
    bit<48> src_addr;
    bit<16> ethertype;
}
```

Jiné to bude v případě, že velikost hlavičky není pevně daná. Pro parsování je nutné přidat typ `varbit` a specifikovat maximální bitovou velikost. Takové hlavičky se označují jako hlavičky dynamické délky. Při extrahování je potřeba zadat výraz definující velikost dynamické části hlavičky.

```
header ipv4_h {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> total_len;
    bit<16> identification;
    bit<3> flags;
    bit<13> frag_offset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdr_checksum;
    bit<32> src_addr;
    bit<32> dst_addr;
    varbit<320> options;
}
```

Lze se setkat i s příklady aplikací, kde není proměnlivá část hlavičky důležitá a lze ji přeskočit. Například se jedná o exportér v této práci, kdy políčka options v IPv4 nebo TCP hlavičce se nezpracovávají. V takovém případě lze ušetřit místo a `varbit` vůbec nepoužívat. V parsování je možné po přeskočení bitů, které P4 konstrukty dovolují (`extern` bloky), normálně pokračovat.

2.1.3 Extern bloky

Jazyk P4 nabízí možnost, jak v programu interagovat s objekty a funkcemi, které poskytuje daná architektura. Takové objekty se dají definovat pomocí klíčového slova `extern` a popisují množinu funkcí, které objekt implementuje, nikoliv jejich implementaci samotnou. Takovýto přístup je podobný jako použití abstraktní třídy v objektově orientovaném programování [7].

Interní fungování `extern` objektů a funkcí je pevně dané a nelze jej popsat pomocí P4. Jejich chování lze nicméně specifikovat v P4 kompilátoru.

Implementace `extern` objektů má výhody v tom, že je definována kompilátorem pro danou architekturu. To umožnuje zvolit takovou implementaci, která je pro dané cílové zařízení nejvhodnější. Mezi další výhody patří snadná definice a podpora nového objektu nebo funkce v kompilátoru.

V následujícím kódu je ukázka `extern` objektu ze standardního hlavičkového souboru core.p4 [8]. Objekt zde reprezentuje příchozí paket, nad kterým

je možné vykonat řadu operací. Mezi operaci patří extrakce bitů do proměnné (`extract`), extrakce bitů bez posunu kurzoru paketu (`lookahead`), posunutí kurzoru paketu (`advance`) a zjištění délky paketu (`length`). $P4_{16}$ má oproti předchozí verzi výhodu v tom, že podporuje šablony funkcí obdobně jako je tomu v jazyce C++. Díky tomu je možné vytvořit jednu funkci, například `extract`, přijímající celou řadu parametrů různých typů.

```
extern packet_in {
    void extract<T>(out T hdr);
    void extract<T>(out T variableSizeHeader,
                    in bit<32> variableFieldSizeInBits);
    T lookahead<T>();
    void advance(in bit<32> sizeInBits);
    bit<32> length();
}
```

Vzhledem ke skutečnosti, že v P4 není možné popsat mnohé operace v exportéru, se bude v této práci s `extern` bloky často pracovat. Pomocí `extern` bloků budou složité operace prováděné v architektuře přesunuty z P4 do C v backendu kompilátoru.

2.1.4 Parser blok

Parser blok slouží k popisu procesu parsování paketu. Celý parser je popsán jako konečný automat skládající se ze stavů a přechodů mezi nimi. Parser, stejně jako ostatní bloky, může obsahovat celou řadu různých parametrů v závislosti na architektuře.

V parseru existují speciální stavy jako jsou `start`, `accept` a `reject`. Stav `start` je vstupním bodem při vykonávání konečného automatu a musí být vždy přítomen v bloku kódu. Naopak ostatní dva stavy jsou speciální v tom, že se s nimi v parseru počítá přestože nejsou pomocí klíčového slova `state` deklarovány. K signalizaci úspěšného ukončení parsování stačí přejít do stavu `accept` v libovolném místě automatu. Neúspěšné parsování naopak signalizuje přechod do `reject`.

Stav parseru lze, jak bylo zmíněno, deklarovat pomocí klíčového slova `state`. Na konci bloku kódu stavu musí být přítomen přechod do dalšího stavu. Přímý přechod do nového stavu lze signalizovat pomocí `transition` a názvu stavu. Lze také specifikovat více stavů pomocí podmínky, která se specifikuje pomocí klíčového slova `select`.

Výběr podmínky pomocí `select` je podobný jako při využívání `switch` konstrukce v C. V závorce v příkazu `select()` je výraz nebo množina výrazů pro porovnání s hodnotami. Více výrazů je nutné oddělit čárkou (bude se jednat o typ `tuple`).

Podmínka přechodu do dalšího stavu se napíše ve tvaru porovnávacího výrazu odděleného dvojtečkou a názvu stavu ukončeného středníkem. Více podmínek lze napsat pomocí závorek, například `(6w0x5, 2w0x1): accept;`.

Výchozí hodnotu, pokud žádná předchozí podmínka není splněna, lze označit pomocí klíčového slova `default` nebo znaku `_`.

V parser bloku na začátku se mohou objevit lokální proměnné platné pro celý parser. Lokální proměnné lze rovněž vytvořit přímo v kódu stavu. Ty pak budou platné pouze pro daný stav.

V následujícím P4 kódu je popsán jednoduchý parser blok parsující IP protokol verze 6. Kód je celkem intuitivní, v parseru je k dispozici `extern` objekt reprezentující příchozí paket a výstupní parametr, kam se uloží IP hlavička.

```
parser IPv6_parser(packet_in packet, out ipv6_h ipv6)
{
    ethernet_h eth;

    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(eth);
        transition select(eth.ethertype) {
            0x86DD: parse_ipv6;
            default: reject;
        }
    }
    state parse_ipv6 {
        packet.extract(ipv6);
        transition select(ipv6.next_hdr) {
            6: accept;
            17: accept;
            default: reject;
        }
    }
}
```

2.1.5 Control blok

Kontrolní blok slouží k naprogramování logiky pro zpracování a manipulaci s parsovanými hlavičkami a paketem. Například volat různé akce, aplikovat tabulky nebo jen modifikovat vstupní paket.

V kontrolním bloku je před blokem kódu klíčové slovo `apply`. Na začátku lze, jako v parser bloku, libovolně deklarovat lokální proměnné. Rovněž je možné specifikovat akce, které se vykonají nebo tabulky.

Výhodou bloku je možnost využívat `if-else` konstruktu. Ten kupříkladu není podporován v `parser` bloku a podmínka se musí řešit pomocí přidání mezistavu. Vykonávaný blok kódu lze navíc okamžitě přerušit pomocí příkazu `return`.

Následující ukázka je převzatá z [9]. Je v ní popsán zápis akcí a tabulky. Blok kódu, který se začne vykonávat je označen jako `apply`, a aplikuje tabulku na příchozí paket. Pomocí tabulky je rozhodováno, na který výstupní port je paket odeslán a jelikož se jedná o popis routeru, je možné pomocí akce změnit zdrojou MAC adresu.

```
control egress(inout headers hdr, inout metadata meta,
               inout standard_metadata_t standard_metadata)
{
    action rewrite_mac(bit<48> smac) {
        hdr.ethernet.srcAddr = smac;
    }
    action _drop() {
        mark_to_drop(standard_metadata);
    }
    table send_frame {
        actions = {
            rewrite_mac();
            _drop();
            NoAction();
        }
        key = {
            standard_metadata.egress_port: exact;
        }
        size = 256;
        default_action = NoAction();
    }
    apply {
        send_frame.apply();
    }
}
```

2.1.6 Match action tabulky

Match action tabulky jsou tabulky obsahující pravidla. Při spuštění zařízení jsou tabulky prázdné a vyplňují se za běhu. Tabulky jsou rozhodovací logikou v otázce, co se má stát s paketem, které zařízení přijalo. V předchozí ukázce kódu byl příklad použití tabulky.

Pravidlo se skládá z klíče a akcí. Klíčem v tabulce je vyparsované políčko, například IP adresa. Akce jsou definovány v P4 programu a jsou vykonány, když se políčko v paketu shoduje s některým klíčem tabulky. V P4 lze požadovat jako algoritmus hledání shody například přesnou shodu (exact), ternary nebo longest preffix match (lpm).

2.1.7 package

V P4 dokumentaci se o `package` typu uvádí *A package type describes the signature of a package* [7]. Jedná se v podstatě o konstrukt, který umožňuje

shlukovat dohromady, do balíku, různé typy. Nejčastěji se bude jednat o typy `parser` a `control`, ale lze i `package`.

Tento konstrukt bude v implementaci užitečný, neboť pomocí něj lze vytvořit balík komponent stejného zaměření. Například se může jednat o balík podporovaných pluginů ve flow cache, balík s komponentami flow cache nebo balík s komponentami pro export. Ukázkový kód byl převzat z [10].

```
parser parse<H>(packet_in packet, out H headers);
control filter<H>(inout H headers, out bool accept);

package ebpfFilter<H>(parse<H> prs, filter<H> filt);
```

V každém P4 programu musí být přítomna instance balíku, která se jmenuje `main`. Parametry balíku má každá architektura vlastní a balík slouží podobným účelům jako funkce `main` v C.

