VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

OCHRANA PŘED DOS ÚTOKY S VYUŽITÍM JAZYKA P4

PROTECTION AGAINST DOS ATTACKS USING P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **KAMIL VOJANEC**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN KUČERA**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Vojanec Kamil**
Program: Informační technologie
Název: **Ochrana před DoS útoky s využitím jazyka P4**
Protection Against DoS Attacks Using P4 Language
Kategorie: Počítačová architektura

Zadání:

1. Nastudujte vlastnosti jazyka P4, seznamte se s jeho kompilátorem a možnostmi převodu popisu síťového zařízení z P4 do VHDL. Dále se seznamte s problematikou útoků typu odepření služby (DoS) a zařízením vyvíjeným v rámci sdružení CESNET pro ochranu před těmito útoky.
2. Analyzujte klíčové vlastnosti a požadavky na toto zařízení a identifikujte možnosti jejich popisu v jazyce P4.
3. S ohledem na popis systému v jazyce P4 dále navrhněte potřebná rozšíření kompilátoru, hardwarové architektury a softwarového API generovaného z jazyka P4. Zaměřte se na možnost využití externí paměti u Match-Action bloků.
4. Navržené úpravy implementujte.
5. Vyhodnoťte implementované řešení z hlediska dosažených vlastností.
6. V závěru diskutujte výsledky a možnosti dalšího pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce: **Kučera Jan, Ing.**
Konzultant: Kuka Mário, Ing., CESNET
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 25. října 2019

Abstrakt

Bakalářská práce se zabývá přepracováním architektury existujícího zařízení pro ochranu před útoky typu DoS (Denial of Service, odepření služby) do prostředí vysokoúrovňového programovacího jazyka P4. Důvodem využití jazyka P4 je usnadnění adaptace funkční části zařízení na měnící se typy útoků. Nově vytvářené zařízení je navrženo jako modulární a umožňuje snadnou modifikaci změnou zapojených komponent. Cílovou platformou pro tuto práci jsou akcelerační karty s FPGA čipy. Výsledkem práce je návrh řady firmwarových modulů pro ochranu před DoS útoky a implementace cílové aplikace sestavené z těchto modulů. Dílčí výsledky práce byly prezentovány na mezinárodní konferenci IEEE ANCS (Symposium on Architectures for Networking and Communication Systems) v září 2019 na University of Cambridge.

Abstract

This thesis focuses on reimplementation of existing DoS (Denial of Service) attack mitigation device with high-level P4 programming language. The main reason for using P4 is to enhance adaptability and functionality to different types of DoS attacks. The created device is designed in a modular way and enables easy alterations by using interchangeable components. The target platform for this thesis is an FPGA acceleration card. The work results in designing several DoS mitigation components and implementing applications composed of these components. Pats of this work have been presented at IEEE ANCS (Symposium on Architectures for Networking and Communication Systems) in September 2019 at University of Cambridge [14].

Klíčová slova

odepření služby, DoS, P4, FPGA, DDoS Protector, vysokorychlostní sítě, hardwarová akcelerace

Keywords

Denial of Service, DoS, P4, FPGA, DDos Protector, High speed networking, Hardware acceleration

Citace

VOJANEC, Kamil. *Ochrana před DoS útoky s využitím jazyka P4*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Kučera

Ochrana před DoS útoky s využitím jazyka P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kučery. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Kamil Vojanec
26. května 2020

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Janu Kučerovi za vedení, odborné konzultace a ostatní pomoc při zpracování této práce. Dále děkuji pracovníkům Oddělení nástrojů pro monitoring a konfiguraci při sdružení CESNET za odbornou pomoc, zejména pak svému odbornému konzultantovi Ing. Mário Kukovi. Rovněž děkuji své rodině za jejich podporu po dobu celého studia.

Obsah

Kapitola 1

Úvod

Rychlý rozvoj komunikačních technologií a snaha spolu propojit čím dál tím více různých zařízení s sebou přináší stále se zvyšující nároky na zkvalitňování komunikační infrastruktury. Jedním z hlavních prvků této infrastruktury jsou počítačové sítě. Tyto prostředky umožňují propojení počítačů, sdílení souborů, vzdálené ovládání různých zařízení a podobně. Současné počítačové sítě dosahují vysokých přenosových rychlostí. Navíc je trendem tuto rychlost stále zvyšovat.

Se zvyšováním přenosové rychlosti však přichází potřeba komunikaci na těchto sítích zabezpečovat a monitorovat. Existující přístupy k zabezpečení a monitorování však na vysoké rychlosti nestačí a je proto nutné přejít k řešením, které využívají hardwarovou akceleraci.

Vytváření systémů pro různé akcelerační platformy je velmi náročná práce, která zabírá velké množství času a vyžaduje zkušené programátory. Útoky na síti se však velmi rychle vyvíjejí a je potřeba na tento vývoj reagovat. Z tohoto důvodu je vhodné využívat aplikačně specifické jazyky, jako například P4. Tento jazyk značně zjednodušuje popis síťových zařízení a tím umožňuje snadno a rychle upravovat vlastnosti síťových zařízení a tím reagovat na měnící se charakter útoků.

Jedním z typů útoků, před kterými je třeba sítě chránit, jsou útoky typu odepření služby (*Denial of Service*, *DoS*), které mají za cíl omezit přístup legitimních uživatelů k cílové službě.

Cílem této práce je navrhnout sadu několika modulů, jejichž účelem je sledovat síťový provoz a v případě možnosti útoku typu DoS škodlivý provoz blokovat. Navržené moduly budou dále složeny do aplikací, které zvoleným způsobem chrání před útoky typu odepření služby. Výsledkem práce je sada komponent v jazyce P4, a rovněž několik aplikací z těchto komponent složených.

Kapitola 2 se zabývá popisem programovacího jazyka P4, který je hlavním jazykem pro implementaci zařízení na ochranu před DoS útoky. V kapitole 3 je proveden teoretický rozbor útoků typu Denial of Service, před kterými má vytvořené zařízení chránit. Návrh jednotlivých komponent a z nich sestavených aplikací je popsán v kapitole 5. Pro ukládání velkého množství záznamů pro ochranu před DoS útoky je vhodné využít externí paměť, její použití je popsáno v kapitole 6. Implementace jednotlivých komponent, aplikací a zapojení externí paměti je popsáno v kapitole 7 a dosažené výsledky po implementaci v kapitole 8.

Kapitola 2

Jazyk P4

Programovací jazyk P4 (*Programming Protocol-independent Packet Processor*) [10] se řadí mezi deklarativní programovací jazyky. Účelem jazyka P4 je určit způsob zpracování paketů v síťovém zařízení. Původně byl jazyk P4 zamýšlen pro programování síťových přepínačů, velmi rychle se však rozšířil i na jiná zařízení jako jsou směrovače nebo síťové karty.

Jazyk P4 je nezávislý na protokolu, což umožňuje popisovat i zařízení, která používají uživatelem definované protokoly. Výhodou nezávislosti na protokolu je možnost snadno přidat podporu nově vzniklých protokolů nebo navrhovat nové, ještě neexistující.

Jazyk P4 se v současnosti vyskytuje ve dvou specifikacích – $P4_{14}$ [5] a $P4_{16}$ [7]. Rozdíly mezi těmito specifikacemi jsou popsány v kapitole 2.2.

2.1 Abstraktní model P4 zařízení

Popis síťového zařízení v jazyce P4 se řídí tzv. *abstraktním modelem*. Jeho účelem je sjednotit způsob, jakým je popsáno zpracování paketů v síťovém zařízení nezávisle na použité technologii (software, FPGA, ASIC) a typu popisovaného zařízení (síťová karta, směrovač, přepínač).

Schéma 2.1 zobrazuje síťové zařízení popsané jazykem P4. Vidíme, že se jedná o zřetězenou linku (*pipeline*) sestávající se z komponenty *parser*, dvou bloků *match+action* a komponenty *deparser*. Varianta jazyka $P4_{14}$ byla omezena na dvě zřetězené linky bloků match+action, novější $P4_{16}$, toto omezení už nemá a programátor si může linek nadefinovat více.

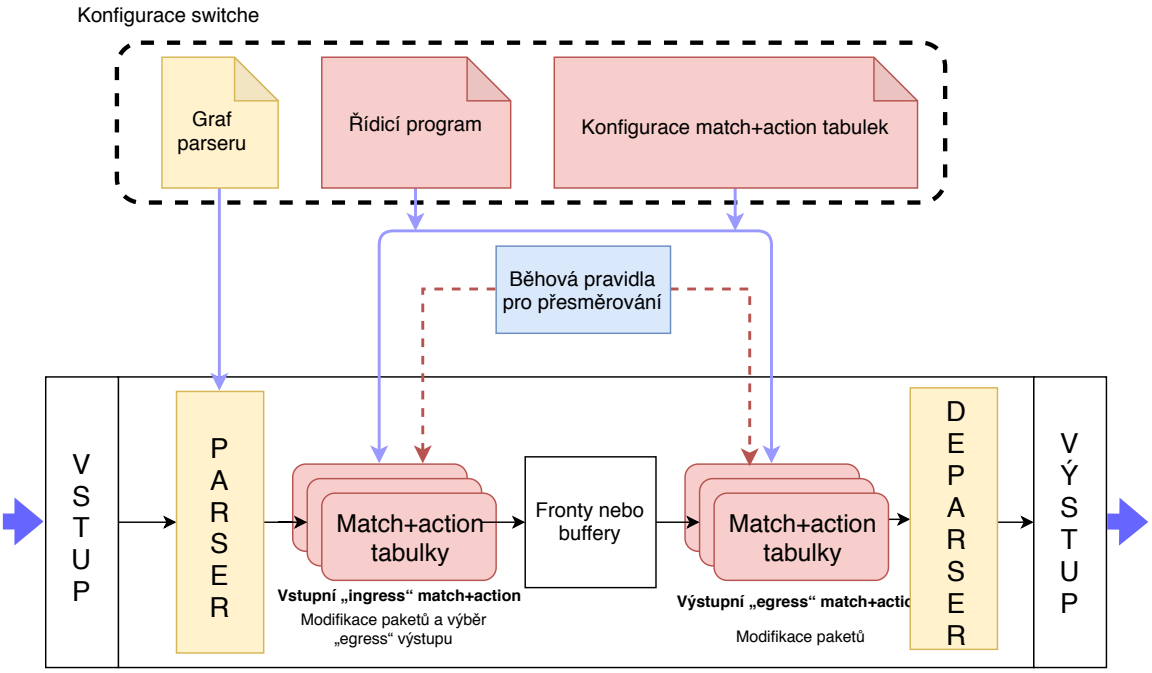
Komponenta parser má za úkol zpracovat příchozí paket podle pravidel specifikovaných v programu (viz kapitola 2.2) do sady protokolových hlaviček. Parser bývá zpravidla implementován jako konečný stavový automat, čemuž odpovídá i způsob zápisu v programu.

Dalším prvkem P4 pipeline jsou sady match+action tabulek. Těch může programátor definovat několik, nejčastěji se však vytvářejí dvě. První sada je umístěna ihned za parserem a nazývá se *ingress*. Účelem ingress tabulek je modifikovat hlavičky paketů nebo metadat. Rovněž je možné upravit výstupní port síťového zařízení. Druhá sada match+action tabulek se nazývá *egress*. Podobně jako u ingress tabulek mohou i egress tabulky modifikovat hlavičky paketů a metadata, nesmí ale modifikovat výstupní port.

Match+action tabulky obsahují část *match* a část *action*. Hlavním úkolem části match je porovnat zpracovávanou hlavičku paketu s údaji v paměti. V případě nalezení odpovídajícího záznamu, vrátí se hodnota *hit* a hlavička paketu se předá části action, která následně provede akci, která je určena programem (například modifikace hlavičky, nastavení meta-

dat ...). V případě, že v paměti není nalezen záznam, je vrácena hodnota *miss* a žádná akce se neaplikuje. Informace o úspěchu vyhledávání (hit nebo miss) se propagují dále a lze je využít pro řízení toku programu, viz sekce 2.2.5.

Poslední částí P4 pipeline je komponenta deparser. Ta znovu sestavuje hlavičky paketů do výstupního paketu. Tento paket následně opouští pipeline.



Obrázek 2.1: Schéma abstraktního modelu P4 zařízení

2.2 Program v jazyce P4

Jazyk P4 v současné době zahrnuje dvě různé varianty – $P4_{14}$ a $P4_{16}$. Specifikace $P4_{16}$ provedla několik významných změn v syntaxi a sémantice jazyka, které nejsou zpětně kompatibilní s předchozí specifikací. Zejména se jedná o odstranění značné části klíčových slov. Další změnou je přesun některých funkčních vlastností jazyka do externích knihoven. $P4_{16}$ je tak sám o sobě menší než $P4_{14}$. Tato práce dále používá specifikaci jazyka $P4_{16}$. Program v jazyce P4 je složen z následujících bloků:

- 2.2.1 Definice hlaviček – definují formát (velikost a složení) každé hlavičky paketu. Rovněž umožňují definovat uživatelská metadata použitá v programu.
- 2.2.2 Popis grafu parseru – definuje povolené posloupnosti hlaviček paketu, zapsán jako konečný stavový automat.
- 2.2.3 Definice akcí – popisují proveditelné akce.
- 2.2.4 Definice tabulek – definují typ vyhledávání, vstupní hlavičky, použitelné akce a velikost jednotlivých tabulek.
- 2.2.5 Rozvržení pipeline a řídicí struktury – popisují složení bloků za sebou.

