SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-104611

ANALÝZA VHODNÝCH STAVOVÝCH JAZYKOV PRE DETEKCIU ÚTOKOV

BAKALÁRSKA PRÁCA

2022 **Alex Teplanský**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-104611

ANALÝZA VHODNÝCH STAVOVÝCH JAZYKOV PRE DETEKCIU ÚTOKOV

BAKALÁRSKA PRÁCA

Študijný program :	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.

2022 **Alex Teplanský**

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program :	Aplikovaná informatika
Vyberte typ práce	Analýza vhodných stavových jazykov pre detekciu útokov
Autor:	Alex Teplanský
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.
Miesto a rok predloženia práce:	Bratislava 2022

V našej bakalárskej práci sme za zaoberali analýzou stavových jazykov ASTD a STATL a ich schopnosťou opisu útokov. Taktiež sme sa zaoberali ontologickým jazykom OWL a snažili sa ho využiť na detekciu útokov, pričom sme vychádzali z predošlej analýzy.. Naším cieľom bolo analyzovať vybrané jazyky ASTD a STATL, porovnať dané jazyky, nájsť spôsob ako využiť OWL pre detekciu útokov, pričom by sme vychádzali z komponentov, ktorými disponujú ASTD a STATL a následne vytvoriť funkčné testy, ktoré by dokázali možnosť, pomocou OWL detekovať útoky. V našej práci sme úspešne analyzovali vybrané jazyky a porovnali ich, našli sme spôsob ako špecifikovať jednotlivé útoky pomocou OWL, no testovanie funkčnosti sa nám nepodarilo z dôvodu absencie autonómneho rozhodovania OWL, čo má za dôsledok nutnosť dodatočného systému ktorý by riadil a prekladal dané špecifikácie zapísané pomocou OWL. Napriek tomu môžeme povedať, že je možné pomocou OWL vytvárať špecifikácie podobné tým ktoré vytvára ASTD alebo STATL.

Kľúčové slová: ASTD, STATL, OWL

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Bachelor Thesis:	Analysis of suitable state languages for attack detection
Autor:	Alex Teplanský
Supervisor:	Ing. Štefan Balogh, PhD.
Consultant:	
Place and year of submission:	Bratislava 2022

In our bachelor thesis we dealt with the analysis of state languages ASTD and STATL and their ability to describe attacks. We also focused on the ontological language OWL and tried to use it to detect attacks, building on the previous analysis. Our goal was to analyze selected ASTD and STATL languages, compare the given languages, find a way how to use the OWL method for attack detection, and we were based on the components available to ASTD and STATL and then create functional tests which would prove the possibility of OWL to detect attacks. In our work, we successfully analyzed selected languages and compared them, we found a way to specify individual attacks using OWL, but we did not test functionality due to the absence of autonomous OWL decision-making, resulting in the need for an additional system to manage and translate the specifications using OWL. Nevertheless, we can say that it is possible to use OWL to create specifications similar to those created by ASTD or STATL.

Key words: ASTD, STATL, OWL

Vyhlásenie autora

Podpísaný Alex Teplanský čestne vyhlasujem, že som Bakalársku prácu Analýza vhodných stavových jazykov pre detekciu útokov vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.
Uvedenú prácu som vypracoval pod vedením Ing. Štefana Balogha, PhD.

V Bratislave dňa 17.06.2022

.....
podpis autora

Poďakovanie

Rád by som sa poďakoval Ing. Štefanovi Baloghovi, PhD. za všetky užitočné rady, poznatky a vedenie danej práce.

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek a značiek

ASTD – Algebraic State Transition Diagram

STATL - State Transition Analysis Technique Language

OWL – Web Ontology Language

IDS - Intrusion Detection System

TCP - Transmission Control Protocol

DL – Deskripčná logika

AWS – Amazon Web Services

DR – Detection Rate

FPR – False Positive Rate

TP – True Positive

FP – False Positive

PD – Positive Detection

FPD – False Positive Detection

Úvod

V súčasnej dobe výrazne vnímame pokroky v oblasti informačných technológii, ktoré vo veľkej miere uľahčujú každodenný život. Rozvoj informačných technológií však so sebou prináša okrem veľkého množstva pozitívnych aspektov aj tie negatívne. Jedným z negatívnych aspektov sú riziká kybernetických útokov. Tieto útoky je možné obmedziť pomocou efektívneho systému kybernetickej bezpečnosti, ktorý chráni kritické systémy a citlivé informácie pred digitálnymi útokmi.

Systém kybernetickej bezpečnosti predchádza kybernetickým útokom pomocou kľúčových technológií a osvedčených postupov. Systém na detekciu útokov by mal spĺňať určite kritéria a mal by vedieť spoľahlivo detekovať a filtrovať škodlivý typ dát. Keďže mnohé z týchto útokov prebiehajú sekvenčne s postupnosťou krokov, je vhodné pre ich detekciu využiť možnosti stavových jazykov, ktoré sa javia ako spoľahlivý a vhodný spôsob.

Stavové jazyky využívajú vlastnosti ako sú stavy, prechody alebo udalosti, pričom dokážu efektívne opísať celý priebeh útoku a odchytiť podozrivé správanie. Výhodou využitia stavových jazykov je hlavne ich schopnosť abstrakcie a to vďaka stavovým diagramom, vďaka čomu sme schopní jednoduchým a prehľadným spôsobom znázorňovať a vytvárať scenáre útokov.

V našej práci sa budeme zaoberať dvoma stavovými jazykmi a to ASTD a STATL, ktoré sú v súčasnej dobe využívané pre detekciu útokov, a analyzovať ich schopnosť detekcie. Taktiež budeme analyzovať ontologický jazyk OWL a skúmať jeho schopnosti pre detekciu útokov v zmysle stavových jazykov, pričom budeme vychádzať z vlastností a funkcií, ktoré ponúkajú ASTD a STATL, a hľadať spôsob ako tieto vlastnosti využiť v OWL a do akej miery je možné použiť jazyk OWL na detekciu útokov. Ďalej sa budeme zaoberať už existujúcim výskumom, ktorý sa zameriaval na 3 systémy pre detekciu útokov, pričom dva systémy využívali stavové jazyky STATL a ASTD, a jeden systém využíval detekciu na základe signatúr a pravidiel. Z tohto výskumu zhodnotime výhody systémov fungujúcich so stavovými jazykmi oproti systémom využívajúcim signatúry, ako aj rozdiely medzi STATL a ASTD.

Na záver sme zhodnotíme vybrané stavové jazyky ako aj schopnosť jazyka OWL detekovať útoky a porovnáme jeho riešenia, ktoré sú potrebné pre detekciu útokov, s riešeniami ktoré ponúkajú ASTD a STAL.

1 Teoretický úvod k stavovým jazykom

Hlavnou témou v našej bakalárskej práci sú stavové jazyky. Preto je dôležite definovať čo sú stavové jazyky a ako fungujú.

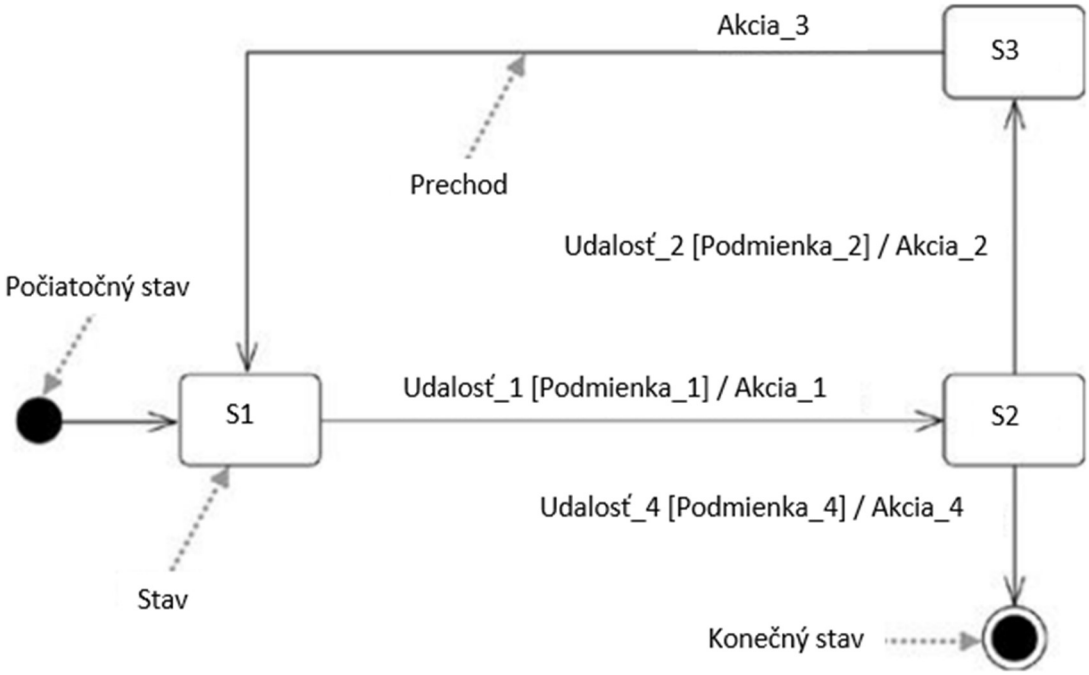
1.1 Stavový jazyk

Počítačové a sieťové útoky môžu byť modelované ako postupnosť krokov, ktoré privedú systém z počiatočného bezpečného stavu do konečného narušeného stavu. Tento modelovací prístup je prirodzene podporovaný jazykom založeným na stavoch a prechodoch. Jednou z výhod tohto prístupu je, že špecifikácie stavov a prechodov možno znázorniť pomocou stavových diagramov. [3]

Stavový diagram opisuje všetky stavy, ktoré môže objekt nadobudnúť, udalosti, pri ktorých objekt mení svoj stav (prechod), podmienky, ktoré musia byť splnené kým prechod nastane (podmienka), a aktivity vykonávané počas životnosti objektu (akcia). Stavové prechody sú užitočné pri opisovaní správania jednotlivých objektov v rámci celého súboru prípadov použitia, ktoré tieto objekty ovplyvňujú. [1]

Jednotlivé časti stavového diagramu sú [2]:

- **Stav:** Za stav považujeme stav objektu počas životnosti, v ktorom spĺňa nejakú podmienku, vykonáva nejakú činnosť alebo čaká na nejakú udalosť.
- **Udalosť:** Je to prípad, ktorý môže spustiť prechod stavu. Udalosti obsahujú jasný signál zvonku systému, vyvolanie zvnútra systému, uplynutie určeného časového obdobia alebo splnenie určitého stavu.
- **Podmienka:** Je Boolean výraz, ktorý ak je pravdivý, umožňuje udalosti vykonať prechod.
- **Prechod:** Je to zmena stavu v rámci objektu.
- **Akcia:** Akcia je jedna alebo viac akcií vykonaných objektom ako reakcia na zmenu stavu.



Obrázok 1: Jednoduchý stavový diagram [1]

Diagram na Obrázku 1 zobrazuje päť stavov, z toho dva stavy sú počiatočný a konečný stav. Taktiež udáva pravidlá, na základe ktorých systém rozhodne ísť z jedného stavu do druhého. Tieto pravidlá sú vo forme prechodov: čiara spájajúca stavy. [1]

Ako sme už spomínali, prechod indikuje prechod z jedného stavu do druhého. Každý prechod môže mať zápis tvorený tromi časťami: udalosť [podmienka] / akcia. Tieto časti sú voliteľné. Chýbajúca akcia indikuje, že po prechode sa nič nevykoná. Chýbajúca podmienka zasa indikuje, že prechod sa vykoná vždy, keď nastane jeho udalosť. Neprítomnosť udalostí pri prechode je nezvyčajná, no môže nastať. Indikuje, že prechod nastane okamžite. Konečný stav indikuje, že stavový automat je dokončený. Napríklad na Obrázku 1 môžeme prechod zo stavu S1 čítať ako „Ak v stave S1 nastane Udalosť_1 a Podmienka_1 je splnená, vykoná sa Akcia_1 a systém sa presunie do stavu S2. [1]

1.2 Vybrané stavové jazyky

Útoky na informačné systémy sa každým rokom stávajú čoraz sofistikovanejšie a bežné, a preto je potreba zdieľať a klasifikovať informácie o prienikoch, exploitoch

a technikách narušenia. Je možné definovať najmenej šesť tried jazykov, ktoré sa zaoberajú útokmi: jazyky udalostí, jazyky odozvy, jazyky nahlasovania, korelačné jazyky, jazyky exploitov a jazyky detekcie [3]. Hoci sú tieto jazyky užitočné a opisujú rôzne oblasti a ciele, žiaden z nich nespĺňa požiadavky bežného jazyka na detekciu útokov, ktoré sú [3]:

- **Jednoduchosť:** jazyk by mal poskytovať práve tie funkcie, ktoré sú potrebné na reprezentáciu útokov.
- **Expresívnosť:** jazyk by mal byť schopný reprezentovať akýkoľvek signál útoku, ktorý je detekovateľný.
- **Prísnosť:** jazyk by mal mat presne definovanú syntax a sémantiku nezávisle od implementácie, takže význam každého popisu útoku je jednoznačný.
- **Možnosť rozšírenia:** nové domény majú nové typy udalostí a vyžadujú funkcie pre tieto typy. Malo by byť možné rozšíriť daný jazyk pre nové domény dobre definovaným a jednoduchým spôsobom.
- **Spustiteľnosť / preložiteľnosť:** malo by byť možné začleniť popisy útokov do IDS aplikácie bez akéhokoľvek manuálneho prekladu. Buď musia byť spustiteľné špecifikácie, alebo techniky a nástroje musia byť poskytnuté na preklad zo špecifikácie útoku na efektívnu implementáciu.
- **Prenosnosť:** nástroje na spracovanie jazyka by mali byť jednoducho prispôsobiteľné rôznym prostrediam.
- **Heterogenita:** malo by byť možné opísať útoky pomocou udalostí z viacerých domén, ako sú IP pakety a záznamy auditu hostiteľa.