```
package ebpfFilter<H>(parse<H> prs, filter<H> filt);
ebpfFilter(myParser(), myFilter()) main;
```

2.1.8 P4 kompilátor

V současné době existují dva kompilátory P4 programu. Jeden slouží pro kompilaci jen $P4_{14}$ programů a je napsaný v pythonu [11]. Tento kompilátor není již udržovaný a neplánuje se v budoucnosti přidání podpory pro verzi $P4_{16}$. Druhý kompilátor podporuje kompilaci programů obou verzí jazyka a je napsaný v C++ [5]. Je stále ve vývoji a obsahuje několik backendů mezi které patří generátor grafů z `control` a `parser` bloků, generátor eBPF programů a generátor přepínačů.

V této práci bude využit novější backend podporující verzi 16. Ten ve stručnosti obsahuje sadu tříd pro procházení P4 programu ve formě stromu zvaného IR. Mezi takové třídy patří velké množství `visitor` patternů jako Inspector (pouze čtení uzlů), Modifier (mění data v uzlech) a Transform (mění strukturu stromu).

2.2 IPFIX

IPFIX neboli Internet Protocol Flow Information Export [12] je standardizovaný protokol pro přenos záznamů o síťových tocích. Tento protokol má oproti jiným výhodu v tom, že umožňuje definovat libovolné množství vlastních šablon s exportovanými políčky.

Uživatelsky definované šablony jsou zahrnuty v takzvaném setu, kterých může být několik. Šablona v IPFIX exportu obsahuje identifikátor šablony a množinu políček. Políčko je definováno trojicí skládající se z identifikátoru organizace (PEN), identifikátoru políčka uvnitř organizace a délky políčka v bajtech. Políčka mohou mít fixní délku a nebo proměnlivou, například pro

přenos řetězců. Definované sety s šablonami jsou periodicky odesílány na kolektor.

Bajty jednotlivých políček fixní délky se v paketu musí objevit vždy. Políčka dynamické délky se vyplní tak, že se jako první uvede celkový počet bajtů elementu a za něj se vloží samotná data. Data se pro jednotlivé šablony musí do paketu vyplnit v tom samém pořadí v jakém byly políčka šablon odeslány na kolektor toků. Při vyplňování se musí přesně dodržet jednotlivé délky políček, jinak je kolektor špatně interpretuje.

2.3 Exportér `flow_meter`

V rámci mé bakalářské práce [2] jsem vyvíjel open source exportér síťových toků do projektu NEMEA. Systém NEMEA je projekt zabývající se proudovou bezpečnostní analýzou síťového provozu. Celý systém je modulární a skládá se z jednotlivých modulů, detektorů a reportéru, které mohou mezi sebou jednoduše komunikovat. Mezi tyto moduly patří exportér `flow_meter`, který v tomto systému hraje klíčovou roli, neboť poskytuje ostatním modulům data ve formě síťových toků.

Základní architektura každého exportéru síťových toků obsahuje komponentu pro získávání paketů. Ty lze získat buď ze síťového rozhraní a nebo ze souborů uložených ve formátu PCAP nebo PCAPNG. Přečtené pakety jsou vloženy do komponenty flow cache, kde se agregují do flow záznamů a na základě splnění daných pravidel se předají komponentě, která se stará o export těchto záznamů.

Mezi pravidla patří nejčastěji kontrola takzvaných aktivních a neaktivních timeoutů. Aktivní timeout nastane, jestliže rozdíl časů přijetí prvního a posledního paketu daného toku je větší než stanovená mez. Neaktivní timeout naopak nastane, jestliže rozdíl aktuálního času a času posledního přijatého paketu daného toku je větší než stanovená mez. Aktivní timeout je jednodušší na kontrolu, neboť je kontrolován s příchodem paketu. Naopak neaktivní timeout je nutné kontrolovat asynchroně.

Co se týče komponenty pro export, nejvíce využívanými formáty pro posílání záznamů o tocích jsou IPFIX a NetFlow. Ve `flow_meter`u je navíc používán interní formát zvaný UniRec, který ale v této práci nebude podporován v generovaném exportéru. Podporován bude zatím pouze univerzální exportní formát IPFIX.

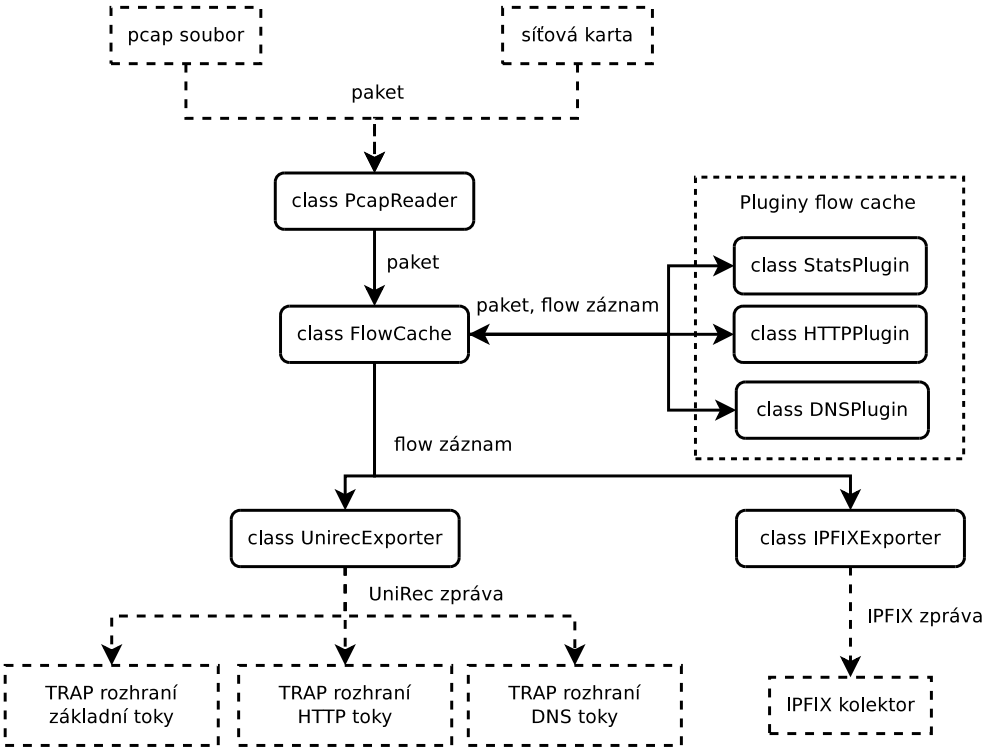
Kromě výše zmíněné architektury `flow_meter` navíc obsahuje vylepšenou flow cache s API pro podporu L7 parsovacích pluginů. Pluginy jsou schopné parsovat payload paketů ze sedmé vrstvy ISO/OSI modelu a k existujícím flow záznamům vkládat informace navíc. Tyto informace poté poslouží detektorům k lepší analýze provozu a detekci hrozeb na síti.

Mezi pluginy, které `flow_meter` obsahuje, patří pluginy pro parsování HTTP, HTTPS (extrahuje jméno serveru z SNI rozšíření hlavičky při usta-

vování spojení), DNS, SMTP, SIP a NTP. Flow záznamy obohacené o nové informace z pluginů, je exportér schopný exportovat ve formátu IPFIX a UniRec.

2.3.1 Architektura

Architektura je zobrazena na obrázku 2.1, který původně pochází z mé bakalářské práce. Zobrazuje klíčové komponenty v C++ implementaci exportéru. Mezi ně patří komponenty pracující buď s pakety a nebo se záznamy o tocích. Tyto komponenty jsou v této podsekci analyzovány.



Obrázek 2.1: Architektura exportéru `flow_meter`

Parser paketů v tomto exportéru je vysoce optimalizovaný a je zaměřen na klasické protokoly síťové a transportní vrstvy. Parser a ani flow cache neumožňuje v aktuální implementaci zpracovávat tunelovaný provoz. Vysoká optimalizace, přehlednost a snadná rozšiřitelnost většinou nejdou dohromady a z tohoto důvodu je často obtížné přidávat podporu pro nové protokoly. Stejný problém platí i pro ostatní komponenty exportéru, které se zparsovaným paketem pracují.

Exportér popsaný v P4 problémy s rozšiřitelností a přehledností odstraní. Cena za toto vylepšení bude v podobě nižší rychlosti parsování. Jak moc

bude mít vysokoúrovňový popis dopad na rychlost bude předmětem měření v samostatné kapitole.

Analyzujeme-li flow cache zjistíme, že obsahuje tři důležité části pro práci s parsovaným paketem. Těmito částmi je funkce pro vytvoření klíče toku, vyplnění záznamu toku pokud neexistuje a funkce pro aktualizaci nalezeného toku ve flow cache. Všechny tři funkce pracují s políčky z extrahovaných hlaviček z parseru.

V první funkci se políčka definující tok řetězí za sebou v poli bajtů. Toto pole následně vstoupí do hash funkce a jejím výstupem je číslo, identifikátor, toku. Tímto číslem lze tok, v interní struktuře pro ukládání toků, nalézt.

Při hledání buď tok existuje a je aktualizován, nebo neexistuje a je potřeba vytvořit záznam. Záznam se vytvoří tak, že se volná struktura se všemi daty o toku vynuluje a pak se vyplní políčky ze zpracovávaného paketu. Vyplňují se zejména statické položky jako jsou IP adresy, porty a protokol. Čítače se vyplní zavoláním aktualizační funkce.

Aktualizační funkce toku, jak bylo zmíněno, upravuje čítače. Jedná se o čítače celkového počtu paketů v toku a celkový počet bajtů v paketech od síťové vrstvy nahoru.

V P4 je potřeba tyto funkce pro práci s extrahovanými políčky dodat. Díky tomu bude možné jednoduše reagovat na změnu v parsovaných hlavičkách a přizpůsobit vytváření klíče nebo vytvářet úplně nové flow, například agregovat navíc provoz pomocí VLAN identifikátoru.

Další komponentou jsou parsovací pluginy zpracovávající aplikační protokoly. Tyto pluginy mají ve flow cache k dispozici pět API callback funkcí pro práci s pakety a toky. Jedná se o funkce pre create (speciální případy použití), post create, pre update, post update a pre export.

Funkce z předchozího odstavce se volají v určitých okamžicích. Pre create se volá před vytvořením toku a používá se pro speciální parsovací pluginy z nižších vrstev, například pro ARP. Funkce post create se volá po vytvoření záznamu o toku a jeho umístěním do flow cache. V této funkci se provádí parsování, jelikož velká část protokolů obsahuje jeden dotaz na server a jednu odpověd pro klienta a tím tok končí. Další funkce, pre update, se zavolá před aktualizací toku a je nejvyužívanější. V ní se zpracovávají kontinuální provoz protokolů jako například SMTP, kde se posílá několik příkazů v jedné komunikaci. Předposlední funkcí je post update, zavolá se po aktualizaci toku a většinou ji pluginy moc nepoužívají. Poslední funkcí je pre export, ta se zavolá před exportováním záznamu na kolektor toků a rovněž se moc nevyužívá v aktuálních pluginech.

Jako parametr těchto funkcí je ukazatel na payload příchozího paketu a ukazatel na záznam o toku. Výjimkou je funkce pre export, kde zavolání probíha asynchroně s příchodem paketu a tudíž má pouze ukazatel na záznam. Díky nim je možné v exportéru zparsovat payload paketu a nové informace vložit k záznamu o toku jako rozšiřující záznam. Případně již existující rozšiřující záznam upravit.

Vzhledem k faktu, že se nejvíce využívají funkce post create a pre update, budou v implementaci P4 exportéru zatím podporovány jen tyto dvě. V případě potřeby složitějšího pluginu je lze v budoucnosti doimplementovat. V P4 je nejvhodnější způsob reprezentace pluginu `parser` blokem. Ovšem v implementaci je nutné dodat funkce pro práci s řetězci, a to pomocí extern bloku. Více informací o aplikačních pluginech je v sekci o aplikačních protokolech, kde je vysvětleno, co se jak parsuje a jaké jsou požadavky pro podporu v P4.

Poslední komponenta slouží pro export záznamů o tocích na kolektor. Popsaný bude pouze export pomocí IPFIX. Ten je realizován díky využití existující implementace exportního pluginu pro FlowMon exportér [13], jejímž autorem je Petr Velan.

Každý plugin si při startu exportéru definuje šablony obsahující množinu exportovaných políček. Dostupné šablony exportéru se periodicky odesílají na kolektor. Jakmile dojde k exportu toku, příslušný plugin u toků obsahující jeho rozšiřující záznamy vyplní IPFIX paket políčky dle pořadí v šabloně, které zparsoval. Toho je docíleno callback funkcí, jelikož pouze v pluginu je známo, jak jím definovanou šablonu vyplnit.

V P4 lze specifikaci šablon docílit pomocí funkcí v `extern` bloku a program se všemi šablonami je možné reprezentovat pomocí kontrolního bloku. Navíc vyplnění příslušné šablony pluginem lze zařídit tak, že každý plugin bude obsahovat vlastní program pro vyplnění, také pomocí kontrolního bloku. S danými třemi funkcemi u pluginů post create, pre update a vyplnění šablony lze v P4 programu pracovat ve formě balíku, `package`.

Poslední je třeba dořešit funkci pro vyplnění základního záznamu o toku, tedy bez aplikačních rozšířeních. I v tomto případě je vhodné použít `control` blok.

2.4 Aplikační protokoly

Aplikační protokoly nejsou v této práci detailně popisovány. Je zmíněna základní struktura a princip fungování každého protokolu. Analýza poslouží pro stanovení požadavků pro parsovací pluginy, které tyto protokoly budou zpracovávat.

Jedním z cílů práce je snažit se pomocí P4 konstruktů jednoduše napsat plugin, který by daný aplikační protokol zpracovával. Exportér `flow_meter` obsahuje celkem 6 aplikačních parsovacích pluginů. Jedná se o pluginy pro parsování DNS, SMTP, HTTP, HTTPS, SIP a NTP.

Podpora pro aplikační protokoly bude v této práci řešena zejména pomocí regulárních výrazů. Ne všechny protokoly lze pomocí P4 bloků popsat. Analyzovány jsou v následujících podsekcích.

2.4.1 DNS

DNS protokol se využívá pro překlad doménových jmen. Komunikace probíhá ve většině případů pomocí UDP protokolu, přenášet data lze i pomocí TCP protokolu. V rámci jednoho spojení je odeslán jeden požadavek pro server a jedna odpověď.

Struktura požadavku i odpovědi je stejná. Jedná se o strukturované zprávy, obsahující hlavičky, které by se daly parsovat klasickým způsobem (funkce z `packet_in`) pomocí `parser` bloku. Problém je ovšem v ukládání doménových jmen. Ty jsou uloženy speciálním způsobem kvůli šetření místa.

Komprese doménových jmen je dosaženo pomocí takzvaných ukazatelů. Máme-li domény `images.example.com` a `mail.example.com`, jsou v paketu uloženy na různých místech jako `images` a `mail` a `example.com`. Za názvem poddomén `images` a `mail` se nachází ukazatel (offset od začátku paketu) na doménu `google.com`. Tímto způsobem jsou domény komprimovány.

Právě tato komplikace neumožňuje v P4 programu jednoduché zjištění celého doménového jména. Ty jsou přitom klíčové, neboť je exportér `flow_meter` pomocí DNS pluginu exportuje. Plugin je možné v novém exportéru přesto podporovat, nebude jej možné popsat pomocí jazyka P4, ale pomocí C.

2.4.2 SMTP

SMTP protokol slouží k přenosu pošty mezi klientem a poštovním serverem nebo mezi servery. Spojení probíhá přes protokol TCP a je nešifrované. V rámci jednoho TCP spojení jsou odesílány příkazy na server a přijímány odpovědi od něj.

Příkazy pro server jsou uvedeny na začátku zprávy, například HELO, RCPT atd. Poté ve většině případů následuje mezera a data příkazu. Nakonec je příkaz ukončen sekvencí znaků `\r\n`.

Odpovědi od serveru mají podobný formát. Na začátku řádky s odpovědí je číslo určující výsledek operace. Následuje mezera a hláška v textové podobě.

V exportéru `flow_meter` se zpracovávají a zaznamenávají výskyty příkazů od klienta a jejich data jako například adresát u příkazu RCPT. V odpovědích od serveru se zpracovávají pouze návratové kódy a hledá se, jestli se v odpovědi nevyskytuje řetězec SPAM, přičemž nezáleží na velikosti písmen.

Pro zachování stejné funkcionality je potřeba umět v P4 převést řetězec na číslo kvůli zpracovávání zmíněných návratových kódů. Pro hledání klíčového slova SPAM postačí vhodně upravený regulární výraz. A extrakce parametrů a příkazů lze dosáhnout například pomocí specifikace takzvaných capture groups, používaných v POSIX regulárních výrazech. Stačí jednoduchým způsobem označit závorkami (a) text, který se má extrahovat.

Příklad jednoduchého regulárního výrazu pro parsování příkazu pro server je `([A-Za-z]{4,})([ ]|"\r\n")` (syntaxe regulárních výrazů použitá v práci je z generátoru re2c). Výraz také počítá s případem, kdy klient odešle příkaz

QUIT, který neobsahuje žádná data (a tudíž ani mezeru za příkazem). Parsování odpovědi od serveru je možné vyřešit pomocí `([0-9]{3})[ -]`. Pomlčka značí, že následuje za kódem komentář.

2.4.3 HTTP

Protokol pro komunikaci s webovými servery se nazývá HTTP. HTTP protokol funguje na principu odeslání požadavku a příjmu odpovědi od serveru. Jedná se o čistě textový protokol.

Formát zprávy začíná tak, že na začátku je obsažen jeden řádek s HTTP hlavičkou. Poté následují řádky s datovými políčky (*Host*, *Referer*, *User-Agent* atd.), které jsou ve formátu název políčka, dvojtečka a data políčka. Veškeré řádky jsou odděleny sekvencí znaků `\r\n`. Pokud se objeví prázdný řádek, je to signalizace konce hlavičky.

V prvních verzích HTTP se pro každý požadavek vytvořilo nové TCP spojení, které se po obdržení odpovědi již nevyužívalo. Neefektivnímu využívání zdrojů mělo zabránit zavedení tzv. TCP keepalives. Slouží k tomu, aby se spojení po prvním požadavku neuzavíralo a využilo se pro všechny požadavky.

Díky TCP keepalives je potřeba sledovat payload a v případě nalezení dalšího požadavku nebo odpovědi tok okamžitě exportovat ještě před aktualizací. Signál, že se má tok exportovat by se v P4 mohl indikovat pomocí výstupní proměnné nebo například přechodem do vlastního speciálního stavu.

Co se týče extrakce políček v hlavičce, postačí stejný způsob jako je zmíněn v analýze SMTP protokolu. Například pro parsování prvního řádku HTTP hlavičky v požadavku by se mohl použít (zde zjednodušený) regulární výraz ve tvaru `("GET"|"POST")[ ]([^ ]*)[ ]"HTTP"[/][0-9][.][0-9]"\r\n"`. Tímto způsobem lze zparsovat například řádek ve tvaru `GET /gzip HTTP/1.1`.