2.2.1 Definice hlaviček

Definice hlavičky v jazyce P4 začíná klíčovým slovem `header` (viz výpis 2.1), následuje název hlavičky a poté výčet jednotlivých částí hlavičky. Při popisu dílčích částí je dovoleno použít zanořené struktury, definované před samotnou definicí hlavičky.

```
header Ethernet_h {
    bit<48> srcAddr;        // Délka 48 bitů
    bit<48> dstAddr;
    bit<16> etherType;
}
```

Výpis 2.1: Příklad definice hlavičky v P4 programu

2.2.2 Popis grafu parseru

Definice parseru v jazyce P4 popisuje stavový automat, který řídí zpracování paketu. Příklad popisu je zobrazen na výpisu 2.2. Každý parser musí začínat klíčovým slovem `parser` (řádek 3). Vstupem parseru je nezpracovaný paket a jeho výstupem jsou jednotlivé hlavičky paketu. Parser může rovněž zpracovávat metadata, a to jak vstupní, tak i výstupní. Jednotlivé stavy konečného automatu jsou uvozeny klíčovým slovem `state` (řádek 7). Každý parser musí obsahovat jeden stav s názvem `start`, kde začíná zpracování paketu. Přechody mezi stavy automatu se provádí příkazem `transition` (řádek 12). Výběr následujícího stavu se v P4 provádí příkazem `select`, který realizuje pattern matching, tedy srovnání se vzorem. Poslední stav parseru se zpravidla označuje jako `accept`, proběhlo-li zpracování v pořádku, nebo `reject`, vyskytla-li se chyba. Pro vyhodnocení hodnoty v hlavičce se využívá příkaz `extract`. P4 rovněž umožňuje zpracování části paketu pomocí subparseru, což je zanořený parser, který zpracovává určitou část paketu stejným způsobem jako hlavní parser.

```
// Parser přijímá paket, jeho výstupem je struktura headers,
// a zároveň zpracovává  vstupní a výstupní metadata
parser Top(packet_in packet, out Headers headers, inout Metadata meta,
           inout standard_metadata_t sm) {

    // Každý parser obsahuje stav start
    state start {
        // Vyčte hlavičku ethernet z~paketu
        packet.extract(headers.ethernet);

        // Zvolí přechod do dalšího stavu
        transition select(headers.ethernet.etherType) {
            0x0800 : parse_ipv4;    // Přesun do stavu parse_ipv4, je-li hodnota 0x0800
            _      : accept;        // Jinak konec
        }
    }
    // Zde by parser pokračoval stavem parse_ipv4
}
```

Výpis 2.2: Příklad popisu parseru v P4 programu

2.2.3 Definice akcí

Akce jsou částí programu, které mohou číst a upravovat zpracovávaná data. Definice akce začíná klíčovým slovem `action`, viz výpis 2.3, za nímž následuje seznam parametrů akce,

což jsou data, s nimiž bude manipulováno. Tělo akce se skládá z posloupnosti příkazů, kterými mohou být obyčejné operace (sčítání, násobení, přiřazení, negace...) nebo invokace jiné akce. Akce může také obsahovat větvení. Invokace akce může rovněž být provedena implicitně, při zpracování v match+action tabulkách.

```
// Akce přijímá parametr dstAddr o~velikosti 48 bitů
// a číslo výstupního portu.
action ipv4_forward(bit<48> dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port; // Přiřazení do metadat deklarovaných globálně
    headers.ethernet.srcAddr = headers.ethernet.dstAddr; // Přiřazení do hlavičky paketu
    headers.ethernet.dstAddr = dstAddr;
    headers.ipv4.ttl = headers.ipv4.ttl - 1;    // Odečtení hodnoty v~hlavičce paketu
}
```

Výpis 2.3: Příklad definice akce v P4 programu

2.2.4 Definice tabulek

Tabulky v P4 programu definují match+action tabulky 2.4. Deklarace tabulky je uvozena klíčovým slovem `table` (řádek 2). Invokace tabulky je provedena příkazem `apply`. Tabulka se skládá z deklarace vyhledávacího klíče (klíčové slovo `keys`, řádek 3), typu vyhledávání záznamu (součást deklarace klíče) a seznamu akcí (klíčové slovo `actions`, řádek 8), které tabulka podporuje.

P4 podporuje 3 typy vyhledávání:

Exact Výsledek vyhledání je úspěšný v případě, že se hodnota přesně shoduje s vyhledávacím klíčem.

Ternary Vyhledávací klíč se skládá ze dvou částí – samotného klíče a masky klíče. Maska klíče určuje bity klíče, které budou porovnávány, ostatní budou zanedbány.

Longest Prefix Match Vyhledávací klíč je určen samotným klíčem a délkou *prefixu*. Ta určuje počet bitů, které budou pro porovnání použity.

Volitelně může tabulka obsahovat i záznam o implicitním pravidlu (`default_action`, řádek 13). Dále může definice tabulky volitelně obsahovat i seznam výchozích záznamů, které budou ve výchozím stavu vloženy do tabulky (klíčové slovo `entries`). Dalším volitelným prvkem tabulek je jejich velikost (klíčové slovo `size`, řádek 12).

```
// Definice tabulky
table ipv4_match {
    keys = {
        // Vyhledávání podle cílové adresy pomocí longest prefix match
        headers.ipv4.dstAddr : lpm;
    }
    // Seznam akcí
    acions = {
        ipv4_forward;
        drop;
    }
    size = 1024;    // Velikost tabulky
    default_action = ipv4_forward;  // Výchozí akce
}
```

Výpis 2.4: Příklad definice tabulky v P4 programu

2.2.5 Řídicí struktury a tok programu

Tok programu je rozdělen do control bloků 2.5. Tyto bloky (uvozeny klíčovým slovem `control`, řádek 2) umožňují ve svém těle deklarovat lokální proměnné (řádek 5), definovat akce (řádek 8), tabulky (řádek 9) a rovněž je invokovat. Samotné řízení programu je provedeno ve složeném příkazu uvozeném klíčovým slovem `apply` (řádek 12). Uvnitř tohoto bloku je povoleno invokovat tabulky, akce a rovněž větvit tok programu pomocí příkazu `if-else` (řádek 14).

```
1  // Control blok přijímá zpracované hlavičky a metadata
2  control Top(inout Headers headers, inout Metadata meta,
3              inout standard_metadata_t sm) {
4
5      bit<32> nextHopAddr;    // Deklarace lokální proměnné
6
7      // Definice tabulky a akce
8      action ipv4_forward { ... }
9      table ipv4_match { ... }
10
11     // Řízení toku programu
12     apply {
13         // Větvení, pokud je time to live roven 0
14         if (headers.ipv4.ttl == 0) {
15             drop();     // Invokace akce drop
16         }
17         else {
18             ipv4_match.apply();     // Invokace tabulky ipv4_match
19         }
20     }
21 }
```

Výpis 2.5: Příklad definice control bloku v P4 programu

2.2.6 Bloky definované mimo jazyk P4

V případě nutnosti využít komponent, které samotný jazyk P4 neobsahuje, je povoleno využít speciální konstrukci *extern*. Ta umožňuje definovat komponenty mimo jazyk P4 (například v jazyce C, VHDL apod.) a používat je v prostředí jazyka P4 pomocí určeného rozhraní. Příkladem externích komponent mohou být bloky pro výpočet kontrolních součtů, čítače, registry ...

Pro použití externích komponent je nejprve nutné deklarovat její rozhraní 2.6. Deklarace je uvozena klíčovým slovem `extern` (řádek 1). Následují deklarace metod, které jsou externí komponentou podporovány.

Při použití externí komponenty 2.7 se nejprve vytvoří externí objekt pomocí definovaného konstruktoru (řádek 1). Následně je možné používat metody definované externím objektem. V uvedeném příkladu má externí objekt paměť, je tedy potřeba před každým použitím zavolat metodu *clear*, která vymaže paměť objektu a připraví jej na další použití.

```
extern Checksum16 {
    Checksum16();   // Konstruktor externího objektu
    void clear();   // Příprava objektu k~výpočtu
    void update<T>(in T data);  // Přidání hodnoty kontrolního součtu
    void remove<T>(in T data);  // Odebrání hodnoty kontrolního součtu
    bit<16> get();  // Získání hodnoty kontrolního součtu
}
```

Výpis 2.6: Příklad deklarace externí komponenty v P4 programu

```
Checksum16 ck; // Vytvoření externího objektu pomocí konstruktoru
action update_checksum() {
    headers.ipv4.checksum = 0;  // Vynulování kontrolního součtu IP hlavičky
    ck.clear();                 // Příprava externího objektu pro výpočet
    ck.update(headers.ipv4);    // Výpočet kontrolního součtu hlavičky IP
    headers.ipv4.checksum = ck.get();   // Přiřazení nově vypočtené hodnoty do hlavičky IP
}
```

Výpis 2.7: Příklad použití externí komponenty pro výpočet kontrolního součtu IP hlavičky

2.3 Překladače jazyka P4

2.3.1 Překladač $P4_{14}$

Pro starší standard jazyka P4 existuje projekt *P4-HLIR* (*High level intermediate representation*) od tvůrců jazyka P4, který reprezentuje přední část překladače (lexikální, syntaktická a sémantická analýza). Výstupem tohoto programu je mezikód, který lze dále zpracovat až do výsledného kódu cílové architektury. Na tento projekt dále navazuje překladač P4-VHDL, který generuje kód v jazyce VHDL určený pro použití na čipech FPGA. Jak P4-HLIR, tak P4-VHDL (verze pro jazyk $P4_{14}$) jsou napsány v jazyce *Python*. Architektura starší verze překladače P4-VHDL odpovídá návrhu [8].

2.3.2 Překladač $P4_{16}$

K jazyku P4 byl jeho autory vytvořen překladač *p4c* [3]. Tento překladač podporuje jak $P4_{14}$, tak $P4_{16}$. Narozdíl od staršího překladače lze tento rozdělit do tří základních komponent:

Frontend Provádí lexikální, syntaktickou a sémantickou analýzu zdrojového programu na základě referenčních gramatik a pravidel. Jejím výstupem je vnitřní reprezentace (*Intermediate Representation, IR*) programu. Frontend je zcela nezávislý na cílové architektuře.

Midend Optimalizuje vnitřní reprezentaci programu. Optimalizace mohou být nezávislé na cílové platformě, na základě její specifikace však mohou být některé optimalizace vynuceny nebo naopak zakázány.

Backend Provádí generování kódu pro cílovou platformu na základě optimalizované vnitřní reprezentace programu. Může provádět některé optimalizace, které jsou pro cílovou platformu specifické (pro FPGA například vkládání registrů pro zlepšení časování).

Celková architektura je navržena jako modulární. Tato vlastnost je zejména výhodná pro komponentu backend, která tak může být vytvořena pro jakoukoliv platformu. Referenční překladač obsahuje základní sadu backendů:

1. *p4c-bm2-ss* – Překládá P4 do behaviorálního popisu jednoduchého síťového přepínače.
2. *p4c-ebpf* – Generuje kód v jazyce C, který může být přeložen do podoby EBPF [4] a použit jádrem operačního systému pro filtrování.
3. *p4test* – Backend určený pro testování a ladění překladače p4c.
4. *p4c-graphs* – Vytváří grafy P4 řídicích struktur.

Pro převod popisu zařízení v jazyce P4 do firmware je třeba přeložit P4 zdrojový kód do nízkoúrovňového popisu v jazyce VHDL. Pro tyto účely vznikl v rámci sdružení CESNET backend P4-VHDL pro překladač p4c.

Překladač vychází z původní architektury navržené v práci [8]. Původní překladač však podporoval jen $P4_{14}$. Nový překladač přeloží i zdrojové kódy $P4_{16}$.

Překladač je napsán v jazyce *C++* a pro svoji přední část (lexikální, syntaktická a sémantická analýza) využívá *p4c* [3]. Na to navazuje zadní část překladače, která má na starosti samotné generování kódu. Z důvodu širokých možností zápisu match+action tabulek v $P4_{16}$ se pro jejich překlad využívá technologie *High level synthesis*. V době psaní této práce byl překladač v rané verzi a řadu vlastností jazyka P4 nepodporoval. Zejména se jedná o komponenty typu extern, match+action tabulky využívající různé typy vyhledávání pro různé části klíče a invokaci akcí uvnitř control bloků.

Kapitola 3

Útoky typu odepření služby

Útoky typu Denial of Service, česky odepření služby, představují na internetu vážný problém. Cílem těchto útoků je narušit fungování služby omezením přístupu dané služby nebo serveru, na němž služba běží, k síti. Tohoto cíle lze dosáhnout zahlcením síťových rozhraní, které služba používá, která následně přestanou fungovat a cílová služba je nepoužitelná.

Tyto útoky sice nepoškozují data přímo, ani nedochází k jejich odcizení, ale znemožněním přístupu ke službě mohou jejímu poskytovateli vzniknout finanční či jiné škody.

Dle způsobu provedení můžeme DoS útoky rozdělit do následujících kategorií [11]:

Útoky na úrovní síťového zařízení Útoky na této úrovni cílí na využití chyb v softwaru nebo snahu vyčerpat veškeré hardwarové zdroje cílového síťového zařízení.

Útoky na úrovni operačního systému DoS útoky útočící na operační systém se většinou snaží zneužít způsobů, kterými operační systém cílového počítače implementuje síťové protokoly.

Útoky cílené na aplikace Zaměřují se na aplikace běžící na počítači oběti. Nejčastěji využívají nedokonalostí a chyb těchto aplikací nebo zneužívají funkčnost aplikací pro zahlcení zdrojů cílového počítače.