1.2.1 Jazyk STATL

State Transition Analysis Technique Language - STATL je rozšírený stavový jazyk na opis útokov, navrhnutý na podporu detekcie narušenia [3]. Jazyk umožňuje opísať prieniky do počítača, ako je napríklad postupnosť akcií vykonávaná útočníkmi, aby ohrozili počítačový systém. Opis útoku jazykom STATL môže byť použitý systémom zameraným na detekciu narušenia, aby analyzoval tok udalostí a odhalil možné prebiehajúce prieniky. Keďže detekcia narušenia sa vykonáva v rôznych doménach (t.j. sieť alebo host) a v rôznych operačných prostrediach (napr. Solaris, Linux alebo Windows), je dôležité mať jazyk, ktorý je ľahké rozšíriť a prispôsobiť rôznym cieľovým prostrediam. Jazyk STATL definuje vlastnosti útokových scenárov nezávisle od domény a poskytuje možnosti pre rozšírenie

jazyka na popis útokov v konkrétnych doménach a prostrediach. Jazyk STATL bol úspešne použitý pri popise sieťových aj hostiteľských útokov a bol prispôsobený odlišným prostrediam, ako napríklad už spomínaný Solaris, Linux či Windows. Pre jazyk STATL bola vyvinutá implementácia runtime podpory a implementovaná sada nástrojov pre detekciu narušení [3].

Pôvodný model STAT reprezentoval útoky pomocou neformálneho jazyka, založeného na stavových diagramoch, kde stavy reprezentovali vlastnosti, zdroje bezpečnosti systému a prechody popisovali kritické akcie útoku. Takouto akciou je napríklad otvorenie TCP pripojenia alebo spustenie aplikácie [3].

Jazyk STATL bol navrhnutý tak, aby spĺňal požiadavky na jazyk popisujúci útok, tak ako bolo spomenuté v časti 1.2. Jazyk je jednoduchý a explicitne zameraný na reprezentovanie útokových scenárov. Výsledný jazyk obsahuje niekoľko dobre definovaných abstrakcií (scenár, stav, prechod, akcia), ktoré sa používajú na špecifikáciu útokových signatúr. [3]

1.2.2 Jazyk ASTD

Algebraic State-Transition Diagrams (ASTD) je nadstavba obvyklých stavových diagramov a automatov [4]. Najskôr bol predstavený pre špecifikáciu, grafickú reprezentáciu a doklad o informačných systémoch. Základné ASTD je automat, ktorý je možné ľahko spájať s algebraickými operátormi, ako je sekvencia, Kleeneho uzáver, výber, podmienka, paralelná kompozícia so synchronizáciou, kvantifikovaný výber a kvantifikovaná paralelná kompozícia. Podobne ako pri stavových diagramoch, môžu stavy automatov sami byť komplexnými ASTD [4].

V snahe použiť ASTD na špecifikáciu detekcie kybernetických útokov, musí byť identifikovaných množstvo chýbajúcich funkcií v systéme. Je potrebné tento jazyk rozšíriť o podporu deklarovania atribútov (napr. stavové premenné) a akcie, ktoré môžu modifikovať tieto atribúty, keď sa vykoná prechod [4]. Atribúty môžu byť deklarované lokálne v rámci každého ASTD. Akcie môžu byť vykonané na prechodoch, ale tiež na úrovni samotného ASTD. Pri využití ASTD na detekciu útokov sa rozlišuje syntax ASTD a jeho stav. Za stav môžeme označovať aj množinu stavov ASTD. Každý typ ASTD vedie k podtypu stavov. Medzi základné podtypy ASTD môžeme zaradiť: elem (jednoduchý stav), automat, sekvencia, synchronizácia a nový operátor flow [4].

Práca [4] taktiež rozširuje ASTD zápis o stavové premenné, akcie na prechodoch a už vyššie spomínaný flow. Potrebné rozšírenia pre ASTD a stavové podtypy rozoberieme v časti 4.

1.3 Nástroje využívajúce stavové jazyky

V našej bakalárskej práci sa budeme tiež zaoberať nástrojmi, ktoré využívajú spomínané stavové jazyky. Ako prvý spomenieme nástroj Zeek, ktorý využíva jazyk STATL [20] a následne nástroj iASTD, ktorý využíva jazyk ASTD. V časti 5 sa budeme zaoberať porovnávaním aj spomínaných dvoch nástrojov, ktoré využívajú stavové jazyky STATL a ASTD, preto je dôležité si tieto nástroje rozobrať.

1.3.1 Zeek

Zeek je pasívny open-source analyzátor sieťovej prevádzky [5, https://zeek.org/]. Často je používaný na monitorovanie sieťovej bezpečnosti na podporu skúmania podozrivých alebo škodlivých aktivít [7].

Písanie Zeek skriptov je v podstate programovanie pomocou funkcií a globálnych premenných. Stavový a doménovo nezávislý jazyk STATL umožňuje abstraktnejšiu reprezentáciu scenárov útoku pomocou stavových automatov s akciami a stavovými premennými. Zeek je riadený pomocou event-driven skriptovacieho jazyka, pomocou ktorého presne špecifikuje a identifikuje útoky [20]. Event-driven programovanie je model programovania, v ktorom je priebeh programu určený udalosťami. Napríklad akcia vykonaná používateľom, akou je napríklad kliknutie myšou, stlačenie klávesu alebo správa operačného systému, alebo iného programu. Event-driven aplikácia je navrhnutá tak, aby keď sa vyskytnú udalosti, vedela správne vyhodnotiť danú udalosť pomocou vhodnej procedúry spracovania udalostí. [20] Viac informácií o stavových jazykoch môžeme nájsť na oficiálnej stránke [7].

1.3.2 iASTD

Počiatočný vývoj ASTD interpreta bol navrhnutý pre informačné systémy. Bol rozšírený o podporu nových funkcií pre detekciu kybernetických útokov. Tento interpreter implementuje operačnú sémantiku ASTD výpočtom prechodových dôkazov. Spracuje špecifikácie vstupného útoku na vstupné zdroje. Špecifikácie útoku sa pred odoslaním

interpretovi skonvertujú do serializovaného formátu. Počas vykonávania sú nespracované udalosti z hosta a/alebo siete spracované do enumeračnej formy. Interpreter číta vopred spracované udalosti v offline režime alebo v reálnom čase a vypočítava možné prechody. Akcie ako sú upozornenia sa zobrazia správcovi. ASTD má svoju interpretáciu založenú na jazyku OCaml a taktiež môže byť ASTD preložený do programovacieho jazyka B pre formálnu analýzu a kontrolu, a teda môže byť pretvorený na model B automatu v štýle Event-B modelovacej metódy. Tým pádom je možné využiť kontrolovanie modelov pomocou ProB. [4]

OCaml je jednoduchý, všeobecný programovací jazyk [8, https://ocaml.org/] s dôrazom na bezpečnosť a citovosť. Je vyvíjaný už viac ako 20 rokov skupinou významných vedcov, je podporovaný aktívnou komunitou a má bohatú sadu knižníc a vývojových nástrojov. Medzi najsilnejšie stránky jazyka OCaml patria [8]:

- Výkonný typový systém obsahujúci parametricky polymorfizmus a odvodzovanie typu. Napríklad typ kolekcie môžeme parametrizovať podľa typu jej prvkov. Takýto prístup nám umožňuje niekoľko operácií nad kolekciou a to nezávisle na type jej prvkov. Príkladom je napríklad triedenie poľa. Ďalej interferenciou typu môžeme definovať takéto operácie a to bez toho, aby sme potrebovali typ ich parametrov a výsledku.
- Užívateľom definované algebraické dátové typy a porovnávanie vzorov. Kombináciou záznamov a súčtov je možné definovať nove algebraické dátové typy a metódy, ktoré pracujú nad takýmito dátovými štruktúrami je možné definovať za pomoci porovnávania vzorov, čo je zovšeobecnená forma známeho príkazu switch a nástroj, ktorý ponúka prirodzený spôsob skúmania a pomenovania dát.
- Automatické spravovanie pamäte vďaka rýchlemu inkrementálnemu zberaču nežiaducich dát.
- Samostatná kompilácia nezávislých aplikácií. Prenosné kompilátory bitecodu umožňujú vytvárať samostatné aplikácie z programov Caml Light alebo OCaml.

ProB je animátor a kontrolér modelov pre B-method (vývojový software založený na jazyku B [10]) [9]. Umožňuje animáciu mnohých B špecifikácií a môže sa použiť na systematickú kontrolu chybovosti špecifikácie. Schopnosť ProB riešiť obmedzenia, možno

použiť aj na vyhľadávanie modelov, kontrolu uviaznutia a generovanie testovacích prípadov. ProB pokrýva veľkú časť jazyka B a snaží sa o úplne pokrytie konštrukcií nástroja Atelier B a B4Free. ProB podporuje funkcie jazyka B, ako sú nedeterministické operácie, akékoľvek príkazy, operácie s komplexnými argumentami, množinami, sekvenciami, funkciami, lambda abstrakcie, chápania množín, záznamy, konštanty, vlastnosti a mnohé ďalšie. ProB možno použiť aj na automatizovanú kontrolu spresnenia a kontrolu modelu LTL (linear temporal logic). Stavový priestor špecifikácií je možné graficky znázorniť. [9]

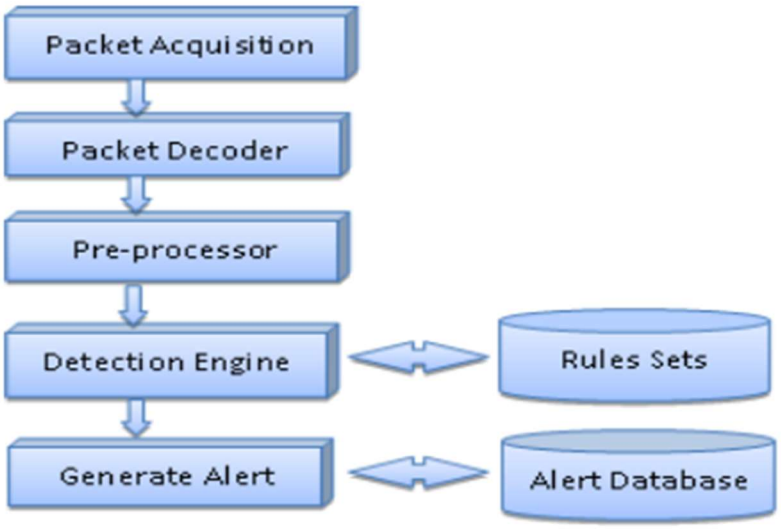
1.4 Nástroje nevyužívajúce stavové jazyky

V časti 5, tak ako aj Zeek a iASTD budeme porovnávať aj nástroj ktorý nevyužíva pre detekciu stavové jazyky a tým nástrojom je Snort. Preto je dôležité si definovať čo je Snort a akým spôsobom funguje na rozdiel od nástrojov založených na stavových jazykoch.

1.4.1 Snort

Snort je open-source systém pre odhaľovanie prienikov [11, https://www.snort.org/]. Systém používa skupinu pravidiel, ktoré lepšie definujú škodlivú sieťovú aktivitu a používa tieto pravidlá na nájdenie paketov, ktoré sa s nimi zhodujú a generuje upozornenia pre používateľov. Nevýhodou mnohých prístupov založených na detekcií na základe signatúr je neschopnosť odhaľovať nové útoky [11].

Architektúra Snortu pozostáva zo štyroch komponentov [12]: sniffovanie paketov, predprocesor, detekčný mechanizmus a generovanie výstupov. Snort sa pri získavaní paketov sústredí hlavne na knižnice libcap (pre UNIX/Linux) a winpcap (pre Windows). Tieto knižnice sú špecifické platformy, ktoré sa používajú na získavanie dát prúdiacich na



Obrázok 2: Snort architektúra [12]

sieti. Získavanie paketov monitoruje čas, kedy paket prichádza a vypočíta jeho celkovú dĺžku a skontroluje typ prepojenia rozhrania, na ktorom bol paket zachytený. [12]

Snort generuje upozornenia podľa pravidiel, ktoré sú definované v konfiguračnom súbore. Snortovský pravidlový jazyk je flexibilný a tvorba nových pravidiel je relatívne jednoduchá. Pravidlá pomáhajú pri rozlišovaní medzi bežnými internetovými aktivitami a škodlivými aktivitami. [11]

```
log tcp !192.168.0/24 any -> 192.168.0.33 (msg: "mounted access" ; )
```

Obrázok 3: Príklad Snort pravidla [13]

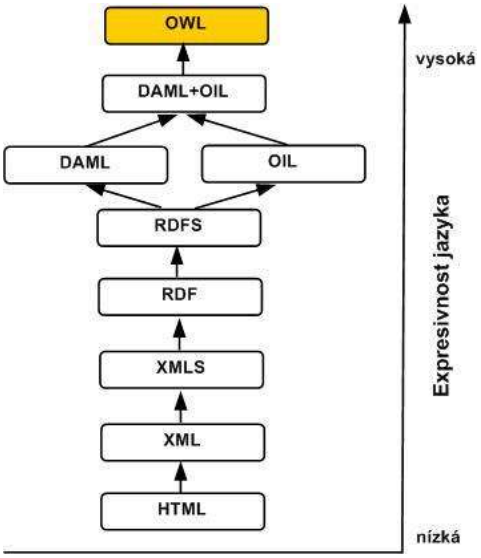
Z úvodnej teórie k stavovým jazykom, vieme povedať, že ich dôležitou vlastnosťou sú stavové diagramy, ktoré sa skladajú zo stavov, prechodov, udalosti, podmienok a akcii. Z práce [3] vyplýva, že jazyk pre opis útokov by mal spĺňať niekoľko podmienok, ktoré sme spomenuli a rozobrali v časti 1.2 a medzi jazyky, ktoré spĺňajú tieto podmienky môžeme zaradiť jazyk STATL a ASTD. STATL špecifikuje útoky pomocou scenárov útokov a je využitý v nástroji Zeek. Z práce [4] vyplýva, že rozšírením bežného jazyka ASTD o stavové premenné, akcie na prechodoch a nový operátor flow, je možné detailne popisovať špecifikácie útokov. Nástroj, ktorý využíva tento jazyk, je nástroj iASTD, ktorý je

interpretovaný programovacím jazykom OCaml. Taktiež sme rozobrali nástroj Snort, ktorý nevyužíva stavový jazyk, no bude spomínaný neskôr v časti 5, pri porovnávaní jednotlivých nástrojov. Keďže naša práca sa tiež zaoberá využitím ontologického jazyka OWL pre detekciu útokov, v ďalšej časti rozoberieme tento jazyk.