Je ale v implementaci potřeba volit takovou knihovnu pro regulární výrazy,která bude capture groups podporovat. A v API konečného automatu, který regulární výraz bude přijímat, mít přístup k proměnným s řetězci obsahující data z capture groups.

Při parsování políček hlavičky by se mohl použít regulární výraz ve tvaru `([^:]*)": "(.*)"\r\n"`. `flow_meter` exportuje jen data políček *Host*, *Referer*, *User-Agent* a *Content-Type* a k jejich odlišení mezi sebou lze rovněž využít regulárního výrazu. Máme-li v C proměnné obsažen řetězec s názvem políčka například *Accept*, dá se pro hledání shody použít regulární výraz `"Accept\x00"`. Ten porovná řetězec i s ukončujícím nulovým znakem a nestane se, že by se chybně vyhodnotila shoda například s *Accept-Encoding*.

2.4.4 HTTPS

U HTTPS se HTTP protokol přenáší šifrovaným tunelem až k serveru nebo klientovi. Rozdíl je zde navíc v tom, že před odesíláním požadavků je nutné ustavit spojení.

HTTPS plugin v exportéru `flow_meter` zpracovává pouze jedno políčko. Jedná se o políčko server name indicator (SNI), což je řetězec s doménovým názvem serveru nacházející se v takzvaném hello paketu. Hello paket je odeslán před ustavením tunelu a slouží k výměně parametrů jako jsou podporované šifrovací algoritmy atd. Paket je strukturovaný, obsahuje množství políček a hlaviček fixní velikosti, případnou dynamickou velikost políčka lze zjistit a nepotřebná data přeskočit.

Jedno zmíněné políčko je potřeba zparsovat a uložit jako řetězec. Vzhledem k tomu, že velikost políčka je uložena v 16 bitové proměnné a řetězec není ukončen terminálním znakem, nelze regulární výraz použít. V implementaci P4 programu bude potřeba přidat podporu, funkci, pro extrakci řetězců z paketu pomocí zadané délky.

2.4.5 SIP

Formát hlavičky SIP protokolu je úplně stejný jako HTTP. Spojení je ale realizováno pomocí protokolu UDP a požadavky se posílají kontinuálně. Je zde také potřeba od sebe oddělovat jednotlivé požadavky a toky exportovat jako je tomu u HTTP protokolu.

2.4.6 NTP

Protokol NTP se používá k synchronizaci času mezi síťovými zařízeními. Využívá se zde protokol UDP a v rámci jednoho spojení je opět odeslán jeden požadavek a přijímána jedna odpověď.

Paket obsahuje jedinou velikostně fixní NTP hlavičku. Ve zprávě nejsou obsaženy žádné textové řetězce.

V exportéru `flow_meter` jsou exportované políčka celé hlavičky. Navíc jsou některá políčka převáděny na řetězec, kupříkladu všechny časové značky (po konverzi ze speciálního formátu) a referenční identifikátor (IP adresa na řetězec). Z těchto důvodu by bylo potřeba mít v P4 programu podporu pro práci s řetězci jako operace zřetězení a vkládání konstant.

2.5 Existující řešení

V této sekci jsou popsány existující P4 aplikace. Nejčastěji se bude jednat o aplikace, kde klíčovou roli hraje parsování a filtrování paketů.

eBPF backend [10] je backend pro generování C kódu, který se může zkompilovat do podoby rozšířených paketových filtrů (extended Berkley Packet Filter). Tyto filtry se poté mohou nahrát do linuxového jádra a využít pro rychlé filtrování síťového provozu.

Architektura se skládá z `parser` bloku pro parsování paketů a `control` bloku s rozhodovací logikou pro filtrování. Pro účely této práce je zají-

mavý parser paketů. Ten je backendem překládán jako funkce v C, stavy konečného automatu jako bloky kódu a přechody mezi nimi jsou realizovány pomocí `goto` příkazu. Rozhodovací logika pro přechod do dalšího stavu je překládána jako `switch`.

XDP backend [14] je zkratkou pro eXpress Data Path a jedná se o backend velice podobný eBPF.

XDP program je speciální případ eBPF programu pro zpracování paketů na nejnižší úrovni síťového stacku. XDP byl primárně vyvinut pro rychlé rozhodnutí, zda se má přijatý paket zahodit před tím, než se vyčerpá příliš mnoho systémových zdrojů. To slouží k prevenci útoků typu denial of service [15].

FPP backend [16] je backend pro generování flexibilního parseru (Flexible Packet Parser) v jazyce C. Generovaný kód parseru vychází z eBPF backendu.

Tento parser je schopný parsovat hlavičky paketů a ty, které si vývojář v P4 zvolí, extrahovat na výstupu parseru. Výstup hlaviček v parseru je ve formě spojového seznamu tak, jak se hlavičky při parsování nacházely. Tato P4 aplikace umožňuje využít flexiblity jazyka P4 pro jednoduché psaní parseru paketů a dovoluje zpracovávat zapouzdřené pakety (tunelovaný provoz).

Tento generátor parseru v C vznikl pro použití v exportérech síťových toků. Klasické exportéry mají většinou omezený parser paketů, který nedovoluje zpracovávat zapouzdřený provoz. Jejich společným problémem je rovněž horší rozšiřitelnost a schopnost reagovat na příchod nových protokolů. Práce je dostupná v [17].

Způsob a styl generovaného C parseru, pomocí tohoto backendu, bude v této práci využit pro exportér a navíc rozšířen.

VHLD backend [18] je backend pro generování VHDL kódu s parserem paketů z P4 popisu. Architektura pro zpracovávání paketů se nazývá HFE M2 a na síťové kartě s FPGA lze dosáhnout rychlosti zpracování provozu až 100 Gb za sekundu.

Vyvažování zátěže [19] je bakalářská práce věnující se využití jazyka P4 pro vyvažování zátěže na síti. Práce je demonstrována na rozložení zátěže pro HTTP servery. Diskutována je rovněž portace na 100 Gb/s COMBO síťovou kartu.

BMV2 backend [20] (Behavioral Model version 2) slouží pro generování kódu pro behavioral model [21]. Behavioral model je prostředí pro testování a ladění P4 programů. K dispozici je několik architektur síťových zařízení jako jsou simple switch, simple router, L2 switch a PSA (Portable Switch Architekture).

Krom ostatních backendů jsou eBPF a BMV2 výchozí součástí $P4_{16}$ kompilátoru.

2.6 Generátor regulárních výrazů

Pro účely zpracování aplikačních protokolů je v implementaci využito generátoru konečného automatu přijímající řetězce popsané regulárním výrazem. Jako generátor jsem zvolil re2c [22]. Ten překládá regulární výraz do konečného automatu v C jazyce, který je reprezentovaný pomocí `goto`, `if` a `switch` příkazů.

Mezi hlavní výhody re2c patří rychlost, o které autoři tvrdí, že je alespoň taková jako u optimalizovaného ručně napsaného automatu. Další výhodou je snadné ladění narozdíl od automatů implementovaných pomocí tabulky. A nakonec podporuje POSIX capture groups.

Použitá syntaxe regulárního výrazu by mohla být matoucí. V práci se pracuje se syntaxí, která je dostupná na [23].

Kapitola 3

Návrh

V analýze byl popsán jazyk P4 a architektura exporéru `flow_meter`. Kromě analýzy komponent exportéru byly uvedeny i možnosti, jak by se tyto komponenty daly teoreticky popsat pomocí P4. V této kapitole je věnována pozornost návrhu na architekturu nového exportéru popsaného v P4.

3.1 Architektura

Popisovat celý exportér v P4 by bylo komplikované a nepřehledné, velká část kódu by se nakonec nevyužívala, nemodifikovala tak často jako jiné části. Z tohoto důvodu budou popsány pouze čtyři klíčové části exportéru. Mezi ně patří parser paketů pro který použiji `parser` blok. Dále funkce flow cache, mezi které patří vytvoření klíče toku, vyplnění a aktualizace záznamu o toku. Pro tyto funkce použiji `control` blok, který umožňuje využívat `if-else` podmínek. K popisu IPFIX exportu použiji jeden `control` blok pro definování šablon a druhý blok pro naprogramování vyplnění šablony záznamem o toku. Pluginy pro parsování protokolů aplikační vrstvy vyžijí jeden `parser` blok pro post create funkci a druhý blok pro pre update funkci. Každý plugin bude zároveň obsahovat jeden `control` blok, kde se specifikuje, jak se má IPFIX paket parsovanými daty vyplnit.

3.2 Kompilátor P4

Exportér bude popsaný v novějším jazyce $P4_{16}$ a jako kompilátor bude použit [5]. Do kompilátoru vytvořím vlastní backend, který bude překládat z P4 program do C.

Generování kódu bude rozdělen na pět hlavních části. V první části se budou překládat P4 typy do C a v ostatních částech se budou překládat čtyři vyjmenované komponenty z předchozí sekce. Výsledné soubory se zdrojovými kódy vzniknou tak, že si napíši kostru celého exportéru v C a na vybranná

místa doplním přeložený kód pomocí backendu. Tím bude možné po vygenerování vzniknout spustitelná aplikace.

Kostra exportéru bude získána tak, že použiji zdrojové kódy existujícího exportéru `flow_meter` a ty modifikuji. Modifikaci provedu tím způsobem, že přepíši exportér z C++ do C, pro jednodušší generování, a vybranná místa v architektuře uzpůsobím pro použití s generovanými zdrojovými C kódy z backendu. Tím zůstane velká část nově generovaného exportéru stejná jako s původním a bude snažší oba exportéry porovnat. Pluginy původního exportéru se pokusím přepsat do P4 a vylepšit.

3.3 Parser paketů

Pro parser paketů bude použit existující implementace v FPP backendu. V tomto backendu [17] jsem rozšířil parser paketů z existujícího eBPF backendu o možnost vygenerovat parser paketů v C. Tento parser podporuje možnost parsovat na výstup funkce tunelovaný provoz. Rovněž pomocí speciální struktury umožňuje specifikovat, které hlavičky paketů se objeví na výstupu, který je implementován pomocí spojového seznamu. Dále použitý parser rozšířím o lepší práci s hlavičkami ve spojovém seznamu a přidám podporu pro extrakci bitů paketu do 64 bitové proměnné. Rychlost parseru z FPP navíc optimalizuji tím, že odstraním alokace paměti.

3.4 Flow cache

Výstup z parseru, spojový seznam, bude zpracován pomocí kontrolního bloku, který se aplikuje na každý prvek tohoto seznamu. V této části vytvořím nový `extern` blok, který bude reprezentovat potřebné funkce pro práci s flow cache. Zároveň přidám do P4 speciální strukturu, reprezentující záznam ve flow cache, pomocí které si vývojář může přidávat nové políčka do záznamu.

3.5 Export toků

Do komponenty pro export toků pomocí IPFIX protokolu přibude `extern` blok pro práci s šablonami a možností vyplňovat políčka do IPFIX paketu.

3.6 Pluginy

Pro pluginy bude vytvořen `extern` blok reprezentující funkce pro práci s textovými a obyčejnými protokoly v pluginech. Funkce pro práci s textovými protokoly budou implementovány pomocí regulárních výrazů, k tomu bude využit existující generátor konečného automatu re2c z výrazu.

V každém pluginu v P4 bude muset být definována struktura obsahující parsovaná pole. Tato struktura se při úspěšné zparsování (přechod do stavu `accept`) uloží k záznamu o toku ve flow cache.

V pluginech bude k dispozici `extern` blok z předchozí komponenty pro IPFIX export. To umožní v pluginu vyplnit IPFIX paket daty, které plugin zparsoval.

Kapitola 4

Implementace

V této kapitole je popsána implementace backendu překladače a generovaný kód exportéru. Pozornost je také věnována vyskytlým problémům a jejich vyřešení.

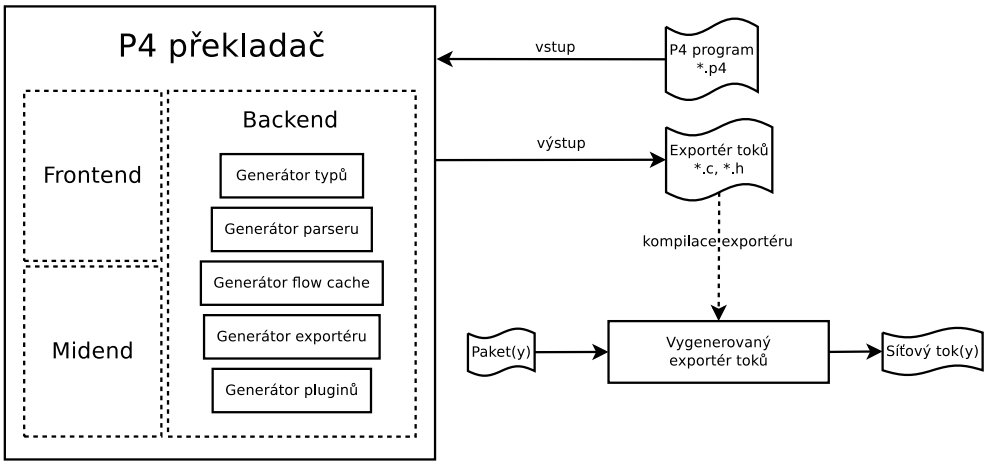
Celý backend P4 překladače je napsaný v jazyce C++ a jím generovaný exportér je napsán v jazyce C a lze jej rozdělit na 4 klíčové části. Těmito částmi jsou parser paketů, flow cache, exportér toků a aplikační pluginy.

V následující sekcích je popsáno, jak se překládají jednotlivé datové typy z P4 jazyka do jazyka C, architektura generovaného parseru paketů spolu se specifikací hlaviček k parsování a výstupní formát parsovaných hlaviček. Dále architektura flow cache, jak se v P4 specifikuje vytvoření unikátního ID toku a jaké hlavičky musí být přítomny pro jeho vytvoření a nebo aktualizaci již existujícího. V další části je popsáno, jak se definují šablony pro IPFIX exportér a jak se tato šablona dá pomocí P4 vyplnit a odeslat, exportovat, na kolektor toků. Poslední částí je sekce o aplikačních pluginech. Popsán je způsob definování proměnných pro ukládání řetězců, způsob extrakce řetězců pomocí regulárních výrazů a popis algoritmu fungování pluginu.

4.1 Architektura překladače

Co se týče architektury překladače, ta je popsaná na obrázku 4.1. Je zde zobrazen překladač se třemi klíčovými částmi. Jedná se o frontend, midend a backend. Frontend se stará o zpracování vstupního P4 programu a midend provádí vybranné optimalizace. Po optimalizaci vstupuje upravený program do poslední části, kde jsou vypsány generátory jednotlivých komponent exportéru, které jsem vytvořil.

Po kompilaci P4 programu jsou vygenerované soubory zdrojových kódů a hlavičkových souborů. Dále je nutné učinit jeden krok, a to zpracovat soubory s koncovkou `re` programem re2c pro převod regulárního výrazu na automat. Ty se poté mohou zkompilovat libovolným C překladačem do podoby spustitelné aplikace.



Obrázek 4.1: Architektura překladače.

Důležitou částí je, jak se zdrojové kódy exportéru toků generují do souborů. Lze je generovat dvěma způsoby. Buď je celé generovat přímo z kompilátoru a nebo generovat minimální část přeloženého P4 kódu a tu vložit do existujících souborů se šablonami zdrojových kódů, kde se doplní na vyznačená místa.

První způsob má výhodu ve vytvoření exportéru bez nutnosti vlastnit soubory se šablonami. Nevýhodou je naopak nemožnost snadné modifikace kódu exportéru, protože kód se musí vložit do zdrojových kódu překladače. Nakonec jsem zvolil druhou možnost, která umožňuje snadnou modifikaci při využívání. K vyplňování šablon se používá šablonovací knihovna Inja C++ [24], která přijímá JSON s přeloženým C kódem.

Generování kódu pomocí šablon je velice flexibilní. Při použití výše uvedené knihovny stačí ve zdrojových kódech vyznačit místa, například tímto způsobem `{{ plugin/name }}` a při generování se zde doplní zdrojový kód z JSON specifikace. Lze použít i `for` cykly a podmínky `if`. Tyto konstrukty usnadnily vytváření zdrojových kódu exportéru.

Překladač má k dispozici dva parametry. První slouží k nastavení složky, kam se vygenerují zdrojové kódy s exportérem a jmenuje se `--gen-dir`. Druhý slouží k nastavení cesty ke složce s šablonami zdrojových kódů exportéru a parametr se nazývá `--template-dir`.

Překladač potřebuje k vygenerování finálních zdrojových kódů šablony s C kódem exportéru. Jednotlivé komponenty překladače využívající šablony jsou zobrazeny v tabulce 4.1. Tyto soubory je potřeba dodat, jinak kompilace P4 programu skončí s chybou.

Důležité je poznamenat, že kompilátor s tímto backendem akceptuje pouze P4 programy, které jako `main package` obsahují balík se stejnými parametry, jako je uvedeno v následujícím kódu. Tento balík se v kódu jmenuje `top` a sdružuje všechny důležité části P4 programu. Zéjména je důležité dodžet jména

Tabulka 4.1: Komponenty využívající soubory šablon

komponenta	šablony
-	main.c.tmplt, Makefile.tmplt
generátor typů	types.h.tmplt
generátor parseru	parser.c.tmplt, parser.h.tmplt, xxhash.c.tmplt, xxhash.h.tmplt
generátor flow cache	cache.c.tmplt, cache.h.tmplt
generátor exportéru	ipfix.c.tmplt, ipfix.h.tmplt
generátor pluginů	plugin.c.tmplt, plugin.h.tmplt, regex.c.re.tmplt, regex.h.tmplt

jednotlivých parametrů (`prs, create, update, init,...`), protože v backendu se na ně odkazuje jménem. Inicializovaný balík jménem `main` musí být vždy v programu přítomen a je možné ho inicializovat jako v ukázce.