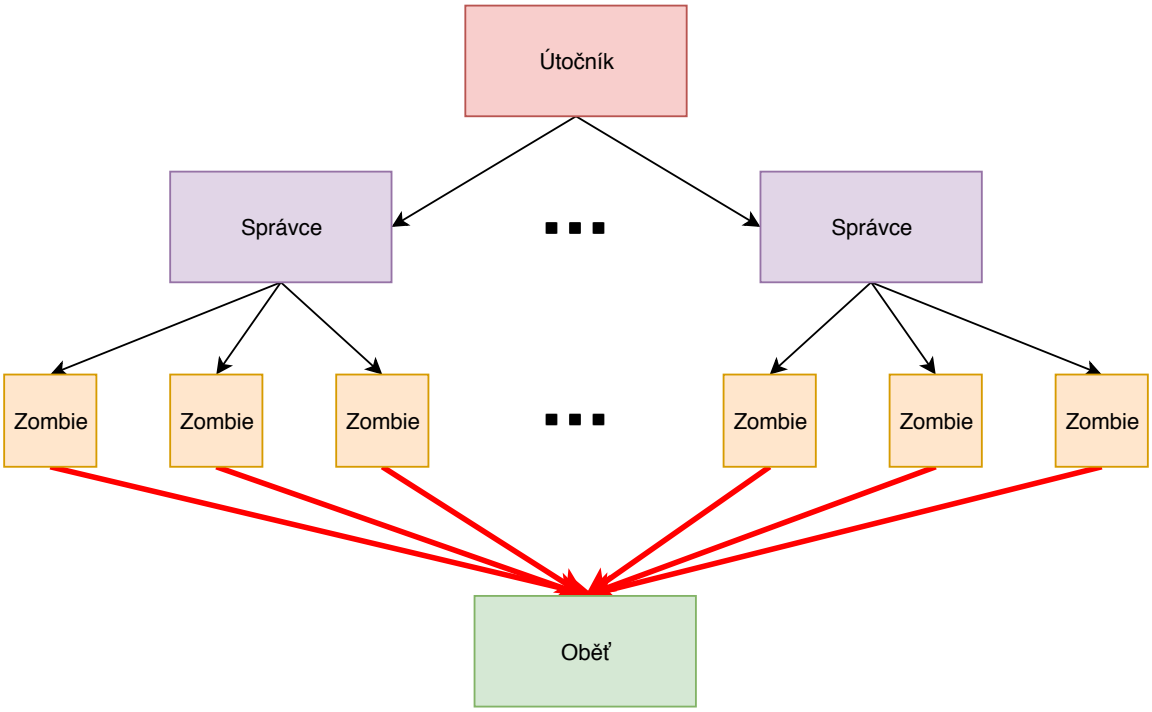
Datová záplava Využívá co největší šířku pásma posíláním extrémního množství dat po síti. Tímto nutí cílový počítač zpracovávat tato data a ten přestává přijímat ostatní požadavky.

Útoky na vlastnosti protokolů Některé útoky typu odepření služby mohou využívat určitých vlastností protokolů používaných na síti. Do této kategorie se mohou řadit útoky SYN flood [9].

Specifickým typem útoků typu odepření služby jsou *DDoS* útoky (Distributed Denial of Service). Ty se vyznačují tím, že se na nich podílí velké množství počítačů, které byly napadeny a slouží jako tzv. *zombie hosts*. Ty se pak podílí na samotném útoku. Uskupení takto vytvořených zombie počítačů nazýváme *botnet*. Tímto způsobem je možné dosáhnout lepší efektivity útoků a ztížit možnosti jejich prevence. Ne všechny zombie počítače podílející se na DDoS útoku jsou však oběťmi napadení. V případě, že je vlastník počítače příznivcem důvodu útoku, může si na svůj počítač dobrovolně nainstalovat DDoS software [1].

Na obrázku 3.1 můžeme vidět zjednodušené schéma útoku DDoS. Prvky tohoto schématu jsou následující:

- Útočník (attacker) – počítač, že kterého je zahájen útok.



Obrázek 3.1: Schéma rolí při útoku DDoS

- Správce (handler) – počítač, který je ovládán útočníkem, má za úkol řídit zombie počítače.
- Zombie – počítač, který většinou bez vědomí svého majitele realizuje útok.
- Oběť – napadený počítač.

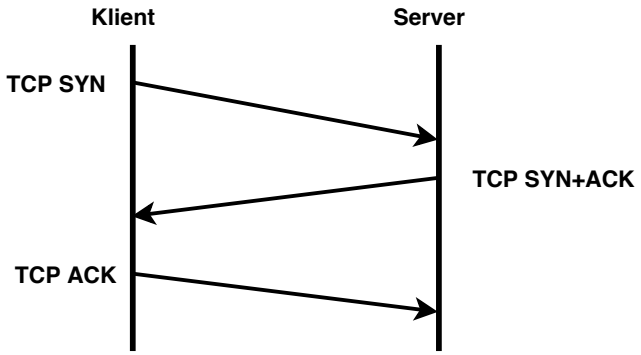
DDoS útoky většinou probíhají v několika fázích [1]. Většinou začínají manuálním nebo automatizovaným průzkumem části sítě, kde útočník hledá počítače, které by mohl napadnout a použít jako zombie nebo handlery. Po prvním průzkumu je často ustaven užší seznam zranitelných počítačů, které jsou dále zkoumány, často pomocí techniky *port scanningu* [13]. Tímto způsobem může útočník zjistit více informací o cílovém počítači, jako například běžící síťově služby, typ operačního systému a podobně. Tyto informace je dále možné použít k identifikaci bezpečnostních chyb a zranitelností cíle. Pro napadení počítače však není vždy zapotřebí využívat některou z bezpečnostních chyb, další metodou bývají phishingové útoky nebo do počítače stažený škodlivý software.

Poté, co byly identifikovány zranitelné počítače dochází k jejich napadení. V závislosti na požadavcích daného útoku se z napadeného počítače může stát jeden z handlerů nebo zombie. Po čase může počet takto napadených počítačů narůst na tisíce až miliony hostitelů [1].

Po napadení počítačů dochází k jejich zapojení do *command and control (C&C)* sítě, která umožňuje jejich snadné ovládání přes internet. Řízení může probíhat různými způsoby. Původně se používala technologie *IRC* (*Internet Relay Chat*), která byla velmi jednoduchá na implementaci.

Jedním z častých typů útoků DDoS je SYN flood [9]. Tento útok využívá principu ustanovení spojení, který je použit protokolem TCP, viz diagram 3.2. Při tomto procesu

nejprve klient pošle TCP segment s nastaveným příznakem SYN, který serveru signalizuje žádost o spojení. Server odpovídá segmentem s příznaky SYN a ACK, což značí úspěšné ustavení spojení ze strany serveru. Poslední zprávou je segment s příznakem ACK, jež zasílá klient serveru. Ten značí úspěšné spojení ze strany klienta. Následně může následovat TCP provoz.



Obrázek 3.2: Ustanovení spojení protokolem TCP

Pokud však nedojde k poslednímu kroku zaslání ACK zprávy klientem, vzniká velké množství neustavených spojení. Každé takové spojení pak zaplňuje paměť serveru a při úplném zahlcení může dojít až k pádu systému a odepření služby.

Dle zprávy společnosti Kaspersky [16] tvořily SYN flood útoky v posledním čtvrtletí roku 2019 až 84,6 % všech DDoS útoků.

Kapitola 4

Klíčové vlastnosti zařízení DDoS Protector

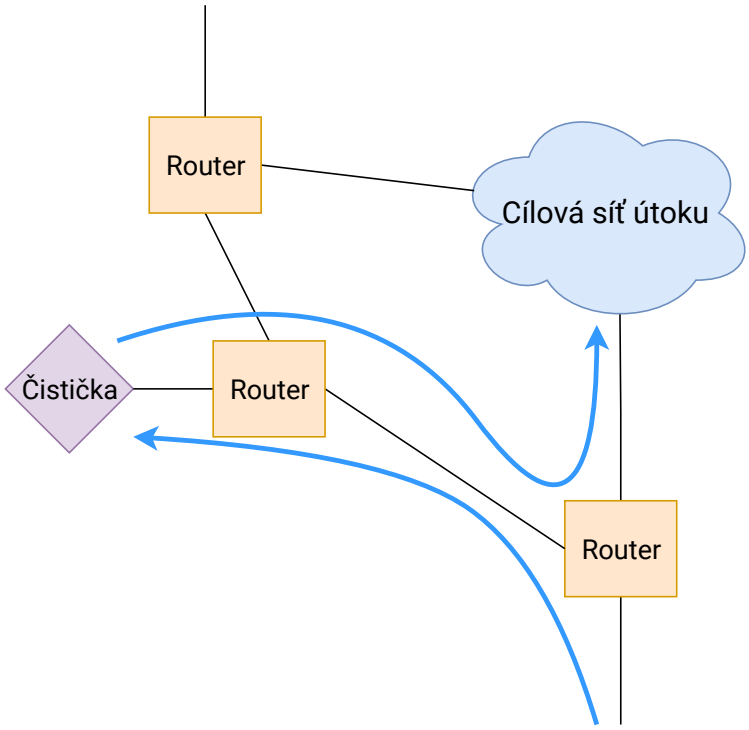
Zařízení pro ochranu před DoS útoky (*čistička provozu*) musí zajišťovat blokování nežádoucího provozu při zachování legitimního provozu s vysokou propustností. Čistička je nasazena v páteřní síti sdružení CESNET, a je tedy nutné dosahovat propustností alespoň 100 Gbps (Gigabitů za sekundu).

Zařízení pro ochranu před DoS útoky může být v síti zapojeno více způsoby. V současné verzi se jedná o dva způsoby zapojení:

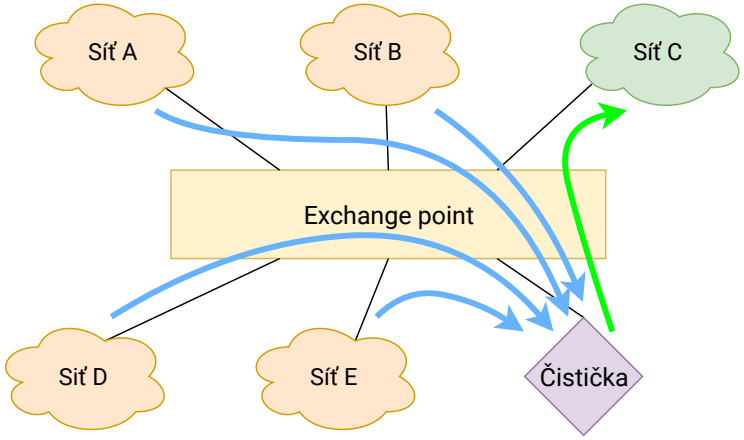
Zapojení v síti CESNET Tento způsob zapojení je znázorněn na schematu 4.1. Jedná se o konfiguraci, ve které je čistička připojena k některému z routerů v síti. Na některém z routerů v síti je identifikován potenciálně nebezpečný provoz a ten je následně směrován do čističky provozu metodou *VRF (Virtual Routing Function)* [2], kdy nebezpečnému provozu je přiřazena hodnota VLAN tagu odlišná od běžného provozu. Firmware zařízení potřebuje mít možnost mapování VLAN tagů, aby mohla vyčištěný provoz vracet zpět do sítě. Jelikož toto zapojení nevyžaduje, aby zařízení provádělo vlastní směrování, je provedena pouze výměna zdrojové a cílové MAC adresy.

Zapojení v peeringovém uzlu NIX.CZ Tento způsob zapojení popisuje schéma 4.2. Jedná se o plnohodnotné zapojení na úrovni L3, kdy čistička provozu musí umožňovat samostatné směrování. V případě, že některý člen peeringového uzlu zjistí, že do jeho sítě směřuje DoS útok, upozorní všechny ostatní členy uzlu, a ti poté provoz směřující do dané sítě pošlou nejprve přes čističku provozu. Ta provoz zpracuje a následně jej směruje dále do původní cílové sítě. Klíčovou komponentou pro tento způsob zapojení je *prefixový filtr*, který implementuje jednoduchou směrovací tabulku. Tento způsob může také využívat virtualizované sítě pomocí VRF. Využití VRF pak umožní jednotlivým členům peeringového uzlu přeposílat nebezpečný provoz v jejich síti na čističku provozu, čímž mohou zabezpečit provoz ve své síti.

Do budoucna čistička počítá s použitím *whitelistu*, tedy seznamu povolených IP adres. Schéma 4.3 popisuje zapojení komponent ve firmware existujícího zařízení [15].



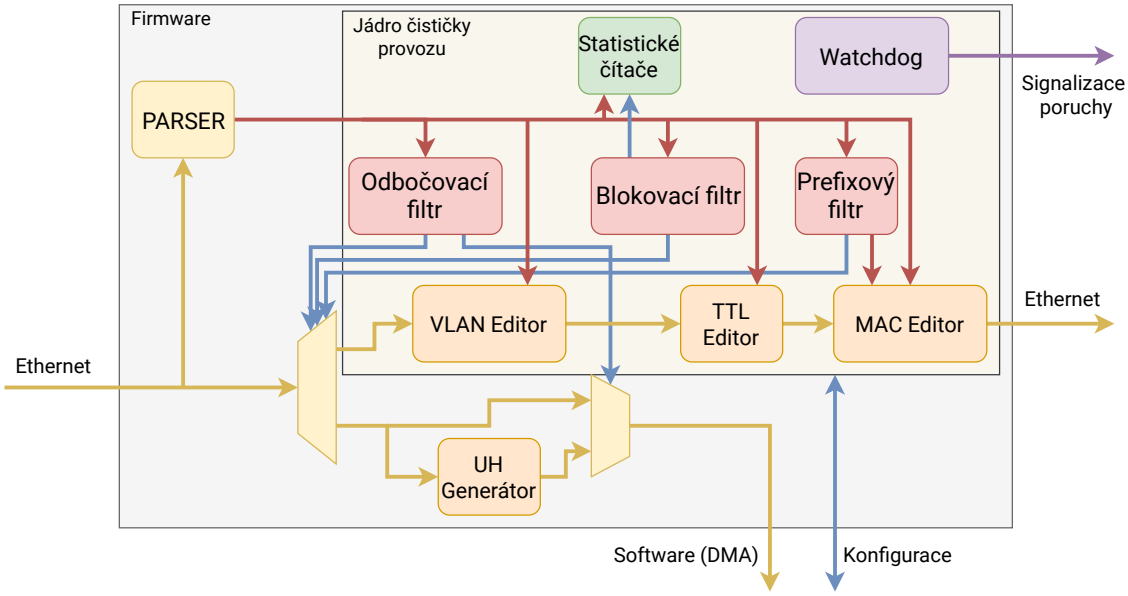
Obrázek 4.1: Schéma zapojení čističky provozu v síťové infrastruktuře CESNET



Obrázek 4.2: Schéma zapojení čističky provozu v peeringovém uzlu NIX.CZ

4.1 Funkční komponenty zařízení

Hlavní funkčnost zařízení je zajištěno dvojicí filtrů, jejichž výstup následně konfiguruje vstupní a výstupní multiplexory a statistické čítače, popřípadě zahazuje pakety. Data z hlaviček paketů jsou nejprve pomocí komponenty parser převedena na vnitřní reprezentaci se kterou pracují ostatní komponenty.