2 Web Ontology Language

Web Ontology Language (OWL) je sémantický webový jazyk navrhnutý tak, aby reprezentoval bohaté a komplexné znalosti o veciach, skupinách vecí a vzťahoch medzi vecami. Je navrhnutí na použitie v aplikáciách, ktoré potrebujú spracovať obsah informácií namiesto toho, aby informácie len ďalej prezentovali ľuďom [14]. Bol vytvorený v roku 2004 skupinou Web Ontology Working Group [14] a v roku 2012 vyšla verzia OWL 2 [15]. OWL je jazyk založený na výpočtovej logike, takže znalosti, ktoré sú vyjadrené pomocou OWL, môžu byť využité počítačom. OWL uľahčuje väčšiu strojovú interpretovateľnosť webového obsahu než tá, ktorú podporujú XML, RDF a RDF Schema, vďaka poskytnutím ďalšej slovnej zásoby spolu s formálnou sémantikou. OWL ma tri čoraz výraznejšie podjazyky: OWL Lite, OWL DL a OWL Full. OWL je súčasťou technologického zásobníka sémantického webu W3C. [14]

OWL je vyvinutý z kombinácie jazykov DAML a OIL, ktoré tvorili jazyk DAML+OIL, ktorý sa dá považovať za štartovací bod pre reprezentáciu znalostí v prostredí sémantického webu. Cieľom vytvorenia OWL bola potreba jazyka, ktorý by bol bohatší na sémantiku než spomínaný jazyk DAML+OIL a pritom by bol kompatibilný s jazykmi XML a RDF a taktiež by bol schopný odvodzovať nové skutočnosti z už existujúcich. Výsledkom bol jazyk OWL pre tvorbu webovej ontológie, pričom OWL je dokonalejším, čo sa týka obsahu jazyka, než už spomínané jazyky XML, RDF a RDF Schema [14]. Na obrázku č. 4 môžeme vidieť zoradenie jazykov podľa ich expresívnosti.



Obrázok 4: História OWL [5]

Ako sme spomenuli, pre tvorbu ontológie v jazyku OWL je možné využiť jednu z troch verzií [14]:

- OWL-Lite - je najjednoduchšia verzia OWL z pohľadu syntaxe, ktorú je vhodné použiť, ak chceme vytvárať jednoduchšie štruktúry ontológie. Poskytuje nižšie úrovne zložitosti, nepoužíva určité elementy, ktoré obsahujú vyššie verzie OWL, napríklad zjednotenie (owl:unionOf), doplnok (owl:complementOf) alebo nepoužíva disjunktnosť tried (disjointWith).
- OWL-DL je zložitejšia ako OWL-Lite a založená na deskripčnej logike (Description Logic). Pomocou OWL-DL vieme realizovať odvodzovanie, tvoriť komplexnejšie popisy a definície tried. Obsahuje všetky konštrukcie OWL Full, ale ich použitie je obmedzené.
- OWL-Full nemá žiadne obmedzenia a obsahuje všetky konštrukcie a elementy jazyka, no odvodzovanie je zložitejšie.

Medzi hlavné komponenty OWL ontológie patria [14]:

- Trieda (Class) - Pojem trieda v jazyku OWL nie je odlišná od objektového orientovaného princípu a teda sa jedná súbor jedincov s rovnakými vlastnosťami. Sú hierarchicky organizované, pričom tieto hierarchie sa nazývajú taxonómie (napr.: nadtrieda a podtrieda). Tvorenie tried dodáva ontológií štruktúru. V OWL existujú triedy, ktoré sú buď preddefinované alebo nami vytvorené. Jednou z preddefinovaných je trieda **owl:Thing**, čo je najobecnejšia trieda, do ktorej patria všetky ostatné triedy. Opak triedy owl:Thing je trieda **owl:Nothing**.

- Jedinec - Jedinec je reprezentáciou objektu domény, ktorá patrí určitej triede. Jedinec môže patriť naraz do viacerých tried. V OWL musí byť explicitne povedané, určitý jedinci sú si navzájom rôzni alebo rovnakí.

- Vlastnosť - vytvárajú vzťah medzi jedincam, a teda spájajú k sebe dva objekty. Vlastnosti môžu byť: objektové, datatypové a anotačné. Objektové vlastnosti vytvárajú spojenie medzi triedami, datatypové spájajú jedinca

s hodnotou, ktorá je určitého dátového typu, a anotačné pridávajú triedam a jedincom ďalšie informácie.

2.1 Deskripčná logika

OWL Lite a OWL DL využíva formálny jazyk nazývaný Deskripčná logika (DL) [12]. Základným prvkom DL je axióma, teda logické vyhlásenie určitých rolí alebo konceptov. V DL môžeme rozlišovať takzvane TBox a ABox. TBox obsahujú vety, ktoré popisujú vzťahy medzi pojmami, zatiaľ čo ABox obsahuje vety, ktoré hovoria, kam v hierarchií jednotlivci patria. [14]

V nasledujúcej časti práce analyzujeme jednotlivé časti stavových jazykov STATL a ASTD, ktoré sú potrebné pre detekciu útokov. Taktiež poukážeme na ontológie vytvorené pomocou OWL, ktoré vychádzajú z analýzy predošlých stavových jazykov, a mali by sme byť schopní vďaka ním dostatočne špecifikovať útoky.

3 Analýza stavových jazykov

Pri analýze sme sa zamerali na jednotlivé časti daných jazykov a hľadali sme, ako dané jazyky popisujú útoky a čo pri tom využívajú. Ako prvý sme analyzovali jazyk STATL. STATL popisuje útoky pomocou tzv. scenárov útoku, pričom sme rozobrali túto vlastnosť, ako aj spôsob ako znázorňuje stavy a prechody. Pri ASTD sme sa zamerali na 3 dôležité operátory a to automat, sekvencia a flow. Pre OWL sme našli spôsob, ako špecifikovať ontológiu automatu, pričom sme vychádzali z jazyku ASTD, a ontológiu sekvencie.

3.1 STATL

Špecifikácia jazyka STATL je popisovaná kompletnými scenármi útoku. Tieto útoky sú modelované ako sekvencie krokov, ktoré opisujú presun systému z bezpečného stavu do stavu ohrozenia. STATL ako stavový jazyk využíva stavy a prechody pre opis útokových scenárov, ktoré je možné zobraziť pomocou diagramov [3].

Stavy sa využívajú na opis rôznych snímok systému počas vývoja útoku, pričom STATL opisuje len tie časti stavu systému, ktoré sú potrebné na definovanie útoku. Prechod má priradenú akciu, ktorá špecifikuje udalosť, ktorá môže preniesť scenár útoku do ďalšieho stavu [3].

3.1.1 Scenár útoku

Scenár útoku môže používať knižnice, špecifické pre danú aplikáciu, a teda môže využívať jej udalosti, funkcie a typy. Scenár ďalej pozostáva z názvu, môže obsahovať parametre, anotácie, deklarované premenné a dôležité je, že obsahujú stavy a prechody, ktoré definujú útok [3]. Blok kódu môže vyzerať nasledovne:

```
Scenario ::=
    { use LibraryID {',' LibraryID} ';' }
    scenario ScenarioID
    [ScenarioParameters]
    '{'
        [FrontMatter]
        {State | Transition | NamedAction}
    '}'
    { FunctionDefinition }
```

Obrázok 5: Scenár útoku v STATL [3]

Scenár musí obsahovať aspoň jeden prechod a dva stavy – počiatočný a konečný stav. Parametre sú špecifikované ako zoznam identifikátorov oddelených čiarkami [3].

3.1.2 Stav

Stav je jedným zo základných pojmov v STATL. Stavy sú označené názvom, aby sa na ne dalo odkázať v prechodoch a pri grafickom znázornení scenára pomocou diagramu. Každý stav ma voliteľné prvky ako anotácia, tvrdenie alebo blok kódu.[3]

State ::=
[initial]
state *StateId* {*Annotation*}
'{'
[*StateAssertion*]
[*CodeBlock*]
'}'

Obrázok 6: Stav v STATL [3]

Ak je pri stave prítomné tvrdenie (*StateAssertion*), je testované ešte pred vstupom do stavu po tom, ako je otestované tvrdenie pri prechode, ktorý vstupuje do tohto stavu. Blok kódu (*CodeBlock*) je vykonaný po tom, čo je otestované tvrdenie pri prechode a pri vstupe do stavu, pričom tieto tvrdenia musia byť splnené [3].

3.1.3 Prechod

Prechod je ďalším základným pojmov v STATL. Každý prechod musí mať názov a pár stavov, ktoré spája. Prechod môže mať ten istý počiatočný stav aj koncový stav. Ďalej môže obsahovať anotácie, typ prechodu, udalosť a blok kódu [3].

```
Transition ::=
    transition TransitionID '(' StateId '->' StateId ')'
    (consuming | nonconsuming | unwinding)
    {Annotation}
    '{'
        ( '[' EventSpec ']' | ActionId )
        {Annotation}
        [':' Assertion]
        [CodeBlock]
    '}'
```

Obrázok 7: Prechod v STATL [3]

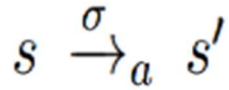
Typ prechodu môže byť konzumný, nekonzumný alebo odvíjací. Kód pri prechode sa vykoná po tom, čo je splnené jeho tvrdenie a tvrdenie pri stave, do ktorého smeruje, a pred vykonaním kódu tohto stavu. [3]

Všetky ostatné sčasti jazyka STATL môžeme nájsť opísané v práci [3].

3.2 ASTD

Syntax všetkých ASTD je definovaná použitím typovej hierarchie [4]. Každý ASTD operátor je reprezentovaný typom. Abstraktný typ ASTD je definovaný ako $ASTD = (n, P, V, A_{astd})$, kde n je názov $ASTD$, P je voliteľný zoznam parametrov, V je súbor atribútov, A_{astd} je akcia, ktorej predvolená hodnota je skip, čo znamená, že nerobí nič. Parametre P sa používajú na príjem hodnôt pri volaní ASTD, atribúty V sú stavové premenné, ktoré je možné upravovať akciami a testovať v podmienkach v rámci ASTD [4].

ASTD sémantika pozostáva z označovacieho prechodového systému – Labelled Transition Systém (LTS). LTS je podmnožina *Stav x Udalosť x Stav* a prechod je označený ako [4]:



Obrázok 8: Prechod ASTD [4]

To znamená, že ASTD ***a*** môže vykonať udalosť σ zo stavu ***S*** a presunúť sa do stavu ***S'.*** [4]

ASTD využíva niekoľko stavových podtypov. Detailnejšie sme rozobrali 3 z nich. Detailne popísané všetky stavové podtypy môžeme nájsť v práci [6].

3.2.1 Automat

Automat je základným ASTD, preto sme sa rozhodli detailne popísať jeho funkčnosť. Štruktúra ASTD automatu je (aut, Σ, S, ζ, ν, δ, SF, DF, n_0), kde platia nasledujúce obmedzenia. $\Sigma \subseteq$ *Udalosť* je udalosť, ktorá sa označuje písmenom z abecedy. $S \subseteq$ *Názov* je súbor názvov stavov, $\nu \in S \rightarrow$ *ASTD* mapuje názov každého stavu na jeho pod ASTD, ktorý môže byť elementárny (značený ako *elem*) alebo komplexný. Prechod v automate medzi stavmi n_1 a $n_2 \in S$ je označený ako $\sigma[g]/A_{tr}$ a je reprezentovaný ako (obr. 9) [4]:

$$((\mathsf{loc}, n_1, n_2), \sigma, g, A_{tr}, final?) \in \delta$$

Obrázok 9: Prechod v ASTD [4]

Symbol *final?* je typu Boolean. Ak *final? = pravda*, zdroj prechodu je označený guľôčkou, ktorá značí, že prechod môže byť vykonaný len ak n_1 je finálny stav. Toto označenie je užitočné len ak n_1 je komplexný stav. $SF \subseteq S$ je súbor plytkých finálnych stavov (shallow states) a $DF \subseteq S$ označuje súbor hlbokých finálnych stavov (deep states). Pri *SF* a *DF* platí, že $DF \cap SF = \emptyset$. $n_0 \in S$ je názov počiatočného stavu. Hlboký konečný stav (DF) je konečný len vtedy, keď je jeho pod ASTD tiež konečný stav, zatiaľ čo plytký konečný stav (SF) je konečný bez ohľadu na stav jeho pod ASTD [4]. Aby sme vysvetlili rozdiel medzi počiatočným a konečným stavom, musíme si zadefinovať stav.

Stav automatu nemôže byť reprezentovaný len názvom. Je to komplexná štruktúra typu $(aut_\circ, n, E, s)$. $aut_\circ$ je konštruktor stavu automatu, $n \in S$ je označenie pre aktuálny stav automatu a E obsahuje hodnoty atribútov automatu. Ak je *n* komplexný stav, *s* značí jeho stav. Ak je *n* elementárny stav, potom *s = elem* [4].