```
// Parser
parser parse_packet(packet_in packet, out headers_s headers);

// Cache
control cache_create_flow(in headers_s headers, flowcache c,
   out flowrec_s flow, out bool success);
control cache_update_flow(in headers_s headers, flowcache c,
   out flowrec_s flow);

// Exporter
control exporter_init_templates(ipfix_exporter e);
control exporter_export_flow(in flowrec_s rec, ipfix_exporter e);

// Plugins
package cache_plugins(/* ... */);

package top(
   parse_packet prs,
   cache_create_flow create, cache_update_flow update,
   exporter_init_templates init, exporter_export_flow export,
   cache_plugins plugins
);

top(prs(), flow_create(), flow_update(),
   exporter_init(), exporter_export(), plugins) main;
```

4.2 Překlad typů

V jazyce P4 lze nalézt omezené množství datových typů, které je potřeba přetransformovat na typy v C. Datové typy v P4, které mají pro tuto práci

smysl, jsou `header`, `header_union`, `struct`, `bit`, `int` a `bool`. V této sekci popsáno, jak se tyto typy překládají do C.

Překlad z P4 do C není nijak složitý. `header` a `struct` se z P4 přeloží na `struct` v C, `header_union` se přeloží na `union`. Jednoduché typy `bit` nebo `int` se přeloží na odpovídající variantu `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` nebo `int8_t`, `int16_t` atd. Pokud má `bit` větší bitovou délku než 64 bitů, přeloží se na statické pole typu `uint8_t` odpovídající velikosti, podobně se přeloží i typ `int`.

Velikosti přeložených typů se volí tak, aby se do nich daný P4 typ vešel a neplýtvalo se místem. Pokud máme například `bit<20>`, bitová délka se zvýší na nejbližší vyšší mocninu 2 a výsledný typ tím pádem je `uint32_t`. Poslední zbývající typ `bool` se přeloží na `uint8_t`.

Pro aplikační pluginy je práce s řetězci nutností, jelikož velké množství aplikačních protokolů jsou textové a nebo obsahují části s řetězci. Takovými čistě textovými protokoly jsou například HTTP, SMTP nebo SIP a dále protokol, kde je nutné mít možnost ukládat řetězec je HTTPS (SNI).

P4 bohužel práci s řetězci nepodporuje a ani zavedení vlastních datových typů. Jediná možnost, jak do P4 programu vložit řetězec je pomocí takzvaných anotací. Anotace je možné vložit k lokální proměnné v blocích, k členské proměnné ve struktuře a je to způsob, jak do kompilátoru vložit nějakou vlastní informaci. Zapisují se před anotovanou proměnnou pomocí `@jméno(seznam výrazů)` nebo jen `@jméno`.

Vytvořený backend touto prací podporuje anotace `regex` a `string` a to pouze při psaní parseru pro aplikační plugin. Příkladem použití vlastních anotací je specifikace regulárního výrazu `@regex("[a-z]*")` a nebo označení proměnné jako string o velikosti 10 znaků `@string("10")`. Proměnné označné anotací `string` se v tomto backendu mohou objevovat ve funkcích přijímající řetězce jako parametr. Více se lze dozvědět v sekci o implementaci těchto pluginů.

Ostatní prvky P4 programu jako konstanty jsou překládány tak jak jsou. To znamená, že číselné konstanty v P4 programu se objeví jako číslo v C programu. Konstanty jako jsou `true` a `false` se překládají jako 1 a 0. Kompletní seznam přeložených typů je zobrazen v tabulce 4.2.

4.3 Parser paketů

P4 nabízí pomocí parser bloku jednoduchý a snadno rozšiřitelný popis procesu parsování paketů. V paketu se může nacházet tunelovaný provoz například přes L2TP, GRE, VXLAN nebo IP protokol. Při návrhu bylo potřeba přijít na to, jak reprezentovat výstup parseru a jak tyto tunelované pakety zpracovávat. Dále bylo potřeba vyřešit, jak specifikovat požadované hlavičky, které se mají z paketu extrahovat. Této problematice je věnována celá tato sekce.

Tabulka 4.2: Překládané typy

P4	C	popis
struct	struct	
header	struct	
header_union	union	
int	int8_t, int16_t, ...	$délka \leq 64$
bit	uint8_t, uint16_t, ...	$délka \leq 64$
int	int8_t[ ]	$délka > 64$
bit	uint8_t[ ]	$délka > 64$
bool	uint8_t	
@string("32")	uint8_t[32]	libovolná proměnná je řetězcem
@regex(".*")		libovolná proměnná je regex
control	funkce	pouze podporované bloky
parser	funkce	pouze podporované bloky
volání extern fce	specifický kód	pouze podporované funkce
error	enum	pouze knihovní chyby z core.p4

4.3.1 Překlad

Architektura a styl generovaného C kódu je z velké části stejná, jako kód parseru pro eBPF filtery z existujcího backendu [10] a zároveň z FPP backendu [16]. Generování kódu parseru z obou backendů je touto prací navíc vylepšeno.

Vylepšení spočívá například v podpoře extrakce do 64 bitových proměnných. A lepší práci se spojovým seznamem oproti FPP.

P4 `parser` blok se přeloží backendem jako samostatná funkce, která přijímá ukazatel na paket, jeho délku a jako návratovou hodnotu vrací ukazatel na spojový seznam s extrahovanými hlavičkami. Na začátku funkce jsou definovány veškeré lokální proměnné použité v parser bloku, mimo lokální proměnné ve stavech. Architektura přeloženého automatu se sestává z příkazů `goto` sloužící pro přechod mezi jednotlivými stavy a návěstími. Stavy konečného automatu backend přeloží jako blok kódu a název stavu slouží jako návěstí pro `goto` do daného stavu.

Kompilátor podporuje použití standardních funkcí používaných v parserech z `packet_in` `extern` objektu. Mezi ně patří `extract` pro extrakci hlavičky fixní délky do proměnné nebo spojového seznamu, `advance` pro přeskočení bitů (posunutím kurzoru), `length` pro zjištění velikosti paketu a `lookahead`, která extrahuje bity bez posunutí kurzoru. Překlad extrakce bitů je podporován pouze pro P4 datový typ `header` a navíc jsou z hlediska generování kódu rozlišovány 2 scénáře. Těmi jsou extrakce do lokální proměnné parseru a nebo do spojového seznamu na výstup funkce.

V FPP a eBPF backendech jsou podporovány pouze extrakce proměnných do velikosti 32 bitů, ostatní velikosti se extrahují jako pole bajtů. Tato práce navíc přidává podporu pro extrakci do 64 bitové proměnné `uint64_t`. Zmí-

něné extrakce je dosaženo pomocí načtení bajtů z paketu velikosti `uint64_t` a `uint8_t`.

Extrakce bitů jednotlivých políček hlavičky je v implementaci dosaženo pomocí načtení 1, 2, 4 nebo 8 bajtů, případně 9 bajtů, a následného maskování a bitových posunů. Důležité je poznamenat, že proces extrahování bitů z paketu musí začít na adrese dělitelné 8, jinak parser nebude fungovat správně. Z toho vyplývá, že velikost každé hlavičky v počtu bitů musí být násobkem 8. Toto omezení má za výhodu jednodušší a rychlejší implementaci, jelikož se do další extrakce nemusí přenášet informace o offsetu v bajtu.

Pro zjištění, kolik bajtů pro jednotlivé políčko se má celkem extrahovat záleží na bitovém offsetu políčka. Nejmenší datovou jednotkou, kterou lze z paketu načíst je bajt. Z tohoto důvodu záleží, jestli se políčko nachází uvnitř bajtu nebo na pomezí více bajtů. Například pokud 64 bitové políčko se nachází na nultém offsetu v bajtu, pak z paketu stačí načíst 8 bajtů a není potřeba maskování ani bitových posunů. Pokud je offset v bajtu například 4, pak je nutné načíst celkem 9 bajtů, provést posun doprava o 4 bity a vymaskovat prvních 8 bajtů.

K přechodu do dalších stavů slouží v P4 příkaz `transition` buď s názvem stavu a nebo v případě více stavů jsou zvoleny pomocí `select` výrazu. Pro přechod mezi stavy je zvolen obyčejný příkaz `goto`. V případě použití `select` se výběr stavu přeloží do C jako `switch` s výběrovou podmínkou a `case` pro každý přechod.

P4 navíc podporuje v `select` příkazu možnost použití více podmínek pomocí listu výrazů. Místo jednoduchého výrazu `select(ipv4.protocol)` lze zvolit přechod do dalšího stavu pomocí množiny výrazů například použitím `select(ipv4.protocol, ipv4.ihl)`. Backend momentálně podporuje pouze jeden výraz, jelikož `switch` neumožňuje řetězit podmínky. Tento problém lze vyřešit pomocí nahrazení `switch` konstruktu za obyčejné podmínky a množinu výrazů v nich řetězit pomocí logického AND. Vzhledem k tomu, že více podmínek se využije nezřídka, jsem zatím nechal podporu pouze pro jednu.

Dále je v kompilátoru zavedena podpora pro překlad standardních logických, aritmetických a bitových operací. Mezi další překládané prvky jazyka P4 jsou logické konstanty, překládané jako 0 nebo 1, číselné konstanty, jména proměnných, volání extern funkcí z `packet_in` objektu a přiřazení do proměnných.

Prototyp parser bloku je vypsán v následujícím kódu. Vstupním parametrem je příchozí paket, který je reprezentován `packet_in` extern blokem. Výstupním parametrem je specifikace parsovaných hlaviček ve struktuře `headers_s`. Více o specifikaci hlaviček se lze dozvědět v podsekci o extrahovaných hlavičkách.

```
parser Parser(packet_in packet, out headers_s headers);
```

4.3.2 Tunely

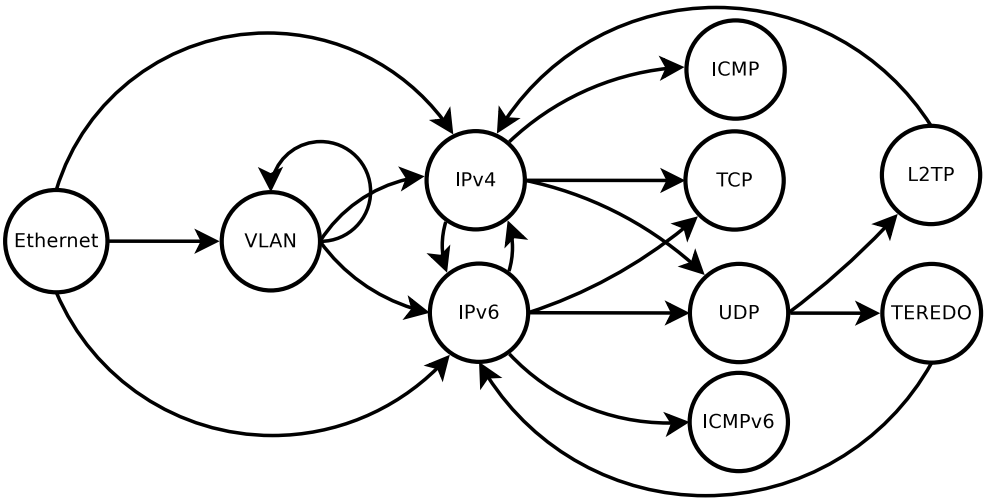
Díky rozmanitosti protokolů a velkému počtu možností pro tunelování provozu je nutné podporovat přidávání více hlaviček na výstup parseru. K tomuto účelu se nejlépe hodí spojový seznam, který se dá snadno rozšiřovat o parsované hlavičky paketů. Hlavičky se ve spojovém seznamu nachází ve stejném pořadí, jako byly zpracovány a díky tomu je možné rozdělovat korektně jeden paket do více toků.

V generovaném kódu exportéru jsem spojový seznam nejprve implementoval pomocí alokování paměti funkcí `malloc`. Toto je problémové řešení, protože při čtení velkého množství paketů menší velikosti se projeví režie spojené s alokací paměti. Výhodou naopak je možnost zpracovat teoreticky neomezené množství hlaviček, ale v běžném provozu se nesetkáme s velkým množstvím tunelů.

Nevýhodu spojenou s režií alokace jsem vyřešil pomocí statické prealokace paměti pro hlavičky paketů a článků spojového seznamu. P4 kompilátor vygeneruje zdrojový kód exportéru se strukturou, kde jsou statická pole pro každou hlavičku a navíc pro články seznamu.

Článek spojového seznamu obsahuje ukazatel na data, označení o jakou extrahovanou hlavičku se jedná, offset hlavičky v paketu a ukazatel na další prvek. Nevýhodou je, že je podporován pouze omezený počet hlaviček v paketu. Krom snížení režie pro alokaci paměti jsou navíc položky umístěny v paměti za sebou a tím se využívá prostorová lokalita cache.

Příklad parseru pro zpracování tunelů je zobrazen na obrázku 4.2. Parser dokáže zpracovat tunely jako jsou 4in6, 6in4, L2TP, TEREDO a jejich libovolnou kombinaci v jednom paketu.



Obrázek 4.2: Příklad parseru

V původním parseru v FPP backendu nebylo možné se odkazovat na na-

posledy zpracovanou hlavičku ve spojovém seznamu. Tento problém jsem v práci vylepšil a nyní je možné se na poslední hlavičku odkázat bez potřeby ukládat data do dočasné proměnné.

4.3.3 Extrahované hlavičky

Paket může obsahovat libovolně velké množství hlaviček a bylo by zbytečné je extrahovat všechny. V P4 programu, který přijímá vyvtvořený backend, lze označit hlavičky paketů pro extrakci tím, že je programátor přidá jako členskou proměnnou do speciální struktury. Tato struktura se nazývá `headers_s` a jedná se o obyčejný datový typ `struct`. V backendu lze poté s těmito hlavičkami jednoduše pracovat a přidávat je do spojového seznamu.

Extrakci do spojového seznamu vyjádří programátor v P4 klasickým voláním funkce `extract`. Je ale nutné jako argument funkce se odkázat na danou hlavičku ve výstupním parametru parser bloku. Například zavoláním `packet.extract(headers.ipv4)`. S extrahovanými hlavičkami lze dále v P4 programu stejným způsobem pracovat, více se lze dozvědět v sekci o flow cache. Ukázka definice struktury je v následujícím kódu.

```
struct headers_s {
    ethernet_h eth;
    ipv4_h ipv4;
    ipv6_h ipv6;
    tcp_h tcp;
    udp_h udp;
    icmp_h icmp;
    icmpv6_h icmp6;
    payload_h payload;
}
```

V předchozím kódu se objevila nestandardní hlavička `payload_h`. Tato hlavička neobsahuje žádné členské proměnné, což P4 povoluje, a v exportéru plní speciální funkci. Slouží k indikaci kde začíná payload paketu, aby bylo možné parsovacím pluginům předat korektně ukazatel na bajty paketu. To je možné zjistit pomocí offsetu od začátku paketu, kde byla hlavička zparsována. Ten se nachází v článku spojového seznamu spolu s ukazatelem na data hlavičky a typem.

4.3.4 Přidané protokoly

V ukázkové implementaci exportéru v P4 jsem přidal podporu pro několik tunelovacích protokolů. Mezi ně patří ETHERIP, EoMPLS, PPTP, 4in6, 6in4, GRE, L2F, L2TP, VXLAN, GTP (verze 0, 1 a 2), TEREDO, GENEVE. Dále jsem přidal podporu pro parsování protokolů 802.1q, 802.1ah, TRILL a MPLS. Exportér `flow_meter` obsahuje podporu pouze pro protokoly MPLS, PPTP, 802.1q, 802.1ah, PPPOE a ostatní jako ethernet, IP protokoly, ICMP, TCP a UDP.

4.4 Flow cache

Flow cache je část exportéru, která přijímá spojový seznam s vyparsovanými hlavičkami z parseru popsaného výše. Při návrhu bylo nutné vymyslet, do jaké míry popsat funkcionalitu flow cache pomocí jazyka P4. Nakonec není nutné popisovat celou funkcionalitu, ale stačí pouze tu klíčovou, týkající se zpracování vstupních hlaviček z parseru. To znamená popsat vyplnění a nalezení záznamu o flow a u nalezeného záznamu jej aktualizovat, typicky se jedná o aktualizaci čítačů paketů a bajtů. Této problematice je věnována celá tato sekce.

Implementace flow cache je víceméně stejná jako v exportéru `flow_meter`. Liší se ve zpracování vstupních zparsovaných hlaviček. V exportéru `flow_meter` je flow cache hash tabulka a skládá ze záznamů, jejichž počet je ve tvaru 2^n. Tabulka je členěná na řádky, kde velikost řádku je ve tvaru 2^m, a index řádku se počítá tak, že se políčka identifikující klíč toku, typicky IP adresy, porty a protokol transportní vrstvy, vloží do hash funkce a hash se použije pro vyhledání řádku.

Po nalezení řádku se řádek prochází a hledá se existující záznam se stejným klíčem kvůli aktualizaci. Pokud není nalezen, použije se volný záznam a pokud není žádný k dispozici, tak se exportuje a uvolní záznam pomocí algoritmu LRU (least recently used). Výchozí hodnoty velikosti flow cache je 2^{17} a pro řádek 2^4, tento tvar umožňuje zjistit řádek cache pomocí bitové operace AND, jinak by se musela použít pomalejší operace výpočtu zbytku po dělení. V exportéru z projektu NEMEA lze hash klíče vypočítat přímo, protože exportér podporuje omezený počet hlaviček a všechna parsovaná data se dala uložit do jedné struktury. Typicky se očekává přítomnost ethernetové, IP a TCP nebo UDP hlavičky a s možností tunelovaného provozu se zde nepočítá.

Nová verze exportéru generovaná backendem musí na možnosti existence více parsovaných hlaviček reagovat a to tím, že je nutné procházet spojový seznam. Složitost výpočtu klíče a aktualizace záznamu nalezeného toku se zvýší z $\mathcal{O}(1)$ na $\mathcal{O}(n)$. Podrobnější popis se lze dočíst v podkapitolách.

Dalším, nově vzniklým, problémem je existence tunelů v paketu. Nově je potřeba řešit existenci více stejných nebo podobných hlaviček vstupujících do flow cache. Tyto hlavičky se musí rozdělit a vložit do cache jako nový paket. V případě výskytu takových hlaviček je i vhodné řešit hierarchickou strukturu toků. Tu lze reprezentovat několika způsoby, které jsou popsané v následující podkapitole.

V P4 jsem přidal další speciální strukturu, pomocí které lze vkládat do záznamu o flow nové proměnné. Jedná se o strukturu `flowrec_s` a hraje klíčovou roli ve zbytku exportéru podobně jako seznam parsovaných hlaviček v `headers_s`. S touto strukturou lze pracovat zde ve flow cache, exportéru nebo v pluginech.

Backend do této struktury při generování přidává navíc políčka `first` a `last`, které reprezentují čas prvního a posledního paketu v toku. Důvodem je

nemožnost pracovat v P4 s časem, která by se dala řešit pomocí `extern`, ale je to zbytečná komplikace.

Parsovací pluginy mohou přidávat další data například z aplikačních protokolů v paketu. Z tohoto důvodu obsahuje navíc i ukazatel na spojový seznam s rozšířeními toku, který přidává rovněž backend při generování. Obsah této struktury pro nový ukázkový popis exportéru je zobrazena v následujícím P4 kódu.