Obrázek 4.3: Firmware zařízení na ochranu před DDoS

4.1.1 Odbočovací filtr

Odbočovací filtr specifikuje, který síťový provoz bude přeposílán do operační paměti počítače přes DMA sběrnici a vybírá konkrétní DMA kanál. Zároveň nastavuje podobu přenášených dat, může se jednat o celé rámce nebo jen hlavičky *UH*, viz kapitola 4.2.1.

V případě posílání provozu dále do ethernetu tento filtr vybírá výstupní síťové rozhraní. Filtr ale také rozhoduje o zahození paketu.

4.1.2 Blokovací filtr

Při DDoS útocích zaznamenáváme řádově tisíce útočících IP adres, které je potřeba filtrovat. Blokovací filtr obsahuje paměť, která všechny tyto adresy uchovává a na jejich základě rozhoduje, zda daný paket zahodí. K blokovacímu filtru je připojena jednotka statistických čítačů, která uchovává počty zahozených bajtů a paketů z blokovaného provozu. Z důvodu velkého množství uchovávaných statistických dat (ke každé IP adrese v paměti připadá jeden čítač) je vhodné tuto komponentu zapojit jako externí paměť. Na základě těchto čítačů je software schopen určit, které IP adresy se nejvíce podílejí na probíhajícím útoku a blokovat je, rovněž je díky nim schopen rozhodovat, zda útok již neskončil.

4.2 Ostatní komponenty

4.2.1 Generátor hlaviček UH

Unified Headers UH je speciální typ hlaviček, který nahrazuje jakoukoli hlavičku paketu. Důvodem použití této hlavičky je snazší analýza v software z důvodu pevně dané struktury pro jakýkoli paket. Součástí hlavičky UH jsou tyto prvky:

- Zdrojová IP
- Cílová IP

- Zdrojový port
- Cílový port
- Délka paketu v bajtech
- Použitý protokol transportní vrstvy
- Příznaky hlavičky IP
- Poslední label MPLS nebo VLAN
- Příznaky TCP

Komponenta UH generátor má na starosti zpracování celých hlaviček paketů a vytvoření hlavičky UH pro každý paket. Na základě konfigurace ze softwaru je poté rozhodnuto, zda se budou do software přenášet původní hlavičky paketů nebo hlavičky UH.

4.2.2 Watchdog

Komponenta *watchdog* je implementována jako konfigurovatelný čítač, v případě dosažení nastavené maximální hodnoty signalizuje na síťovém rozhraní chybu. Resetování čítače probíhá kdykoli je jeho hodnota vyčtena. Cílem je zajistit opakované vyčítání hodnoty čítače, a tím ověřit správnou komunikaci mezi firmware a software.

Kapitola 5

Návrh zařízení v jazyce P4

Původní implementace čističky provozu byla psána v jazyce VHDL, který je jedním z nejpoužívanějších jazyků pro popis číslicových obvodů. Tento jazyk je však velmi komplexní a implementace nových vlastností nebo úprava existujících vlastností zabere velké množství času a znalostí. Pracovníci v odvětví správy sítě však potřebují rychle reagovat na rychle se měnící typy útoků. P4 rovněž umožňuje snadno upravit firmware v závislosti na způsobu zapojení zařízení a přidávat nebo odebírat jednotlivé funkční bloky, což dodává popsaným zařízením potřebnou flexibilitu. V této kapitole je popsán návrh několika nezávislých komponent, které je možné v jazyce P4 popsat a libovolným způsobem zapojit.

5.1 Návrh komponent

Následující část se zabývá návrhem jednotlivých nezávislých komponent v jazyce P4. Některé z těchto komponent vycházejí z původního zapojení DDoS protectoru, některé však byly navrženy na základě požadavků pro další verze čističky.

5.1.1 Mapování VLAN tagů

Jedním z klíčových požadavků pro zapojení zařízení na ochranu před DoS útoky je možnost mapovat VLAN tagy. Tohoto lze v P4 dosáhnout snadno pomocí jednoduché match+action tabulky, která bude přesně porovnávat (exact match, viz sekce 2.2.4) hodnotu VLAN tagu v paketu s pravidlem v tabulce. Klíčovou vlastností této tabulky je její kapacita, která musí pojmout všechny možné hodnoty VLAN tagu, kterých může být $2^{12} = 4096$.

Příklad implementace mapování VLAN tagů v jazyce P4 viz výpis 5.1. Na řádku 2 vidíme definici akce, která mění hodnotu VLAN tagu, na řádku 7 je poté samotná tabulka, která provádí mapování. Řádek 14 specifikuje potřebnou velikost tabulky.

5.1.2 Směrování provozu na L3 vrstvě

Zapojení zařízení pro ochranu před DoS útoky může vyžadovat, aby samo zařízení provádělo směrování. Jednoduché směrování může být v jazyce P4 popsáno jako match+action tabulka, která porovnává cílovou IP adresu pomocí longest prefix match a na jejím výstupu je MAC adresa cílového routeru.

Některé případy užití ale mohou kromě jednoduchého směrování navíc vyžadovat oddělené směrovací tabulky pomocí VRF. Pro tento účel lze sloučit tabulku pro mapování VLAN

```
// Definice akce pro nahrazení VLAN tagu
action replace_vlan(bit<12> new_vid) {
    vlan[0].vid = new_vid;
}

// Tabulka mapující VLAN tagy
table t_replace_vlan {
    keys = {
        vlan[0].vid : exact;
    }
    actions = {
        replace_vlan;
    }
    size = 4096;    // Specifikace velikosti tabulky.
}
```

Výpis 5.1: Příklad popisu mapování VLAN tagů v jazyce P4

tagů (viz kapitola 5.1.1) a tabulku pro jednoduché směrování do jedné tabulky, na jejímž vstupu bude cílová IP adresa porovnávaná pomocí longest prefix match a zároveň hodnota VLAN tagu porovnávaná pomocí exact match. Výstupem této sloučené tabulky bude MAC adresa cílového zařízení a také hodnota VLAN tagu.

Implementace takovéto tabulky by v jazyce VHDL byla velmi náročná, jazyk P4 je ji však schopen popsat velmi snadno, viz výpis 5.2. Na řádku 2 je definována akce, která nahradí cílovou MAC adresu za adresu ve směrovací tabulce a rovněž mapuje VLAN tag. Řádek 7 pak definuje samotnou tabulku. Příklad obsahuje pouze definici tabulky pro směrování IPv4, směrování IPv6 by bylo provedeno analogicky.

V případě zapojení čističky způsobem uvedeným na schématu 4.1 není potřeba plnohodnotné směrování, dostačuje pouze zaměnit zdrojovou a cílovou MAC adresu tak, aby provoz odcházející z čističky provozu pokračoval zpět do routeru, ze kterého původně vycházel. Taková komponenta je v jazyce P4 snadno popsatelná, viz výpis 5.3. Na začátku (řádek 2) vidíme definici lokální proměnné, která je využita pro výměnu MAC adres a je použita v akci definované níže na řádku 7. Instance match+action tabulky na řádku 14 v tomto případě není potřeba, jelikož jazyk $P4_{16}$ dovoluje přímé volání akcí z control bloku (viz 2.2.5), tuto možnost ale nedovoluje použitý překladač a match+action tabulka je uvedena.

5.1.3 Sběr statistik o filtrovaném provozu

V původní implementaci zařízení na ochranu před DDoS útoky existovala komponenta blokovacího filtru. Tento filtr následně sbírá statistiky o zahozených paketech, viz kapitola 4.1.2. Jazyk P4 poskytuje snadný způsob popisu takové komponenty ve dvou fázích, příklad implementace je na výpisu 5.4.

První fází je match+action tabulka (instancovaná na řádku 20), která provádí jednoduché porovnání typu exact na základě zdrojové IP adresy. Tato tabulka by měla obsahovat řádově několik desítek tisíc záznamů, čehož lze dosáhnout s použitím externí paměti. V případě, že při zpracování paketu touto tabulkou dojde k nalezení odpovídajícího záznamu, vyhodnocuje se i následující tabulka (podmíněné vyhodnocení na řádku 50). Při úspěšném

```
// Definice akce pro aktualizaci MAC adresy a VLAN tagu
action prefix_filter(bit<48> new_dst_mac, bit<12> new_vlan_tag) {
    ethernet.dstAddr = new_dst_mac;
    vlan[0].vid = new_vlan_tag;
}

table t_ipv4_prefix_filter {
    keys = {
        ipv4.dstAddr : lpm;
        vlan[0].vid  : exact;
    }
    actions = {
        prefix_filter;
    }
}
```

Výpis 5.2: Příklad jednoduchého směrování s využitím VRF

```
// Nejprve je nutné definovat pomocnou proměnnou pro výměnu
header tmp_mac_h {
    bit<48> tmp_mac;
}

// Definice akce pro záměnu zdrojové a cílové MAC adresy
action swap_mac() {
    tmp_mac_h.tmp_mac = ethernet.dstAddr;
    ethernet.dstAddr  = ethernet.srcAddr;
    ethernet.srcAddr  = tmp_mac_h.tmp_mac;
}

// Tabulka provádějící záměnu.
table t_mac_swap {
    actions = {
        swap_mac;
    }
}
```

Výpis 5.3: Příklad nahrazování MAC adres v jazyce P4

porovnání se rovněž mohou nastavit interní metadata, která mohou být navazující tabulkou využita.

Tato tabulka (řádek 31) obsahuje podstatně menší množství záznamů (řádově stovky), avšak neporovnává pouze IP adresy, ale rovněž několik dalších položek z hlaviček paketu. Tyto položky jsou porovnávány způsobem ternary. Pokud je porovnání úspěšné, je paket vyhodnocen jako nebezpečný a je zahozen. Kromě zahození paketu však dochází navíc k inkrementaci čítače, který odpovídá záznamu v této tabulce.

Komponenta čítače však není součástí jádra jazyka P4 a musí být implementována jako extern. Jeden z možných způsobů deklarace a použití čítačů připojených k tabulkám (tzv. *direct counters*) je uveden v návrhu architektury P4 přepínače [6]. Zmíněná komponenta musí být připojena k nějaké match+action tabulce a obsahuje pole čítačů pro jednotlivé záznamy v tabulce. Voláním metody `count` dochází k inkrementování čítače odpovídajícího záznamu v tabulce.

5.1.4 Filtrování provozu metodou whitelist

Jedním ze způsobů, jakým lze filtrovat provoz na síti, je metoda whitelistu. Tato metoda využívá seznamu povolených adres, jejichž provoz je propuštěn dále ke zpracování. Pokud není adresa nalezena v seznamu, je paket automaticky zahozen. Tato metoda navíc umožňuje sbírat statistiky počtu paketů pro každou IP adresu v seznamu povolených adres a také statistiky počtu zahozených paketů.

V jazyce P4 lze takovou komponentu vyjádřit jako jednu match+action tabulku s jedním čítačem. Tabulka má jako vyhledávací klíč IP adresu zdrojové sítě a vyhledává metodou longest prefix match. V případě úspěšného vyhledání se provede inkrement čítače příslušícího záznamu v tabulce a paket v nezměněné podobě pokračuje k dalšímu zpracování. V případě neúspěšného vyhledání je paket zahozen a je inkrementován čítač zahozených paketů.

5.1.5 Sestavení hlavičky UH

Sestavení hlaviček UH je v jazyce P4 velmi snadný úkol. Lze implementovat akci, která na základě hodnot v paketu vyplní strukturu předem deklarované hlavičky. Tato akce pouze jednoduše nakopíruje data do nové hlavičky viz výpis 5.5. Na řádku 1 vidíme vytvoření struktury pro ukládání metadat. Naplnění této struktury je provedeno podmíněně jen v případě, že je daná hlavička paketu přítomna a tedy je validní. Tato podmínka je zobrazena na řádku 10.