Počiatočný a koncový stav je teda označovaný nasledovne. *a* označíme ASTD automat [4].

$$init(a) \triangleq (\mathsf{aut}_\circ, a.n_0, a.E_{init}, init(a.\nu(n_0)))$$
$$final(a, (\mathsf{aut}_\circ, n, E, s)) \triangleq n \in a.SF \vee (n \in a.DF \wedge final(a.\nu(n), s))$$

Obrázok 10: Počiatočný a koncový stav v automate ASTD [4]

Symbol E_{init} označuje počiatočné hodnoty atribútov. *Init(a.v(n₀))* vracia počiatočný stav pod ASTD *v(n₀)* stavu n_0. Napríklad z automatu na obrázku č.10 vidíme, že *init(A.v(1) = elem* ako elementárny stav a *init(A) = (aut∘, 1, {(x,0)}, elem)* [4].

Pri automate existuje šesť pravidiel, ktoré definujú sémantiku automatu a vďaka ktorým vieme popísať rôzne typy prechodov a stavov [4]. V našej práci sme opísali dve najčastejšie používané pravidlá. Zvyšné pravidlá môžeme nájsť v práci [6].

Pravidlo *aut₁*, opisuje prechod medzi lokálnymi stavmi.

$$\mathsf{aut}_1 \ \frac{a.\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, final?) \quad \Psi \quad \Omega_{loc}}{(\mathsf{aut}_\circ, n_1, E, s) \xrightarrow{\sigma, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', init(a.\nu(n_2)))}$$

Obrázok 11: Pravidlo aut1 v ASTD [4]

Toto pravidlo hovorí o tom, že prechod pri udalosti *σ* sa môže vykonať z n_1 do n_2 s pred a po hodnotami atribútov *E, E'*. Stav pod ASTD n_2 je jeho počiatočný stav. Predpokladá sa, že takýto prechod je možný, ak existuje zodpovedajúci prechod v *δ*, ktorý je reprezentovaný ako *δ((loc, n1, n2), σ' , g, A_{tr}, final?)*. *σ'* je udalosť opisujúca prechod a môže obsahovať premenné. Hodnoty týchto premenných sú dané z prostredia E_e a pomocou lokálnych hodnôt *E*, ktoré môžu byť použité ako substitúcia[4]. To je na prechode zabezpečené predpokladom

$$\Psi \triangleq \big((final? \Rightarrow final(a, (\mathsf{aut}_\circ, n_1, E, s))) \wedge g \wedge \sigma' = \sigma\big)([\![E_g]\!])$$

Obrázok 12: Predpoklad Ψ na prechode [4]

Ψ definovaným nasledovne (obr. 12) [4]:

$$\Omega_{loc} \triangleq \left\{ \begin{array}{c} A = A_{tr} \ ; \ a.A_{astd} \\ E_g = E_e \lhd E \\ A(E_g, E'_g) \\ E'_e = E_e \lhd (V \lhd E'_g) \\ E' = V \lhd E'_g \end{array} \right\}$$

Obrázok 13: Predpoklad Ω_{loc} [4]

Ak je prechod konečný, potom aktuálny stav musí byť tiež konečný. Podmienka (guard) *g* drží prechod a udalosť σ sa rovná udalosti σ', ktorá označuje prechod automatu po aplikovaní prostredia E_g ako substitúcie. Prostredie E_g je definované za predpokladu Ω_{loc} [4].

Tento predpoklad môžeme chápať nasledovne. Vykonané akcie sú akcie na prechodoch A_{tr}, za ktorými nasleduje akcia $a.A_{astd}$, ktorá je deklarovaná v hlavičke automatu. ASTD akcia je užitočná na definovanie zmien stavu, ktoré je potrebné vykonať pri každom prechode ASTD. E_g opisuje globálny zoznam premenných deklarovaných v uzatvárajúcich ASTD (E_e) a premenných deklarovaných lokálne (E). Ak lokálna premenná nesie rovnaký názov ako premenná deklarovaná v uzatvárajúcom ASTD, je prepísaná. Hodnoty E_g' sú použité na deklarovanie E' (lokálne atribúty) použitím obmedzení na atribúty V deklarované v ASTD a hodnôt E_e' (atribúty deklarované v uzatvárajúcom ASTD) [4].

Pravidlo aut_6 zasa hovorí o prechodoch v pod ASTD stavu *n* [4].

$$\text{aut}_6 \ \frac{s \xrightarrow{\sigma, E_g, E''_g}_{a.\nu(n)} s' \qquad \Theta}{(\text{aut}_\circ, n, E, h, s) \xrightarrow{\sigma, E_e, E'_e}_{a} (\text{aut}_\circ, n, E', h, s')}$$

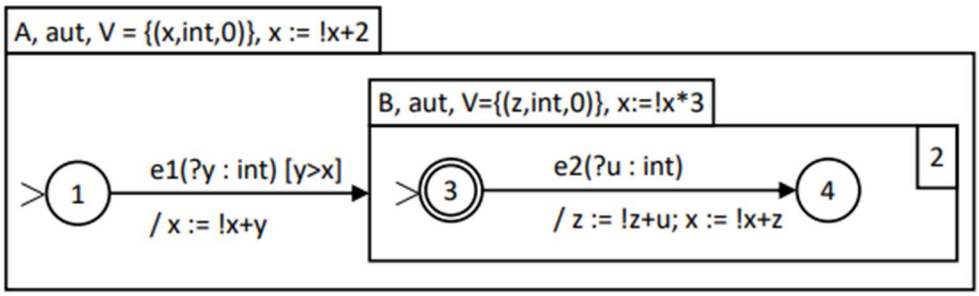
$$\Theta \triangleq \left(\begin{array}{cc} E_g = E_e \lhd E & a.A_{astd}(E''_g, E'_g) \\ E'_e = E_e \lhd (V \lhd E'_g) & E' = V \lhd E'_g \end{array} \right)$$

Obrázok 14: Pravidlo aut6 v ASTD [4]

Prechod začína z podstavu *s* a prechádza do podstavu *s'*, jeho rodičovského stavu *n*. Akcie sú vykonané od konca. E_g'' sú hodnoty vypočítané pomocou pod ASTD. Označenie Θ hovorí o vypočítaní E_g' z E_g'' pomocou vykonania ASTD akcie definovanej v hlavičke

rodičovského ASTD (A_{astd}). *E_e'* a *E'* sú určené rozdelením *E_g'* za pomoci atribútov V definovaných v hlavičke ASTD. Toto označenie Θ je použité v každom ASTD pravidle, kde sa rozoberajú prechody v pod ASTD [4].

Na obrázku č. 15 je znázornený automat A, ktorý deklaruje súbor atribútov *V={(x,int,0)},* ktorý obsahuje atribút *x* typu *int* a jeho počiatočná hodnota je 0. Automat A tiež deklaruje akciu *x:=!x+2*. Keďže jazyk ASTD je interpretovaný pomocou OCaml, akcia je vyjadrená v tomto jazyku a hovorí, že atribút x inkrementuje o 2. Stav 1 je elementárny stav a stav 2 je komplexný stav a to *automat B*. Prechod zo stavu 1 do stavu 2 je označený udalosťou *e1(?y : int),* ktorá deklaruje lokálnu premennú *y*, ktorej rozsah je len prechod. Taktiež tento prechod obsahuje podmienku *[y > x]* a akciu *x:=!x+y*. ASTD B, ktorý znázorňuje stav 2, deklaruje lokálnu premennú *z* a akciu na úrovni celého ASTD *x:=!x+3*. Jeho počiatočný stav je tiež aj konečný stav, definovaný funkciou *init(v(2)) = ($aut_\circ$, 3, {(z,0)}, elem)* [4].



Obrázok 15: ASTD automat s komplexným stavom 2 [4]

Automat A teda môže vykonať nasledujúce dva prechody [4]:

$$
\begin{aligned}
& (\mathrm{aut}_\circ, 1, \{(x,0)\}, \mathrm{elem}) \\
\xrightarrow{e1(1)}_A \quad & (\mathrm{aut}_\circ, 2, \{(x,3)\}, (\mathrm{aut}_\circ, 3, \{(z,0)\}, \mathrm{elem})) \\
\xrightarrow{e2(1)}_A \quad & (\mathrm{aut}_\circ, 2, \{(x,14)\}, (\mathrm{aut}_\circ, 4, \{(z,1)\}, \mathrm{elem}))
\end{aligned}
$$

Obrázok 16: Prechody automatu [4]

Daný automat teda funguje tak, že pri prechode zo stavu 1 do stavu 2 sa vykoná udalosť *e1* a teda za *y* sa dosadí číslo 1. Podmienka sa tým pádom splní pretože za *x* sa pri podmienke dosadí hodnota 0, vykoná sa akcia na prechode a teda *x* sa zvýši o hodnotu *y*. Po

tejto akcii bude mať *x* hodnotu 1 a prechod bude vykonaný, tým pádom sa vykoná udalosť definovaná v hlavičke ASTD, v tomto prípade sa x inkrementuje o 2 a teda nová hodnota *x* bude 3 a automat vojde do komplexného stavu 2, čo je automat B, a do jeho stavu 3. Zo stavu 3 sa vykoná prechod do stavu 4 tak, že sa udalosť *e2* dosadí do premennej *u* hodnotu 1. V tomto prípade podmienka nie je definovaná, takže sa hneď vykoná akcia na prechode *z:=!z+u; x:=!x+z*. Po tejto akcií bude hodnota *z=1* a *x=4*. Automat prejde do stavu 4 a vykoná sa akcia definovaná v hlavičke automatu B, a *x* bude mať hodnotu 12. Následne sa vykoná aj akcia definovaná v automate *A*, keďže automat *B* je jeho pod ASTD, a x bude mať po akcií *x:=!x+2* hodnotu 14 [4].

3.2.2 Sekvencia

Sekvencia v ASTD umožňuje sekvenčnú kompozíciu dvoch ASTD. Keď prvá zložka príde do konečného stavu, druhá sa môže vykonať. Toto umožní uložiť problémy do súboru úloh, ktoré sa musia vykonávať postupne [4].

Štruktúra sekvencie v ASTD je *(⇨, fst, snd)*, pričom *fst*, *snd* sú ASTD označujúce v poradí prvé a druhé ASTD. Stav sekvencie je typu *(⇨∘, E, [fst | snd], s), kde* ⇨∘ označuje konštruktor sekvencie, *E* označuje hodnoty atribútov deklarovaných v tejto sekvencii, označenie *[fst | snd]* je výber medzi dvoma označeniami, ktoré indikujú, či je sekvencia v prvom pod ASTD alebo v druhom, a $s \in Stav$. Počiatočný a koncový stav je teda označovaný nasledovne. *a* označíme ASTD sekvenciu [4].

$$
\begin{aligned}
init(a) &\triangleq (\diamond\!\!\rightarrow_{\circ}, a.E_{init}, \mathsf{fst}, init(a.fst)) \\
final(a, (\diamond\!\!\rightarrow_{\circ}, E, \mathsf{fst}, s)) &\triangleq final(a.fst, s) \quad \wedge \\
&\quad\ final(a.snd, init(a.snd)) \\
final(a, (\diamond\!\!\rightarrow_{\circ}, E, \mathsf{snd}, s)) &\triangleq final(a.snd, s)
\end{aligned}
$$

Obrázok 17: Počiatočný a koncový stav sekvencie ASTD [4]

Počiatočný stav sekvencie je vlastne počiatočný stav jeho prvého pod ASTD. Za koncový môžeme považovať dva prípady. V prvom prípade je označený konečný stav prvého pod ASTD, ktorý je vo svojom konečnom stave a zároveň jeho druhý pod ASTD je tiež v konečnom stave. V druhom prípade je označený konečný stav druhého pod ASTD, ktorý musí byť vo svojom konečnom stave [4].

Sekvencia ASTD používa tri pravidlá, ktoré sú potrebné pre vykonanie sekvencie [4]. Prvé pravidlo hovorí o prechodoch iba v prvom pod ASTD (*fst*). Druhé pravidlo hovorí o prechodoch medzi prvým (*fst*) a druhým (*snd*) pod ASTD. Tretie pravidlo hovorí o prechodoch v rámci druhého pod ASTD (*snd*). [4]

$$\Rightarrow_1 \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.fst} s' \qquad \Theta}{(\Rightarrow_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\Rightarrow_\circ, E', \mathsf{fst}, s')}$$

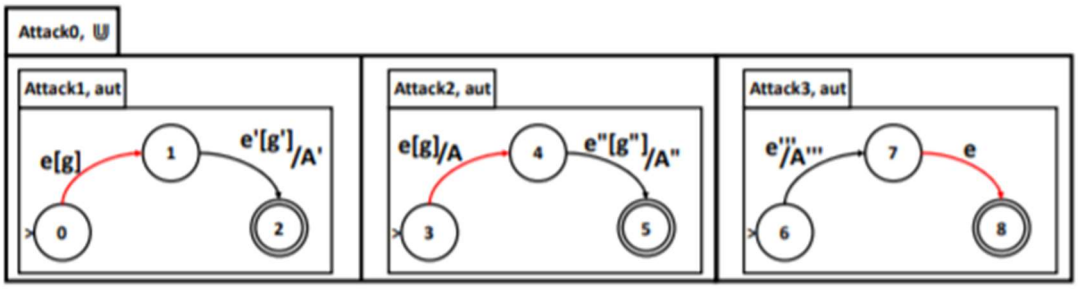
$$\Rightarrow_2 \frac{final(a.fst, s)([E_g]) \quad init(a.snd) \xrightarrow{\sigma, E_g, E_g''}_{a.snd} s' \quad \Theta}{(\Rightarrow_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\Rightarrow_\circ, E', \mathsf{snd}, s')}$$

$$\Rightarrow_3 \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.snd} s' \qquad \Theta}{(\Rightarrow_\circ, E, \mathsf{snd}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\Rightarrow_\circ, E', \mathsf{snd}, s')}$$

Obrázok 18: Pravidlá pri sekvenciách ASTD [4]

3.2.3 Flow

Je celkom bežné, že jedna udalosť môže byť súčasťou niekoľkých kybernetických útokov. Systém, ktorý detekuje narušenia, vykonáva túto udalosť pri každej špecifikácii útoku, ktorá ho môže vykonať. Preto je potrebné zadefinovať nový operátor nazývaný flow, ktorý kombinuje viacero ASTD a na každom vykoná udalosť z tých, ktoré sú možné vykonať. Ide teda o formu slabej synchronizácie. Toto rozšírenie je potrebné pri modelovaní kybernetických útokov. [4]



Obrázok 19: Diagram využívajúci operátor flow [4]

Na obrázku č. 19 možno vidieť, že Attack0 kombinuje viaceré útoky použitím operátora flow: ATTACK1, ATTACK2, ATTACK3. Flow operátor skontroluje všetky

možné prechody a vykoná ich. Ak je dosiahnutá udalosť *e*, prechody 0-1, 3-4, 7-8 sú spustené [4].