```
struct flowrec_s {
    bit<64> bytes;
    bit<32> packets;
    bit<8> tcpflags;

    bit<8> ip_version;
    bit<8> tos;
    bit<8> ttl;

    bit<8> protocol;
    bit<16> src_port;
    bit<16> dst_port;
    ipaddr_u src_addr;
    ipaddr_u dst_addr;

    bit<48> src_hwaddr;
    bit<48> dst_hwaddr;
}
```

4.4.1 Hledání záznamu

K vytvoření nebo obecně nalezení záznamu je nutné projít celý spojový seznam dodaný z parseru. Z každé hlavičky využít určitá políčka k výpočtu hodnoty hash funkce a toto číslo pak použít pro nalezení řádku a poté nalezení existujícího záznamu nebo volného místa.

Problémem bylo najít způsob, jak tuto operaci popsat v P4, která bohužel nepodporuje `for` ani `while` cykly. Nakonec jsem našel způsob, jak toho docílit a to pomocí `control` bloku a vhodných operací reprezentovaných `extern` blokem.

Odstranění nutnosti použití smyčky v P4 je touto prací řešeno pomocí `control` bloku. A to tím způsobem, že se přeložený kontrolní blok v exportéru aplikuje na každou z hlaviček spojového seznamu. Tím vlastně došlo k přesunu nutnosti použití smyčky z P4 do C. Kontrolní blok umožňuje použít `if` kontrukt, případně `if-else` a pomocí těchto kontruktů lze od sebe jednoduše oddělit kód pro zpracování jednotlivých hlaviček. V bloku je navíc možné použít speciální funkce zavedené pomocí nového `extern` bloku pro práci s hlavičkami.

V přidaném `extern` bloku, který se jmenuje `flowcache`, je obsažena reprezentace několika málo operací pro práci s hlavičkami spojového seznamu a políčky hlaviček. První operací je `add_to_key` a způsobí, že políčko, které do této funkce vstupuje se použije pro výpočet klíče. V implementaci exportéru to znamená, že se obsah políčka jednoduše vloží na konec pole, které klíč reprezentuje, a toto pole se nakonec použije jako vstup do hash funkce.

Pro práci s hlavičkami spojového seznamu jsou nezbytné funkce pro zjištění, o jakou hlavičku se vlastně jedná. Pomocí funkce `is_present` lze zjistit, zda hlavička, která vstupuje jako parametr, je aktuálně zpracovávána. Další funkcí je `is_next`, která zjišťuje typ následující hlavičky. Obě tyto funkce vrací jako hodnotu `true` nebo `false`.

Poslední nezbytná funkce slouží pro zaregistrování konfliktních hlaviček. Například se může v paketu objevit IPv4 a IPv6 hlavička a správně by se měly vytvořit 2 toky. Rozdělení zajistí funkce `register_conflicting_headers`, ale pokud se ve spojovém seznamu objeví například 2 hlavičky stejného typu, je rozdělení provedeno automaticky bez nutnosti registrace. Popisovaný extern blok v P4 je v následujícím kódu.

```
extern flowcache {
    void add_to_key<T>(in T hdr);
    bool is_present<T>(in T hdr);
    bool is_next<T>(in T hdr);
    void register_conflicting_headers<T1, T2>(
                          in T1 hdr1, in T2 hdr2);
}
```

Co se týká zmíněného rozdělování flow, existuje několik přístupů v reprezentaci pořadí tunelovaného provozu ve flow cache. Například v článku [17] popisujeme reprezentaci pomocí vkládání klíče tunelovaných flow do jednoho záznamu, kde nevýhodou je dynamická délka záznamu. Dále popisujeme druhý přístup v reprezentaci pomocí stromové struktury. Výhodou je statická délka záznamu, navíc přibude pouze políčko reprezentující identifikátor rodičovského toku.

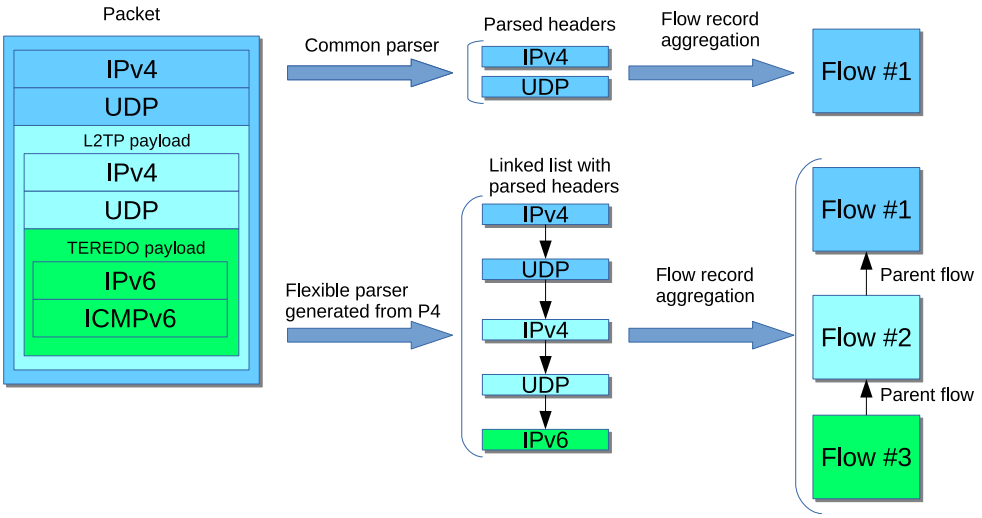
Popisovány jsou dva přístupy v reprezentaci identifikátorů. První je unikátní číslo, jako které se může zvolit i sekvence čísel. Je potřeba zařídit, aby se po restartování exportéru pokračovalo v sekvenci, jinak identifikátory v exportovaných záznamech mohou kolidovat. Druhým typem je reprezentace pomocí časových značek. Musí se ovšem zajistit, aby každý paket obsahoval unikátní časovou značku, zvýšením například přesnosti.

V novém exportéru jsou závislosti mezi toky reprezentovány pomocí identifikátoru toku používaného pro hledání záznamu. Jedná se o 64 bitový hash, kde je malá pravděpodobnost kolize. Do záznamu o toku jsou přidány 2 proměnné, identifkátor toku a identifikátor rodiče. Tyto dvě políčka jsou rovněž exportovány na kolektor toků, kde je snadné zjistit stromovou strukturu.

Problémem by mohlo být, že jeden stejný tok se může v čase několikrát vyskytnout a tím pádem má i stejný identifikátor. Rodiče tunelovaného toku

lze přesto jednoznačně dohledat pomocí času a identifikátoru. Tunelované toky mají totiž stejný čas vzniku jako jejich rodič.

Příklad tunelovaného provozu lze nejlépe zobrazit pomocí obrázku 4.3. Ukázkový paket obsahuje dokonce dva tunely, použité jsou protokoly L2TP a TEREDO. Obrázek je z našeho publikovaného článku z [17] a znázorňuje, jak si exportér poradí s tunelovaným provozem oproti běžnému exportéru.



Obrázek 4.3: Příklad tunelovaného paketu

Když už víme, jak zpracování hlaviček funguje, máme popsaný způsob, jak vytvořit klíč pro hledání ve flow cache. V případě, že záznam o toku nelze pomocí klíče nalézt, je nutné vytvořit nový tok. Bylo by zbytečné procházet spojový seznam znovu a vyplňovat do struktury, reprezentující tok, data z hlaviček.

Oba přístupy jsem měřil, obě operace odděleně nebo vytvoření klíče a vyplnění záznamu zároveň v jednom bloku kódu. Výsledkem bylo, že čas jednotlivých operací je přibližně stejný. Z tohoto důvodu jsem obě operace sloučil do jednoho kontrolního bloku. Problémem je, že bez klíče není možné zjistit, jaký záznam se má vyplnit. Proto je nutné nejprve vyplnit záznam do dočasné proměnné a poté jej umístit do flow cache. To je v exportéru řešeno pomocí výměny ukazatelů.

V novém exportéru obsahuje položka uložená v cache 2 proměnné. Jedna je identifikátor toku, tedy hash z pole bajtů obsahující položky toku umístěné za sebou, a ukazatel na záznam o toku, kde jsou uloženy položky toku jako IP adresy, porty, protokol atd. Cache navíc obsahuje pole s ukazateli na nevyplněné záznamy o toku, odkud se vezme jeden ukazatel na volný záznam, do kterého se vyplní políčka hlaviček. Pokud by tok v cache již existoval, vyplněný záznam se nechá být a nalezený záznam se aktualizuje. Pokud neexistuje, pouze se vymění ukazatele mezi vyplněným záznamem a neplatným

záznamem uloženým ve flow cache. V případě, že je celá flow cache úplně plná by nebylo možné vyplnit nový záznam, proto je nutné mít celkem $2^n + 1$ záznamů. To zaručí, že vždy bude alespoň jeden k dispozici, přestože by tok již v cache existoval.

V kontrolním bloku má programátor přístup k výstupní proměnné obsahující nevyplněné položky flow záznamu. Jeho úlohou je při procházení hlaviček jejich položky vyplnit do záznamu pomocí klasického operátoru přiřazení. Veškeré položky záznamu jsou před aplikováním kódu vynulované, takže se není nutné obávat, že by některé byly neinicializované. Na aktuálně zpracovanou hlavičku se lze odkázat jako při používání extern funkcí, tedy pokud aktuální hlavička je IPv4, pak se na ni odkáže pomocí `headers.ipv4`. Pokud by byla zvolena jiná hlavičku, bude se jednat o nedefinované chování a při čtení by obsahovala data jiné hlavičky.

Prototyp kontrolního bloku obsahuje 4 parametry. Prvním je vstupní parametr `headers` se seznamem hlaviček, které se mohou objevit ve spojovém seznamu. Stejný parametr je použit jako výstupní v prototypu parseru. Druhý parametr je `extern` blok reprezentující operace ve flow cache ke zpracování spojového seznamu. Třetí výstupní parametr je záznam, který se ukládá ve flow cache. Při vytvoření klíče se zároveň vyplní i tato struktura. Poslední výstupní parametr určuje, jestli se vytváření klíče zdařilo. Například si může programátor zvolit, že vytváření se bude považovat za úspěšné, pokud paket obsahuje alespoň IPv4 nebo IPv6 hlavičku.

```
control flow_create(in headers_s headers, flowcache c,
                    out flowrec_s flow, out bool success);
```

Praktické použití extern bloku je znázorněno v následujícím fragmentu kódu. Ten v exportéru řeší zpracování ethernetové a IPv4 hlavičky. U ethernetové hlavičky se pouze zkopírují adresy do flow záznamu. Pokud jde o druhou hlavičku, políčka se použijí pro vytvoření klíče a vyplnění záznamu. Pro indikaci úspěšného zpracování je vyplněn výstupní parametr `success`.

```
if (c.is_present(headers.eth)) {
   flow.src_hwaddr = headers.eth.src_addr;
   flow.dst_hwaddr = headers.eth.dst_addr;
} else if (c.is_present(headers.ipv4)) {
   success = true;
   flow.ip_version = 4;
   c.add_to_key(flow.ip_version);
   c.add_to_key(headers.ipv4.src_addr);
   c.add_to_key(headers.ipv4.dst_addr);
   c.add_to_key(headers.ipv4.protocol);

   flow.src_addr.v4.addr = headers.ipv4.src_addr;
   flow.dst_addr.v4.addr = headers.ipv4.dst_addr;
   flow.protocol = headers.ipv4.protocol;
   flow.tos = headers.ipv4.diffserv;
   flow.ttl = headers.ipv4.ttl;
```

```
}
```

Například funkce `is_present(headers.ipv4)` se z výše uvedeného kódu přeloží na kód `hdr->type == ipv4_h`, kde `hdr` je ukazatel na článek spojového seznamu a `ipv4_h` je hodnota výčtového typu `enum`. Druhá použitá funkce `add_to_key(headers.ipv4.src_addr)` se pro ukázku přeloží na kód `*(uint32_t *)(key + *key_len)` a do ní se pomocí operátoru = přiřadí `((struct ipv4_h *)hdr->data)->src_addr`. Políčko mělo velikost 4 B a proto se ještě aktualizuje celková délka klíče pomocí `*key_len += 4`.

4.4.2 Aktualizace záznamu

Aktualizaci záznamu lze také specifikovat pomocí kontrolního bloku. Princip fungování je stejný jako při vytváření klíče. Na každou hlavičku ve spojovém seznamu je aplikován kód bloku. I registrace konfliktních hlaviček zde má svůj smysl. Při existenci konfliktních hlaviček nebo hlaviček stejných typů je provádění přerušeno a daný tok neobsahuje hodnoty čítačů jiných toků.

Prototyp kontrolního bloku, narozdíl od předchozího, neobsahuje parametr `success`. P4 kód pro aktualizaci toku je v referenčním popisu exportéru kratší než pro vytváření klíče. Obsahuje zejména aktualizaci čítačů bajtů a paketů pro IPv4 a IPv6 toky, jelikož každý tok musí obsahovat minimálně IPv4 nebo IPv6 hlavičku.

```
control flow_update(in headers_s headers, flowcache c,
                    out flowrec_s flow)
{
   apply {
      c.register_conflicting_headers(
         headers.ipv4, headers.ipv6);

      if (c.is_present(headers.ipv4)) {
         flow.bytes = flow.bytes +
            (bit<64>) headers.ipv4.total_len;
         flow.packets = flow.packets + 1;
      } else if (c.is_present(headers.ipv6)) {
         flow.bytes = flow.bytes +
            (bit<64>) headers.ipv6.payload_len + 40;
         flow.packets = flow.packets + 1;
      } else if (c.is_present(headers.tcp)) {
         flow.tcpflags = flow.tcpflags | headers.tcp.flags;
      }
   }
}
```

Z výše uvedeného kódu se volání funkce `register_conflicting_headers` přeloží jako podmínka, která okamžitě ukončí zpracování seznamu. Při volání stejné funkce v `flow_create` se navíc uloží ukazatel na poslední přerušenou hlavičku, aby se mohlo ve zpracování tunelovaného toku později pokračovat.

4.5 Export toků

Export toků je realizován pomocí exportního protokolu IPFIX. V sekci je popsáno, jak se definují šablony exportéru a jak se vyplňuje IPFIX paket políčky.

Funkce `extern` bloku reprezentující IPFIX export lze rozdělit na tři skupiny. První skupina obsahuje funkce pro přidávání nových šablon a políček. Druhá obsahuje funkce pro nastavování šablon pro účely vyplnění políček. Třetí skupinou jsou funkce pro přidávání políček.