5.2 Návrh cílových aplikací

Z výše zmíněných komponent lze v jazyce P4 sestavit celou řadu aplikací, jejichž cílem je blokování potenciálně nebezpečných zdrojů provozu. Pro většinu aplikací můžeme v P4 využívat stejnou definici hlaviček linkové, síťové a transportní vrstvy. Stejně tak může být pro většinu aplikací použit stejný parser. Hlavní funkční část P4 aplikací se odehrává v match+action pipeline.

```
// Deklarujeme lokální proměnnou pro pomocná metadata
header local_id {
    bit<8> id;
}

// Deklarujeme instanci čítače
DirectCounter<bit<64>>(PACKETS) packet_counter;

action set_local_id(bit<8> id) {
    local_id.id = id;
}

// Definice akce, která inkrementuje čítač a zahodí paket
action count_drop() {
    packet_counter.count();
    drop();
}

// První fáze -- filtrování IP adres
table t_ipfilter {
    keys = {
        headers.ipv4.srcAddr : exact;
    }
    actions = {
        set_local_id;
    }
    size = 65536;
}

// Druhá fáze -- podrobnější filtrování
table t_stfilter {
    keys = {
        // Složený vyhledávací klíč
        local_id.id           : ternary;
        headers.ipv4.srcAddr  : ternary;
        headers.ipv4.dstAddr  : ternary;
        headers.ipv4.protocol : ternary;
        ...
    }
    actions = {
        count_drop;
    }
    size = 256;
    // Přiřazení čítače tabulce není součástí jádra P4
    counter = packet_counter;
}

apply {
    // Podmíněná invokate tabulky t_stfilter
    if (t_ipfilter.apply().hit) {
        t_stfilter.apply();
    }
}

}
```

Výpis 5.4: Příklad popisu statického filtru v jazyce P4

```
header unified_header {
    bit ip_version;
    bit<128> src_ip;
    bit<128> dst_ip;
    bit<16>  src_port;
    bit<16>  dst_port;
    ...
}
action set_uh_data() {
    if (ipv4.valid()) {
        uh.ip_version = 0;
        uh.src_ip = ipv4.srcAddr;
        uh.dst_ip = ipv4.dstAddr;
    }
    else {
    ...
}
```

Výpis 5.5: Příklad sestavení hlavičky UH v jazyce P4

5.2.1 Směrovač s podporou VLAN

Jednou z aplikací, která lze v jazyce P4 snadno vytvořit, je směrovač s podporou VLAN. Tento směrovač může být využit při zapojení DDoS protectoru v peeringovém uzlu NIX, jak je uvedeno v kapitole 4. Hlavním důvodem pro využití tohoto zařízení je možnost individuálně směrovat pakety v závislosti nejen na jejich IP adresách, ale i na identifikátoru VLAN, který přísluší paketu.

V jazyce P4 by se mohlo jednat o jednu match+action tabulku, která vyhledává na základě VLAN identfikátoru a IP adresy. Podobné řešení je zobrazeno v sekci 5.1.2.

5.2.2 Filtr provozu s využitím whitelistu

Další možností, jak blokovat zdroje škodlivého provozu je použití whitelistu. Komponenta whitelistu je detailně popsána v sekci 5.1.4.

Pro efektivní filtrování do whitelistu ukládáme prefixy jednotlivých povolených zdrojových sítí. Na jejich základě pak whitelist rozhoduje o zahození paketů. Další důležitou vlastností whitelistu při tomto použití je velká kapacita pro záznamy v tabulce, aby bylo dosaženo dostatečného počtu povolených sítí.

Rozhraní čítače může být zapojeno na rozhraní P4 zařízení, což dovoluje sledovat počty propuštěných a zahozených paketů řídicím software, který je rovněž zodpovědný za vkládání záznamů do tabulky.

5.2.3 Akcelerace algoritmu SYN Drop

Implementace původního zařízení na ochranu před DDoS útoky obsahovala možnost rozšíření o modul pro detekci útoků typu SYN flood (viz sekce 3). Tento modul však fungoval pouze na úrovni softwaru, použitý algoritmus však lze popsat i v jazyce P4 za použití komponent navržených výše, a díky tomu jej akcelerovat v FPGA. Detekce SYN flood útoků spočívá v počítání počtu přijatých segmentů s příznaky SYN nebo ACK pro jednotlivé zdrojové IP adresy. Následně je podle hodnot čítačů rozhodnuto o propuštění nebo zahození daného paketu. Po určitých časových intervalech jsou čítače segmentů vynulovány.

V jazyce P4 je pro implementaci takovéto aplikace potřeba využít několika externích komponent. Zejména se jedná o registry a hashovací funkce. Rozhraní a chování těchto komponent uvádí dokument [6]:

Registr Tato komponenta je generický paměťový prvek, který uchovává specifikovaný počet záznamů jakéhokoli typu. Jedná se tak spíše o pole registrů. Narozdíl od čítačů však není přímo připojen k match+action tabulce. Při instanciaci registru je pak specifikován typ záznamů, které bude registr ukládat a rovněž kapacita registru. Volitelným parametrem konstruktoru je pak výchozí hodnota záznamů v registru

Registr poskytuje metody `read` pro čtení z registru při specifikování indexu, na který se bude přistupovat a `write` pro zápis hodnoty na index.

V případě akcelerace SYN Drop algoritmu bude registr použit pro ukládání časové značky posledního přijatého SYN nebo ACK segmentu, čítače SYN segmentů, čítače ACK segmentů a zdrojové IP adresy, která slouží jako uložený klíč.

Hashovací funkce Komponenta určená pro hashování je velmi podobná komponentě počítající kontrolní součty. Navíc však umožňuje zvolit použitý hashovací algoritmus, který je zvolen při instanciaci komponenty.

Pro výpočet hashe se využívá metoda `get_hash`, jejíž parametrem jsou data, ze kterých má hash být počítán. Volitelně může být tato metoda volána s argumenty specifikující minimální a maximální hodnotu výsledného hashe. V tom případě je vypočítáno modulo původního hashe a maximální hodnoty a následně přičtena minimální hodnota.

Samotná akcelerace algoritmu SYN Drop může být implementována jako jedna tabulka match+action, která podporuje pouze default akci. Tato akce má tři parametry – konstantu pro offset časové značky aktuálního paketu, spodní práh (S) a horní práh (H) pro vyhodnocení zahazování paketů.

Ve svém těle pak sleduje TCP segmenty s příznaky SYN a ACK, ostatní pakety ignoruje. Na základě zdrojové IP adresy se počítá hash pomocí externí hashovací komponenty. Podle spočítaného hashe se přistoupí na index externího registru. Následně je třeba ověřit správnost indexu, na který bylo přistoupeno. To je provedeno porovnáním zdrojové IP adresy aktuálně zpracovávaného paketu s hodnotou uloženého klíče na zvoleném indexu registru. V případě, že se IP adresy shodují, vyhodnocuje se ještě časová značka paketu, a to sečtením hodnoty časové značky v registru s hodnotou argumentu akce. Pokud je tato sečtená hodnota větší, než časová značka aktuálně zpracovávaného paketu, pokračuje paket ještě k dalšímu vyhodnocení.

V každém případě dojde k inkrementaci příslušného čítače SYN nebo ACK a aktualizaci časové značky.

Vyhodnocení pak probíhá podle jednoduché rozhodovací tabulky 5.1, kde intervaly hodnot SYN a ACK odpovídají hodnotám čítačů SYN a ACK pro sledovanou IP adresu.

		SYN			
	Rozsah	$\langle 1; 1\rangle$	$\langle 2; S\rangle$	$(S; H)$	$\langle H, \infty)$
ACK	$\langle 0; 0\rangle$	Drop	Allow	Drop	Drop
	$\langle 1; \infty)$	Allow	Allow	Allow	Drop

Tabulka 5.1: Rozhodovací tabulka pro akceleraci SYN Drop algoritmu

Tato tabulka používá hodnoty spodního (S) a horního prahu (H), které jsou předány jako argument akce. Segmenty s příznakem SYN mohou být podle rozhodovací tabulky propuštěny nebo zahozeny, segmenty ACK jsou vždy propuštěny, pro jejich zahazování není důvod.

Kapitola 6

Využití externích pamětí

U aplikací využívajících jazyk P4 může být důležité ukládat značné množství pravidel v match+action tabulkách, záznamů v registrech nebo čítačů. Z důvodů omezeného množství zdrojů na čipu FPGA není však vhodné uchovávat velké množství dat v paměťových blocích FPGA. Z tohoto důvodu je na síťových kartách, které využívají FPGA umístěno ještě několik externích paměťových modulů, které jsou na FPGA napojeny.

6.1 Externí paměti na síťových kartách NFB

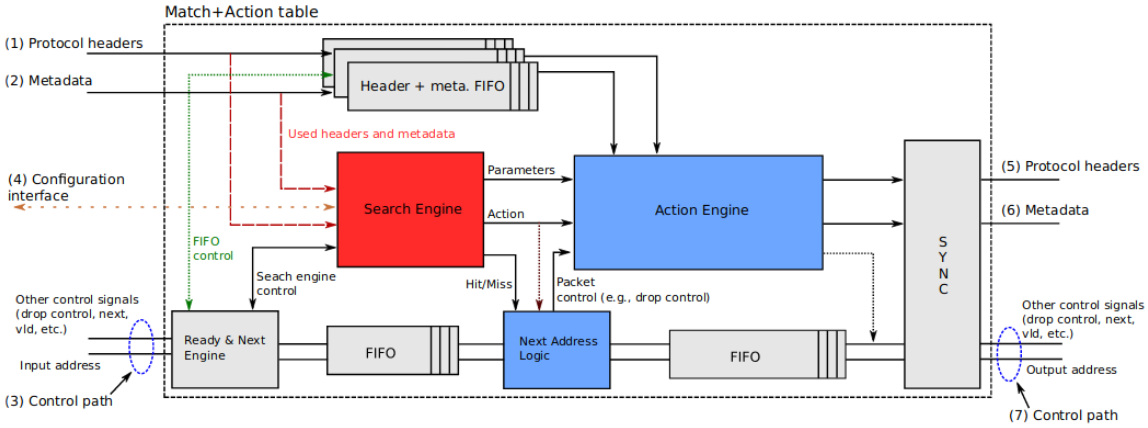
Pro využití externích pamětí v jazyce P4 je potřeba nejprve určit cílovou architekturu, která externí paměti obsahuje. Síťové karty NFB [12] obsahují dva typy externích pamětí – *QDR-III+* a *DDR3*. Z důvodu vyšší propustnosti (více než 50 Gb/s) a možnosti přenášet až 4 datová slova v jednom hodinovém taktu u paměti typu QDR, je tato paměť vhodná pro velké množství transakcí v krátkém časovém rozsahu, což je v případě zpracování vysokorychlostního síťového provozu potřeba.

U síťových karet *NFB100G2Q* je řadič paměti QDR na čipu FPGA zapojen a otestován, bylo možné jej přímo použít. Novější karty *NFB200G2QL* však řadiče externích pamětí sice mají, nejsou však zapojeny do celkového rozhraní. Jedním z dalších úkolů bylo tedy zapojit řadič novějšího typu paměťového modulu QDR na síťové kartě NFB200G2QL do platformy NetCOPE na FPGA. Pro tuto síťovou kartu již existoval experimentální řadič, ten však nebyl nikdy nasazen. U testovací aplikace s tímto řadičem se navíc vyskytl problém nekompatibility se současným systémem, kterým je VHDL kód překládán. Aplikace nejprve byla aktualizována, aby bylo možné ji otestovat. Zde se již objevily problémy se splněním požadavků pro časování, řadiče paměti pro jednotlivé QDR byly nástrojem Vivado umístěny na různé části FPGA čipu. Testování tedy dále probíhalo za použití pouze jednoho řadiče a jedné QDR. Experimentální řadič se ukázal jako funkční a bylo možné jej zapojit do platformy NetCOPE. Pro zachování kompatibility s předchozí řadou karet NFB100G2Q byl tento nový řadič zapojen do stejné obálky, jako řadič starší QDR, což umožnilo zachovat shodné rozhraní. Každou QDR na síťové kartě je před použitím třeba zkalibrovat. K tomuto účelu je použito prostředí *PyNFB*, což je rozšíření knihovny pro práci se síťovými kartami NFB do jazyka Python. V tomto prostředí byl napsán jednoduchý skript, který pomocí sběrnice MI32 zapisuje do konfiguračního registru řadiče QDR paměti a sleduje, zda se kalibrace zdařila. V případě zdařilé kalibrace zaznamená použité kalibrační konstanty. Po kalibraci je paměť QDR připravena k použití.

6.2 Návrh match+action tabulky využívající externí paměť

Jedna z klíčových součástí jazyka P4 je match+action tabulka. Tyto tabulky mohou ukládat velké množství pravidel pro zpracování paketů. Implementace match+action tabulek existující v současných verzích překladače $P4_{14}$ a $P4_{16}$ nepodporují použití externí paměti pro uložení velkého množství pravidel. Pro některé aplikace použité pro ochranu před DoS útoky je však žádoucí uchovávat řádově tisíce až desítky tisíc pravidel.

Schéma 6.1 ukazuje způsob zapojení match+action tabulky v rámci překladače P4-VHDL. Vidíme dvě klíčové komponenty – *search engine* a *action engine*, které dohromady realizují hlavní funkcionalitu match+action tabulek, a to vyhledání akce a aplikování dané akce.



Obrázek 6.1: Schéma match+action tabulky v překladači P4-VHDL [8]

Vyhledávání záznamů v paměti match+action tabulky provádí *search engine*, který proto musí přistupovat k nějakému druhu paměti. Pro zapojení match+action tabulky je tedy nutné vytvořit nový search engine, který bude podporovat zapojení na externí paměť.

Návrh nového search engine s napojením na externí paměť vychází z existující komponenty *search* ze systému *Software Defined Monitoring* [17] a jedná se o komponentu s vyhledáváním typu *exact*. Kromě samotného search engine je z původního SDM použit i paměťový arbitr, jehož účelem je poskytovat jednotné rozhraní pro požadavky na čtení nebo zápis do správné paměti, je-li zapojeno více pamětí.

Zápis velkého množství záznamů vyžaduje využití efektivního algoritmu. Search engine systému SDM využívá algoritmu kukaččího hashování (*cuckoo hash*) [18], který vytváří několik hashí, a tím se snaží minimalizovat kolize při zápisu do paměti. Znázornění algoritmu je na schématu 6.2. Šedá tabulka reprezentuje samotnou paměť, jednotlivé bílé čtverce znázorňují pomocný registr. Šipka s plnou čárou ukazuje, kam je záznam vložen a přerušovaná čára ukazuje výsledek druhé hashovací funkce, který bude do paměti uložen společně se záznamem. Tečkované šipky vpravo od šedé tabulky znázorňují adresu paměti určenou uloženou hashí. Zobrazen je postupný zápis 4 položek (A, B, C, D) s využitím 2 hashovacích funkcí v 7 krocích:

1. Položka A je nejprve vložena do pomocného registru. Následně se vypočtou hashe $h1(A)$ a $h2(A)$, které určují možné adresy paměti, na které se bude zapisovat. Záznam je uložen na adresu určenou $h1(A)$. Společně se záznamem je uložena hodnota $h2(A)$.

2. Při zápisu položky B nedochází k žádné kolizi, může proto být do paměti vložena stejným způsobem.

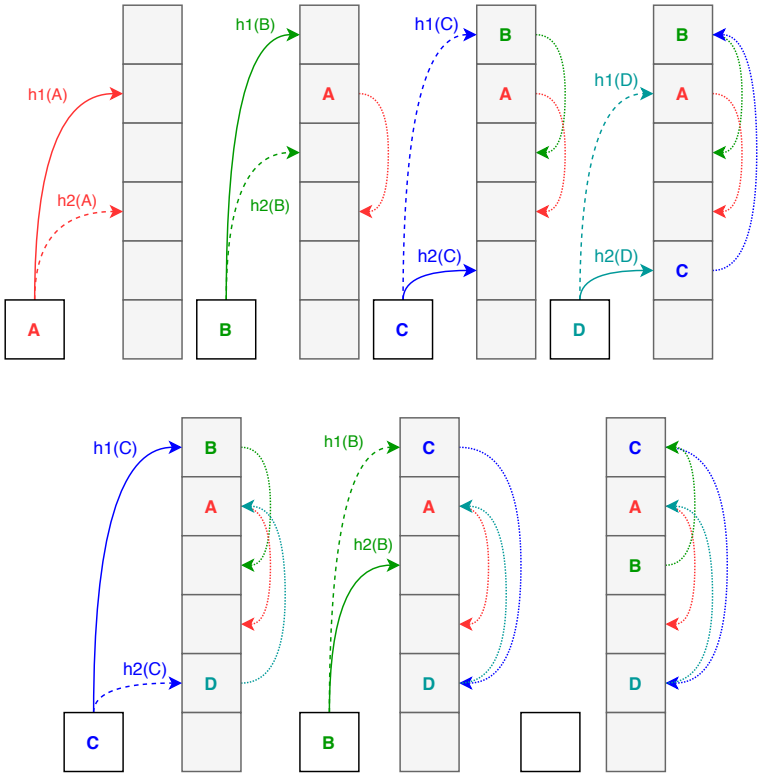
3. Zápis položky C však vyvolá kolizi na adrese určené funkcí $h1(C)$, a proto je záznam vložen na místo určené funkcí $h2(C)$ (označeno plnou čárou) a do paměti je uložena hodnota funkce $h1(C)$.

4. Položka D však vyvolá kolizi jak u adresy určené $h1(D)$, tak $h2(D)$. Dojde proto k reorganizaci záznamů v paměti. Prvním krokem je vyjmutí záznamu C z paměti a zápis položky D na jeho místo.

5. Nyní je potřeba znovu uložit položku C do paměti. Pokus o uložení znovu vyvolá kolizi na obou možných adresách. Je tedy využita hash $h1(C)$, která byla uložena v paměti společně se záznamem, a dojde k vyjmutí položky B na adrese určené $h1(C)$ a zápisu C na toto místo společně s hodnotou $h2(C)$.

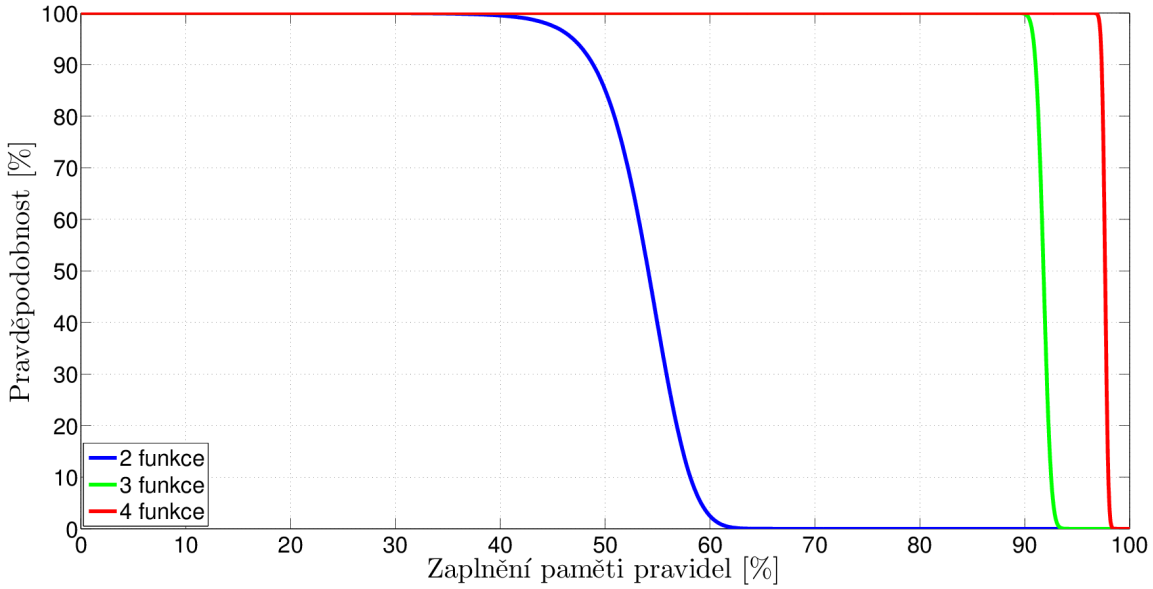
6. Dalším krokem je opakované vložení položky B do paměti. Vidíme, že při pokusu o zápis na adresu určenou funkcí $h1(B)$ dojde ke kolizi, nicméně místo $h2(B)$ je volné a položka B je vložena tam.

7. Tímto způsobem byly uloženy všechny 4 prvky.



Obrázek 6.2: Ukázka algoritmu kukaččího hashování

Algoritmus kukaččího hashování využívající dvě hashovací funkce je však s rostoucím počtem vkládaných záznamů nucen častěji reorganizovat uložené položky a efektivita se snižuje. Nakonec se algoritmus dostane do fáze, kdy není možné vložit nový záznam z důvodu



Obrázek 6.3: Pravděpodobnost zaplnění paměti algoritmem kukaččího hashování v závislosti na počtu použitých hashovacích funkcí

nekonečné smyčky přeskládávání záznamů. Graf očekávané maximální kapacity je zobrazen na schématu 6.3. Vidíme, že při použití pouze dvou hashovacích funkcí maximální zaplnění dosáhne zhruba poloviny kapacity paměti. Implementace match+action tabulky, která využívá tento algoritmus musí obsahovat mechanismus, který umožní detekovat vznik nekonečné smyčky a umožnit smazání libovolné položky z paměti, aby byla reorganizace ukončena.

Schéma 6.4 ukazuje návrh vytvoření nového search engine pro překladač P4. Nejvýraznější úpravou, kterou je potřeba pro zapojení do překladače P4 provést, je úprava rozhraní search komponenty, tak aby odpovídalo jednotnému rozhraní pro všechny P4 search enginy. Pro vkládání záznamů, mazání záznamů a vyčištění celé tabulky je využito rozhraní MI32, které je v rámci překladače P4 používáno standardně a slouží k vkládání záznamů, mazání záznamů a resetování tabulky. Tuto funkcionalitu bude realizovat i v nově vytvořené match+action tabulce. Celkový návrh se skládá z následujících komponent (řazeno shora dolů ve schématu):

Vstupní multiplexor Rozhoduje, bude-li search engine vyhledávat, zapisovat, mazat či provádět reset tabulky a podle toho nastaví příslušné řídicí signály, které ovládají prvky dále v enginu.

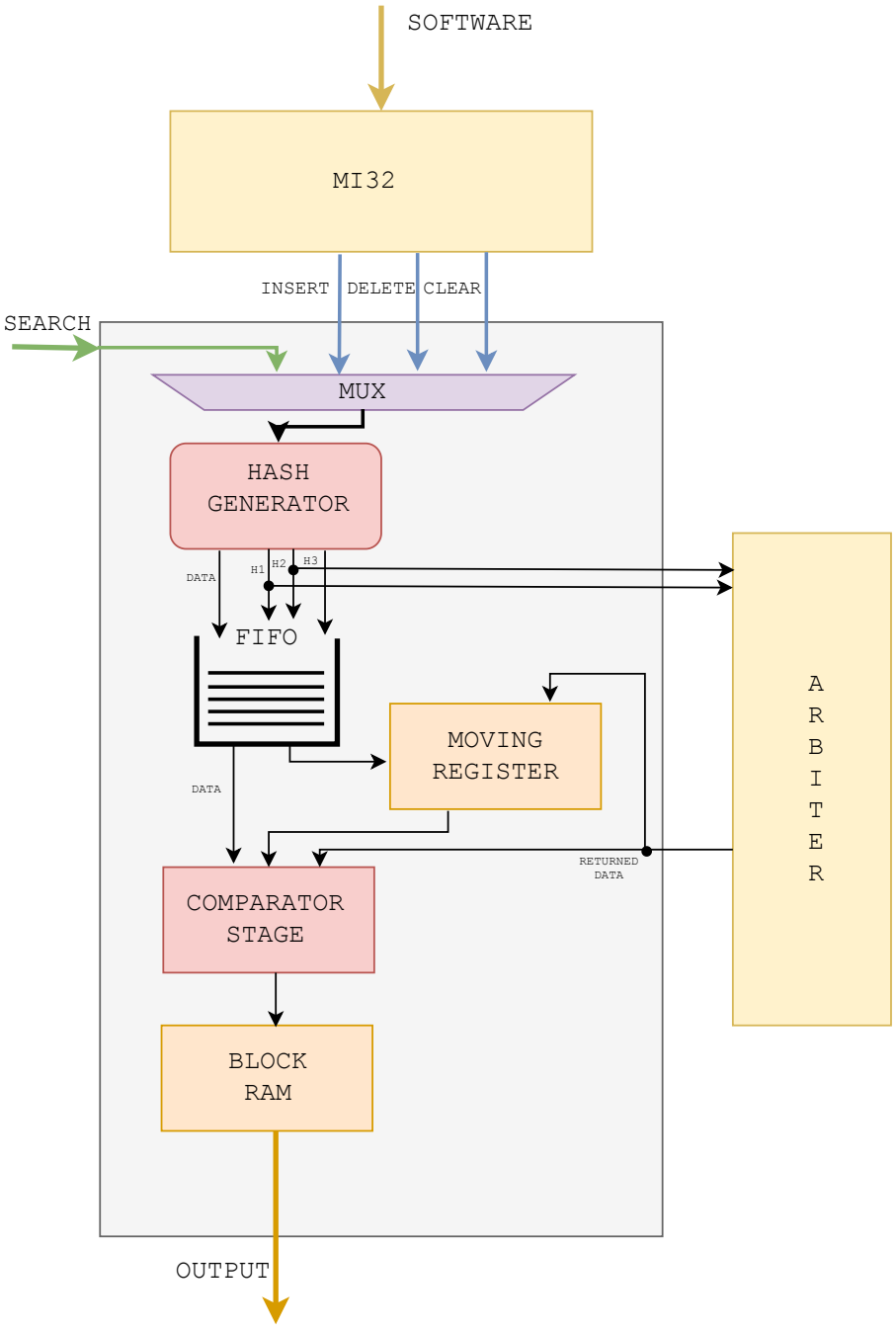
Generátor hash Vytváří hashe určené pro algoritmus cuckoo hash.

FIFO Fronta seřazující požadavky a zajišťující synchronní zpracování.

Moving register Registr obsahující aktuálně zpracovávanou položku. Může získávat hodnoty z fronty FIFO nebo zpětně z komponenty arbiter, pokud dojde ke kolizi při vkládání záznamu, viz 6.2.

Comparator stage Sada komparátorů ověřující validitu dat získaných z paměti. Porovnává hodnoty získané z paměti s hodnotami zpracovávaného požadavku.

Block RAM Paměť na čipu FPGA, která obsahuje jeden bit pro každý záznam v externí paměti. Určuje, zda je záznam v externí paměti validní. Díky tomu není potřeba pro smazání položky přistupovat do externí paměti, ale stačí změnit jeden bit v této block RAM. Stejně tak pro resetování paměti postačí vynulovat block RAM. Navíc je tento mechanismus jednoduše schopen ukončit reorganizační smyčku.



Obrázek 6.4: Celkové schéma komponenty search systému SDM

Kapitola 7

Implementace

Tato kapitola popisuje implementaci jedné z navržených aplikací pro ochranu před DDoS útoky, a to konkrétně směrovač s podporou VLAN. Jedná se o jedinou aplikaci, kterou lze s použitím současné verze překladače P4-VHDL přeložit. Dále se tato kapitola zabývá implementací match+action tabulky využívající externí paměť. Obě sekce obsahují výsledky ze syntézy a implementace designu na FPGA a výsledky experimentálního měření výkonnosti.

7.1 Zařízení pro ochranu před DoS útoky

Prvním krokem implementace bylo vytvoření minimálního funkčního programu v jazyce P4, který lze překladačem P4-VHDL přeložit. Tento program neprováděl žádnou modifikaci hlaviček paketů, pouze zkopíroval paket ze vstupu na výstup. Následovalo vytvoření obálky nad designem, který byl překladačem vytvořen. Tato jednoduchá obálka pouze instancuje samotné vygenerované jádro P4 programu a paralelně s ním používá frontu FIFO určenou pro pomocné hlavičky, které nejsou zpracovány P4 programem. Tato fáze zahrnovala vytvoření top level komponenty pro síťovou kartu *NFB-200G2QL*. Schéma 7.1 znázorňuje komponenty, které jsou zapojeny. Vidíme, že bylo třeba zapojit několik komponent:

slr_crossing Komponenta zajišťuje optimalizaci časování při přechodu obvodu mezi fyzicky oddělenými oblastmi (*Super Logic Region*) čipu FPGA. Je zapojena pro každé vstupní rozhraní.

flu_binder V případě, že je zapojeno více vstupních datových toků, seřazuje tyto toky do jednoho.

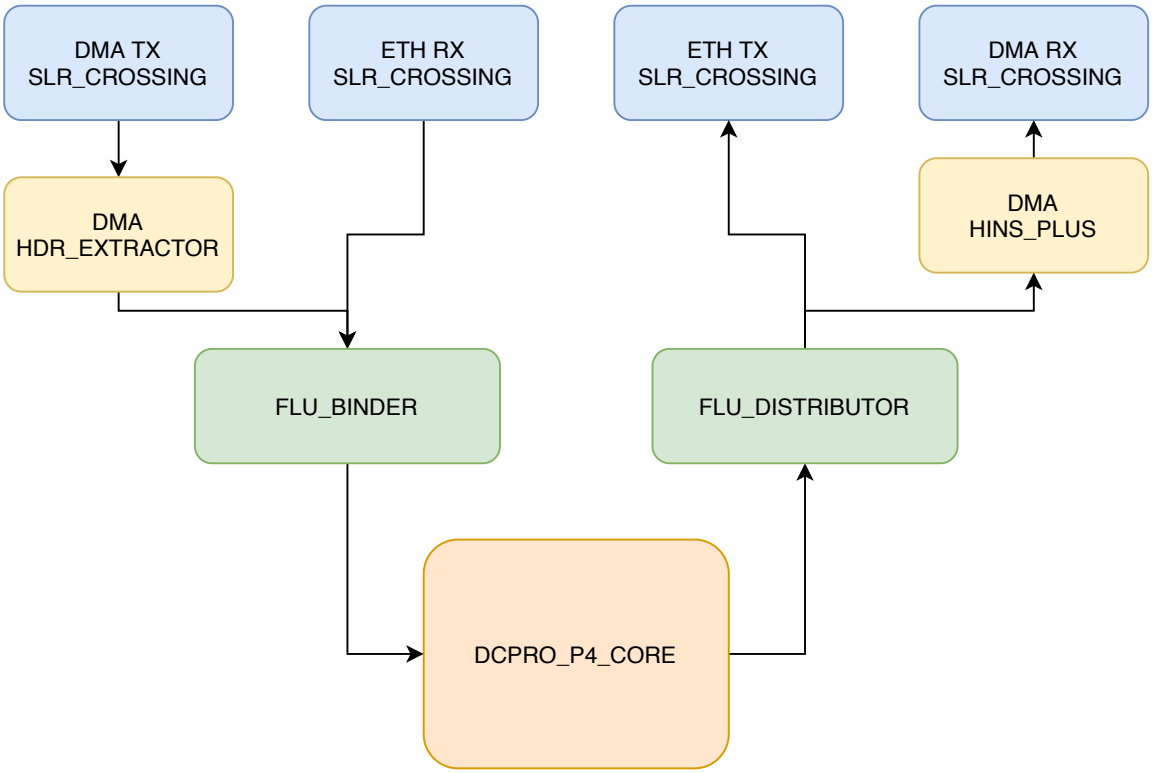
flu_distributor V případě, že je zapojeno více výstupních datových toků, rozděluje jeden tok mezi více výstupních toků.

dma_hdr_extractor Odděluje hlavičku použitou při přenosu DMA od zbytku dat.

hins_plus Přidává hlavičku pro DMA přenos ke zbytku dat.

dcpro_p4_core Jádro P4 programu společně s frontou FIFO pro pomocné hlavičky.

Funkční prázdná aplikace byla poté testována a bylo zkoumáno, zda pakety procházejí jak z ethernetového portu, tak z několika DMA kanálů. Do tohoto prvotního programu byly následně přidány ladící sondy, které pomohly s odhalením chyb v zapojení a synchronizaci



Obrázek 7.1: Schéma zapojení jádra P4 ve firmware DDoS Protectoru

signálů. Po zprovoznění všech vstupních směrů byl do aplikace přidán registr pro nastavení výstupního rozhraní. V této první testovací verzi byl registr ovládán pouze pomocí MI32, a mohlo díky němu být ověřeno, zda data směřují na správné výstupní rozhraní (ethernet nebo DMA kanál).

Dalším krokem bylo sestavit funkční P4 aplikaci. Z důvodu rané fáze vývoje překladače P4-VHDL pro jazyk $P4_{16}$ byl jako vhodná aplikace zvolen router s podporou VLAN mapování (viz sekce 5.2.1). Kromě napsání samotné aplikace bylo potřeba zapojit P4 metadata pro ovládání výstupních rozhraní namísto MI32 registru. Tato metadata byla v P4 programu deklarována podobně jako hlavičky paketů, nicméně protože se jedná o vstupní a výstupní signály, je nutné, aby metadata byla napojena na příslušné signály v obálce nad P4 jádrem. Pro modifikaci hodnot těchto metadat pak již stačí použít jednoduchou match+action tabulku, která bude rozhodovat o výběru správného rozhraní, popřípadě paket zahodí.

Posledním krokem bylo aplikaci syntetizovat a vygenerovat firmware pomocí nástroje *Vivado*. Cílovým FPGA pro tuto aplikaci je čip *Xilinx Virtex UltraScale+*, který je zapojen na síťové kartě NFB200G2QL. Tabulka 7.1 zobrazuje výčet využitých zdrojů na čipu FPGA pro celou aplikaci. Vidíme, že je využito relativně velké množství pamětí BRAM, které jsou využívány match+action tabulkami. Frekvence vygenerovaného obvodu je 125 MHz, maximální možná frekvence je 126 MHz. Je zřejmé, že pro vytvoření výkonnějších aplikací bude potřeba použít optimalizované VHDL komponenty v překladači P4-VHDL.

Blok	Využito	Dostupné	%
LUT	121803	788160	15.454096
LUTRAM	20067	394560	5.0859184
FF	155011	1576320	9.833726
BRAM	345	1440	23.958332
URAM	14	640	2.1875
DSP	121	4560	2.653509

Tabulka 7.1: Tabulka využitých zdrojů na čipu FPGA

Blok	Využito	Dostupné	%
LUT	8334	362800	2.297
LUTRAM	783	141600	0.553
FF	12419	725600	1.712
BRAM	43	940	4.574

Tabulka 7.2: Tabulka využitých zdrojů na čipu FPGA jádrem P4 s SDM search enginem

7.2 Match+action tabulky využívající externí paměť

Funkční jádro jednotky bylo implementováno v jazyce VHDL za pomoci nástroje *Vivado*. Součástí implementace byla rovněž změna používaného generátoru hashí. V původní implementaci měl generátor hashí pevnou vstupní bitovou šířku, v upravené verzi byla tato komponenta nahrazena za generickou implementaci. To umožňuje použití search enginu i s nestandardními formáty vstupních dat, což je vhodné zejména pro jazyk P4. Úpravou rozhraní prošla kromě hlavní komponenty search engine ještě komponenta *arbiter*, která byla rovněž převedena do generické podoby. Tabulka 7.2 zobrazuje zdroje využité jádrem P4 na čipu *Xilinx Virtex 7*. V tabulce 7.3 jsou poté využité zdroje samotnou SDM search jednotkou. Pracovní frekvence tohoto designu je 200 MHz.

Po vytvoření funkční search jednotky a jejím otestování v prostředí *Modelsim* následovalo začlenění search engine do překladače P4. Z důvodu lepší funkčnosti byl pro tuto komponentu zvolen starší překladač P4-VHDL s podporou pouze standardu $P4_{14}$. Tento překladač je napsán v jazyce *Python* a pro sestavení jednotlivých komponent jsou použity sady šablon napsané pro nástroj *Cheetah*, který umožňuje snadno šablony vyplnit zadanými daty.

Pro zapojení nového search engine bylo potřeba vytvořit novou šablonu, která kromě standardního rozhraní search enginů v match+action jednotkách obsahuje navíc ještě rozhraní pro napojení externí paměti. Toto rozhraní je v šablonách ostatních komponent (nadřazených search enginu) generováno podmíněně jen v případě, že P4 program vyžaduje

Blok	Využito	Dostupné	%
LUT	708	362800	0.195
LUTRAM	176	141600	0.124
FF	658	725600	0.091
BRAM	33.5	940	3.564

Tabulka 7.3: Tabulka využitých zdrojů na čipu FPGA samotnou SDM search jednotkou

zapojení komponenty s externí pamětí. Pro zapojení externí paměti je také zapojena komponenta arbiter napojená přímo na rozhraní FPGA čipu.

V samotném programu P4 je pak tabulka s externí pamětí vytvořena pomocí konstrukce *pragma*, viz výpis 7.1. Na stejném příkladu vidíme nastavení maximální velikosti tabulky. Tento parametr je nutné zadat, jelikož výchozí hodnota kapacity v překladači P4-VHDL je 32 a software, který bude tabulku ovládat by nepovolil vložení více pravidel.

```
1  // Definice tabulky
2  @pragma use_external_mem true
3  @pragma engine_type sdm_search
4  table ext_mem_match {
5      reads {
6          headers.ipv4.dstAddr : exact;
7      }
8      acitons  {
9          act_noop;
10         act_drop;
11     }
12     max_size = 500000;    // Velikost tabulky
13 }
```

Výpis 7.1: Vytvoření tabulky s externí pamětí v jazyce $P4_{14}$

Aplikace pracující s designem vygenerovaným překladačem P4 využívají pro komunikaci s firmwarem v FPGA knihovnu *libp4dev*. Tato softwarová knihovna napsaná v jazyce C poskytuje určitou formu abstrakce nad standardním rozhraním MI32 pro zápis a mazání pravidel tabulky a také funkce pro resetování tabulky. Před samotným testováním tak bylo potřeba upravit tuto knihovnu, aby podporovala nově vytvořený search engine. Největším rozdílem mezi search jednotkami, které už v knihovně jsou podporovány a nově vytvořenou jednotkou je množství pravidel, které engine uchovává. Implementace knihovny nepočítala s search enginy, které by mohly podporovat takto velké množství pravidel a obsahovala vnitřní reprezentaci všech pravidel uloženou v jednoduchém poli. Tato vlastnost znemožňovala rozumné testování maximální kapacity paměti, protože při každém vložení pravidla docházelo k procházení celého pole a hledání vhodného místa v poli pro uložení vnitřní reprezentace záznamu. K odhalení tohoto razantního zpomalení byly využity nástroje *sprof* a *callgrind*, pomocí kterých byly nalezeny konkrétní funkce, které nejvíce přistupují do pole záznamů. Pro nový search engine s využitím externí paměti je však použití vnitřní reprezentace záznamů nežádoucí, a proto bylo přidáním jednoduché podmínky zastaveno opakované přistupování do paměti.

Další změnou oproti ostatním search enginům je signalizace zaplnění kapacity tabulky. Zatímco ostatní search jednotky mají kapacitu jednoznačně danou nastavením v P4 programu, popřípadě využívají nějakou implicitní hodnotu, u nově vytvořené jednotky se maximální kapacita z důvodu využití kukaččího hashování liší, viz graf 6.3. Jedinou možností, jak určit plné zaplnění paměti záznamů je sledovat vznik cyklu při reorganizaci položek v paměti. Do knihovny tak byla přidána jednoduchá čekací smyčka, která čeká na signalizaci úspěšného vložení záznamu do paměti. Pokud však tento signál nepřijde do vypršení časovače, je vkládání považováno za neúspěšné a knihovna signalizuje úplné zaplnění paměti. Díky vyšší prioritě operace mazání záznamu je možné reorganizační smyčku ukončit smazáním některého záznamu nebo resetováním tabulky.

Kapitola 8

Dosažené výsledky

8.1 DDoS protector

Kromě implementace samotné P4 aplikace byly navíc vytvořeny testy pro jednotlivé klíčové komponenty a pro měření propustnosti. Pro tyto testy bylo využito existující testovací prostředí původního projektu DDoS protectoru. Toto prostředí je napsáno v jazyce Python, a je navrženo modulárně, což umožňuje snadno přidat vlastní testy včetně testovacích dat. Klíčovým prvkem testovacího prostředí je *Spirent*. Jedná se o generátor a analyzátor síťového provozu o rychlostech až 100 Gb/s, který obsahuje aplikační programové rozhraní pro jazyk Python. Pro testování jsou vytvořeny síťové toky, které jsou zapojeny na vstup síťové karty a sleduje se, jakým způsobem jsou tyto toky modifikovány na výstupu.

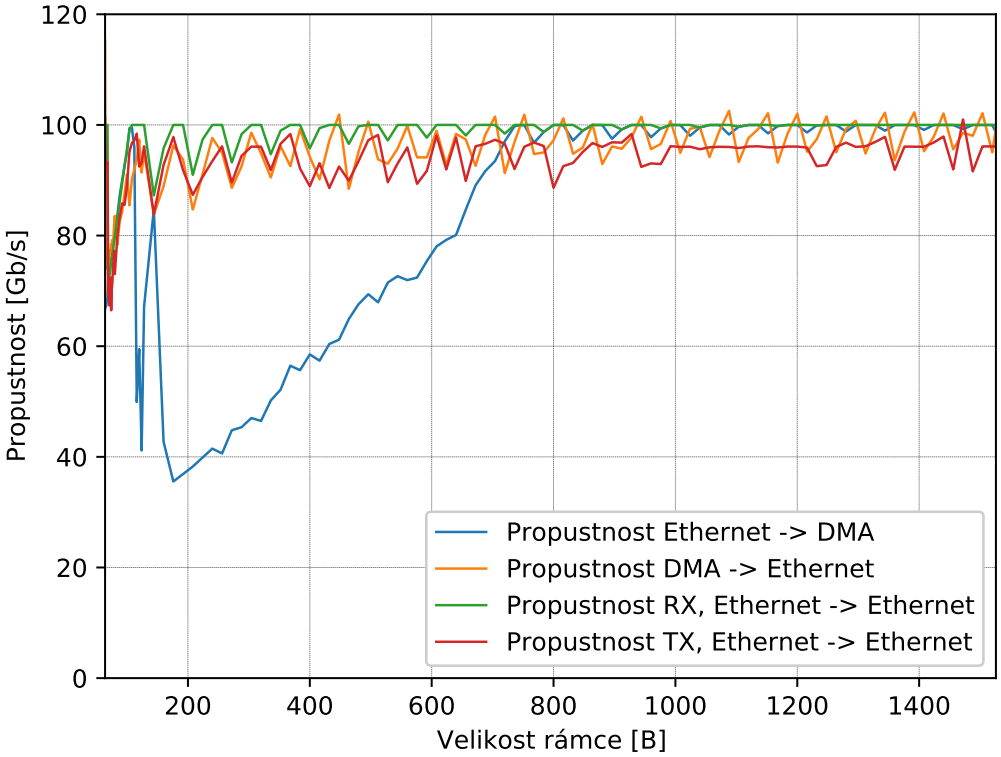
Propustnost aplikace byla testována také pomocí modulu Spirent. Ten je také schopen měřit, kolik paketů prošlo zařízením za určitou dobu a tím určit propustnost. Měření probíhalo třemi způsoby:

1. Vstup z generátoru Spirent na ethernetové rozhraní a výstup směřován na DMA rozhraní, měřeno na ethernetovém vstupu. V grafu 8.1 je toto měření zobrazeno modrou barvou. Vidíme, že malé pakety do velikosti 750 B aplikace není schopná zpracovávat data na plné rychlosti 100 Gb/s. S většími pakety ale aplikace již nemá problém.

2. Vstup generovaný v softwaru, přenášený přes DMA do aplikace a výstup na ethernetové rozhraní, měřeno na ethernetovém výstupu. Na grafu 8.1 je tento směr zobrazen žlutou barvou. Problém při zpracování je u tohoto směru o něco menší, nicméně se ukazuje pilovitý tvar křivky, pravděpodobně způsobený nedokonalým generováním paketů v software.

3. Vstup z generátoru Spirent na ethernetové rozhraní, výstup na ethernetové rozhraní, měřeno na vstupním i výstupním ethernetovém rozhraní. Graf 8.1 zobrazuje výsledky na vstupu (RX) zelenou barvou a na výstupu (TX) červenou barvou. Problém zpracování malých paketů je stále přítomen, nicméně u větších paketů se daří dosahovat propustnosti, která se blíží 100 Gb/s.

U všech měření se vyskytuje problém nedostatečné rychlosti při zpracování velmi malých paketů. Tento problém vzniká z důvodu nutnosti zpracovávat každý paket zvlášť, což u malých paketů vede k zahlcení vstupu pakety. Z tohoto důvodu by pro odstranění tohoto problému bylo potřeba optimalizovat většinu komponent uvnitř překladače. Měření ve směru z DMA do DMA prováděno nebylo, jelikož je pravděpodobné, že software by ne-

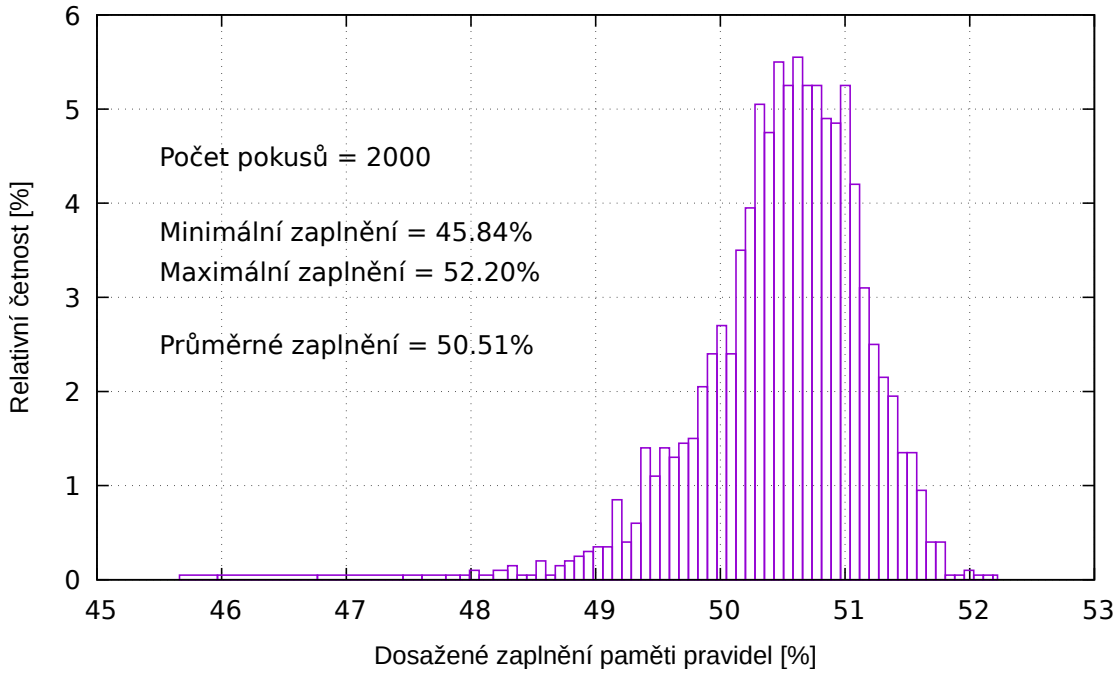


Obrázek 8.1: Graf propustnosti

byl schopen zároveň pakety generovat na DMA a přijímat na DMA, což by vedlo k velmi nízké propustnosti, což by však samotný firmware nemohl nijak ovlivnit.

8.2 Match+action tabulka využívající externí paměť

Pro testování search enginu byly vytvořeny dva testovací programy využívající knihovnu *libp4dev*. První program měl za úkol testovat správnost vkládání pravidel – načítal seznam pravidel ze souboru a poté je vkládal do tabulky. Tímto způsobem bylo ověřeno, že se pravidlo správně vyhledá a aplikuje. Maximální kapacita paměti byla testována jiným způsobem. Program vytvářel pravidla náhodně a tato pravidla vkládal do externí paměti. Tento testovací program byl poté volán opakovaně a při zaplnění paměti byl uložen počet záznamů, kterého bylo dosaženo. Graf 8.2 zobrazuje naměřené maximální zaplnění. Je zřejmé, že experimentální výsledek potvrdil teoretický odhad z grafu 6.3. Přidání další hashovací funkce však kromě větší maximální kapacity také zvýší množství dat, které je potřeba v paměti ukládat.



Obrázek 8.2: Dosažené zaplnění paměti pravidel s využitím kukaččího hashování

Kapitola 9

Závěr

Jazyk P4 umožňuje jednoduše popsat různá síťová zařízení. Jedním z nich je i zařízení na ochranu před DoS útoky. Značnou výhodou tohoto přístupu je jednoduchá adaptace na různé typy zapojení zařízení i rychlé přidání vlastností v případě odhalení nového typu útoku. Nevýhodou zatím zůstává nemožnost přeložit některé komponenty, jako například registry nebo čítače, překladačem P4-VHDL. Zmíněné komponenty jsou důležité pro sběr statistik o zpracovávaném síťovém provozu, což je u některých navržených aplikací klíčové.

Klíčovou částí této práce bylo nastudování principů jazyka P4, možností ochrany před útoky typu Denial of Service a následný návrh komponent a aplikací v jazyce P4, které před DoS útoky mohou chránit. Implementována byla poté sada navržených komponent, které je možné nezávisle na sobě sestavovat do větších aplikací. Pro ukázku funkčnosti navrženého řešení byla jedna z navržených aplikací implementována za pomocí dříve vytvořených komponent. Pro možnosti ukládání velkého množství pravidel v match+action tabulkách byl navržen nový typ tabulky, který využívá externí paměť typu QDR na síťových kartách NFB100G2Q. Po implementaci všech částí práce byly veškeré komponenty testovány jak po funkční, tak po výkonnostní stránce. Match+action tabulka s připojenou externí pamětí byla testována vkládáním náhodných pravidel, a tímto byla ověřena její kapacita, která odpovídá teoretickému předpokladu za použití dvou hashovacích funkcí, tedy více než 250 000 záznamů. U zařízení pro ochranu před DoS útoky je klíčovým výkonnostním prvkem jeho propustnost. Ta rovněž odpovídala očekávanému stavu a dosahovala rychlostí až 100 Gb/s.

Výsledky této práce budou dále sloužit pro rozvoj projektu DDoS Protector v rámci výzkumu sdružení CESNET. Dílčí výsledky práce byly prezentovány v rámci konference IEEE ANCS (Symposium on Architectures for Networking and Communication Systems) v září 2019 na University of Cambridge ve Spojeném Království [14].

Literatura

[1] *A Cisco Guide to Defending Against Distributed Denial of Service Attacks*. Cisco Systems, Inc. [Online], [Cit. 2019-12-18]. Dostupné z: https://tools.cisco.com/security/center/resources/guide_ddos_defense.

[2] *Virtual Route Forwarding Design Guide*. Cisco Systems, Inc. [Online], [Cit. 2019-12-30]. Dostupné z: https://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucme/vrf/design/guide/vrfDesignGuide.html.

[3] *P4c*. Barefoot Networks, Inc, 2013. [Online], [Cit. 2019-12-17]. Dostupné z: https://github.com/p4lang/p4c.

[4] *Berkeley Packet Filter*. FreeBSD, 2016. [Online], [Cit. 2020-01-08]. Dostupné z: https://www.freebsd.org/cgi/man.cgi?bpf(4).

[5] *The P4 Language Specification*. The P4 Language Consortium, listopad 2018. [Online], Verze 1.0.5 (2018), [rev. 2018-11-26], [cit. 2019-12-19]. Dostupné z: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[6] *P4$_{16}$ Portable Switch Architecture*. The P4.org Architecture Working Group, listopad 2018. [Online], Verze 1.1 (2018), [rev. 2018-11-22], [cit. 2020-04-16]. Dostupné z: https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf.

[7] *P4$_{16}$ Language Specification*. The P4 Language Consortium, říjen 2019. [Online], Verze 1.2.0 (2019), [rev. 2019-10-23], [cit. 2019-12-19]. Dostupné z: https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf.

[8] Benáček, P. *Generation of High-Speed Network Device from High-Level Description*. 2016. Disertační práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

[9] Bogdanoski, M., Shuminoski, T. a Risteski, A. Analysis of the SYN Flood DoS Attack. *International Journal of Computer Network and Information Security*. Červen 2013, roč. 5, č. 8, s. 1–11. Copyright - Copyright Modern Education and Computer Science Press Jun 2013. Dostupné z: https://search.proquest.com/docview/1623861730?accountid=17115.

[10] Bosshart, P., Daly, D., Izzard, M., McKeown, N., Rexford, J. et al. Programming Protocol-Independent Packet Processors. 2013.

[11] Douligeris, C. a Mitrokotsa, A. DDoS attacks and defense mechanisms: a classification. In: *Proceedings of the 3rd IEEE International Symposium on Signal Processing and Information Technology (IEEE Cat. No.03EX795)*. Prosinec 2003, s. 190–193. DOI: 10.1109/ISSPIT.2003.1341092.

[12] Friedl Štěpán, Puš, V., Matoušek, J. a Špinler, M. Designing a Card for 100 Gb/s Network Monitoring. Prosinec 2013. Dostupné z: https://www.cesnet.cz/wp-content/uploads/2014/02/card.pdf.

[13] Katterjohn, K. *Port scanning techniques.* 2007. [Online], [Cit. 2019-12-18]. Dostupné z: https://dl.packetstormsecurity.net/papers/attack/port-scanning-techniques.txt.

[14] Kuka, M., Vojanec, K., Kučera, J. a Benáček, P. Accelerated DDoS Attacks Mitigation using Programmable Data Plane. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* 2019, s. 1–3.

[15] Kuka, M. *Hardwarově akcelerované zařízení pro ochranu před DoS útoky.* 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[16] Kupreev, O., Badovskaya, E. a Gutnikov, A. DDoS attacks in Q4 2019. Únor 2020. [Online], [cit. 22.4.2020]. Dostupné z: https://securelist.com/ddos-report-q4-2019/96154/.

[17] Kučera, J. *Aplikačně specifický procesor pro stavové zpracování síťových dat.* 2014. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[18] Pagh, R. a Rodler, F. F. Cuckoo hashing. Elsevier Inc. 2004, roč. 51, č. 2, s. 122–144. ISSN 0196-6774.