Flow ASTD má štruktúru *(⩒, l, r)*. Stav flow-u je typu *(⩒∘, E, S_l, S_r)*, kde S_l a S_r sú stavy ľavého a pravého pod ASTD. Počiatočný a konečný stav sú definované nasledovne. *a* označíme ASTD flow [4].

$$init(a) \triangleq (⩒_\circ, a.E_{init}, init(a.l), init(a.r))$$
$$final(a, (⩒_\circ, E, s_l, s_r)) \triangleq final(a.l, s_l) \wedge final(a.r, s_r)$$

Obrázok 20: Počiatočný a koncový stav operátora flow ASTD [4]

3.3 OWL

OWL dokáže byť vhodným jazykom pre reprezentáciu modelov automatov [18]. Pre správnu reprezentáciu je nutné niektoré časti automatu opísať inou ontológiou. Automat môže využívať pri svojich prechodoch podmienku (guard), pričom táto podmienka je opísaná pomocou ontológie výrazu (expression) [18]. Pri akciách, ktoré sa vykonávajú v stavových automatoch, môže byť správanie tejto akcie opísane pomocou kontrolnej ontológie [18].

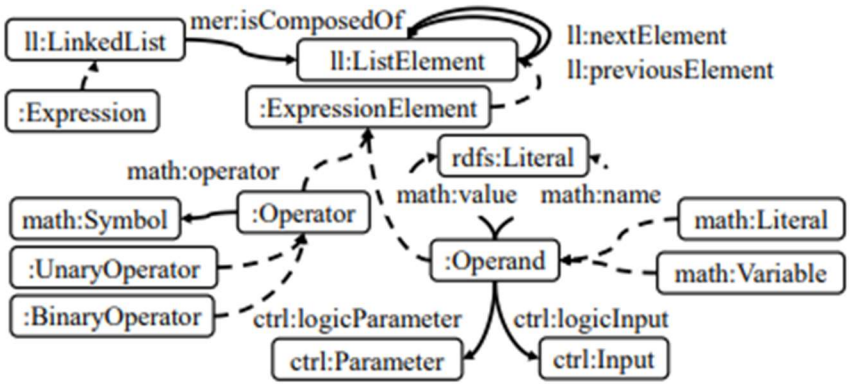
3.3.1 Ontológia podmienky

Pri podmienke je základným komponentom matematický výraz [18]. Napríklad pri stavových automatoch, podmienka opisuje, kedy má daný automat prejsť z jedného stavu do druhého.

Výraz pri tejto podmienke je modelovaný ako zoznam elementov, kde vzťah medzi dvoma elementmi umožňuje vyjadriť ich roztriedenie. Napríklad pri výraze $x < 1$, prvým elementom je premenná *x* [18].

Elementy môžu byť rozdelené na operátory a operandy. [18] Operátor predstavuje matematický symbol, ktorý špecifikuje, čo sa stane s operandmi, ktoré sú k nemu pridelené. Operand je na druhú stranu hodnota alebo premenná. Symbol operátora je vyjadrený pomocou vzťahu *operator*. Operátor môže byť buď binárny alebo unárny, pričom binárny môže byť aplikovaný iba medzi dva operandy a unárny operátor je naviazaný iba na jeden operand. Binárny operátor môže byť napríklad > a < a za unárny operátor môžeme pokladať

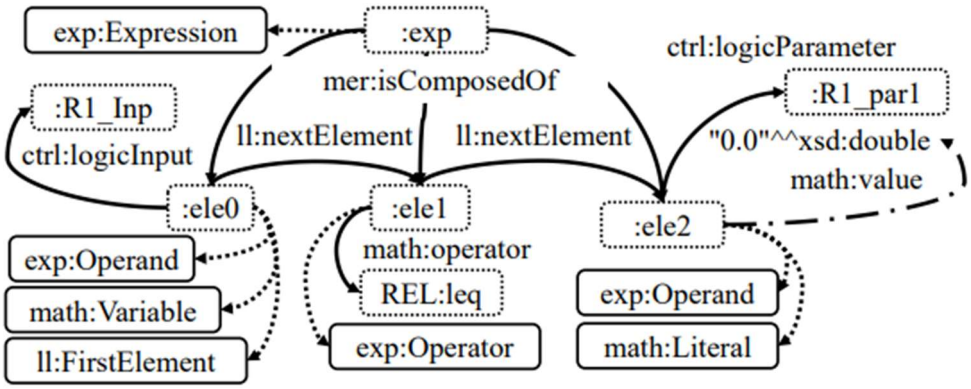
napríklad „!“, čím vieme negovať boolovské výrazy. Operand môže byť buď premenná alebo presná hodnota. [17]



Obrázok 21: Všeobecná ontológia podmienky [18]

Na obrázku č. 22 je znázornená ontológia pre jednoduchý výraz „R1_Inp <= 0.0“. Výraz je v tomto prípade rozdelený na tri elementy: ele0, ele1 a ele2. Tieto elementy sú zoradené sekvenčne za sebou, pomocou vzťahu nextElement. Ontológia začína elementom

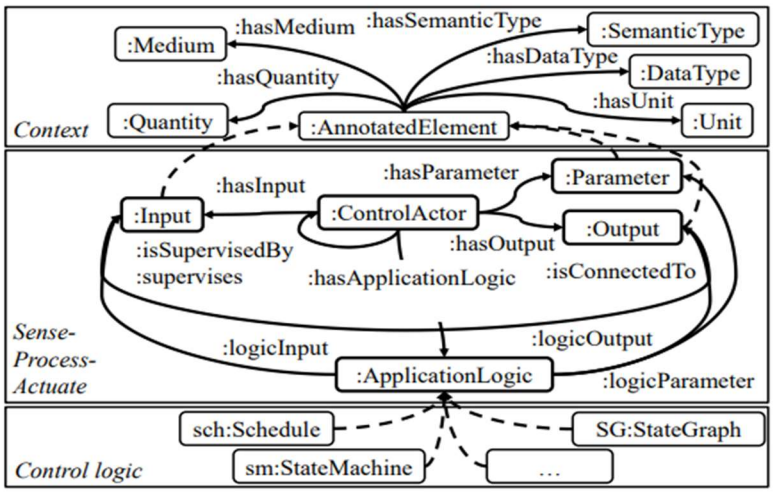
ele0, ktorý je modelovaný ako operand, s pridelenou premennou R1_Inp. Za ním nasleduje ele1, ktorý je namodelovaný ako operátor, ktorý má pridelený symbol „<=“. Posledný je element ele3, ktorý je operand a má priradenú presnú hodnotu 0.0 [18].



Obrázok 22: Príklad ontológie podmienky [18]

3.3.2 Ontológia kontroly

Ontológia kontroly je potrebná pri automatických systémoch pre kontrolu stavu pomocou jeho vstupov [18]. Tieto vstupy sú generované logickým systémom, napríklad stavovým automatom. Výsledkom sú výstupy, ktoré môžu byť použité buď v ďalšej kontrole, alebo môžu byť priradené do systému, ktorý je pod kontrolou [18]. Táto kontrolná jednotka obsahuje vstupy (inputs), parametre (parameter) a výstupy (outputs), ktoré sú priradené vzťahom hasInput, hasPrameter a hasOutput. Tieto dáta môžu byť ďalej rozšírené o doplňujúce údaje, ako typ jednotky alebo hodnotu [18].

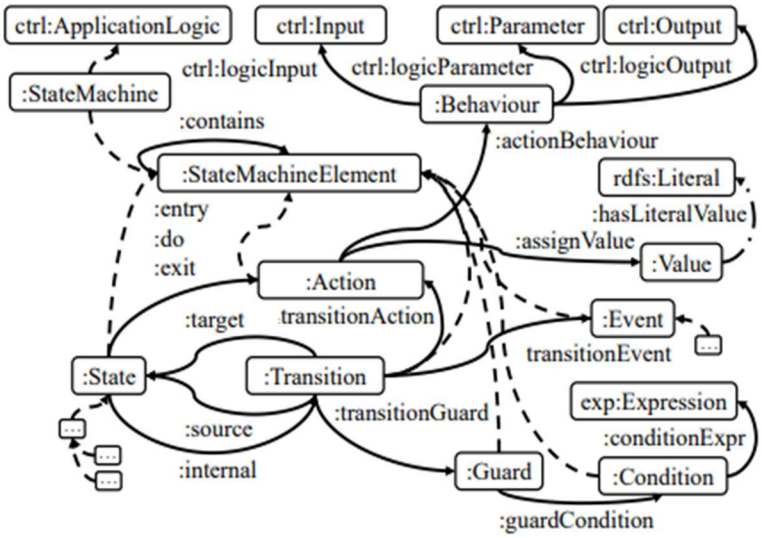


Obrázok 23: Všeobecná ontológia kontroly [18]

3.3.3 Stavový automat

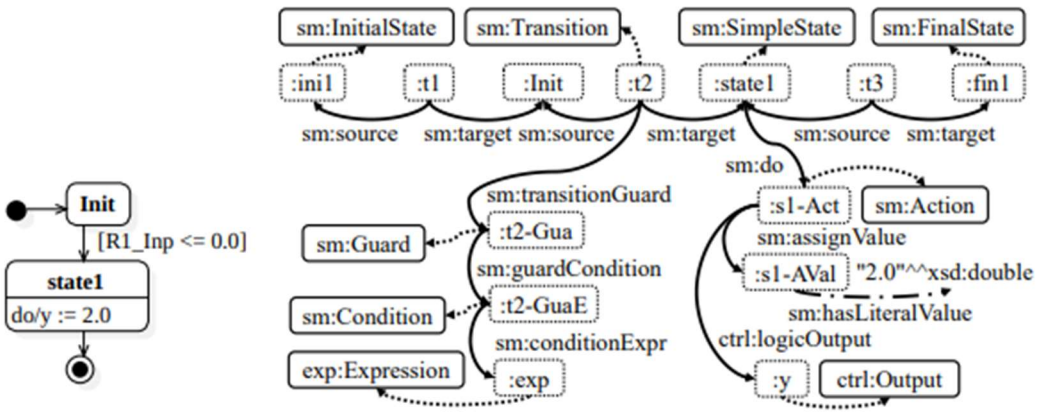
Základom stavového automatu je zreťazenie stavov a prechodov [18]. Prechod má jeden počiatočný stav (source) a jeden cieľový stav (target). Na rozdiel od toho, stav môže mať jeden alebo viac prechodov [18]. Akcia pri stavoch a prechodoch je jasne opísaná pomocou troch označení, a to entry, do a exit, pričom tieto označenia označujú stav, ktorý sa má vykonať pri vstupe do stavu, vo vnútri stavu a pri vychádzaní zo stavu [18]. Správanie akcie môže byť opísané algoritmom alebo iným stavovým automatom. Ak je toto správanie opísané pomocou algoritmu, tento algoritmus je ďalej opísaný pomocou kontrolnej ontológie (CTRLont). Pri jednoduchých správaniach, kde je priradená nejaká hodnota ak je stav aktívny, akcia môže mať hodnotu priradenú pomocou vzťahu priraďujúceho hodnotu (assignValue), ktorý hneď definuje hodnotu pomocou atribútu hasLiteralValue [18]. Prechod je spustený pomocou udalosti a môže byť kontrolovaný pomocou podmienky

(guard). Táto podmienka môže byť explicitne špecifikovaná pomocou ontológie výrazu (Expression) a pridelená k podmienke pomocou vzťahu ConditionExpr. Prechod, jeho podmienka a jej špecifikácia môžu byť medzi sebou prepojené pomocou vzťahov transitionGuard a guardCondition [18].



Obrázok 24: Všeobecná ontológia stavového jazyka [18]

Na obrázku č. 24 je znázornená ontológia jednoduchého stavového automat, opísaného pomocou UML diagramu. Správanie stavu *state1* je spustené po tom, čo je vytvorený prechod do stavu Init a následne je splnená podmienka R1_Inp <= 0.0, ktorá ak ma hodnotu true, prechod prejde do stavu state1. *state1* má iba *do* akciu, ktorá na výstup premennej y priradí číslo 2.0. Následne automat prejde do konečného stavu. [18]

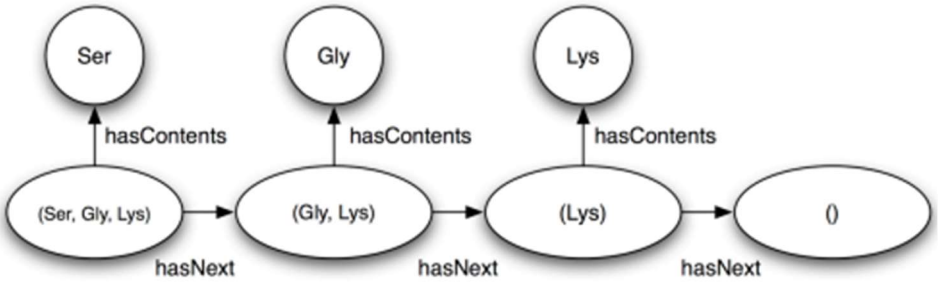


Obrázok 25: Príklad ontológie stavového automatu [18]

3.3.4 Sekvencia

Jazyk OWL nemá vbudovanú špecifikáciu, ktorá podporuje sekvenčnosť, no disponuje prostriedkami, ktoré sú schopné túto vlastnosť dosiahnuť [19].