```
extern ipfix_exporter
{
    void register_template(bit<8> id);
    void add_template_field(bit<16> en, bit<16> id, int<16> len);

    void set_template(bit<8> id);
    void set_finish();

    void add_field<H>(in H field);
    void add_field_empty();

}
```

4.5.1 Definice šablony

Šablony exportéru lze definovat v `control` bloku jménem `exporter_init`. Ten jako jediný parametr obsahuje zmíněný `extern` blok. Tento blok se spustí při zapnutí exportéru.

Proces vkládání nových šablon se skládá z registrace šablony a registrace políček. Pro registraci šablony je nutné zvolit unikátní identifikátor, pomocí kterého se lze na danou šablonu ve zbytku P4 programu odkazovat. Registrace šablony je dosaženo zavoláním funkce `register_template` s identifikátorem jako parametrem. Tím je možné začít šablony plnit novými políčky pomocí funkce `add_template_field`. Registraci šablony lze provést jen jednou. Druhé volání funkce by skončilo s chybou.

Funkce pro registraci políček obsahuje tři parametry. Prvním parametrem je identifikační číslo společnosti (PEN), druhý je identifikátor políčka a třetí velikost políčka v počtu bajtů. Jako hodnoty parametrů lze volit pouze číselné konstanty.

Příklad je v následujícím kódu. Číslo 0 v prvním parametru je PEN pro orgranizaci IANA, ve kterém jsou předdefinovány často používaná políčka. Například identifikátory 8 a 12 jsou IPv4 adresy, IPv6 má 27 a 28 (zdrojová a cílová adresa).

```
#define IPFIX_TEMPLATE_IPV4 0
control exporter_init(ipfix_exporter e)
{
```

```
    apply {
        e.register_template(IPFIX_TEMPLATE_IPV4);

        e.add_template_field(0, 8, 4);    // src ipv4
        e.add_template_field(0, 12, 4);   // dst ipv4
        e.add_template_field(0, 60, 1);   // l3 proto
        //...

        e.register_template(IPFIX_TEMPLATE_IPV6);
        //...
    }
}
```

Co se týče překladu, ze specifikovaných políček se vezmou jen hodnoty parametrů. Ty se při překladu doplní do inicializační funkce v IPFIX implementaci.

4.5.2 Export záznamu

Naprogramovat exportování nerozšířeného záznamu o toku je možné v kontrolním `exporter_export` bloku. K tomu se využije `extern` funkcí `set_template`, `add_field` a `set_finish`. Tyto tři funkce slouží k vybrání šablony, přidání pole do IPFIX paketu a k signalizaci dokončení vyplnění šablony.

Šablonu lze zvolit tak, že se použije stejný identifikátor, který byl použit při jejím vytváření v bloku `exporter_init`. Pole k vyplnění se jednoduše vloží jako parametr funkce `add_field`. Tady je potřeba být opatrný, protože velikost vyplněných bajtů do šablony určuje typ, který jako parametr vstupuje. Pokud by existovalo políčko šablony s 64 bity a do parametru této funkce by vstoupila proměnná s bitovou délkou 32, pak se do výsledného paketu uloží jen 4 bajty (nutnost přetypovat na `bit<64>`). Pro správné vyplnění šablony je potřeba přesně dodržet pořadí jednotlivých políček a přesnou délku. Jakmile je vyplnění hotové, je nutné signalizovat konec plnění pomocí `set_finish` funkce.

```
control exporter_export(in flowrec_s flow, ipfix_exporter e)
{
    apply {
        e.set_template(IPFIX_TEMPLATE_IPV4);
        e.add_field(flow.src_addr.v4.addr);
        e.add_field(flow.dst_addr.v4.addr);
        e.add_field(flow.protocol);
        // ...
        e.set_finish();
    }
}
```

Funkce jsou do C překládány tím způsobem, že `set_template` vybere z pole pomocí ID tu správnou šablonu a zkontroluje se, jestli je v paketu dost

místa na všechna políčka. To se kontroluje tak, že v backendu jsou délky polí šablony sečteny a při použití `set_template` se doplní podmínka. Pokud existují v šabloně položky variabilní délky, je potřeba kontrolu provést v místě kopírování (na začátku není známá velikost). Celý kód vyplnění šablony je vložen do `while` cyklu a jakmile se v místě vyplnění, například řetězce, zjistí, že řetězec se do paketu nevejde, exportují se pakety na kolektor a pomocí `continue` se paket vyplní znovu. Druhá funkce `add_field` se přeloží, například pro 8 bitové pole, jako `(*(uint8_t *) buffer = (flow->ttl))`. Backend pozná, jestli se jedná o řetězec nebo bitovou proměnnou a zvolí příslušné délky. Poslední funkce `set_finish` provede to, že ve struktuře s daty šablony (ukazatel na buffer paketu a délka obsažených dat v něm) se aktualizuje délka.

4.6 Aplikační pluginy

Tato sekce je věnována problematice zpracování aplikačních protokolů. Chybějící podpora pro práci s řetězci je přidaná pomocí `extern` bloků. Inspiroval jsem se již existujícím `extern` blokem `packet_in` a přidal ekvivalentní funkce pro extrakci řetězců. Navíc jsem přidal funkce pro práci s řetězci jako konverze řetězce na číslo, porovnání a kopírování řetězce.

Plugin flow cache je v P4 reprezentován třemi bloky. První blok slouží jako post create funkce, druhá jako pre update funkce ve flow cache a obě jsou reprezentovány jako `parser` blok. Třetí blok je kontrolní a slouží pro vyplnění rozšířených záznamů o tocích do IPFIX paketu.

Obě funkce z flow cache mají za parametr vytvořený `extern` blok pro práci s payloadem paketu. Druhým, výstupním, parametrem je struktura, kam se uloží data zparsované pluginem. Pro každý plugin se jedná o jinou strukturu, kterou si definuje. Tato struktura je při úspěšném zparsování payloadu, který je signalizován přechodem do stavu `accept`, uložena k záznamu o toku.

Při exportu toku se zavolá vygenerovaný C kód z exportní funkce a tuto strukturu správně vyplní do IPFIX paketu. Šablony pluginů je třeba definovat ve speciálním `control` bloku, kde se definují všechna políčka jednotlivých šablon.

Blok pro vyplnění rozšířeného záznamu do IPFIX paketu obsahuje tři parametry. Vstupními dvěma parametry je struktura reprezentující tok a rozšířený tok. V pluginu je třeba vyplnit nejprve základní políčka toku a rozšířená políčka. Aby to bylo možné, je k dispozici jako třetí parametr `extern` blok reprezentující IPFIX komponentu.

Ze všech tří bloků reprezentující plugin je, pro lepší manipulaci, vytvořen balík. Balíky jednotlivých pluginů jsou seskupeny do dalšího balíku, reprezentující množinu pluginů exportéru. Ten je nakonec vložen jako parametr pro top level, `main`, balík.

```
// HTTP plugin
parser http_plugin_create_(payload p, out http_extension_s ext);
```

```
parser http_plugin_update_(payload p, out http_extension_s ext);
control http_plugin_export_(in flowrec_s flow,
   in http_extension_s ext, ipfix_exporter e);
package http_plugin(http_plugin_create_ create,
   http_plugin_update_ update, http_plugin_export_ export);

// plugins
package cache_plugins(
   http_plugin http,
   smtp_plugin smtp,
   https_plugin https,
   ntp_plugin ntp
);

cache_plugins(
   http_plugin(http_plugin_parser(), http_plugin_parser(),
      http_plugin_export()),
   smtp_plugin(smtp_plugin_parser(), smtp_plugin_parser(),
      smtp_plugin_export()),
   https_plugin(https_plugin_parser(), https_plugin_parser(),
      https_plugin_export()),
   ntp_plugin(ntp_plugin_parser(), ntp_plugin_parser(),
      ntp_plugin_export())
) plugins;
```

4.6.1 Extern blok

V této podsekci je ukázáno, jak vypadá `extern` blok pro práci s payloadem v pluginech. Funkce v bloku lze rozdělit na tři části. První část slouží k práci s řetězcem v paketu, druhá pro práci s vyextrahovanými řetězci v proměnných a třetí pro klasické operace známé z `packet_in` bloku ze standardní knihovny. Veškeré funkce jsou detailně popisovány v následujících podsekcích

```
extern payload
{
   bool extract_re<R, V>(in R regex, in V~vars);
   bool lookahead_re<R>(in R regex);

   bool match<R, V>(in R regex, in V~str);
   void strcpy<S1, S2>(in S1 dst, in S2 src);
   void to_number<N, S>(in S~str, out N number);
   void extract_string<T>(out T var, in bit<32> length);

   void extract<T>(out T var);
   T lookahead<T>();
   void advance(in bit<32> bytes);
   bit<32> length();
}
```

4.6.2 Extrahování řetězce

První přidanou funkcí používanou pluginy je `extract_re`. Ta je ekvivalentem funkce `extract` z `packet_in` bloku a slouží k extrakci řetězců. Obsahuje dva parametry, první slouží ke specifikaci regulárního výrazu a druhý ke specifikaci proměnných, do kterých se mají řetězce uložit. Celé to funguje tak, že ve vstupním regulárním výrazu vývojář označí místa, která se mají extrahovat pomocí POSIX capture groups (závorky). Poté musí vložit do druhého parametru tolik proměnných, kolik vytvořil skupin pomocí závorek.

Funkce vrací hodnotu true, pokud řetězec odpovídá popisu celého regulárního výrazu. Pokud se tak stane, kurzor bajtů paketu se posune o daný počet bajtů tam, kde porovnávání regulárním výrazem skončilo. Pokud ne, funkce vrací hodnotu false a nic se neextrahuje a ani se neposouvá kurzor. Funkce vrátí hodnotu false i tehdy, když počet zbývajících bajtů v payloadu je příliš malý.

Proměnná prvního parametru musí obsahovat anotaci `regex` s regulárním výrazem. Nezáleží na tom, jaký má proměnná typ, protože důležitý je pouze řetězec s regulárním výrazem v anotaci. Proto jako parametr může vstupovat libovolná proměnná.

Druhý parametr může obsahovat proměnnou a nebo list proměnných. Proměnné, které vloží do druhého parametru musí být řetězce. Ty se dají deklarovat pomocí anotace `string`. Pokud u proměnné libovolného typu se využije této anotace, jsou brány jako řetězec s uvedenou délkou v závorce. List se dá vytvořit pomocí závorek { a }. Proměnné se v něm oddělují pomocí čárky, například `{ key, delimiter, value }`. Pokud regulární výraz obsahuje jen jednu skupinu, může vývojář psát buď `key` nebo `{ key }`. Ukázka použití je v následujícím kódu. Lze si všimnout, že uvozovky jsou v regulárním výrazu escapovány, jinak by je nešlo specifikovat (v backendu jsou nakonec escape znaky odstraněny).

```
@regex("([^:]*)\": \"(.*)\"\r\n\"")
bit<1> http_keyval;

@string("512") bit<1> key;
@string("512") bit<1> val;

state parse_fields_request {
   // Parse HTTP hdr fields

   transition select(p.extract_re(http_keyval, {key, val})) {
      true: check_host;
      false: accept;
   }
}
```

Jedním z problémů využívání záchytových skupin pomocí závorek (a) je v tom, že v závorkách se může objevit více regulárních výrazů. Například se

může jednat o použití výrazu `( |\r\n)`, kde se očekává mezera nebo konec řádku v aplikačním protokolu. Vývojář může tuto konstrukci chtít použít, aniž by měl v plánu onen řetězec extrahovat. Vyřešit to lze jednoduchým způsobem a to extrahování do proměnné s anotací `@string("1")`. Tam se poté, díky nedostatku místa, uloží pouze ASCII ukončovací znak a zbytečně se nekopírují data z payloadu paketu.

Další problém souvisí s použitím listu ke specifikaci více proměnných. V `extern` funkci je parametr uveden jako vstupní, `in V vars`. Vzhledem k tomu, že se extrahuje do proměnných, by měl být parametr výstupní, `out V vars`. Pokud bych použil parametr správně jako výstupní, pak by nešel zadat list s proměnnými, protože ten je pouze pro čtení a kompilátor by ho ve dvojici s `out` parametrem odmítl.

Problém lze vyřešit tak, že by se nevyužíval list, ale muselo by se pro každé použití funkce `extract_re` definovat v `extern` bloku novou funkci s více parametry. Například pro extrahování do tří parametrů by musela existovat funkce s `out V var1, out V var2, out V var3`. To by bylo dost nepohodlné, proto pro zatím nechávám použití listu i přes drobné porušení konvencí.

Konečně se dostáváme k samotnému překladu funkce do C. Pro každé použití funkce s unikátním regulárním výrazem se do nového exportéru vygeneruje nová funkce. Regulární výraz se vloží do hash funkce a vzniklé číslo je použito jako identifikátor této funkce. Regulární výraz je použit pouze pro účely vytvoření automatu a ve vygenerovaných funkcí v C nakonec v parametrech nevystupuje.

Každá vygenerovaná C funkce obsahuje parametry `const uint8_t *payload` a `const uint8_t *payload_end`. Ty slouží pro účely zpracování konečného automatu. Dalším parametrem je `const uint8_t **payload_cursor`, pomocí kterého se po úspěšném zpracování upraví ukazatel na payload. Další parametry jsou proměnné sloužící pro ukládání vyparsovaných řetězců.

Pro každý parametr v listu proměnných ve funkci `extract_re` se k výše uvedeným parametrům funkce C doplní ukazatel na řetězec a velikost místa. Těmito parametry jsou `uint8_t *arg0, size_t arg0_len` (pro 1 proměnnou). Velikost bufferu slouží k případnému oseknutí extrahovaného řetězce, aby nedošlo k přetečení. Velikost daného řetězce pochází z anotace proměnné, například `@string("42") bit<1> key` se do C přeloží jako `uint8_t key[42]`, a při volání funkce se jako parametry použijí `key, sizeof(key)`.

Dalším krokem je vygenerování těla funkce pro pozdější generování konečného automatu. To je uvedeno v příkladu v následujícím kódu pro výraz `("GET"|"POST")[ ]([^ ]*)[ ]"HTTP"[/][0-9][.][0-9]"\r\n"`, extrahující typ požadavku a URL v HTTP hlavičce. V těle funkce jsou definovány lokální proměnné, se kterými bude konečný automat pracovat. Mezi ně patří například proměnné `yynmatch` a pole `yypmatch`. Proměnná `yynmatch` slouží pro specifikaci počtu skupin v regulárním výrazu. V příkladu jsou sice skupiny dvě, ale celý výraz je brán jako jedna samostatná skupina, takže ve výsledku jsou tři. Proměnná `yynmatch` obsahuje ukazatele na počáteční a koncový znak

pro každou skupinu, pomocí ní se dají později extrahovat požadované řetězce. Dalším krokem je vygenerování maker pro konečný automat, kde se definuje, kde je například začátek, konec a nebo jak se dá vzít ze vstupní posloupnosti další znak.

Poslední krok je vložení regulárního výrazu do komentáře, kde na začátku existuje speciální řetězec `!re2c` sloužící pro generátor automatu re2c. V komentáři se objeví regulární výraz a za ním v závorce je kód, který se vykoná, pokud vstupní posloupnost bajtů splňuje popis výrazem. V závorce se použijí proměnné yypmatch pro vykopírování řetězců do výstupních proměnných. Nakonec se upraví kurzor paketu a vrací se hodnota 1 jako úspěch.

```
int regex_http_292902314824198396(
        const uint8_t *payload,
        const uint8_t *payload_end,
        const uint8_t **payload_cursor,
        uint8_t *arg0, size_t arg0_len,
        uint8_t *arg1, size_t arg1_len)
{
    const uint8_t *backup, *marker;
    int yynmatch = 3;
    const uint8_t *yypmatch[6], *yyt1, *yyt2, *yyt3;
    #  define YYCTYPE      uint8_t
    #  define YYPEEK()     (payload < payload_end ? *payload : 0)
    #  define YYSKIP()     ++payload
    #  define YYFILL(n)    return 0;
    #  define YYCURSOR     payload
    #  define YYLIMIT      payload_end
    #  define YYMARKER     marker
    #  define YYBACKUP()   backup = payload
    #  define YYRESTORE()  payload = backup
    /*!re2c
       * { return 0; }
       ("GET"|"POST")[ ]([^ ]*)[ ]"HTTP"[/][0-9][.][0-9]"\r\n" {
          size_t len = yypmatch[3] - yypmatch[2];
          if (len >= arg0_len) {
             len = arg0_len - 1;
          }
          memcpy(arg0, yypmatch[2], len);
          arg0[len] = 0;
          len = yypmatch[5] - yypmatch[4];
          if (len >= arg1_len) {
             len = arg1_len - 1;
          }
          memcpy(arg1, yypmatch[4], len);
```

```
            arg1[len] = 0;
            *payload_cursor = payload;
            return 1;
        }
    */
    return 0;
}
```

Výraz `* { return 0; }` v komentáři slouží pro signalizaci neúspěchu, pokud se matching nepovede.

Po vygenerování takovéto funkce je nakonec potřeba zavolat na soubor se zdrojovým kódem program re2c. Ten přeloží uvedený komentář na kód konečného automatu, který zde nebude rozepisován.

4.6.3 Porovnání bez posunu

Funkce `extract_re` v předchozí podsekci sloužila jak pro zjištění, jestli daný řetězec v paketu splňuje daný regulární výraz. Když ano, kurzor paketu se posunul a řetězec se extrahoval. Pokud by bylo v programu potřeba zjistit, jestli se daný řetězec vyskytuje v paketu aniž by se posunul kurzor, nebylo by to možné. Z tohoto důvodu je touto prací přidána do P4 programu funkce `lookahead_re`.

Ta má pouze jeden jediný parametr a to vstup pro regulární výraz. Návratová hodnota je stejná jako v předchozím případě. Vrací true, pokud se matching povedl.

Překlad funkce je jednodušší než v případě funkce `extract_re`. Liší se v tom, že není potřeba přidávat parametry pro extrakci řetězců. Tím pádem ani nepotřebuje logiku pro kopírování řetězců do výstupních proměnných. Vygenerovaný automat je totožný jako v minulé funkci.

Následující kód je příkladem použití v SMTP pluginu. V P4 programu je potřeba zjistit, jestli se v payloadu paketu někde nachází slovo SPAM.

```
state check_spam {
    @regex(".*'SPAM'") bit<1> re_spam;
    transition select(p.lookahead_re(re_spam)) {
        true: process_spam;
        default: accept;
    }
}
```

Další variantou funkce je extrakce do proměnné bez posunu kurzoru. Tu jsem během psaní parsovacích pluginů nepotřeboval a není v této práci přidána.

4.6.4 Porovnávání proměnné

Další nepostradatelnou funkcí je funkce pro porovnávání obsahu proměnné. Funkce se v P4 `extern` bloku nazývá `match` a obsahuje dva parametry. První je klasický parametr specifikující regulární výraz a druhý parametr je proměnná, která se bude porovnávat. Funkce vrací hodnotu true nebo false, pokud se matching povedl.

Překlad funkce do C je jednoduchý, využíváno je stejných principů jako v předchozím případě. Do funkce se vygeneruje konečný automat a logika pro extrahování řetězců se backendem nevyplní. Parametry funkce zůstávají stejné jako v případě `lookahead_re`, jen se při volání místo ukazatele na payload uvede ukazatel na proměnnou a na její konec. Tím se pracuje s daty v proměnné, u které je potřeba provést matching.

Následující příklad ilustruje použití této funkce v HTTP parsovacím pluginu. Plugin extrahuje klíč a hodnotu z HTTP hlavičky a v dalším stavech jsou porovnávany, jestli se mezi nimi nenachází hlavička *Host*. Porovnávání zahrnuje i terminální řetězec v proměnné, aby nedošlo k nesprávnému vyhodnocení (*Host* by byl preffixem jiného řetězce).

```
state parse_fields_request {
   // Parse HTTP hdr fields

   transition select(p.extract_re(http_keyval, {key, val})) {
      true: check_host;
      false: accept;
   }
}
state check_host {
   @regex("\"Host\x00\"")
   bit<1> host_str;

   transition select(p.match(host_str, key)) {
      true: parse_host;
      default: check_agent;
   }
}
```

Toto porovnávání pomocí funkce `match` a přechodů do dalších stavů nahrazuje `if-else` podmínku. Podmínka není v `parser` bloku podporovaná a protože je potřeba porovnat proměnnou `key` na shodu s *Host*, *User-Agent* a *Referer*, je to v P4 implementaci exportéru řešeno právě více stavy.

4.6.5 Konverze řetězce na čísla

U některých protokolů je třeba překládat řetězce na čísla. Příkladem jsou návratové kódy serveru v SMTP nebo HTTP komunikaci. Tyto kódy jsou konvertovány pomocí funkce `to_number`.

Funkce obsahuje dva parametry. První parametr slouží pro vstup řetězce, který se má konvertovat. Druhý parametr je výstupní a uloží se do něj konvertované číslo. V backendu se obě funkce kontrolují, jestli mají správné typy parametrů.

Příklad použití je zobrazen na konvertování návratového kódu v HTTP. Ten je poté uložen do struktury obsahující rozšiřující informace HTTP.

```
state parse_header_response_ {
   p.to_number(resp_code, ext.data.resp.code);

   ext.type = HTTP_RESPONSE;
   transition parse_fields_response;
}
```

Funkce je překládána jako `strtoll` a `strtoull` pro konverzi řetězce do `long long int` nebo `unsigned long long int` typu. Backend dokáže poznat v jakých případech použít kterou funkci. Pro proměnné typu `int` se použije `strtoll` a pro `bit strtoull`. V ostatních případech kompilátor indikuje chybu. Podporovány jsou zápisy čísel v desítkové, oktalové a šestnáctkové soustavě.

4.6.6 Extrakce řetězce zadaného délkou

Poslední přidanou potřebnou funkcí je `extract_string`. Ta umožňuje extrahovat bajty z paketu a tyto bajty uložit jako řetězec. Používá se v případě, kdy nelze použít regulární výraz. Například je před řetězcem v paketu uložená jeho délka a řetězec není ukončený žádným ukončovacím znakem a za ním následují bajty jiného políčka.

Do funkce se vkládá parametr s proměnnou, kam se má uložit řetězec, a délkou pro extrakci. Jediné využití je zatím v pluginu pro parsování HTTPS.