Pri vytváraní dátovej štruktúry sekvencie, postupujeme podľa štandardného vzoru spájania zoznamov, v ktorom každý prvok je držaný v bunke (OWLList). Každá bunka obsahuje hlavičku (head), smerník do ďalšej bunky (tail) a koniec zoznamu, ktorý je indikovaný pomocou prázdneho zoznamu (EmptyList). V OWL definujeme, že každá bunka má práve jeden obsahujúci prvok a práve jednu nasledujúcu bunku (zoznam). Toto môžeme docieliť vzťahmi hasContents a hasNext. Inými slovami povedané, prvky každého zoznamu, sú obsah prvej bunky plus obsah každej nasledujúcej bunky. [19]



Simplified Syntax	OWL Abstract Syntax
A AND B	intersectionOf(A B)

```
OWLList AND
       hasContents SOME Ser AND
       hasNext SOME (
            OWLList AND
                hasContents SOME Gly AND
                hasNext SOME (
                    OWLList AND
                        hasContents SOME Arg AND
                        hasNext SOME EmptyList))
```

Obrázok 26: Ontológia sekvencie [19]

V tejto časti sme podrobne analyzovali jednotlivé časti jazykov STATL, ASTD a OWL. Pri jazyku OWL sme vychádzali hlavne z jazyka ASTD, preto sme sa snažili nájsť spôsob zapísania ontológie automatu alebo sekvencie. Zistili sme, že pre správne definovanie špecifikácií, je potrebné definovať pomocné ontológie pre kontrolu a pre podmienky. Výsledky našej práce spomenieme v nasledujúcej časti, kde si rozoberieme aj porovnania jazykov STATL a ASTD, a porovnanie jazyku ASTD a OWL.

4 Výsledky práce

Hlavným cieľom v našej bakalárskej práci bola implementácia detekcie útokov pomocou jazyku OWL, pričom sme vychádzali zo stavových jazykov ASTD a STATL. No ako prvé sme tieto jazyky porovnávali a poukázali na ich výhody a nevýhody. Po dôkladnej analýze jednotlivých častí vybraných stavových jazykov sme hľadali spôsob ako dané časti použiť v OWL, poprípade čím OWL už disponuje.

4.1 Analýza ASTD a STATL

Pri analýze stavových jazykov ASTD a STATL sme zistili, že systém iASTD vykonáva pri kompilácii automatizované testy funkčnosti a teda testuje, či jednotlivé časti ASTD ako napríklad automat, sekvencia alebo flow, fungujú správne. Fungovanie týchto testov sme analyzovali a našli nástroj s názvom cASTD, z ktorého dokumentácie vyplýva, že je schopný preložiť dané testy do Zeek skriptov, ako aj do jazyku Java alebo C++. Inštalácia systému iASTD a nástroja cASTD je rozobraná v prílohách A a B..

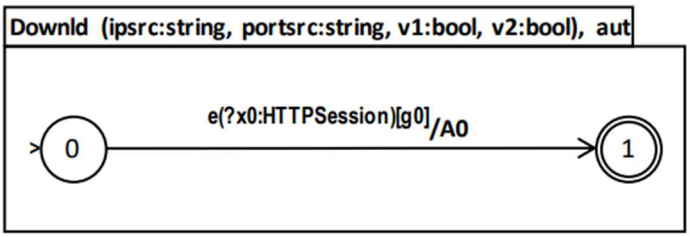
Testy funkčnosti boli v systéme rozdelené do viacerých časti, podľa toho čo mali testovať. Testovanie prebieha vždy pri builodvaní iASTD, no je možné ho po buildovaní dodatočne spustiť skriptom *func_tests.py* alebo spustiť ktorýkoľvek test použitím príkazu *./xASTD -s [špecifikácia] -i [vstup] -vv*. My sme sa zamerali na testovanie jednotlivých operátorov (napr. automat, sekvencia, flow atď.). Príklad špecifikácie pre automat je zobrazený v prílohe C. (K tejto špecifikácii bol priradený súbor so vstupmi, ktoré boli zadefinovane v poradí *e1, e2, e3* a *e4*.) Po spustení testu, boli do konzoly vypísané výsledky a postupnosť udalostí, ktoré sa vykonali. Tieto výpisy boli definované v dodatočnej funkcii napísanej v jazyku OCaml, ktorá bola pripojená do špecifikácie automatu, pričom tieto výpisy bolo možné meniť, poprípade pridávať, a upraviť si tak výstup daného testu tak, ako potrebujeme, čo nám pomohlo lepšie pochopiť fungovanie ASTD automatov.

Nástroj cASTD by mal byť schopný prekladať spomínané špecifikácie aj do Zeek skriptov. Nástroj má preddefinovaných niekoľko špecifikácii ASTD operátorov, ktoré je možné prekladať príkazom: *java -jar castd.jar –spec [špecifikácia] --target [jazyk] --output [miesto výstupu]*. Týmto spôsobom by sme boli schopní porovnať jednotlivé prípady riešenia stavových podtypov jazyka ASTD s ich riešeniami zapísanými pomocou Zeek skriptov, teda pomocou jazyku STATL. No dostupná verzia tohto nástroja napriek

dokumentácii nemala implementovanú možnosť prekladu funkčných testov do Zeek skriptov ale len do jazyku Java a C++. Z tohto dôvodu nebolo možné zrealizovať testovanie, ktoré porovnali ASTD a STATL a tak sme použili už existujúci experiment, ktorý popíšeme v časti 5.

Pre lepšiu predstavu špecifikácií uvedieme aj príklad z práce [20]. Na obrázku č. 27 vidíme model automatu, zapísaný pomocou ASTD a následne na obrázku č. 28 pomocou Zeek skriptu. Automat dostáva parametre ipsrc, portsrc, v1 a v2. Počiatočný stav 0 má vychádzajúci prechod označený udalosťou e(?x0:HTTPSession) a za ním nasleduje podmienka [g0]. Ak je podmienka splnená, automat vykoná akciu A0 a presunie sa do koncového stavu 1 [20].

Zeek, ktorý využíva STATL musí najprv daný model preložiť do Zeek skriptov a výsledok skriptu by mal podobu ako na obrázku 28. Z tohto porovnania môžeme vidieť riešenie pre automat, zapísaný pomocou Zeek skriptu.



Obrázok 27: Príklad automatu v ASTD [20]

```
...
#BLOCK 0
export {
  ...
  global v1: bool = F;
  global v2: bool = F;
  global state_downld: int = 0;
  global state_net_exec_code: int = 2;
  ...
}
...
event http_request (c: connection, ...)
{
 #BLOCK 1 (CAPEC-98)
 local g0: bool = (c$http$method == "GET")
  && (/\/[a-z]+\.(bin|exe)/ in c$http$uri)
  && (/Mozilla/ in c$http$user_agent)
  && (/([0-9]{1,3}\.){3}[0-9]{1,3}/
   in c$http$host);

 if (state_downld == 0 && g0)
 {
   A0(v1, v2);
   state_downld = 1;
 }
 #BLOCK 3 (CAPEC-549)
 local g2: bool = (c$http$method=="GET")
  && (|c$http$uri| == 1)
  && (/[a-z]+\.(bit|ru)/ in c$http$host);

 if (state_net_exec_code == 3 && g2)
 {
   state_net_exec_code = 4;
 }
 #BLOCK 4 (CAPEC-549)
 local g3: bool=(c$http$method=="POST")
  && (/[a-z]+\.(bit|ru)/ in c$http$host)
  && (/Mozilla/ in c$http$user_agent);

 if (state_net_exec_code == 4 && g3)
 {

  # Issue an alert
  A1(v1, v2);
  state_net_exec_code = 5;
 }
} # end http_request

event dns_request (c: connection, ...)
{
 #BLOCK 2 (CAPEC-549)
 local g1: bool =(/\x00\x01\x00\x00/
  in c$dns$query)
  && (/whatismyipaddress/
  in c$dns$query);
 if (state_net_exec_code == 2 && g1)
 {
   state_net_exec_code = 3;
 }
} # end dns_request
```

Obrázok 28: Zeek script automat [20]

Pri porovnávaní efektivity sme sa zamerali na už existujúci výskumu, kde bolo porovnanie vykonané na troch systémoch: Snort, iASTD a Zeek, pričom iASTD a Zeek

využívajú spomínané stavové jazyky ASTD a STATL. Z testovania vyplýva, že špecifikácie útokov opísané pomocou ASTD sú stručnejšie, viac modulárne a majú lepší grafický zápis ako pri nástrojoch Snort a Zeek. Pokiaľ ide o detekciu narušení, iASTD (inrepreter stavového jazyku ASTD) a Zeek poskytli podobné výsledky. iASTD produkoval menej falošne pozitívnych výsledkov a menší počet skutočne pozitívnych výsledkov na jeden útok ako Snort, ktorý je dôležitým faktorom pri zvládaní veľkého množstva udalostí. Čas spracovania iASTD na testovacom zariadení v reálnom čase je pomalší ako Snort a Zeek, ale dá sa vylepšiť kompilovaním ASTD špecifikácií do Zeek skriptov. [11] Detailnejšie rozoberieme dané porovnanie v časti 5.

Z našej analýzy sa javí jazyk ASTD ako jazyk so zaujímavejším a efektívnejším prístupom. Preto sme sa v ďalšej práci zamerali na jazyk ASTD.

4.2 Porovnanie ASTD a OWL

Pre využitie OWL sme sa rozhodli hlavne kvôli jeho jednoduchej prístupnosti a bohatej podpore zabezpečenou W3C. Hľadali sme spôsob ako vytvoriť ontológiu automatu, ktorý by bol schopný popísať útok dostatočne dôkladne. Taktiež sme hľadali spôsoby ako vytvoriť ontológie pre ostatné dôležité ASTD, pričom sme našli spôsob definovania sekvencii v OWL.

V časti 3.3 sme analyzovali jednotlivé stavové podtypy ASTD, pričom základným ASTD je automat. Pri automate je dôležitých niekoľko vlastností aby bol automat schopný zachytávať útoky. Je dôležite mať detailne popísane základné prvky automatu ako sú napríklad stavy, prechody a udalosti. Pre detekciu útokov je potrebne pre ASTD zadefinovať aj akcie na prechodoch, či stavové premenné.

V snahe využiť OWL pre detekciu útokov podobne ako ASTD, sme museli nájsť spôsob definície automatu v tomto jazyku. OWL sám o sebe nedisponuje prvkami, ktoré by jednoznačne vedeli opísať automat, preto bolo nutné nájsť spôsob ako tento level dosiahnuť.

V sekcii 3.4 sme rozoberali prácu [18], ktorá využíva OWL pre definovanie ontológie stavového automatu, pričom ontológia tohto automatu je primárne určená pre opis automatizácie vo výrobe, no disponuje všetkými potrebnými prvkami, ktoré sú definované v ASTD pre detekciu útokov. OWL model automatu dokáže detailne popísať stavy daného automatu vrátane premenných, pričom vie rozlíšiť viacero typov stavov, ako napríklad hlboký stav (deep state) a plytký stavu (shallow state), takže dokáže pracovať aj s

komplexnými stavmi. Pre zložitejšie akcie a udalosti, je potrebné definovanie ontológiu, ktorá detailnejšie definuje udalosť alebo akciu ktorá sa má vykonať. Podmienku pri prechodoch je nutné taktiež definovať pomocou ontológie výrazu, ktorý presne definuje podmienku.

Pomocou OWL vieme vytvoriť ontológiu, ktorá popisuje synchronizáciu, na základe zoznamu prvkov, čo v našom prípade podobne ako v ASTD môže byť zoznam automatov. OWL berie vždy prvý prvok zo zoznamu s ktorým pracuje a ďalej posúva zoznam zvyšných prvkov, pokiaľ nepríde na prázdny zoznam.

4.3 Testovanie OWL

Ako ďalšie sme hľadali spôsob ako otestovať riešenia vytvorené v OWL, pričom sme predpokladali, že OWL bude vedieť riadiť dané riešenie a sám rozhodovať, čo sa má stať. Naším plánom bolo vytvoriť podobné testy funkčnosti, akými disponoval systém iASTD. Napokon sme prišli k záveru, že OWL nie je schopný riadiť daný automat a rozhodovať, aké prechody a akcie majú byť vykonané. Z toho dôvodu je potrebné, tak ako pri ASTD a STATL, použiť doplňujúci jazyk, ktorý by riadil logiku automatu, tak ako napríklad OCaml pri ASTD alebo Zeek scripts pri STATL, pričom ontológia vytvorená pomocou OWL by mohla slúžiť na špecifikáciu útoku, podobne ako stavové diagramy.

V nasledujúcej časti budeme opisovať porovnávanie nástrojov využívajúcich stavové jazyky ASTD a STATL, a nástrojov ktoré pre detekciu využívajú signatúry Snort a Zeek-sig. Toto porovnanie vyplýva z práce [20].

5 Porovnanie nástrojov pre detekciu útokov

Keďže naše testovanie jazykov STATL a ASTD nebolo možné zrealizovať, pre prehľad sme sa zamerali na existujúci experiment, ktorý testoval nástroje iASTD, Zeek a Snort na rôznych typoch útokov, pričom vo výsledku tohto experimentu, môžeme vidieť aj výhody a nevýhody nástrojov využívajúcich stavové jazyky oproti Snortu, ako aj výhody a nevýhody nástroja iASTD oproti Zeek.

Pre porovnanie sme z práce vybrali iba namerané výsledky. Celý priebeh, postup a konfigurácia testovania je detailne popísaná v práci [20].

5.1 Experiment

V experimente boli vybrané dve existujúce databázy z reálneho sveta a to databázy CSE-CIC-IDS2018 a CTU, a vlastná vytvorená databáza podobajúcu sa databázam z reálneho prostredia.

CSE-CIC-IDS2018 [21] je výsledkom spoločného projektu medzi organizáciami Communications Security Establishment (CSE) a Canadian Institute for Cybersecurity. Tento súbor údajov zahŕňa podrobný popis prienikov spolu s abstraktnými distribučnými modelmi pre aplikácie, protokoly alebo sieťové entity [21]. Dataset CSE-CIC-IDS2018 je dostupný na stránke Kanadského Inštitútu Kybernetickej bezpečnosti [https://www.unb.ca/cic/datasets/ids-2018.html].

CTU je súbor údajov o návštevnosti botnetov, ktorý bol zachytený na univerzite ČVUT v českej republike. Cieľom súboru údajov bolo zachytiť skutočnú prevádzku botnetov (normálnu aj na pozadí), pričom bolo vykonaných 7 rôznych botnetov. [22] Dataset CTU je dostupný na stránke [https://www.stratosphereips.org/datasets-ctu13].

Pre vlastnú databázu bolo špecifikovaných 20 útokov v IDS nástrojoch a boli vykonané na platforme AWS. [20]

Vo výsledkoch experimentu, sú zobrazené výsledky niekoľkých útokov a dát z botnetov z každého datasetu. V našej práci sme opísali iba výsledky testovania pre každý jazyk, pričom pre prehľadnosť sme každý jazyk dali zvlášť do tabuliek.

5.2 Výsledky experimentu

Na porovnanie presnosti a výkonu IDS nástrojov boli použité dve metriky: miera detekcie (DR) a miera falošnej detekcie (FPR) [20].

- **Miera detekcie (DR)** je pravdepodobnosť, že IDS vydá výstrahu, keď príde k narušeniu.
- **Miera falošne pozitívnych výsledkov (FPR)** je pravdepodobnosť, že IDS vydá výstrahu napriek tomu, že správanie systému je normálne.

Dané hodnoty sú dosiahnuteľné vzorcami [20]:

$$DN = \frac{TP}{TP+FN} \qquad FPR = \frac{FP}{FP+TN}$$

- **FP (False Positive)** je počet normálnych upozornení, ktoré sú nesprávne zaradených ako škodlivé
- **TP (True Positive)** je počet škodlivých upozornený, ktoré sú správne zaradené ako škodlivé
- **FN (False Negative)** je počet škodlivých upozornení, ktoré sú nesprávne zaradené ako normálne
- **TN (True Positive)** je počet normálnych upozornení, ktoré sú správne zaradené ako normálne

Výsledky hodnotenia nástrojov pre datasety CSE-CIC-IDS2018, CTU a vlastnú vytvorenú databázu, sú uvedené v tabuľke 1. Výsledky pre samostatné nástroje sú ďalej uvedené v tabuľke 2, tabuľke 3 a tabuľke 4. Zápis Zeek-sig/Zeek-script je určený na rozlíšenie výsledkov dvoch verzií nástroja Zeek. Zeek-sig je označením pre špecifikáciu, ktorá používa iba signatúry, naopak Zeek-script je označenie pre špecifikáciu pomocou zeek scriptov [20].

CSE-CIC	Zeek-sig/Zeek-script						Snort						iASTD					
-IDS2018	TP	TN	FP	FN	**DR(%)**	**FPR(%)**	TP	TN	FP	FN	**DR(%)**	**FPR(%)**	TP	TN	FP	FN	**DR(%)**	**FPR(%)**
LOIC DDoS UDP	0/1	0/0	164/0	0/0	0/100	100/0	8904	4914	912	0	100	15.65	1	0	0	0	100	0
HOIC DDoS HTTP	0/1	0/0	289342/0	0/0	0/100	100/0	197	5071	755	0	100	13.31	1	0	0	0	100	0
SSH BruteForce	0/1	0/0	94216/0	0/0	0/100	100/0	67	94	0	0	100	0.00	1	0	0	0	100	0
FTP BruteForce	0/1	0/0	193392/0	0/0	0/100	100/0	3868	94	0	0	100	0.00	1	0	0	0	100	0
GoldenEye DoS	0/1	0/0	27751/0	0/0	0/100	100/0	1	96	8	0	100	7.84	1	0	0	0	100	0
Web BruteForce	71/1	0/0	0/0	0/0	100/100	0/0	3	2682	73	0	100	2.64	1	0	0	0	100	0
XSS BruteForce	19/1	0/0	0/0	0/0	100/100	0/0	1	2482	0	0	100	0.00	1	0	0	0	100	0
SQL Injection	15/1	0/0	0/0	0/0	100/100	0/0	2	2482	0	0	100	0.00	1	0	0	0	100	0
CTU																		
Neris	3/1	0/0	0/0	0/0	100/100	0/0	1	131	2	0	100	1.50	1	0	0	0	100	0
Rbot	2/1	0/0	0/0	0/0	100/100	0/0	10	5249	77	0	100	1.45	1	0	0	0	100	0
Rbot DoS	4/1	0/0	0/0	0/0	100/100	0/0	3	6684	20	0	100	0.30	1	0	0	0	100	0
Virut	2/1	0/0	0/0	0/0	100/100	0/0	1	11	0	0	100	0	1	0	0	0	100	0
Donbot	178/1	0/0	0/0	0/0	100/100	0/0	91	92	10	0	100	9.8	1	0	0	0	100	0
Sogou	2/1	0/0	0/0	0/0	100/100	0/0	1	15	0	0	100	0	1	0	0	0	100	0
qvod	2/1	0/0	0/0	0/0	100/100	0/0	2	501	36	0	100	6.7	1	0	0	0	100	0
NSIS.ay	3/1	0/0	0/0	0/0	100/100	0/0	3	97	0	0	100	0.00	1	0	0	0	100	0
Real-time																		
WannaCry	6/3	0/0	0/0	0/0	100/100	0/0	6	2434	58	0	100	2.32	2	0	0	0	100	0
Petya	13/4	0/0	0/0	0/0	100/100	0/0	22	1336	47	0	100	3.40	2	0	0	0	100	0
TeslaCrypt	4/1	0/0	0/0	0/0	100/100	0/0	8	20	0	0	100	0	2	0	0	0	100	0
Gandcrab	4/1	0/0	0/0	0/0	100/100	0/0	10	39	0	0	100	0	2	0	0	0	100	0
Normal	0/0	0/0	0/0	0/0	0/0	0/0	0	0	0	0	0	0	0	0	0	0	0	0

Tabuľka 1: Výsledky experimentu [20]

5.2.1 Zeek-script / Zeek-sig

CSE-CIC-IDS2018	Zeek-sig / Zeek-script					
	TP	**TN**	**FP**	**FN**	**DR(%)**	**FPR(%)**
LOIC DDoS UDP	0/1	0/0	164/0	0/0	0/100	100/0
HOIC DDoS HTTP	0/1	0/0	289342/0	0/0	0/100	100/0
SSH BruteForce	0/1	0/0	94216/0	0/0	0/100	100/0
FTP BruteForce	0/1	0/0	193392/0	0/0	0/100	100/0
GoldenEye DoS	0/1	0/0	27751/0	0/0	0/100	100/0
Web BruteForce	71/1	0/0	0/0	0/0	100/100	0/0
XSS BruteForce	19/1	0/0	0/0	0/0	100/100	0/0
SQL Injection	15/1	0/0	0/0	0/0	100/100	0/0
CTU						
Neris	3/1	0/0	0/0	0/0	100/100	0/0
Rbot	2/1	0/0	0/0	0/0	100/100	0/0
Rbot DoS	4/1	0/0	0/0	0/0	100/100	0/0
Virut	2/1	0/0	0/0	0/0	100/100	0/0
Donbot	178/1	0/0	0/0	0/0	100/100	0/0
Sogou	2/1	0/0	0/0	0/0	100/100	0/0
qvod	2/1	0/0	0/0	0/0	100/100	0/0
NSIS.ay	3/1	0/0	0/0	0/0	100/100	0/0
Real-time						
WannaCry	6/3	0/0	0/0	0/0	100/100	0/0
Petya	13/4	0/0	0/0	0/0	100/100	0/0
TeslaCrypt	4/1	0/0	0/0	0/0	100/100	0/0
Gandacrab	4/1	0/0	0/0	0/0	100/100	0/0
Normal	0/0	0/0	0/0	0/0	0/0	0/0

Tabuľka 2: Výsledky pre Zeek [20]

V miere FPR, Zeek-script vyprodukoval menej správne pozitívnych (TP) ako Zeek-sig. V Zeek-sig, bola snaha o špecifikáciu signatúr pre Distribuovaný DoS (DDoS) a SSH/FTP BruceForce, založená na protokoloch a slabom pozorovanom obsahu, ktoré neboli dostatočne jedinečné (FPD = 100%). Zeek sig ale detekoval SQL injection, XSS a Web BruteForce s DR 100. [20].

Zeek-script produkoval menej TP ako Zeek-sig a žiadne FP pre WannaCry a Petya útoky. Pre Zeek-script, boli 3 TP pre WannaCry a 4 TP pre Petya. Zeek-script negeneroval žiadne FP [20].

Z výsledkov pre Zeek vyplýva, že Zeek-script, úspešne detekoval všetky útoky v testovaní s DR = 100% a FPR = 0%, zatiaľ čo Zeek-sig úspešne detekoval len 3 útoky.

5.2.2 iASTD

CSE-CIC-IDS2018	iASTD					
	TP	TN	FP	FN	DR(%)	FPR(%)
LOIC DDoS UDP	1	0	0	0	100	0
HOIC DDoS HTTP	1	0	0	0	100	0
SSH BruteForce	1	0	0	0	100	0
FTP BruteForce	1	0	0	0	100	0
GoldenEye DoS	1	0	0	0	100	0
Web BruteForce	1	0	0	0	100	0
XSS BruteForce	1	0	0	0	100	0
SQL Injection	1	0	0	0	100	0
CTU						
Neris	1	0	0	0	100	0
Rbot	1	0	0	0	100	0
Rbot DoS	1	0	0	0	100	0
Virut	1	0	0	0	100	0
Donbot	1	0	0	0	100	0
Sogou	1	0	0	0	100	0
qvod	1	0	0	0	100	0
NSIS.ay	1	0	0	0	100	0
Real-time						
WannaCry	2	0	0	0	100	0
Petya	2	0	0	0	100	0
TeslaCrypt	2	0	0	0	100	0
Gandacrab	2	0	0	0	100	0
Normal	0	0	0	0	0	0

Tabuľka 3: Výsledky pre iASTD [20]

Podobne ako Zeek a Snort, iASTD nevygeneroval FP v bežnej sieťovej prevádzke (riadok Normal) [2]. iASTD podobne ako Zeek-script generoval len TP upozornenia a teda z výsledkov vyplýva, že všetky útoky detekoval úspešne s DR = 100% a FPR = 0% [20].

5.2.3 Snort

CSE-CIC-IDS2018	Snort					
	TP	TN	FP	FN	DR(%)	FPR(%)
LOIC DDoS UDP	8904	4914	912	0	100	15.65
HOIC DDoS HTTP	197	5071	755	0	100	13.31
SSH BruteForce	67	94	0	0	100	0.00
FTP BruteForce	3868	94	0	0	100	0.00
GoldenEye DoS	1	96	8	0	100	7.84
Web BruteForce	3	2682	73	0	100	2.64
XSS BruteForce	1	2482	0	0	100	0.00
SQL Injection	2	2482	0	0	100	0.00
CTU						
Neris	1	131	2	0	100	1.50
Rbot	10	5249	77	0	100	1.45
Rbot DoS	3	6684	20	0	100	0.30
Virut	1	11	0	0	100	0.00
Donbot	91	92	10	0	100	9.80
Sogou	1	15	0	0	100	0.00
qvod	2	501	36	0	100	6.70
NSIS.ay	3	97	0	0	100	0.00
Real-time						
WannaCry	6	2434	58	0	100	2.32
Petya	22	1336	47	0	100	3.40
TeslaCrypt	8	20	0	0	100	0.00
Gandacrab	10	39	0	0	100	0.00
Normal	0	0	0	0	0	0.00

Tabuľka 4: Výsledky pre Snort [20]

Snort generoval množstvo FPR pre LOIC DDoS UDP (15.65%) a HOIC DDoS HTTP (13.31%). Snort taktiež vygeneroval signifikantný počet TP pre LOIC DDoS UDP (8904), HOIC DDoS http (197), FTP BruteForce (3868), SSH BruteForce (67) a Donbot (91). Okrem toho Snort produkoval významný FPR v porovnaní so Zeek-script a iASTD pre WannaCry a Petya útoky (2.32% pre WannaCry a 3.4% pre Petya) [20].

Výsledkov môžeme usúdiť, že napriek vysokej miere FDP pri niektorých útokoch, Snort úspešne detekoval všetky útoky s DR = 100%, no nie tak presvedčivo ako Zeek-script a iASTD [20].

Celkovo Zeek-script a iASTD dosiahli lepší detekčný výkon ako Zeek-sig a Snort s vysokým DR 100% a bez FP. Nástroje dokázali korelovať viacero pripojení http a DNS z CSE-CIC-IDS2018 a CTU útokov.

5.3 Diskusia k testovaniu

Snort je nízkoúrovňový, bezstavový jazyk so vzorom udalostí [20]. Zeek je scriptovací jazyk, ktorého mechanizmy sú založené na imperatívnom programovaní, a teda

využíva procedurálnu abstrakciu, programovú kompozíciu s použitím if-then-else a stavových premenných. Zeek využíva stavy prechody a akcie, vďaka ktorým môže spájať viaceré sieťové udalosti. Vytváranie Zeek skriptov je náročné, zložité a náchylné na chyby [20]. ASTD je viac abstraktný, jeho stavové automaty ponúkajú grafickú a detailnú reprezentáciu útokov a stavových prechodov. Tento útok môže byť špecifikovaný modulárnym spôsobom, ktorý sleduje prirodzený popis štruktúry útoku, zapísaný do fáz a krokov. Snort a Zeek signatúry môžu byť ľahko reprezentované pomocou ASTD špecifikácií, využitím ASTD operátorov. ASTD môže byť použité na akomkoľvek zdroji, napríklad sieť/host udalosť [20].

Ako bolo namerané v experimente, Snort mal významnú FPR a vysokú DR. Priemerný čas spracovania Snortu na testovacom zariadení v reálnom čase je ale pomerne nízky (2.336s pre 1Gb paketov, 8.201s pre 10Gb paketov) [20].

Zeek-script má v priemere nízky FPR a vysoký DR. Jeho doba spracovania na testovacom zariadení v realnom čase je v priemere nízka (7.480s pre 1Gb pakety, 29.766s pre 10Gb pakety) [20].

iASTD má v priemere nízke FPR a vysoké DR, ale jeho čas spracovania na testovacej ploche v realnom čase je v priemere relatívne stredná (20,184s pre 1Gb paketov, 96,778s pre 10Gb paketov) [20].

Na detekciu narušení siete je Snort rýchlejší ako Zeek a iASTD [20]. Zeek dokáže korelovať viacero sieťových pripojení a je rýchlejší ako iASTD. Pre sieť a detekciu narušení hosta, iASTD dokázal korelovať sieťové pripojenia a Windows/Syslog udalosti, ale veľké množstvo udalostí značne ovplyvnilo čas detekcie (188.667s pre 10Gb paketov a 87 450 zmiešaných Windows/Syslog udalostí) [20].

Zeek je rýchlejší ako iASTD, ale malo by byť možne skompilovať ASTD špecifikácie do Zeek skriptov, pre využitie ASTD špecifikácií efektívnejšie, pričom by sa dalo stále benefitovať z vlastností systému Zeek. [20]

Záver

V našej bakalárskej práci sme sa zaoberali stavovými jazykmi ASTD a STATL, venovali sme sa taktiež systémom na detekciu útokov, iASTD a Zeek, ktoré využívajú spomínané stavové jazyky, a ontologickému jazyku OWL. Jazyk ASTD môžeme pokladať za rozšírenie jazyku STATL, z toho dôvodu sme sa v našej práci zamerali práve na tento jazyk a na jazyk OWL, ktorý má dobrú podporu zabezpečenú organizáciou W3C.

ASTD využíva množstvo algebraických operácií, vďaka ktorým vie dôkladne popísať špecifikácie útokov a znázorniť ich pomocou stavových diagramov. Z analýzy vyplýva, že pre využitie ASTD ako jazyka, ktorý by plnohodnotne špecifikoval útoky, by bolo potrebné do jazyka pridať stavové premenné, akcie na prechodoch a nový operátor flow. Základným ASTD je automat, ktorý sa vie ďalej rozšíriť do zložitejších ASTD, napríklad pomocou sekvencií alebo spomínaného operátora flow. Hlavnou výhodou ASTD je jeho schopnosť dobre reprezentovať špecifikácie útokov pomocou stavových diagramov, vďaka svojej modularite a dobrej korelácii medzi jednotlivými operátormi.

OWL je schopný popísať stavové automaty a sekvencie, no sám neobsahuje potrebné komponenty pre vytvorenie automatov či sekvencií, preto bolo potrebné nájsť spôsob, ako vytvoriť ontológiu automatu a všetkých operátorov, ktorými disponuje ASTD. Aby sme boli schopní využiť OWL pre detekciu útokov, je potrebný systém, ktorý by spracovával vytvorenú ontológiu a riadil ju. Z toho dôvodu sme neboli schopní otestovať špecifikácie vytvorené pomocou OWL, pretože by sme museli vytvoriť spomínaný systém na riadenia alebo preprogramovať systém iASTD tak, aby dokázal spracovať aj OWL špecifikácie.

Napriek tomu, že špecifikácie automatu a sekvencií vytvorené pomocou OWL neboli otestované, myslíme si, že OWL je schopný tieto špecifikácie plnohodnotne vytvárať, keďže zdroje daných informácií, z ktorých sme čerpali, boli dostatočne dôveryhodné a profesionálne.

Predpokladáme, že pre lepšiu špecifikáciu útokov pomocou OWL by bolo potrebné definovanie viacerých ontológií jednotlivých operátorov, podobne ako v ASTD (flow, synchronizácia, Kleen uzáver atď.). Taktiež pri využívaní OWL pre detekciu útokov, je nutné vytvorenie systému, ktorý by dané špecifikácie ovládal. Takýto systém by mohol byt založený napríklad na programovacom jazyku OCaml podobne ako ASTD alebo v jazyku

JAVA, ktorý disponuje knižnicami (napr. OWL API), ktoré by boli nápomocné pri práci s OWL.

Analyzovanie stavových jazykov prinieslo nepochybne aj mne nové poznatky z oblasti kybernetickej bezpečnosti a ukázalo nový spôsob detekcie útokov pomocou stavových jazykov.

Dúfame, že naša práca bude prínosom do oblasti kybernetickej bezpečnosti, keďže stavové jazyky, ktoré sa využívajú v systémoch detekujúcich útoky, nie sú v súčasnosti tak rozšírené ako práve OWL, pričom základom do budúcej práce by mohli byť špecifikácie automatu alebo sekvencií, ktoré sme analyzované v našej práci a javia sa ako vhodný spôsob pre opis útokov.

Zoznam použitej literatúry

1. FOWLER, M. *UML Distilled*: *A Brief Guide to the standard object modelling language*. 3. vyd. Boston : Addison-Wesley Longman Publishing Co., Inc., 2003. 256s. ISBN 978-0-321-19368-1
2. SAMEK, M. A Crash Course in UML State Machines. In *Practial UML Statecharts in C/C++*. [online]. 2008. [cit. 2021-11-29]. Dostupné na internete: https://www.semanticscholar.org/paper/A-Crash-Course-in-UML-State-Machines-Samek/20d38f346126865eec1bbbed30cd9f3094a469d6
3. ECKMANN, S., GIOVANNI, V., KEMMERER, R. STATL : An attack language for state-based intrusion detection. In *Journal of Computer Security*, vol. 10, no. 1-2, p. 71-103, 2002. [online]. Santa Barbara : University of California. 2002. [cit. 2021-12-13]. Dostupné na: https://content.iospress.com/articles/journal-of-computer-security/jcs158
4. TIDJON, N. L. et al. Extended Algebraic State-Transition Diagrams. In *23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2018. [online]. Sherbrooke : Université de Sherbrooke. 2018. [cit. 2021-12-10]. Dostupné na: https://ieeexplore.ieee.org/document/8595068?fbclid=IwAR1fPMGkr1u05rLMPKhNZSXLhap9M8ve6alhOCFM3x-tPtKVClZvZ4rGuoY
5. HISTORIA OWL. *Cesta k OWL.* [online]. © 2006 [cit 2022-04-05]. Dostupné na internete: https://lide.uhk.cz/fim/ucitel/fshusam2/lekarnicky/zt1/zt1_kap04.html
6. FRAPPIER, M. et al. Algebraic State Transition Diagrams. Sherbrook : Université de Sherbrooke, Canada. 2017. [cit. 2022-01-05]. Dostupné na: https://www.academia.edu/68444695/Algebraic_State_Transition_Diagrams
7. ZEEK. *About Zeek.* [online]. © 2019-2021 [cit. 2022-12-18]. Dostupné na: https://docs.zeek.org/en/master/about.html?fbclid=IwAR2jgAzgB7nFp0ViNzBaV2q7KH7gmAeCEFpb-07_azLB9v8R-IGSxd8uYbk
8. MINSKY, Y., MADHAVAPEDDY, A. *Why OCaml?.* In *Real World OCaml*. 2021. [online]. [cit. 2022-01-06]. Dostupné na: https://dev.realworldocaml.org/prologue.html#why-ocaml

9. ProB. *The ProB Animator and Model Checker.* [online]. © 2021. [cit. 2022-01-15]. Dostupné na internete: https://prob.hhu.de/w/index.php?title=The_ProB_Animator_and_Model_Checker
10. CANSELL, D., MERY, D. Foundations of the B Method. In *Computing and Informatics,* vol. 22, no. 3-4, p. 221-256, 2003. [online]. ISSN 1335-9150. [cit. 2022-01-18]. Dostupné na internete: https://www.researchgate.net/publication/230688722_Foundations_of_the_B_Method
11. SNORT. *What is Snort?* [online]. © 2022 [cit. 2022-12-20]. Dostupné na internete: https://www.snort.org/?fbclid=IwAR3ZDJZvHfcZJpGDBRPlIhWsq4cGmv8Vvln3Uypg8bYvSAsEiJVKK8rGATM
12. MUNIR, R. et al. Performance Security Trade-off of Network Intrusion Detection and Prevention Systems. In *32nd UK Performance Engineering Workshop & Cyber Security Workshop,* 2016. [online]. Bradford : University of Bradford. 2016. [cit. 2022-01-04]. Dostupné na internete: https://www.researchgate.net/publication/312630470_Performance_Security_Trade-off_of_Network_Intrusion_Detection_and_Prevention_Systems
13. SNORT RULES. Understanding and Configuring Snort Rules. [online]. © 2016 [cit. 2022-02-11]. Dostupné na internete: https://www.rapid7.com/blog/post/2016/12/09/understanding-and-configuring-snort-rules/
14. McGUINNESS, D., HARMELEN F. OWL Web Ontology Language. In W3C, 2004. [online]. Stanford : Stanford University 2004. [cit. 2022-04-09]. Dostupné na: https://www.w3.org/TR/2004/REC-owl-features-20040210/
15. HORROCKS, I. et al. OWL 2 Web Ontology Language. In W3C, 2012. [online]. Oxford : Oxford University. [cit. 2022-04-09]. Dostupné na internete: https://www.w3.org/TR/owl2-overview/
16. BAADER, F., HORROCKS, I., LUTZ, C., SATTLER, U. *An Introduction to Description Logic*. Cambridge : Cambridge University Press. 2017. ISBN 9781139025355.

17. OPERATOR A OPERAND. Expressions, Operators, and Operands. [online]. © 2010 [cit. 2022-04-29]. Dostupné na : https://docs.oracle.com/cd/E19957-01/805-4939/6j4m0vn71/index.html
18. SCHNEIDER, G. Semantic Modelling of Control Logic in Automation Systems. [online]. 2019. Karlsruhe : Karlsruher Institut für Technologie. [cit. 2022-05-20]. Dostupné na internete: https://publikationen.bibliothek.kit.edu/1000092405
19. DRUMMOND, N. et al. Putting OWL in Order: patterns for sequences in OWL. In *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions.* [online]. Athens, Georgia, USA, 2006. [cit. 2022-05-10]. Dostupné na internete: https://www.researchgate.net/publication/221218426_Putting_OWL_in_Order_patterns_for_sequences_in_OWL
20. TIDJON, L., FRAPPIER, M., MAMMAR, A. Intrusion Detection Usig ASTDs. In *34th International Conference on Advanced Information Networking and Applications*. [online]. Caserta, Italy. 2020. ISBN 978-3-030-44040-4. [cit. 2022-05-15]. Dostupné na internete: https://www.researchgate.net/publication/340238489_Intrusion_Detection_Using_ASTDs
21. SHARAFALDIN, I., LASHKARI, A., GHORBANI, A. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*. [online]. p. 108-116. ISSN 2184-4356. [cit. 2022-04-10]. Dostupné na internete: https://www.scitepress.org/Link.aspx?doi=10.5220/0006639801080116
22. GARCIA, S., GRILL, M., STIBOREK, J., ZUNINO, A. *An empirical comparison of botnet detection methods.* [online.] In *Computers and Security Journal*, Elsevier. 2014. [cit. 2022-06-17]. Dostupné na internete: http://dx.doi.org/10.1016/j.cose.2014.05.011

Prílohy

Príloha A: Inštalácia ASTD

Pri inštalovaní systému ASTD postupujeme podľa návodu v *readme* súbore. Systém je potrebné inštalovať na operačnom systéme Linux. Postup inštalácie je nasledovný:

```
$ sudo apt-get install ocaml
$ sudo apt-get install opam
$ opam init
$ eval `(opam config env)`
$ opam switch 4.04.0
$ eval `(opam config env)`
$ echo 'eval `(opam config env)`' >> ~/.bashrc
```

Príloha B: Inštalácia cASTD

Inštalácia cASTD musí byť taktiež vykonaná na operačnom systéme Linux. Postup inštalácie je nasledovný:
Ako prvé je potrebné si rozbaliť zip súbor cASTD a následne vykonať nasledujúce príkazy:

```
$ sudo apt-get install build-essential

$ sudo apt-get-repository ppa:openjdk-r/ppa

$ sudo apt-get update

$ suod apt-get install openjdk-8-jdk

$ sudo update-alternatives –config java

$ sudo update-alternatives –config javac

$ sudo apt-get install ant

$ cd build && java -jar castd.jar -v
```

Príloha C: Príklad špecifikácia automatu pri testovaní

```
(MAIN,
 <aut;
  imports : {
   "ModuleMath.ml"
  };
  attributes : {
   (x, int, 0)
  };
  code : "ModuleMath.inc_by_one(x)";
  {
   (A1->elem),
   (A2->elem),
   (A3->
    <aut;
     code : "ModuleMath.inc_by_ten_if_five(x)";
     {
      (s1->elem),
      (s2->elem)
     };
     {
      ((local,s1,s2), e3, {}, "ModuleMath.inc_if_four(x)", False)
     };
     {};
     {
      s1
     };
     s1
    >
   ),
   (A4->elem)
  };
  {
   ((local,A1,A2), e1, {}, "ModuleMath.inc_if_zero(x)", False),
   ((local,A2,A3), e2, {}, "ModuleMath.inc_if_two(x)", False),
   ((local,A3,A4), e4, {file : "GuardIs16"}, False)
  };
  {
   A3
  };
  {};
  A1
 >
)
```