```
state parse_sni {
   p.extract_string(ext.sni, (bit<32>) tls_sni.length);
   transition accept;
}
```

Funkce se v backendu překládá do C jako `if` podmínka, kontrolující velikost zbývajícího payloadu a `for` cyklus. Ten kopíruje bajty z payloadu a zároveň kontroluje, zda cílový buffer nepřetekl. Pokud je buffer příliš malý, řetězec se zkrátí.

4.6.7 Post create

Post create funkce se ve flow cache volá pro každý plugin po vytvoření toku. V této funkci se naalokuje paměť pro strukturu, která obsahuje rozšířený záznam pro daný plugin. Struktura se jednou vynuluje a spolu s ukazatelem na bajty payloadu a záznamu o toku se vloží jako parametr přeložené C funkci s

parserem pluginu. Ten pak vrátí návratovou hodnotu signalizaci úspěchu při parsování.

Jedním z navíc vyřešených problémů touto prácí je ten, že v exportéru `flow_meter` některé pluginy filtrovaly provoz dle portu, například HTTP plugin parsoval pouze pakety s portem 80. To mělo za následek, že ne veškerý HTTP provoz se zachytil a zpracoval, například HTTP provoz na portu 8080 a ostatních zůstal bez povšimnutí.

V přepsaných pluginech v novém exportéru se předchozí problém řeší tak, že se parsuje každý paket a dokud není jasné, že se opravdu jedná o daný protokol, tak se do rozšířeného záznamu nic neukládá. Předchozí způsob je zavedený proto, aby se při každém volání parseru nemusela struktura inicializovat, přepsáním samými nulami. Tím, že je zachována integrita mezi jednotlivými voláními se tímto způsobem ušetří čas ztrávený inicializací. Kdyby se nějaká data uložila a parsování se nakonec nepovedlo, mohla by se ve výsledku exportovat špatná data.

Dokud si tedy plugin není jistý, že paket obsahuje opravdu protokol, který parsuje, neukládá žádná data do struktury. Data si ukládá do lokálních proměnných, například v HTTP pluginu se takto parsuje hlavička, a pak se kopírují do rozšířeného záznamu. Když je jasné, že se o daný protokol jedná, tak to parser signalizuje přechodem do stavu `accept`. Na základě toho se pak rozšířený záznam uloží k záznamu o toku.

4.6.8 Pre update

Při volání pre update funkce může dojít k dvěma situacím. První je ta, že záznam o toku obsahuje rozšíření aplikačním pluginem, který byl přidán ve funkci post create. Druhá situace je opačná, kdy záznam žádné rozšíření neobsahuje, například se post create funkce zavolala na TCP SYN paket s nulovou délkou payloadu.

Tyto situce jsou řešeny tak, že pokud existující záznam není nalezen, je vytvořen nový jako ve funkci post create a inicializován nulami. Záznam se vloží do funkce a pokud se povede parsování, tak se uloží k záznamu o toku. Pokud záznam již existuje vloží se do parsovací funkce a parserem se provede aktualizace. I zde má smysl, aby parser neukládal do záznamu data bez toho, aby si byl jistý protokolem v paketu.

4.6.9 Podporované pluginy

Do P4 jsem přepsal většinu pluginů napsaných v C++ z původního exportéru. Podporované pluginy jsou v tabulce 4.3. Jediný plugin, který není podporován je DNS plugin z důvodů uvedených v analýze. NTP plugin má částečnou podporu tím, že se exportuje celá hlavička. V původní implementaci se prováděly operace pro konverzi časových značek ze speciálního formátu a IP adresy na

řetězec. Zde se jen políčka exportují tak, jak jsou v paketu a převod je možný na jiném zařízení.

Tabulka 4.3: Přepsané pluginy z C++ do P4 a jejich podpora v novém exportéru

plugin	popis
HTTP	plná podpora
HTTPS	plná podpora
SMTP	plná podpora
SIP	plná podpora
NTP	částečná podpora
DNS	nepodporovaný

4.6.10 Flush

V HTTP, SIP, HTTPS a NTP pluginu je potřeba zajistit možnost předčasně exportovat tok. Signál, že se má tok exportovat, lze signalizovat v konečném automatu parseru pomocí vstupu do speciálního stavu se jménem flush. Poté se v cache exportuje aktuálně zpracovávaný tok a paket, který se vkládal se opětovně vloží do cache. Opětovné vkládání do cache má smysl pouze v souvislosti s funkcí pre update, ve funkci post create se tok pouze exportuje (zabraní se zacyklení). Nakonec se parser payloadu zavolá znovu po vytvoření nového toku a zpracuje tento nezparsovaný payload. Mechanismus rozdělování toku umožní, že se například exportují všechny HTTP požadavky a odpovědi bez toho, aby tyto informace byly ztraceny.

4.6.11 Export v pluginech

Pluginy mají vlastní kontrolní blok pro specifikaci vyplnění IPFIX paketu dle šablony. Způsob vyplnění je naprosto stejný jako když se vyplní nerozšířený záznam o toku. Do bloku vstupují celkem tři parametry, záznam o toku, rozšiřující struktura s daty pluginu a `ipfix_exporter extern` blok.

4.7 Shrnutí

Celá kapitola se věnovala implementaci nového exportéru popsaného v jazyce P4. Popsány byly klíčové 4 komponenty exportéru jako je parser paketů, flow cache, export záznamu o toku a parsovací pluginy aplikačních protokolů. Díky tomuto popisu v P4 je možné jednoduchým způsobem modifikovat hlavní části exportéru.

Pro generování kódu byl touto prací vytvořen backend pro kompilátor $P4_{16}$. Ten překládá vstupní P4 program na C kód a následně jej vkládá do souborů se šablonami kódu exportéru. Tím je možné jednoduše upravovat C

zdrojový kód exportéru a přizpůsobit si jej podle vlastních požadavků. Díky použití šablon je navíc možné upravovat, jak bude přeložený P4 kód ve výsledném exportéru vypadat.

V parseru paketů exportéru je možné jednoduše přidávat podporu pro parsování nových protokolů. Nejprve je potřeba definovat hlavičky protokolů pomocí `header` a poté přidat podporu pro jejich zpracování do konečného automatu. Toho se docílí tak, že do parseru přidají nové stavy a definují se přechody mezi stavy.

Nově přidané protokoly je možné vložit na výstup parseru a mít možnost je zpracovat ve flow cache a vytvořit tok. Toho je možné dosáhnout tím způsobem, že se tyto hlavičky protokolů vloží jako členská proměnná do struktury `header_s`. Dalším krokem je přidání podpory pro zpracování těchto hlaviček, které jsou ve spojovém seznamu. To se provede tak, že se přidá zpracování hlavičky do bloků `flow_create` a `flow_update`. Pro zjištění, o jakou hlavičku se ve spojovím seznamu jedná lze využít funkcí `cache extern` bloku jako `is_present`. Poté je možné vytvořit klíč toku pomocí funkce `add_to_key` a také vkládat políčka z hlavičky do záznamu o toku a upravovat čítače.

Tok ve flow cache lze exportovat napsáním kódu pro vyplnění IPFIX paketu v `control` bloku `exporter_export`. Před tím je ještě nutné definovat šablony v bloku `exporter_init`. Pro oba bloky je nezbytné využití funkcí z `ipfix_exporter extern` bloku. V bloku pro vytváření šablon je možné vytvořit šablonu pomocí `register_template` a definovat její políčka pomocí `add_template_field`. Šablona pro vyplnění se zvolí pomocí `set_template` a políčka se vloží funkcí `add_field`. Vyplnění paketu je nutné ukončit funkcí `set_finish`.

Vytvořit parsovací plugin je možné pomocí dvou `parser` bloků a jednoho kontrolního bloku. Parsovací bloky slouží jako post create a pre update funkce a kontrolní slouží k vyplnění šablony novými políčky, které plugin zparsoval. Psaní parseru je možné při využití funkcí z `payload extern` bloku. Zde je možné využít ekvivalentní funkce `packet_in` bloku pro extrahování jako je `extract_re`, `lookahead_re` a ostatních pro práci s řetězci jako `extract_string`, `match`, `to_number` a `strcpy`. V `payload extern` bloku je rovněž k dispozici sada původních funkcí z `packet_in` bloku.

Kapitola 5

Měření

V kapitole je věnována pozornost srovnání mezi exportérem `flow_meter` a exportérem popsaným v P4. Měření bylo prováděno na stroji s procesorem *Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz*, *8 GB RAM* a diskem *Samsung SSD 850 EVO 250 GB*. Použitý kompilátor je *GCC 8.3.1* a aplikace byly zkompilovány s *-O3* parametrem.

5.1 Parser

Metodika měření rychlosti parserů spočívala v tom, že se načetl 1 paket z PCAP souboru a ten se opakovaně vkládal do parseru paketů. Počet opakování je celkem 100 000 000 a zparsované pakety nebyly vkládány do cache. Tím byla měřena pouze výkonnost parseru paketů.

V tabulce 5.1 jsou uvedeny výsledky měření pro parser. V tabulce jsou popsány hlavičky paketu tak, jak jdou za sebou a velikost paketu v bajtech. V závorce je uveden čas bez kopírování payloadu. Parser v exportéru `flow_meter` funguje tak, že se vyparsují políčka hlaviček a payload se zkopíruje do bufferu (z důvodu volání callback funkce v knihovně PCAP je nutné data kopírovat).

Tabulka 5.1: Výsledky měření parseru. Časy jsou uvedeny v sekundách

paket	`flow_meter` [s]	P4 exportér [s]
ETH, IPv4, TCP, 1111 B	4 (1)	3,04
ETH, IPv4, UDP, 90 B	2,1 (0,97)	2,92
ETH, IPv6, TCP, 94 B	2,15 (1)	3,99
ETH, IPv6, UDP 166 B	2,5 (0,97)	3,9
ETH, VLAN, 2× MPLS, IPv4, TCP, 1526 B	4,5 (1,07)	3,35
ETH, VLAN, 2× MPLS, IPv4, UDP, 107 B	2,6 (1)	3,2

Parser paketů je v exportéru `flow_meter` vysoce optimalizovaný, proto bez kopírování paměti dosahuje takové výkonnosti. Celková výkonnost záleží na

tom, jak je paket velký. Pokud payload není potřeba kopírovat (nejsou zapnuté pluginy), pak se dá počet kopírovaných paketů snížit a zvýšit tak výkonnost. Nový exportér vygenerovaný z P4 je ve většině případů naopak pomalejší.

5.2 Flow cache

Měření flow cache probíhalo tak, že jsem ve smyčce vkládal 100 000 000 paketů do cache. V celé cache jsem zakomentoval všechny funkce pro exportování, aby se ve výsledném čase nezahrnoval čas spojený s odesíláním toků po síti. Před voláním funkce pro vkládání jsem vytvořil paket a ten vložil jako parametr této funkci.

Testování flow cache bylo měřeno na dvou scénářích. V prvním se vkládal jeden a ten samý paket, aby se simuloval dlouhý síťový tok. V druhém scénáři se vkládaly unikátní pakety a tím vznikly unikátní jednopaketové toky. Čísla pro inicializaci políček jako jsou IP adresy, porty a protokol jsem generoval pseudonáhodným generátorem čísel a ta samá posloupnost čísel se generovala u obou exportérů.

Časy jednotlivých scénářu jsou uvedeny v tabulce 5.2. Je patrné, že pro jeden dlouhý tok je lepší původní exportér. Nový exportér si lépe vede ve zpracování více různých toků

Tabulka 5.2: Výsledky měření flow cache. Časy jsou uvedeny v sekundách

scénář	`flow_meter` [s]	P4 exportér [s]
jeden dlouhý tok	3,6	6,5
jednopaketové toky	34,2	23,7

5.3 Export

V této sekci je měřena rychlost vyplňování toků do IPFIX paketu. Vytvořil jsem si na začátku strukturu s tokem, kterou jsem 100 000 000 krát vložil jako parametr funkce pro export záznamu o toku. Tato funkce vyplní políčka záznamu do IPFIX paketu. Záměrně jsem z této funkce odstranil odesílání bufferu s vyplněným paketem, aby operace odesílání nebyl zahrnuta do výsledného času.

V tabulce 5.3 jsou uvedeny výsledné časy měření. Měřeno bylo vkládání IPv4 toku a IPv6 toku. V tabulce je vidět, že časy se od sebe moc neliší. Jediná zvláštnost je, že vyplnění IPv6 toku je rychlejší než IPv4 toku přestože jsou adresy delší.

Tabulka 5.3: Výsledky měření vyplnění IPFIX paketu. Časy jsou uvedeny v sekundách

scénář	`flow_meter` [s]	P4 exportér [s]
IPv4 tok	1,53	1,63
IPv6 tok	1,58	1,61

5.4 Pluginy

Měření pluginů probíhalo podobným způsobem jako měření parseru. Načetl se jeden paket s aplikačním protokolem ze souboru a ten byl vkládán 1 000 000 krát do cache. V exportérech jsem zakomentoval funkce pro vyplnění záznamu o toku do IPFIX paketu. Tato metodika má za cíl testovat, jak rychle pluginy dokáží parsovat a jak rychle dokáže cache pracovat s pluginy (požadavek na okamžité exportování toku a práce s pamětí pluginů).

Výsledky jsou zahrnuty v tabulce 5.4. Jediný velký rozdíl v době běhu je u NTP pluginu. To je dáno tím, že NTP plugin v exportéru `flow_meter` provádí konverzi časových značek na řetězce. V novém exportéru se časová značka uložená v paketu nekonvertuje a rovnou se exportuje.

Tabulka 5.4: Výsledky měření pluginů. Časy jsou uvedeny v sekundách

plugin	`flow_meter` [s]	P4 exportér [s]
HTTP	0,84	1,31
HTTPS	0,14	0,14
SIP	0,62	0,88
SMTP	0,16	0,89
NTP	9,32	0,16

5.5 Celkový výkon

Celkový výkon exportérů jsem měřil na PCAP souborech. Obě aplikace byly spouštěny s parametry *-u -x localhost:4739 -r cesta-k-pcapu* a doba běhu byla měřena pomocí příkazu *time*. Metodika testování je několikrát spustit čtení souboru (aby se soubor načetl do cache) a poté několikrát spustit měření (cca 10 krát) a výsledné hodnoty zprůměrovat. Použité PCAP soubory jsou uvedeny v tabulce 5.5.

První PCAP obsahuje běžný provoz na síti. To znamená hlavičky jako ethernet, IPv4, TCP a UDP. Velikosti paketů jsou v PCAP souboru maximálně 200 bajtů.

PCAP číslo 2 je zajímavý v tom, že 88 % paketů neobsahuje tunelovaný provoz, kdežto 12 % paketů obsahuje jeden tunel a 0.006 % obsahuje dva tunely (bráno jako tunelované toky). Průměrná velikost paketu je 791 B. Všechny

Tabulka 5.5: PCAP soubory pro měření. V závorce je uveden celkový počet toků (i tunelovaných)

PCAP	pakety	toky	velikost [MB]	záchyt
1	5 521 395	74 552	920	10 h
2	1 577 658	64 668 (74 528)	1 214	6 s
3	1 891 105	85 322	1 191	26 h

pakety se skládají z ethernet, VLAN, 2× MPLS a IPv4 hlaviček jdoucí v tomto pořadí a pak následuje TCP nebo UDP hlavička. Za UDP hlavičkou v některých případech následuje tunelovaný provoz začínající L2TP tunelovacím protokolem. Poté ve většině případů následuje IPv4 a TCP/UDP.

V extrémních případech nový exportér vytvoří z jednoho paketu tři toky v druhém PCAP souboru. To se děje když paket obsahuje ethernet, VLAN, 2× MPLS, IPv4, UDP, L2TP, PPP, IPv4, UDP, TEREDO, IPv6 a ICMPv6 hlavičky, jdoucí přesně v tomto pořadí zasebou.

Třetí PCAP soubor obsahuje běžný provoz na síti jako první PCAP. Průměrná velikost paketu je 691 B.

Výsledky měření jsou uvedeny v tabulce 5.6. Měření bylo prováděno ve dvou verzích. První verze zahrnovala běh aplikace s kontrolou neaktivních timeoutů. Ty se v obou exportérech kontrolují tak, že každých 5 sekund se prochází celá cache a provádí se kontroly timeoutů. V živém záchytu se kontroly provádí každých 5 sekund ale při čtení souboru kontrola záleží na rozestupu mezi pakety a proto je čtení pomalé. Pokud jsou v PCAP souboru rozdíly v časových značkách mezi všemi pakety větší než 5 sekund, pak se kontrola provádí po každém čtení paketu. Toto chování je v obou exportérech proto, aby se při živém záchytu a čtení souboru exportovaly toky stejným způsobem. Druhá verze tuto kontrolu neaktivních timeoutů neobsahovala a časy jsou uvedeny v závorkách.

Tabulka 5.6: Výsledky měření celkové výkonnosti. V závorce je uveden čas běhu bez kontroly neaktivních timeoutů

PCAP	`flow_meter` [s]	P4 exportér [s]
1	8,5 (0,8)	2,1 (1,15)
2	0,7 (0,7)	0,8 (0,8)
3	17 (0,6)	2,5 (0,66)

V tabulce jsou vidět rozdíly mezi časy obou verzí měření u PCAP souboru 1 a 3. To je dáno tím, že provoz byl v souboru byl 10 a 26 hodinový, takže často docházelo ke kontrole timeoutů. V nové verzi exportéru nejsou tyto časy tak výrazné a too je způsobeno lepší implementací cache v C. Naopak PCAP číslo 2 obsahoval provoz za krátký časový okamžik a časy běhu jsou stejné.

V kapitole o měření rychlosti parseru bylo napsáno, že `flow_meter` při par-

sování navíc kopíruje payload provozu. Díky tomu je `flow_meter` pomalejší, ale zase má optimalizovanější parser hlaviček paketů. Nový exportér naopak payload nekopíruje a má lepší podporu pro hlavičky paketů. Oba exportéry dosahovaly při měření flow cache různých rychlostí pro vkládání paketů do cache a každý je lepší v něčem jiném. Když se výhody a nevýhody obou implementací zahrnou do celkového výkonu, nejsou časy běhů až tak velké, alespoň pokud jde o druhou verzi měření.

Závěr

Práce se věnuje využití jazyka P4 pro generování bezpečnostních síťových aplikací. Využití je demonstrováno na popisu nové verze exportéru síťových toků `flow_meter`. Architektura tohoto exportéru je v práci analyzována a jsou určeny klíčové komponenty pro popis pomocí jazyka P4.

První kapitola práce se věnuje jazyku P4 a již existujícím aplikacím v tomto jazyce. Popsána je architektura exportéru `flow_meter`, struktura aplikačních protokolů a IPFIX protokolu.

V další kapitole je analýza využívána k návrhu popisu exportéru. Je zde popsáno, jakých P4 konstruktů se využívá. Pro parser paketů se jedná o `parser` blok, flow cache a IPFIX export využívá `control` bloků. K popisu pluginu jsou použity dva `parser` a jeden `control` blok.

Předposlední kapitola se věnuje implementaci backendu. Generování kódu exportéru je dosaženo pomocí kompilátoru $P4_{16}$. Popsáno je, jak se kód P4 překládá do C. Například `parser` blok je překládán jako množina bloků kódu, návěstích a `goto` příkazů. V kapitole jsou rovněž popsány `extern` bloky, pomocí kterých je dosaženo funkcionality, která se v P4 nedá popsat. V práci se povedlo přepsat existující pluginy HTTP, HTTPS, SMTP, SIP, NTP z C++ do jazyka P4 se zachováním stejné funkcionality (až na NTP, kde se pouze nekonvertují některá pole).

V poslední kapitole je obsaženo porovnání původního a vygenerovaného exportéru. Oba exportéry jsem měřil a nevýhodou oproti původnímu exportéru je v tom, že jednotlivé komponenty nového exportéru jsou pomalejší. Ukázalo se, že parser paketů je až $4\times$ pomalejší než ručně optimalizovaný parser, flow cache je pomalejší $1,8\times$ na dlouhých tocích a rychlejší $1,4\times$ na jednopaketových tocích, vyplnění IPFIX paketu bylo $1,05\times$ pomalejší. Naopak měření celkového výkonu na 3 PCAP souborech ukázalo, že nový exportér byl ve dvou případech $4\times$ a $17\times$ rychlejší a v druhém případě byl téměř stejně rychlý. Pokud byly vypnuty kontroly neaktivních timeoutů byl v nejhoším případě pomalejší $1,4\times$ na souboru s velkým počtem paketů a u ostatních dvou byl o $1,14\times$ pomalejší.

Další nevýhodou je, že ne všechny aplikační pluginy původního exportéru lze jednoduše popsat v P4, například DNS. To ale nevadí, protože je stále možné tyto pluginy popsat pomocí jazyka C. Pluginy, které se nedají popsat pomocí P4, lze přidat do šablon se zdrojovými kódy ze kterých backend kompilátoru generuje výsledný kód exportéru.

Přínos práce je v jednodušším popisu exportéru toků. Tím, že jsem se v práci zaměřil pouze na klíčové komponenty exportéru a vytvořil jsem jednoduchý a snadno modifikovatelný popis exportéru v P4. Nyní je možné přehledně a rychle měnit parsované protokoly v paketu, jak se vytváří a z čeho se skládá síťový tok, jak tok vyplnit do IPFIX paketu nebo vytváření parsovacích pluginů pro aplikační protokoly.

Exportér díky této práci navíc dokáže zpracovávat libovolné množství hlaviček v paketu a zejména tunelovaný provoz. Paket s tunely je v exportéru navíc rozdělen do více toků a jejich hierarchická struktura zaznamenána pomocí identifikátoru toku a identifikátoru rodiče (tunelu). Parsovací pluginy je možné díky vysokoúrovňovému jazyku P4 psát rychleji než v C nebo C++.

Stejný přístup jako v této práci lze využít pro jiné síťové aplikace jako je například aplikační firewall. Pokračováním práce může být nalezení optimalizací pro takto generovaný exportér.

Literatura

[1] flow_meter exportér. Available from: https://github.com/CESNET/Nemea-Modules/tree/master/flow_meter

[2] Havránek, J. *Exportér síťových toků s podporou aplikačních informací.* Bachelor's thesis, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií. Available from: https://dspace.cvut.cz/bitstream/handle/10467/69654/F8-BP-2017-Havranek-Jiri-thesis.pdf?sequence=1

[3] P4 Language Consortium. Available from: https://p4.org/

[4] Jazyk P4 jako budoucnost SND. Available from: https://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn/

[5] P4 v16 compiler. Available from: https://github.com/p4lang/p4c

[6] Jazyk P4 jako budoucnost SND dokončení. Available from: https://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn-dokonceni/

[7] P4 16 Language Specification version 1.0.0. Available from: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[8] P4-16 core library. Available from: https://github.com/p4lang/p4c/blob/master/p4include/core.p4

[9] P4 16 simple router. Available from: https://github.com/p4lang/p4c/blob/73f100e460e60d8a872522991603a2b2c3cf77ea/testdata/p4_14_samples_outputs/simple_router-first.p4

[10] extended Berkley Packet Filter backend. Available from: https://github.com/p4lang/p4c/tree/master/backends/ebpf

[11] P4 HLIR. Available from: https://github.com/p4lang/p4-hlir

[12] RFC 7011: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. Available from: https://tools.ietf.org/html/rfc7011

[13] Velan, P. Flowmon export IPFIX. Available from: https://dior.ics.muni.cz/~velan/flowmon-export-ipfix/

[14] eXpress Data Path backend. Available from: https://github.com/vmware/p4c-xdp

[15] Tu, W.; Ruffy, F.; et al. Linux Network Programming with P4. In *Linux Plumbers' Conference 2018*, Vancouver, Canada, 2018.

[16] Flexible packet parser backend. Available from: https://github.com/CESNET/fpp-p4c-backend

[17] Havránek, J.; Velan, P.; et al. Enhanced Flow Monitoring with P4 Generated Flexible Packet Parser. In *AIMS 2018*, Munich, Germany, 2018.

[18] Benáček, P.; Puš, V.; et al. P4-to-VHDL: Automatic Generation of 100Gbps Packet Parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM2016)*, Washington, USA, 2016.

[19] Čtrnáctý, M. *Síťová aplikace v jazyce P4*. Bachelor's thesis, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018. Available from: https://dspace.cvut.cz/bitstream/handle/10467/73829/F8-BP-2018-Ctrnacty-Martin-thesis.pdf?sequence=-1&isAllowed=y

[20] Behavioral Model Backend. Available from: https://github.com/p4lang/p4c/tree/master/backends/bmv2

[21] P4 ehavioral model. Available from: https://github.com/p4lang/behavioral-model

[22] re2c lexer generator. Available from: http://re2c.org/

[23] re2c regular expression syntax. Available from: http://re2c.org/manual/syntax/syntax.html

[24] A Template Engine for Modern C++. Available from: https://github.com/pantor/inja

Příloha A

Acronyms

P4 Programming Protocol-independent Packet Processors

eBPF extended Berkley Packet Filter

XDP eXpress Data Path

FPP Flexible Packet Parser

BMV2 Behavioral Model version 2

IPFIX Internet Protocol Flow Information Export

HTTP Hypertext Transfer Protocol

DNS Domain Name System

NTP Network Time Protocol

SIP Session Initiation Protocol

ARP Address Resolution Protocol

SMTP Simple Mail Transfer Protocol

PEN Private Enterprise Number

IANA Internet Assigned Numbers Authority

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové kódy
  - `implementation` ..... zdrojové kódy backendu překladače
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS