Slovenská technická univerzita v Bratislave
Fakulta elektrotechniky a informatiky

Ontologická reprezentácia pre bezpečnosť informačných systémov

Dizertačná práca

Ing. Peter Švec

Bratislava, 2024

Slovenská technická univerzita v Bratislave
Fakulta elektrotechniky a informatiky

Ontologická reprezentácia pre bezpečnosť informačných systémov

Dizertačná práca

Ing. Peter Švec

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Evidenčné číslo:	FEI-104372-8381
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	prof. Ing. Pavol Zajac, PhD.
Konzultant:	Ing. Štefan Balogh, PhD.

Bratislava, 2024

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024
Evidenčné číslo: FEI-104372-8381

ZADANIE DIZERTAČNEJ PRÁCE

Študent:	**Ing. Peter Švec**
ID študenta:	8381
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Školiteľ:	prof. Ing. Pavol Zajac, PhD.
Vedúci pracoviska:	doc. Ing. Milan Vojvoda, PhD.
Konzultant:	Ing. Štefan Balogh, PhD.

Názov práce: **Ontologická reprezentácia pre bezpečnosť informačných systémov**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Cieľom dizertačnej práce je prebádať a zdokumentovať prínosy ontologickej reprezentácie pre oblasť informačnej bezpečnosti v súvislosti s projektom APVV-19-0220. Hlavnou náplňou práce bude návrh, tvorba a vyhodnotenie systémov na detekciu škodlivého kódu využívajúcich ontologickú reprezentáciu dát.

Úlohy:

1. Analýza súčasného stavu problematiky.
2. Návrh ontológie pre detekciu malvéru.
3. Výskum algoritmov konceptového učenia pre rozpoznávanie škodlivého kódu.
4. Vykonanie experimentov a vyhodnotenie úspešnosti.
5. Zhodnotenie prínosov práce.

Zoznam odbornej literatúry:

1. LEHMANN, Jens. *Learning OWL Class Expressions.* Heidelberg : Akademische Verlagsgesellschaft, 2010. 265 s. ISBN 978-3-89838-635-7.
2. Ye, Yanfang, et al. ”A survey on malware detection using data mining techniques.” ACM Computing Surveys (CSUR) 50.3 (2017): 1-40.

Termín odovzdania dizertačnej práce:	31. 05. 2024
Dátum schválenia zadania dizertačnej práce:	06. 11. 2023
Zadanie dizertačnej práce schválil:	prof. Dr. Ing. Miloš Oravec – predseda odborovej komisie

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Ing. Peter Švec
Dizertačná práca:	Ontologická reprezentácia pre bezpečnosť informačných systémov
Vedúci záverečnej práce:	prof. Ing. Pavol Zajac, PhD.
Konzultant:	Ing. Štefan Balogh, PhD.
Miesto a rok predloženia práce:	Bratislava 2024

Táto práca sa zaoberá výskumom ontológií a sémantických technológií v oblasti bezpečnosti informačných systémov. Konkrétne sa práca zaoberá algoritmami konceptového učenia a ich uplatnením pri vývoji vysvetliteľných modelov pre detekciu škodlivého kódu. V prvej časti práce sme sa venovali návrhu ontológie a datasetu pre škodlivé vzorky, ktorú možno využiť pri rôznych algoritmoch konceptového učenia. Druhá časť práce sa venuje analýze existujúcich algoritmov konceptového učenia, dostupných v rámci softvéru DL-Learner. Jedná sa o algoritmy OCEL, CELOE, PARCEL a SPARCEL. V rámci tejto časti uvádzame experimentálne výsledky pri rôznych nastaveniach spolu s vyhodnotením úspešnosti vytvorených klasifikátorov a možnosťami ich interpretácie. Posledná experimentálna časť práce sa venuje bezpečnosti nami vytvorených modelov z pohľadu útočníkov.

Kľúčové slová: Ontológie, Deskripčné logiky, Detekcia malvéru, Konceptové učenie, Bezpečnosť

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Ing. Peter Švec
Dissertation:	Ontological representation for security of information systems
Supervisor:	prof. Ing. Pavol Zajac, PhD.
Consultant:	Ing. Štefan Balogh, PhD.
Place and year of submission:	Bratislava 2024

This thesis deals with the research of ontologies and semantic technologies in the domain of security of information systems. Specifically, the thesis deals with concept learning algorithms and their application in the development of explainable malware detection models. In the first part of the thesis, we focused on the design of the ontology and dataset for malicious samples, which can be used with various concept learning algorithms. The second part of the thesis is devoted to the analysis of existing concept learning algorithms, available within the DL-Learner framework. These algorithms are OCEL, CELOE, PARCEL and SPARCEL. In this section, we present experimental results for different algorithm settings, together with an evaluation of the developed classifiers and their interpretability. The last experimental part of the thesis deals with the security properties of our models from attackers point of view.

Keywords: Ontologies, Description logics, Malware detection, Concept learning, Security

Poďakovanie

Na tomto mieste by som chcel poďakovať viacerým ľuďom. V prvom rade by som sa rád poďakoval vedúcemu práce prof. Ing. Pavlovi Zajacovi, PhD. a vedúcemu špecialistovi Ing. Štefanovi Baloghovi, PhD. za odborné rady a profesionálne vedenie počas celého štúdia.

Ďalej by som rád vyjadril poďakovanie všetkým ľuďom z tímu ORBIS v rámci APVV-19-0220 projektu, ktorí pomohli pri publikovaní článkov v rámci tejto práce, konkrétne doc. RNDr. Martinovi Homolovi, PhD., Mgr. Jánovi Kľukovi, PhD., Ing. Alexandrovi Šimkovi PhD., Bc. Tomášovi Bistákovi, Mgr. Jánovi Mojžišovi, PhD. a Ing. Roderikovi Ploszekovi, PhD.

Veľká vďaka patrí taktiež Mgr. Ing. Matúšovi Jókayovi, PhD., ktorý poskytol svoje výpočtové zariadenie, na ktorom boli vykonané všetky experimenty v tejto práci a ochotne riešil takmer každý týždeň výpadky SSH spojenia.

Veľké poďakovanie patrí priateľke Sabinke, rodičom, rodine a priateľom za neustálu podporu a trpezlivosť počas štúdia a písania tejto práce.

Špeciálne poďakovanie patrí taktiež ľuďom, ktorých som spoznal v rámci PhD. štúdia a aj keď o tom nevedeli, dodali mi motiváciu pokračovať v štúdiu a napokon ho dokončiť. Vďaka patrí Palovi, Kike, Smilovi a Palkovi.

V poslednom rade by som taktiež rád poďakoval všetkým kolegom z Ústavu Informatiky a Matematiky za vynikajúce a inšpiratívne pracovné prostredie.

Obsah

Zoznam obrázkov a tabuliek

Zoznam výpisov

Zoznam skratiek

AE	Adversarial Examples
AEP	Asymptotic Equipartition Property
AMAO	Adversarial Malware Alignment Obfuscation
ANN	Artificial Neural Network
API	Application Programming Interface
BOFM	Bounded Feauture Space Behavior Modeling
C2	Command & Control
CELOE	Class Expression Learner for Ontology Engineering
CFG	Control Flow Graph
ClaMP	Classification of Malware with PE headers
CLR	Common Language Runtime
COFF	Common Object File Format
CPU	Central Processing Unit
CSV	Comma-Separated Values
CWA	Closed World Assumption
DDOS	Distributed Denial of Service
DL	Description Logics
DLL	Dynamic Link Library
DNS	Domain Name System
DOS	Denial of Service
ELTL	EL Tree Learner
EMBER	Elastic Malware Benchmark for Empowering Researchers
FCM	Frequency Centralized Model
FGM	Fast Gradient Method
FN	False Negative
FOIL	First Order Inductive Learner
FP	False Positive
FURIA	Fuzzy Unordered Rule Induction Algorithm
GADGET	Generative API Adversarial Generic Example by Transferability
GAN	Generative Adversarial Network
GMPC	Greedy minimise partial definition count

GMPL	Greedy minimise pardial definition length
GOLR	Greedy online algorithm - first in first out
Grad-CAM	Gradient-weighted Class Activation Mapping
HIT	Hybrid Image Transformation
HPC	Hardware Performance Counter
HTML	HyperText Markup Language
HTTP	Hypertext Trasnfer Protocol
I-REP	Incremental Reduced Error Pruning
IAT	Import Address Table
ILP	Inductive Logic Programming
IoC	Indicators of Compromise
IP	Internet Protocol
IRI	International Resource Identifier
IRL	Inductive Rule Learner
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbors
KOBG	Kernel Object Behavioral Graph
LIME	Local Interpretable Model-agnostic Explanations
LMDB	Lightning Memory-Mapped Database Manager
MAEC	Malware Attribute Enumeration and Characterization
MBO	Malicious Behavior Ontology
NP	Nondeterministic Polynomial-time
OCEL	OWL Class Expression Learner
OOA	Objective Oriented Association mining
OWA	Open World Assumption
OWL	Web Ontology Language
PARCEL	Parallel Class Expression Learner
PE	Portable Executable
QL	Query Logic
RAM	Random Access Memory
RBM	Restricted Boltzmann Machine
RDF	Resource Description Framework
RGB	Red Green Blue
RIPPER	Repeated Incremental Pruning to Produce Error Reduction
RL	Rules Logic
SHAP	SHapley Additive exPlanations
SML	Structured Machine Learning
SMS	Short Message Service
SoReL	Sophos Reversing Labs
SPACEL	Symmetric Parallel Class Expression Learner
SPARQL	SPARQL Protocol and RDF Query Language

SQL	Structured Query Language
SQLi	Structured Query Language injection
SSH	Secure Shell
STIX	Structured Threat Information Expression
SVM	Support Vector Machines
SWRL	Semantic Web Rule Language
TLS	Thread Local Storage
TN	True Negative
TP	True Positive
UCO	Unified Cybersecurity Ontology
UML	Unified Modeling Language
UPX	Ultimate Packer for Executable
URI	Unified Resource Identifier
URL	Unified Resource Locator
USB	Universal Serial Bus
XML	eXtensible Markup Language
XSS	Cross Site Scripting
YARA	Yet Another Recursive Acronym

Úvod

Rozvoj informačných technológií a internetu v posledných desaťročiach so sebou priniesol aj mnohé hrozby pre podniky, vládne inštitúcie alebo používateľov. Jednou z takýchto hrozieb je aj škodlivý kód (malvér), ktorý sa dokáže vďaka dostupnosti internetu šíriť oveľa rýchlejšie a jednoduchšie ako v minulosti. Podľa najnovších štatistík sa denne objavuje až 450 000 nových vzoriek, pričom od roku 1984 bolo zaznamenaných až 1,5 miliardy škodlivých súborov [1]. Veľké množstvo malvéru so sebou prinieslo potrebu vyvíjať rôzne detekčné mechanizmy, ktoré by boli schopné rozlišovať škodlivý kód od legitímneho. V posledných rokoch sa ako najúspešnejšie riešenie javilo použitie algoritmov strojového učenia, ktoré dokážu spracovávať veľké množstvo dát a zároveň dosahovať vysokú úspešnosť pri detekcii. Napriek spomínaným výhodám, dané algoritmy majú aj určité nevýhody. Jednou z takýchto nevýhod je, že väčšina algoritmov nie je vysvetliteľná a fungujú ako čierna skrinka v zmysle, že nevieme pochopiť rozhodnutie detekčného systému, ktorý označil vzorku za škodlivú. Vysvetliteľnosť a schopnosť dôverovať rozhodnutiam algoritmom strojového učenia sa stala dôležitou výskumnou témou v tejto oblasti, najmä v posledných rokoch, kedy sa strojové učenie dostáva do kritických oblastí ako sú medicína alebo finančný sektor.

Táto práca sa zaoberá výskumom ontológií a sémantických technológií a ich využitím v oblasti počítačovej bezpečnosti pri vývoji vysvetliteľných systémov na detekciu škodlivého kódu. Jadro práce tvorí výskum algoritmov konceptového učenia (algoritmy, ktoré sa dokážu učiť nad ontológiami) a ich doposiaľ nepreskúmaná aplikácia pri detekcii škodlivého kódu. Vedecké ciele práce môžeme zhrnúť do nasledujúcich bodov:

1. Návrh ontológie pre škodlivé PE súbory (pre operačný systém *Windows*) spolu s vytvorením a publikovaním štandardizovaných datasetov založených na spomínanej ontológii. Hlavným cieľom tejto tézy bolo vytvoriť dataset s interpretovateľnými vlastnosťami a zároveň poskytnúť možnosť jednoduchšej reprodukcie výsledkov s cieľom efektívnejšieho porovnávania rôznych algoritmov (nielen pre konceptové učenie). Výsledky boli publikované v [2, 3, 4] (kapitola 4).

2. Výskum existujúcich algoritmov konceptového učenia nad nami navrhnutou ontológiou. Cieľom bolo preskúmať jednotlivé algoritmy a overiť ich úspešnosť pri detekcii škodlivého kódu pri rôznych experimentálnych nastaveniach. Výsledky boli publikované v [5, 6, 7, 8] (kapitoly 5.3 až 5.7).

3. Analýza bezpečnostných aspektov konceptových výrazov (výsledkov z

konceptového učenia). Cieľom bolo preskúmať odolnosť konceptového učenia voči rôznym útokom, ktorých cieľom je pomýliť klasifikátor (t.j. označiť škodlivý kód ako legitímny softvér). Výsledky sú popísané v kapitole 5.8.

Celkovo je práca členená do piatich na seba nadväzujúcich kapitol:

- **Kapitola 1**: Kapitola sa venuje problematike detekcie malvéru. V rámci tejto kapitoly sa venujeme popisu malvéru, existujúcim riešeniam založených na rôznych prístupoch a taktiež aj najznámejším datasetom (vrátane nami použitého datasetu EMBER).
- **Kapitola 2**: V tejto časti sa venujeme teoretickým základom ontológií a deskripčných logík spolu so všeobecným úvodom do konceptového učenia.
- **Kapitola 3**: Kapitola sa venuje do hĺbky jednotlivým algoritmom konceptového učenia, ktoré boli skúmané v rámci prace: OCEL, CELOE, PARCEL a SPACEL. V rámci tejto kapitoly uvádzame aj ďalšie existujúce algoritmy konceptového učenia a iné vysvetliteľné metódy.
- **Kapitola 4**: V kapitole sa venujeme návrhu našej ontológie spolu s motiváciou, ktorá stála za jej vývojom a taktiež jej jednotlivým častiam.
- **Kapitola 5**: Posledná kapitola sa venuje praktickej časti tejto práce, kde popisujeme všetky vykonané experimenty spolu s interpretáciou dosiahnutých výsledkov.

Kapitola 1

Detekcia malvéru

Táto časť práce sa venuje problematike detekcie malvéru. V úvodnej kapitole 1.1 si v stručnosti charakterizujeme škodlivý kód a jeho jednotlivé kategórie. V kapitole 1.2 sa venujeme definícii samotného problému detekcie škodlivého kódu spolu s výzvami, ktorým čelí vedecká komunita v tejto oblasti. Následne, kapitola 1.3 popisuje súčasný stav v oblasti detekcie malvéru. Venuje sa štyrom základným prístupom, ktoré sa najčastejšie používajú v tejto oblasti: signatúry, behaviorálne metódy, heuristika a hlboké učenie. Kapitola 1.4 popisuje ďalšie prístupy v tejto oblasti, avšak zamerané na vysvetliteľné metódy, ktoré ponúkajú možnosť určitým spôsobom interpretovať jednotlivé výsledky detekčných modelov. Ďalej, v kapitole 1.5 popisujeme formát spustiteľných PE súborov, keďže základom našej práce boli statické dáta. Dataset používaný v rámci našej práce je následne popísaný v kapitole 1.6. Ďalšie datasety v danej oblasti (ktoré sme však nepoužívali v rámci práce) sú popísané v kapitole 1.7. Kapitola 1.8 sa venuje dôležitej oblasti bezpečnosti klasifikátorov malvéru, pričom sa venuje rôznym formám útokov. Posledná kapitola 1.9 sa venuje prácam, ktoré použili rôzne semantické technológie na detekcie malvéru (resp. príbuzné oblasti v rámci počítačovej bezpečnosti).

1.1 Malvér

Malvér (z angl. *malware*, resp. *malicious software*), môžeme charakterizovať ako ľubovoľný typ softvéru, ktorý bol navrhnutý s cieľom uškodiť používateľovi alebo systému [9]. Škody, ktoré môže malvér spôsobiť typicky zahŕňajú krádež dát a citlivých údajov, sledovanie aktivity používateľa, získanie kontroly na počítačom obete a využitie zdrojov na ďalšie útoky, šifrovanie dát za účelom finančného vydierania a pod.

Spôsobov, akými sa dokáže škodlivý kód dostať ku koncovému používateľovi je viacero. Ako uvádza report od firmy ESET [10], v súčasnosti je stále najpoužívanejším spôsobom phishingový email. Útočníci dokážu takýmto spôsobom napr. poskytnúť obeti prílohu so škodlivým HTML kódom (ktorý je častejšie používaný v porovnaní so štandardnými spustiteľnými súbormi, keďže tie zvyknú byť automaticky blokované). Škodlivý kód môže následne presmerovať obeť na phishingovú stránku, tváriacu sa napr. ako legitímna stránka banky alebo sociálnej siete, kde obeť vyplní svoje prihlasovacie údaje (ktoré sa prepošlú útočníkovi). Častým spôsobom šírenia je taktiež prostredníctvom rôznych exploitov, ktoré zneužívajú zraniteľnosti v softvéri. Takýmto spôsobom dokáže útočník podhodiť obeti napr. špeciálne vytvorený dokument, ktorý zneužíva zraniteľnosť v editore dokumentov. Následným otvorením dokumentu sa spustí útočníkov kód, ktorý môže vykonávať ľubovoľné akcie ako napr. vytvorenie perzistentného spojenia s počítačom obete alebo stiahnutie malvéru. Je nutné poznamenať, že šírenie prostredníctvom exploitov nemusí vždy prebiehať pomocou akcie používateľa ale napr. aj samotným navštívením nebezpečnej stránky alebo prítomnosťou zraniteľnej služby na počítači obete, ktorá beží na pozadí (v takom prípade sa jedná o tzv. *zero click* exploit [11]). Medzi ďalšie spôsoby prenosu malvéru patria obfuskované JavaScript súbory, *Microsoft Office* dokumenty obsahujúce škodlivé makrá, ktoré môžu sťahovať malvér a pod.

V nasledujúcej časti si stručne popíšeme niektoré základné kategórie malvéru. Je však nutné poznamenať, že jednotlivé kategórie predstavujú skôr typy správania. Malvér v súčasnej dobe predstavuje komplexný typ softvéru a každá nová vzorka je väčšinou kombináciou viacerých typov správania [9]:

Vírusy a červy: označujú typ správania, ktorý definuje spôsob, akým sa malvér dokáže šíriť. Vírusy využívajú na šírenie hostiteľské programy. Typicky si vyžadujú určitú formu používateľskej interakcie (kliknutie na emailovú prílohu alebo vloženie USB). Samotný vírus sa vloží do hostiteľského programu (alternatívne aj zmení svoju formu) a automaticky sa spustí pri spustení hostiteľského programu. Červy, na druhú stranu, sa typicky šíria bez akejkoľvek používateľskej interakcie, napr.

prostredníctvom siete, pričom zneužívajú softvérové zraniteľnosti.

Trójske kone: označujú typ správania, ktorý definuje spôsob, akým sa malvér maskuje. Cieľom autorov malvéru, je pochopiteľne vyhnúť sa detekcii a z toho dôvodu aplikujú rôzne metódy maskovania v podobe emailovej prílohy, webstránky alebo maskovaním sa za antivírusový program či iný legitímny softvér. Komplexné trójske kone môžu dokonca vykonávať legitímnu činnosť po spustení, pričom škodlivá časť sa môže vykonávať na pozadí bez vedomia používateľa (resp. sa môže automaticky aktivovať po určitom časovom období).

Rootkit: jedná sa o typ programu, ktorý slúži na maskovanie prítomnosti malvéru v počítači. Cieľom rootkitov je typicky manipulovať zoznam spustených procesov (tak, aby proces, v ktorom beží malvér, nebol viditeľný), schovávať spustiteľné súbory alebo iné súbory, ktoré vytvoril škodlivý kód alebo maskovať prítomnosť niektorých konkrétnych sieťových spojení (typicky tie, ktoré používa útočník). Samotné rootkity môžu bežať v používateľskom priestore ako aj v priestore jadra (vtedy má útočník väčšie možnosti).

Botnet: označuje sieť počítačov (botov), ktoré sú pod kontrolou útočníka a riadi ich pomocou C2 servera. Cieľom útočníkov je ovládnuť čo najväčší počet počítačov, za účelom získania výpočtovej sily. Takáto výpočtová sila sa následne môže využiť na viacero cieľov v podobe hromadného rozposielania malvéru a spamu, DOS a DDOS útoky alebo na ťažbu kryptomien.

Ransomvér: jedná sa o typ malvéru, ktorého cieľom je vydierať používateľa za účelom zisku. Po infekcii ransomvér typicky zašifruje obsah disku a od obete žiada finančnú odmenu (v podobe kryptomeny) výmenou za kryptografický kľúč, ktorý dokáže dešifrovať dáta. Okrem šifrovania dát sa zvyknú útočníci vyhrážať aj zničením dát alebo využitím techniky *doxing* (v prípade, ak obeť nezaplatí, všetky dáta budú zverejnené na internete).

Okrem vyššie spomínaných typov existujú aj ďalšie ako napr. **keylogger** (zaznamenávanie stlačených kláves), **backdoor** (zadné vrátka - perzistentný prístup k počítaču obete) alebo **dropper** (sťahovanie a inštalovanie malvéru). Ako príklad si môžeme uviesť známy ransomvér *WannaCry* z roku 2017 [12]. Okrem samotnej funkcionality ransomvéru sa šíril prostredníctvom siete a exploitov v podobe počítačového červa, pričom taktiež vykazoval známky trójskeho koňa a poskytoval zadné vrátka útočníkovi.

1.2 Definícia problému

Formálna definícia problému detekcie škodlivého kódu neexistuje, môžeme si však samotný problém definovať neformálne. Majme klasifikátor škodlivého kódu $D(x)$, kde x je ľubovolný program. $D(x)$ vracia hodnotu $y \in \{1, 0\}$, kde hodnota 1 definuje, že program x je malvér a hodnota 0 definuje, že program x môžeme považovať za legitímny softvér. Ako sa ukázalo vo výskume, navrhnúť klasifikátor $D(x)$, ktorý by dokázal správne detegovať všetok škodlivý kód, bez falošne pozitívnych vzoriek je nemožné. Samotný fakt sa dá ukázať pomocou dôkazu sporom [13]. Majme vyššie spomínaný klasifikátor $D(x)$. Predpokladajme, že máme program x', ktorý má nasledovnú funkcionalitu. Ak $D(x')$ vráti 1, tak program ukončíme. Ak však vráti 0, vykonáme ľubovoľnú škodlivú aktivitu. Následne môžu nastať dve situácie pri klasifikovaní programu x'. Ak $D(x') = 1$, tak x' sa ukončí, t.j. nejedná sa o škodlivý kód. V druhom prípade, ak $D(x') = 0$, tak x' spustí škodlivú funkcionalitu, t.j. jedná sa o škodlivý kód. V oboch prípadoch klasifikátor D zlyhal.

Samotná detekcia malvéru je ťažký problém aj z toho hľadiska, že dôležitú úlohu zohráva aj kontext, v akom program beží a nie len samotná funkcionalita. Predstavme si program, ktorý obsahuje funkcionalitu predstavujúcu komunikáciu s C2 serverom (napr. ak máme botnet). Vo všeobecnosti je možné takýto program považovať za škodlivý. V realite však samotný server útočníka už nemusí fungovať, čo znamená, že daný program nedokáže prijímať príkazy, t.j. vykonávať skutočnú škodlivú činnosť. Podobný prípad môžeme vnímať aj v podobe SSH serverovej aplikácie, ktorá je vo všeobecnosti vnímaná ako legitímny softvér. Samotná aplikácia však teoreticky mohla byť inštalovaná útočníkom, za cieľom získania perzistentného sieťového spojenia s napadnutým systémom. V takom prípade môžeme považovať aplikáciu za škodlivú.

Ďalší aspekt, ktorý sťažuje detekciu malvéru je snaha útočníkov obísť detekciu (resp. sťažiť aj samotnú analýzu). Malvér, ktorý neimplementuje žiadnu z takýchto techník môže vyzerať na prvý pohľad podobne ako legitímny súbor. Do kategórie techník, ktorých cieľom je skomplikovať detekciu a analýzu môžeme zaradiť napr. šifrovanie škodlivej sekcie kódu [14], polymorfné (časť kódu, ktorá má za úlohu dešifrovať škodlivý blok sa dynamicky mení pri každej infekcii) a metamorfné techniky (bez šifrovania, pri každej infekcii mení svoju štruktúru) [15] alebo použitie nástroja na zbalenie binárky (pomocou šifrovania alebo kompresie) [16].

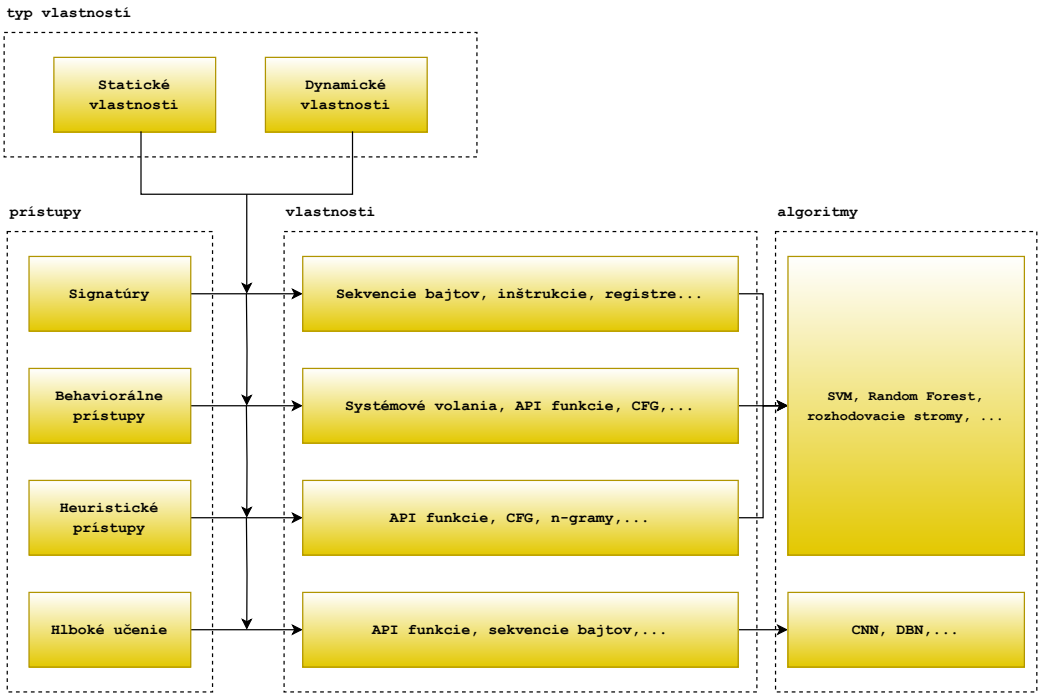
Samotnú detekciu sťažuje aj fenomén degradácie modelu (tzv. *concept drift*) [17]. Zatiaľ čo v iných doménach z oblasti strojového učenia sa predpokladá nemenná distribúcia dát, t.j. že dáta sa časom nemenia, pri škodlivom kóde to neplatí. V aplikačnej doméne ako je napr. počítačové videnie (resp.

aj ďalšie domény, ako spracovanie prirodzeného jazyka, rozpoznávanie hovorenej reči atď.) platí, že charakter datasetu sa časom nemení, napr. pri rozpoznávaní áut bude stále jasne definované, ako vyzerá auto (resp. to bude platiť dostatočne dlhý čas). V doméne detekcie malvéru môžeme degradáciu modelu charakterizovať ako postupný pokles úspešnosti v priebehu času, keďže vznikajú stále nové a nové škodlivé vzorky. Taktiež však pribúda aj množstvo regulérnych súborov. Zatiaľ ako jediným riešením proti degradácii sa javí pravidelné pretrénovanie modelu, čo však môže byť obzvlášť náročné, najmä v produkčnom prostredí.

Dôležitou výzvou v oblasti detekcie malvéru sa v posledných rokoch stala aj vysvetliteľnosť [18]. Mnoho prístupov (ďalej predstavené v kapitole 1.3) dosahuje vysoké percento úspešnosti pri detekcii, avšak fungujú ako čierne skrinky, t.j. nevieme prečo sa daný klasifikátor rozhodol označiť konkrétnu vzorku ako malvér. Detekčné modely, ktoré ponúkajú určitý pohľad dovnútra a umožňujú pochopiť jednotlivé rozhodnutia, sa javia ako vhodné pre bezpečnostných expertov. Pri vysvetliteľných klasifikátoroch dokáže bezpečnostný expert pochopiť, akým spôsobom model označil vzorku ako falošne pozitívnu a na základe toho spraviť finálne rozhodnutie, resp. použiť túto informáciu na vylepšovanie modelu.

1.3 Detekcia malvéru

V rámci tejto podkapitoly si popíšeme súčasný stav v oblasti detekcie škodlivého kódu. Schematické znázornenie jednotlivých prístupov, ktoré si predstavíme, môžeme vidieť na obrázku 1.1. Ako môžeme vidieť, postupne si prejdeme štyri základné prístupy: signatúry, behaviorálne prístupy, heuristické prístupy a riešenia založené na hlbokom učení. Všeobecne, všetky existujúce riešenia využívajú dva typy vlastností: statické a dynamické. Statické vlastnosti patria medzi atribúty, ktoré dokážeme zo vzoriek získať bez samotného spustenia (napr. sekvencie bajtov, importované API volania, hlavičky, atď.). Na získanie dynamických vlastností je naopak potrebné dané vzorky spustiť (sem môžeme zaradiť vlastnosti ako API volania aj s argumentami, vykonané systémové volania, atď.). Niektoré prístupy používajú aj tzv. hybridné vlastnosti, ktoré sú väčšinou kombináciou statických a dynamických vlastností. Na samotnú detekciu sa následne vo veľkej väčšine prístupov používajú rôzne algoritmy strojového učenia.



Obrázok 1.1: Znázornenie existujúcich detekčných prístupov spolu s používanými vlastnosťami a algoritmami.

1.3.1 Signatúry

Detekcia škodlivého kódu pomocou signatúr patrí medzi metódy, ktoré našli svoje najväčšie využitie najmä v antivírusových riešeniach. Základom techník je vytvorenie tzv. signatúry škodlivého súboru a jej uloženie do databázy. Následne, keď chceme overiť neznámy súbor, vyextrahujeme signatúru rovnakým spôsobom a porovnáme ju so signatúrami v databáze. Hlavnými výhodami metódy je rýchlosť detekcie a niektoré techniky dokážu pomocou jednej signatúry odhaliť viacero vzoriek z rovnakej rodiny. Zrejmou nevýhodou je nutnosť vytvárať rozsiahle databázy a taktiež je metóda zraniteľná na rôzne formy obfuskácie. Ako signatúry sa využívali najmä sekvencie bajtov z rôznych častí programu ako napr. hlavička a vstupný bod programu či kryptografický kontrolný súčet v podobe MD5 alebo SHA256 [19].

Napriek tomu že sa jedná o jednu z najstarších metód, v poslednom desaťročí stále vznikali nové a nové prístupy generovania signatúr, využívajúce rôzne vlastnosti spustiteľných súborov spolu s algoritmami strojového učenia. V práci [20] autori vytvorili systém *MalHunter*, ktorý generuje signatúry, zachytávajúce správanie programu. Autori najprv vygenerovali množinu rôznych sekvencií správania zo škodlivých vzoriek (vytvorenie súboru, čítanie registrov, otvorenie mutexu a pod.). Následne tieto sekvencie zoskupili do rôznych klastrov, ktoré boli používané ako signatúry pri porovnávaní sekvencií z neznámych vzoriek. V porovnaní s inými prístupmi, dokáže jedna takáto signatúra zachytiť viacero vzoriek a taktiež si dokáže poradiť s polymorfným malvérom. V práci [21] predstavili autori vytvorenie trojúrovňovej signatúry pre aplikácie na operačný systém *Android*. Samotný heš bol zložený zo zoradenej sekvencie API volaní a názvov tried, ktoré sa nachádzali v aplikácii (ktoré boli tiež zahešované). Baldangombo a spol. [22] vo svojej práci využili techniky na dolovanie dát a použili statické vlastnosti zo spustiteľných súborov ako sú napr. informácie z PE hlavičky, importované DLL knižnice či API volania. Spomínané vlastnosti použili na natrénovanie klasifikátorov pomocou SVM alebo rozhodovacích stromov a dosiahli tak úspešnosť 99.6%. V práci [23] naopak využili techniky na dolovanie textových vzorov z postupnosti inštrukcií, extrahovaných zo spustiteľných súborov. Na hľadanie podobností medzi sekvenciami inštrukcií bol použitý zhlukovací algoritmus *DBScan*. Zaujímavý prístup bol taktiež predstavený v novšej práci [24], kde autori použili na generovanie signatúr množinu registrov, nachádzajúcich sa v moderných procesoroch, ktoré poskytujú rôzne informácie o hardvéri, tzv. Hardware Performance Counter (HPC) ako sú napr. čas strávený v špecifickej vetve, informácie o vyrovnávacej pamäti a pod. Primárne sú tieto registre využívané na ladenie, verifikáciu či sledovanie výkonu.

1.3.2 Behaviorálne prístupy

Ďalšia množina prístupov k detekcii malvéru je založená na tzv. behaviorálnej analýze. V týchto metódach väčšinou nachádzajú využitie rôzne vlastnosti systému, prostredníctvom ktorých vieme popísať škodlivé správanie programu. Medzi najpoužívanejšie vlastnosti môžeme zaradiť monitorovanie systémových volaní, monitorovanie súborového systému (počet zápisov/čítaní alebo počet novovzniknutých súborov), sledovanie zmien v hodnotách registrov alebo sledovanie sieťovej prevádzky či bežiacich procesov. Behaviorálne prístupy sa teda orientujú najmä na dynamické vlastnosti, ktoré vieme extrahovať iba spustením malvéru a sledovaním jeho aktivity. Po extrahovaní dynamických vlastností sa na samotnú klasifikáciu najčastejšie používajú algoritmy strojového učenia. Výhoda behaviorálnych prístupov je, že dokážu detegovať aj nové a neznáme vzorky (najmä v porovnaní so signatúrami) a do istej miery si dokážu poradiť aj s obfuskáciou, keďže tá samotné správanie kódu nemení. Na druhej strane môže byť náročné špecifikovať všetky druhy škodlivého správania a správne rozlíšiť hranicu medzi škodlivým a legitímnym správaním.

V práci [25] definovali škodlivé správanie ako špeciálny graf nazvaný Kernel Object Behavioral Graph (KOBG), kde jednotlivé vrcholy reprezentujú rôzne objekty v jadre operačného systému, ktoré medzi sebou komunikujú počas vykonávania škodlivého kódu. Z viacerých vzoriek z rovnakej rodiny vytvorili všeobecný KOBG graf a samotná klasifikácia bola následne uskutočnená ako hľadanie podgrafu. Podobnú metódu aplikoval aj Ding a spol. [26], avšak použili graf so systémovými volaniami, spolu s ich parametrami. Das a spol [27] naopak zoskupili samotné postupnosti systémových volaní v operačnom systéme *Linux* a vytvorili tak škodlivé/legitímne vlastnosti pomocou techník *n-gramov* a Frequency Centralized Model (FCM). Na samotnú klasifikáciu použili štandardné algoritmy strojového učenia v podobe viacvrstvovej perceptrónovej siete, rozhodovacích stromov či metódy podporných vektorov. V práci [28] využili štandardné dynamické dáta získané zo sandboxu pre analýzu malvéru *CWSandbox*[1]. Výstup v podobe XML reportu bol následne transformovaný na numerickú maticu a použitý ako vstup do viacerých algoritmov strojového učenia. Autori v práci [29] predstavili tzv. Bounded Feauture Space Behavior Modeling (BOFM), ktorý modeluje správanie programov ako interakciu medzi procesmi a kritickými zdrojmi operačného systému ako sú registre či súborový systém. V samotnom rámci okrem iného predstavili aj spôsob akým možno ohraničiť vektor vlastností na hornú hranicu N, keďže v prípade dynamických vlastností dĺžka vektoru do veľkej miery korešponduje s dĺžkou času, počas ktorého je monitorovaný škodlivý kód. Pajouh a spol. [30]

[1] http://cwsandbox.org/

použili kombináciu statických a dynamických vlastností (v podobe frekvencie API volaní) extrahovaných z malvéru a regulárnych súborov pre operačný systém *Mac OS*. Na klasifikáciu použili algoritmus SVM a dosiahli úspešnosť približne 91%. V práci [31] predstavili autori novú reprezentáciu správania programu, nazvanú *CFG-API*. Tá kombinuje Control Flow Graph (CFG), reprezentujúca graf, kde jednotlivé vrcholy sú bloky kódu a hrany predstavujú operácie, ktoré majú za následok zmenu toku programu (priame/nepriame skoky, volania funkcie a pod.) a informácie ohľadom API volaní. Takýto graf bol následne prevedený do vektorovej reprezentácie a použitý ako vstup do algoritmov strojového učenia. Mosli a spol. [32] použili ako základ ich behaviorálneho prístupu identifikátory súborov, ktoré sa používajú v operačnom systéme *Windows* na identifikáciu súborov, registrov, mutexov, vláken a procesov. Tieto informácie extrahovali pomocou sandboxu *Cuckoo*[2], vektorizovali a použili na trénovanie algoritmov KNN, SVM a *Random Forest*. Behaviorálnu detekciu malvéru pre operačný systém *Windows* sme mohli vidieť aj v práci [33], kde využili ako základ súbory, ktoré používa *Windows* na prednačítanie rôznych zdrojov (používajú sa najmä na urýchlenie behu aplikácií) spolu s lineárnou regresiou a SVM.

1.3.3 Heuristické prístupy

Medzi ďalšie populárne metódy v doméne detekcie malvéru patria heuristické prístupy, ktoré používajú rôzne algoritmy na dolovanie dát a algoritmy strojového učenia s podporou pravidiel [34]. Ako vstupné vlastnosti sa najčastejšie používajú API volania, n-gramy, CFG, prípadne hybridné kombinácie statických a dynamických vlastností [35]. V porovnaní so signatúrami, nie je potrebné udržovať rozsiahle databázy a tak dokážu detegovať aj neznáme vzorky. Vo veľkej väčšine prístupov taktiež nie je nutné plné trasovanie spustiteľného súboru tak, ako pri behaviorálnych metódach. Proces tvorby klasifikátorov je však komplikovanejší a taktiež do veľkej miery si nedokáže poradiť so vzorkami, využívajúcimi komplikovanú obfuskáciu.

V práci [36] predstavili metódu založenú na Markovových reťazcoch, kde jednotlivé vrcholy reprezentovali inštrukcie a hrany naopak prechodové pravdepodobnosti získané zo spustenia škodlivých vzoriek. Následne skonštruovali maticu podobností a použili algoritmus SVM ako nástroj na klasifikáciu. Islam a spol. [37] využili hybridnú kombináciu statických a dynamických vlastností a natrénovali klasifikátory pomocou algoritmov SVM, *Random Forest* a rozhodovacích stromov. Naval a spol. [38] naopak vytvorili klasifikátor založený na systémových volaniach škodlivého kódu. V ich práci špecifikovali sémantické

[2]https://cuckoosandbox.org/

cesty v grafe systémových volaní, reprezentujúce určité správanie programu pomocou techniky Asymptotic Equipartition Property (AEP) a použili ich spolu s algoritmom *Random Forest*. Autori navyše uviedli, že ich klasifikátor je odolný voči útokom, ktoré na obídenie detekcie injektujú ďalšie systémové volania do pôvodnej škodlivej vzorky tak, aby nenarušili pôvodnú funkcionalitu. V práci [39] predstavili autori klasifikátor založený na pravidlách, ktoré boli extrahované pomocou techniky Objective Oriented Association mining (OOA). Ich metóda spočívala v statickom extrahovaní API volaní zo škodlivej vzorky a použití algoritmov na dolovanie asociačných pravidiel. V práci testovali tri algoritmy: *FP-Apriori*, *FP-Growth* a *Fast-FP-Growth*. Autori práce taktiež porovnali naučené pravidlá so štandardnými algoritmami strojového učenia ako sú *Naive Bayes* či SVM a dosiahli lepšie výsledky. Zakeri a spol. [40] predstavili riešenie, v ktorom pomocou algoritmov dolovania dát vyextrahovali statické vlastnosti zo škodlivých vzoriek (s najväčším príspevkom k informačnému zisku) a aplikovali učiace algoritmy Fuzzy Unordered Rule Induction Algorithm (FURIA) a Inductive Rule Learner (IRL). V samotnej práci sa okrem detekcie malvéru zamerali aj na rozlišovanie či je spustiteľný súbor zbalený alebo nie (baliace algoritmy sú väčšinou využívané škodlivým kódom na ochranu voči detekcii antivírusovými riešeniami), kde dosiahli úspešnosť 99.7% pomocou fuzzy pravidiel. Khodamoradi a spol. [41] využili ako základ ich detekcie štatistické vlastnosti operačných kódov (v bajtoch zakódované inštrukcie procesora), kde analyzovali 227 inštrukcií z 12 kategórií. Na klasifikáciu využili následne algoritmus *Random Forest* a dosiahli vysokú úspešnosť, najmä pre detekciu samomodifikujúcich sa škodlivých programov. V práci [42] predstavili autori systém *AdDroid*, detekčný nástroj pre operačný systém *Android* založený na pravidlách. Ako základ použili statické vlastnosti z aplikácií, pomocou ktorých definovali 63 pravidiel, reprezentujúcich rôzne udalosti, ktoré môže aplikácia vykonať, ako napr. prístup k internetu, načítanie externých knižníc, prístup k posielaniu správ a pod. Tieto pravidlá následne použili ako vstup na trénovanie klasifikátora pomocou rozhodovacích stromov a dosiahli tak úspešnosť 99.11%. Zaujímavý prístup prezentovali autori v práci [43], kde sa zamerali priamo na bezpečnosť a implementovali tak tzv. monotónny klasifikátor. Autori využili ako základ kombináciu statických a dynamických vlastností spolu s rozhodovacími stromami. Z rôznych vlastností však vybrali iba také, ktoré sú pre útočníkov ťažké na odobranie alebo pridanie do pôvodnej vzorky, čím sa dosiahne vyššia bezpečnosť klasifikátora. Príkladom vlastností, ktoré sa rozhodli nezahrnúť, sú rôzne statické dáta z PE súboru, IP adresy alebo názvy mutexov. Takéto vlastnosti sú pre útočníkov jednoduché na zmenu. Naopak ponechali vlastnosti ako digitálny podpis spustiteľného súboru, operácie so súbormi či registrami, t.j. vlastnosti ktorú sú pre útočníkov netriviálne na zmenu,

ak chcú ponechať pôvodnú škodlivú funkcionalitu. Klasifikátor trénovali na datasete s 1 miliónom vzoriek a dosiahli celkovú úspešnosť 62% a pozorovali tak pokles úspešnosti o približne 13% (v porovnaní s klasifikátorom, ktorý natrénovali pomocou všetkých vlastností).

1.3.4 Hlboké učenie

Hlboké učenie patrí medzi odbor strojového učenia, ktorý získal obrovskú popularitu v rámci iných domén ako sú počítačové videnie či spracovanie prirodzeného jazyka. V posledných rokoch tak pribudlo aj väčšie množstvo prác, ktoré sa snažili využiť metódy hlbokého učenia aj v rámci detekcie malvéru. Zatiaľ čo algoritmy hlbokého učenia sa ukázali ako rýchle a efektívne a do istej miery redukujúce množstvo vlastností potrebných na extrakciu zo škodlivých vzoriek (v niektorých prácach sa aj samotná extrakcia vlastností ponecháva na algoritmus), proces trénovania modelu je značne náročný a taktiež množstvo prác ukázalo, že hlboké siete sú obzvlášť zraniteľné voči špeciálne útočníkmi pripraveným vzorkám (viac v kapitole 1.8).

V práci [44] použili ako vstup vektor veľkosti 1024, ktorý obsahoval histogram bajtov a reťazcov, importované funkcie z PE súboru a rôzne metadáta z hlavičky. Tento vektor následne použili ako vstup do hlbokej ANN siete s dvoma skrytými vrstvami. Na trénovanie modelu použili spolu 350 000 škodlivých vzoriek a 80 000 legitímnych súborov. Finálny model dosahoval úspešnosť 95%. Huang a Stokes [45] predstavili paralelnú architektúru hlbokej siete *MtNet*, ktorá okrem rozlišovania škodlivých a neškodlivých súborov, dokáže taktiež klasifikovať vstupný súbor do 100 rôznych rodín. Samotná sieť bola zložená zo 4 skrytých vrstiev, pričom vstup tvoril vektor o veľkosti 50 000. Ako vstupné dáta použili sekvenciu API volaní z dynamickej analýzy, spolu s ich vstupnými parametrami. Na trénovanie siete použili dataset o veľkosti 4.5 milióna vzoriek a dosiahli úspešnosť cez 90%. V práci však taktiež ukázali, že *MtNet* je zraniteľná na rôzne útoky. Podobne aj v práci [46] použili na trénovanie API volania spolu so špeciálnou hlbokou sieťou Restricted Boltzmann Machine (RBM). Na trénovanie použili približne 10 000 vzoriek, kde sa nachádzali aj neoznačené vzorky. Model dosahoval presnoť 98.82%. Zhu a spol. [47] predstavili *DeepFlow* model pre operačný systém *Android*. Na vytvorenie vstupného vektoru použili aplikáciu *FlowDroid*, ktorá staticky extrahuje tok citlivých údajov v aplikácii ako napr. tok lokalizačných údajov cez SMS správu na konkrétne číslo a pod. Ako model využili štandardnú štvorvrstvovú hlbokú sieť, trénovanú pomocou 11 000 vzoriek. Úspešnosť modelu dosiahla 95.05%. Špeciálny prístup k detekcii bol publikovaný v práci [48]. Nataraj a spol. predstavili nový spôsob reprezentácie malvéru, ktorý spočíval v prevedení spustiteľného súboru na obrázok v šedej škále. Samotné hodnoty bajtov od 0 až po 255 previedli na RGB hodnoty pixelu, čím vznikol

obrázok. Ich prístup bol motivovaný najmä úspechmi algoritmov počítačového videnia. Zatiaľ čo pôvodná práca používala Gaborov filter, novšie práce už využívali moderné konvolučné siete [49, 50]. Spomínané prístupy sa však najviac osvedčili pri klasifikácii podobností vzoriek malvéru z rovnakých rodín. Raff a spol. [51] prezentovali architektúru *MalConv* v ktorej použili priamo celý spustiteľný súbor (v podobe dlhej sekvencie bajtov) ako vstup do konvolučnej neurónovej siete. Hlavnou motiváciou výskumu bolo otestovať, či sa dokážu konvolučné vrstvy v sieti naučiť rôzne vzorce odpozorované na vstupe, podobne ako na obrázkoch. Zatiaľ čo výsledný model, trénovaný pomocou 400 000 vzoriek, vykazoval relatívne vysokú úspešnosť (najlepší experiment dosiahol 94%), samotný návrh architektúry sa ukázal ako extrémne zraniteľný aj voči jednoduchým útokom. Rovnaký prístup zvolil aj Krčál a spol. [52], ktorí navrhli architektúru so 4 konvolučnými vrstvami a 4 plne prepojenými neurónovými vrstvami. Zvolený model trénovali na obrovskom datasete s 20 miliónmi vzoriek a dosiahli vyššiu úspešnosť ako *MalConv* (cca 97%).

1.4 Vysvetliteľné prístupy

V kapitole 1.2 sme si uviedli ako jednu z výziev pri detekcii malvéru tzv. vysvetliteľnosť. Zatiaľ čo v kapitole 1.3 sme sa venovali jednotlivým detekčným prístupom zo všeobecnejšieho hľadiska, v tejto kapitole si v stručnosti predstavíme niektoré práce, ktoré sa venovali detekcii malvéru s dôrazom na vysvetliteľnosť.

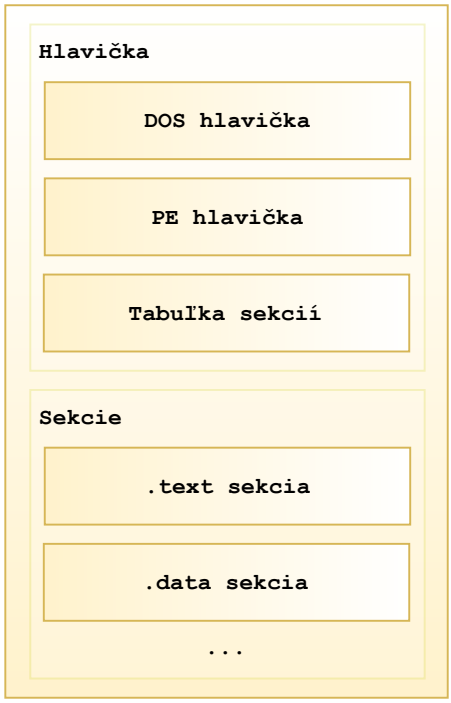
V práci [53] predstavili Mills a spol. systém nazvaný NODENS. Tento systém sa zameriava na dynamické vlastnosti, pričom ako vstupné atribúty používa vlastnosti spustených procesov v systéme ako sú napr. cesty, ktoré používa proces, veľkosť virtuálnej pamäti, čas strávený na procesore, počet otvorených súborov, mutexov a pod. Na samotné učenie autori použili algoritmus strojového učenia *Random Forest*, ktorý môžeme označiť za implicitne vysvetliteľný, aj keď ako negatívum môžeme označiť relatívne malé množstvo vzoriek, ktoré boli použité na trénovanie.

Iadarola a spol. predstavili odlišný prístup založený na hlbokom učení [54]. V práci navrhli konvolučnú neurónovú sieť, ktorú trénovali na 8000 vzorkách škodlivého kódu pre operačný systém *Android*, pričom cieľom bola klasifikácia do jednotlivých rodín. Samotné vzorky boli pred trénovaním prevedené na obrázky v šedej škále. Na poskytnutie vysvetliteľnosti následne využili Grad-CAM algoritmus, ktorý generuje tepelné mapy pre najzaujímavejšie časti vstupu, ktoré prispievajú ku klasifikácii. Hlavnou ideou autorov bolo poskytnúť určitý náhľad do modelu pre reverzných inžinierov s cieľom uľahčiť analýzu, vyznačením dôležitých častí programu. Podobný prístup predstavili aj Marais a spol. [55], pričom využili EMBER dataset a HIT metódu na predspracovanie vzoriek do obrázkov v šedej škále. Ich prístup dokáže rozpoznávať, či je daná vzorka obfuskovaná, pričom ich hlavným cieľom bolo taktiež zefektívniť analýzu malvéru.

Vysvetliteľnosti sa venovali aj Dolejš a spol. a taktiež použili EMBER dataset [56]. V ich práci trénovali viacero rôznych modelov pričom na vysvetlenie modelov použili algoritmy na extrahovanie rozhodovacích zoznamov I-REP a RIPPER [57]. Výstupom je interpretovateľná množina pravidiel, ktorá podporuje štyri operátory: $\geq$, $\leq$, $=$ a $\wedge$ (konjunkcia).

1.5 PE formát

Keďže v našej práci sme sa zamerali na statické vlastnosti malvéru v operačnom systéme *Windows*, v tejto kapitole si v stručnosti definujeme formát Portable Executable (PE) súborov [58]. Aby mohol byť súbor na operačnom systéme *Windows* reálne spustený na procesore, musí dodržiavať presný formát typu PE. Tento formát je potrebný nielen pre spustiteľné súbory, typicky s koncovkou *.exe* ale aj pre dynamické DLL knižnice (*.dll*) alebo objektové súbory (*.obj*). Samotný formát definuje, kde sa nachádzajú dôležité metadáta potrebné pre operačný systém, aby vedel program zaviesť do operačnej pamäte a spustiť ako napr. kde sa nachádza samotný kód, jeho veľkosť, oprávnenia a pod.



Obrázok 1.2: Štruktúra PE formátu.

Schematické znázornenie PE formátu môžeme vidieť na obrázku 1.2. Ako môžeme vidieť, celkovú štruktúru PE súboru tvoria dve hlavné časti: hlavičky a sekcie. Hlavička sa skladá z viacerých častí a definuje samotné metadáta spustiteľného súboru. Sekcie naopak prezentujú už reálne dáta, nahrávané do operačnej pamäti. Hlavička PE súboru sa skladá z nasledujúcich častí:

- **DOS hlavička**: jedná sa o špeciálnu časť hlavičky, ktorá bola ponechaná v rámci PE formátu kvôli spätnej kompatibilite so staršími systémami

MS-DOS. Prvé dva bajty hlavičky obsahujú špeciálnu magickú hodnotu `MZ` a používajú sa pri spúštaní na verifikáciu, či sa jedná o platný PE súbor. Okrem samotnej magickej hodnoty obsahuje DOS hlavička aj krátky kód, ktorý vypíše chybovú hlášku v prípade, ak sa pokúsime spustiť súbor na operačnom systéme *MS-DOS*.

- **PE hlavička**: podobne ako DOS hlavička, aj PE hlavička začína dvoma magickými bajtmi, v tomto prípade (a v prípade správneho formátu) obsahujú hodnoty `PE`. Samotná PE hlavička sa skladá z dvoch podhlavičiek:
  - **COFF hlavička**: obsahuje informácie o architektúre na akú bol kód kompilovaný, čas kedy bola aplikácia skompilovaná, počet sekcií v súbore a metadáta o tom či daný súbor obsahuje voliteľnú hlavičku, spolu s jej veľkosťou.
  - **Voliteľná hlavička**: z angl. *Optional Header*. V tejto hlavičke sa nachádzajú dôležité informácie o tom, aká veľká je sekcia s kódom, jej adresa v rámci súboru a kde presne sa nachádza vstupný bod programu. Ďalej obsahuje bázové adresy, kontrolný súčet, informácie, či sa jedná o konzolovú aplikáciu alebo aplikáciu s používateľským rozhraním. Taktiež hlavička obsahuje dátové priečinky (z angl. *Data Directories*), ktoré obsahujú adresy, kde sa nachádzajú niektoré dôležité dáta programu. Sem patria napr. tabuľka importovaných/exportovaných funkcií (napr. *Windows* API funkcie), tabuľka zdrojov programu (obrázky, písma a pod.), ladiace symboly, certifikát a pod.
- **Tabuľka sekcií**: poslednou časťou metadát je tabuľka s informáciami o jednotlivých sekciách. Ku každej sekcii, ktorá sa nachádza v programe, má tabuľka uložené dáta ako napr. meno sekcie, jej adresa, veľkosť a oprávnenia.

Po hlavičke sa v súbore nachádzajú už samotné sekcie PE súboru s reálnymi dátami. Medzi najčastie sekcie patria:

- **Sekcia s kódom**: v tejto sekcii sa nachádza všetok kód programu vo forme inštrukcií, ktoré sú spustiteľné na procesore a zároveň sa v nej nachádza vstupný bod programu. V prípade, ak kód používa importované funkcie z externých knižníc, tak smerníky v tejto sekcii sa odkazujú na sekciu s importmi. Vo všeobecnosti by sa malo jednať o jedinú sekciu v programe, ktorá obsahuje kód a je spustiteľná. Najčastejšie sa meno sekcie s kódom označuje ako `.text` alebo `.code`.

- **Sekcia s dátami**: jedná sa o sekciu v ktorej sa nachádzajú dáta programu. V tejto kategórii všeobecne môžeme rozlišovať štyri typy sekcií. Sekcia s neinicializovanými dátami (napr. `.bss`), kde sa nachádzajú dáta, ktoré sa inicializujú až za behu a v takom prípade by sa jednalo o neefektívne využitie miesta v spustiteľnom súbore (napr. staticky alokované veľké pole). Inicializované dáta sa môžu naopak nachádzať v sekcii `.idata`. Dáta, ktoré slúžia iba na čítanie (reťazcové literály, konštanty a pod.) sa nachádzajú v sekcii s názvom `.rdata`. Posledným typom je sekcia `.data`, kde sa nachádzajú dáta, ktoré nespadajú ani pod jednu z vyššie spomínaných kategórií.
- **Sekcia so zdrojmi**: v tejto sekcii sa nachádzajú všetky zdroje programu usporiadané v hierarchickej štruktúre. Najčastejšie sa sa takéto sekcie označujú menom `.rsrc`.
- **Sekcia s importmi**: sekcia nazývaná zvyčajne aj `.idata` obsahuje informácie o importovaných funkciách vrátane IAT tabuľky, ktorá si drží samotné adresy importovaných funkcií.
- **Sekcia s exportmi**: táto sekcia, nazývaná aj `.edata`, je platná väčšinou v prípadoch ak je PE súbor dynamická knižnica. V samotnej sekcii sa nachádzajú mená a adresy exportovaných funkcií, ktoré môžu byť po načítaní volané inými programami.
- **Sekcia s ladiacimi informáciami**: sekcia nazývaná `.debug`, obsahuje informácie potrebné pre ladenie programu (názvy premenných, ich adresy a pod.).

1.6 Dataset EMBER

Dataset Elastic Malware Benchmark for Empowering Researchers (EMBER) patrí v súčasnosti medzi najpoužívanejšie datasety v oblasti detekcie malvéru [59]. Celkovo dataset obsahuje 1.1 milióna vzoriek, z čoho je 400 tisíc škodlivých vzoriek, 400 tisíc regulérnych vzoriek a 300 tisíc neoznačených vzoriek. Dataset je taktiež implicitne rozdelený na trénovaciu časť (600 tisíc vzoriek) a testovaciu časť (200 tisíc vzoriek). Samotný dataset sa skladá z JSON objektov, kde jeden objekt reprezentuje staticky extrahované vlastnosti z PE súborov.

EMBER ponúka široké využitie pri trénovaní rôznych modelov. Každá vzorka obsahuje aj rodinu malvéru ku ktorej patrí, takže dataset je možné využiť aj pri riešení klasifikačných problémov. Prítomnosť neoznačených vzoriek taktiež umožňuje využitie datasetu pri učiacich algoritmoch bez dozoru. Zjednodušenú ukážku jednej vzorky možno vidieť vo výpise 1.1.

Samotné statické vlastnosti v jednotlivých vzorkách sú usporiadané do nasledujúcich sekcií:

Informácie o reťazcoch: táto časť JSON objektu zahŕňa informácie o reťazcoch v binárnom súbore. Patrí sem počet reťazcov nachádzajúci sa vo vzorke, priemerná dĺžka reťazca alebo entropia. Taktiež sa tu nachádzajú užitočné informácie o tom, koľko reťazcov spĺňa formát URL (napr. webové adresy), ciest k súborom alebo k registrovým kľúčom. Zaujímavou informáciou je aj počet `MZ` hlavičiek v súbore (viď. kapitola 1.5).

Všeobecné informácie: v tejto časti sa nachádzajú rôzne všeobecnejšie dáta o vzorke. Okrem veľkosti súboru sa tu nachádzajú informácie o tom, či daná vzorka obsahuje ladiace symboly, podpis, zdroje alebo TLS sekciu. Taktiež sa tu nachádza počet exportovaných funkcií, importovaných funkcií a celkový počet ladiacich symbolov.

Informácie o hlavičkách: v tejto časti sú zahrnuté informácie z COFF a voliteľnej hlavičky PE súboru. Táto sekcia JSON súboru do veľkej miery kopíruje rovnaké dáta ako môžeme nájsť v PE súbore (viď. kapitola 1.5).

Informácie o sekciách: táto sekcia je dedikovaná jednotlivým sekciám, ktoré sa nachádzajú v PE súbore. Pre každú sekciu vo vzorke máme uvedený jej názov, typ (či sa jedná o sekciu s kódom, inicializovanými alebo neinicializovanými dátami), oprávnenia, ktorými disponuje daná sekcia (t.j. či má sekcia práva na spúšťanie, čítanie alebo zápis) a taktiež hodnota entropie pre dáta, ktoré sa nachádzajú v sekcii.

Importované funkcie: zoznam importovaných funkcií, vrátane názvu DLL knižnice, z ktorej je daná funkcia importovaná.

Exportované funkcie: zoznam exportovaných funkcií. Tieto sú zahrnuté väčšinou iba vo vzorkách, ktoré sú DLL knižnice.

Okrem vyššie spomínaných dát, každá vzorka obsahuje aj ďalšie položky, ktoré nespadajú do žiadnej kategórie. Sem patrí hodnota hešu pre vzorku, označenie či sa jedná o škodlivú alebo legitímnu vzorku, rodina malvéru alebo histogram bajtov.


```
{
  "sha256": "eb87d82ad7bdc1b753bf91858d2986063ebd8aabeb8e7e91c0c78db21982a0d6",
  "md5": "aba129a3d1ba9d307dad05617f66d8e7",
  "appeared": "2018-01",
  "label": 1,
  "avclass": "fareit",
  "histogram": [ 96506, 8328, 5582, ... ],
  "byteentropy": [0, 4229, 269, 247, ... ],
  "strings": {
    "numstrings": 7762,
    "avlength": 181.60641587219789,
    "printabledist": [591, 51, 96, 46, ... ],
    "printables": 1409629,
    "entropy": 5.037064474164528,
    "paths": 0,
    "urls": 9,
    "registry": 0,
    "MZ": 11
  },
  "general": {
    "size": 2261028,
    "vsize": 1912832,
    "has_debug": 0,
    "exports": 0,
    "imports": 17,
    "has_relocations": 1,
    "has_resources": 1,
    "has_signature": 0,
    "has_tls": 1,
    "symbols": 0
  },
  "header": {
    "coff":{
      "timestamp": 708992537,
      "machine": "1386",
      "characteristics": ["CHARA_32BIT_MACHINE", "BYTES_REVERSED_LO", "EXECUTABLE_IMAGE", ... ]
    },
    "optional": {
      "subsystem": "WINDOWS_GUI",
      "dll_characteristics": [],
      "magic": "PE32",
      ...
    }
  },
  "section": {
    "entry": "CODE",
    "sections": [
      {
        "name": "CODE",
        "size": 443392,
        "entropy": 6.532932639432919,
        "vsize": 442984,
        "props": ["CNT_CODE", "MEM_EXECUTE", "MEM_READ"]
      },
    ...
    ]
  },
  "imports": {
    "kernel32.dll": ["DeleteCriticalSection", "TlsSetValue", "Sleep", ... ],
  },
  "exports": [],
  "datadirectories": [ { "name": "EXPORT_TABLE", "virtual_address": 0 }, ... ]
}
```


Výpis 1.1: Ukážka jednej vzorky z EMBER datasetu.

1.7 Ďalšie datasety

Napriek tomu, že ako základ našej práce sme si vybrali dataset EMBER, existujú aj ďalšie zdroje datasetov, ktoré môžu byť vhodné pre detekciu malvéru. Celkový prehľad verejných datasetov (ktoré majú dostupné štatistiky) uvádzame v tabuľke 1.1.

Ďalej uvádzame popis najznámejších datasetov:

SoReL: dataset SoReL [60] priamo nadväzuje na úspech datasetu EMBER a je takpovediac jeho nasledovníkom. Podobne ako EMBER, obsahuje statické vlastnosti PE súborov v JSON formáte. V porovnaní s datasetom EMBER však chýbajú niektoré predspracované vlastnosti, ktoré je možné extrahovať priamo z PE súboru, ako je napríklad prítomnosť podpisu, ladiacich symbolov a pod. Hlavným rozdielom v porovnaní s datasetom EMBER je jeho robustnosť. SoReL obsahuje spolu až 20 miliónov vzoriek, pričom 15 miliónov vzoriek je označených a ďalších 5 miliónov je neoznačených. Navyše, obsahuje aj 10 miliónov skutočných vzoriek malvéru (nie len extrahované vlastnosti), ktoré však majú z bezpečnostných dôvodov modifikovanú hlavičku (a teda ich nie je možné omylom spustiť). Napriek tomu, že SoReL obsahuje väčší počet novších vzoriek, v našej práci sme dali prednosť datasetu EMBER z dvoch dôvodov. Prvým dôvodom bolo, že v čase vydania datasetu (koniec roku 2020), sme mali rozpracovanú ranú verziu našej ontológie, založenej na datasete EMBER spolu s prvotnými experimentami. Druhým dôvodom bola zaužívanosť datasetu vo vedeckej komunite (v čase písania práce 301 vs. 44 citácií). Samotné dáta v datasete SoReL je však možné priamo aplikovať na našu ontológiu (viac v kapitole 4).

VirusShare: patrí v súčasnosti medzi najrozsiahlejšiu databázu škodlivých vzoriek [70]. V čase písania práce obsahovala databáza takmer 58 miliónov vzoriek malvéru (reálne spustiteľné súbory). Napriek tomu, že sa jedná o veľmi rozsiahlu databázu, jej použitie v oblasti detekcie malvéru je v porovnaní s ostatnými datasetmi značne limitovanejšie. Okrem toho že obsahuje rôzne formáty okrem PE súborov ako sú škodlivé *Word* dokumenty, HTML stránky a pod., samotná databáza neobsahuje žiadne reguléme vzorky.

VirusTotal: patrí medzi veľmi známe služby, ponúkajúce primárne možnosť otestovať nahraný súbor na jeho škodlivosť [71]. Okrem iného však umožňuje sťahovať aj reálne škodlivé/regulérne súbory, pričom obsahuje milióny vzoriek. Nevýhodou je, že sťahovanie vzoriek patrí medzi platené služby.

Tabuľka 1.1: Tabuľka verejných datasetov škodlivého kódu spolu s ich vlastnosťami.

Meno	Vzorky	Malvér	Legitímne vzorky	Neoznačené	Platforma	Vlastnosti	Formát
EMBER [59]	1,100 k	400 k	400 k	300 k	Windows	statické	JSON
SOREL [60]	20,000 k	10,000 k	10,000 k	0	Windows	statické	LMDB+binárne
MSMCC [61]	20 k	20 k	0	0	Windows	statické	disassembler
MALREC [62]	66 k	66 k	0	0	Windows	dynamické	PANDA
Mal-API-2019 [63]	7.1 k	7.1 k	0	0	Windows	dynamické	CSV
AVAST-CTU [64]	49 k	49 k	0	0	Windows	stat.+dyn.	JSON
ClaMP [65]	5.2 k	2.7 k	2.5 k	0	Windows	statické	CSV
MalImg [48]	9.5 k	9.5 k	0	0	Windows	statické	šedá škála
DREBIN [66]	129 k	5.6 k	123 k	0	Android	statické	textový súbor
MOTIF [67]	3.0 k	3.0 k	0	0	Windows	statické	JSON+binárne
Malpedia [68]	7.8 k	7.8 k	0	0	mix	stat.+dyn.	obraz pamäti
MalDozer [69]	71 k	33 k	38 k	0	Android	statické	neznámy

theZoo: patrí medzi veľmi málo známy dataset, ktorý ponúka iba niekoľko stoviek vzoriek malvéru [72]. V porovnaní s inými databázami, je však jedinečný v tom, že okrem samotných škodlivých súborov ponúka aj ich zdrojový kód.

Microsoft Malware Classification: patrí medzi dataset vydaný spoločnosťou *Microsoft* pre účely súťaže o najlepší klasifikátor [61]. V samotnom datasete sa nachádza približne 20 tisíc vzoriek malvéru z 9 rôznych rodín. Cieľom súťaže bolo natrénovať model, ktorý by vedel klasifikovať neznáme vzorky do správnej rodiny. Samotné vzorky v datasete sa nachádzajú v podobe spustiteľného súboru, bez hlavičky (ochrana pred náhodným spustením).

MalShare: jedná sa o podobný zdroj ako *VirusShare*, s veľkým množstvom reálne spustiteľných vzoriek [73]. Hlavným rozdielom je, že *MalShare* je orientovaný na komunitu, t.j. umožňuje nahrávať škodlivé vzorky a zdieľať ich medzi používateľmi. Obsahuje vzorky malvéru od roku 2012 až po súčasnosť, presnejší celkový počet však nie je známy.

Malrec: jedná sa o odlišný dataset v porovnaní s vyššie spomenutými [62]. Zatiaľ čo predchádzajúce datasety obsahovali statické vlastnosti alebo reálne spustiteľné vzorky, *Malrec* zachytáva celkovú činnosť malvéru v systéme. Tvorba jednej vzorky spočíva v uložení stavu systému pred spustením malvéru a logovaním všetkých zdrojov nedeterminizmu v systéme ako sú prerušenia alebo systémové volania. Takýmto spôsobom je možné zopakovať činnosť malvéru a extrahovať rôzne vlastnosti, ktoré môžu následne slúžiť pri trénovaní klasifikátorov. V samotnom datasete sa nachádza približne 66 tisíc vzoriek škodlivého kódu, pričom dáta je nutné určitým spôsobom predspracovať, keďže nie sú priamo vhodné pre strojové učenie.

Mal-API-2019: je pomerne malý a málo známy dataset [63], ktorý sa zameriava primárne na API volania. Samotný dataset obsahuje približne 7 tisíc škodlivých vzoriek, pričom každá vzorka je zložená z postupnosti API volaní, ktoré boli vykonané malvérom.

Avast-CTU: jedná sa o dataset, ktorý kombinuje statické a dynamické vlastnosti [64]. Samotné vzorky obsahujú kombinované vlastnosti, ktoré boli získané prostredníctvom sandboxu *CAPE*[3]. Dataset obsahuje približne 50 tisíc vzoriek škodlivého kódu, pričom jeho hlavné zameranie je klasifikácia malvéru do rodín.

[3]https://capesandbox.com/

ClaMP: dataset Classification of Malware with PE headers (ClaMP) patrí taktiež medzi menšie a menej známe datasety [65]. Celkový dataset obsahuje 5 tisíc škodlivých a regulérnych vzoriek. Na rozdiel od iných datasetov, ClaMP obsahuje iba vlastnosti, ktoré možno extrahovať z PE hlavičiek ako sú hodnoty hešu, počet sekcií, architektúra, čas kompilácie a pod.

MalImg: jedná sa o veľmi špecifický dataset, ktorý bol pripravený priamo pre účely algoritmu prezentovaného v práci [48]. V samotnom datasete sa nachádza približne 10 tisíc škodlivých vzoriek z 25 rôznych rodín. Zaujímavosťou datasetu je, že jednotlivé vzorky sú zakódované ako obrázky v šedej škále. Zrejmou nevýhodou tohoto datasetu je prílišná špecifickosť použitia pre rôzne algoritmy hlbokého učenia, ktoré slúžia najmä pri počítačovom videní.

DREBIN: zatiaľ čo všetky vyššie spomínané datasety boli zamerané primárne na operačný systém *Windows*, dataset *DREBIN* cieli naopak na mobilný operačný systém *Android* [66]. Obsahuje približne 5 tisíc škodlivých vzoriek a 10 tisíc regulérnych vzoriek, pričom jednotlivé vzorky sú prezentované vo forme 215 vyextrahovaných vlastností, ktoré charakterizujú správanie aplikácie.

MOTIF: jedná sa o relatívne menší dataset, ktorý obsahuje 4 tisíc škodlivých vzoriek zo 454 rôznych rodín [67]. Zaujímavosťou datasetu je však fakt, že všetky vzorky boli správne označené expertmi ako škodlivé. Iné datasety (najmä keď sú rozsiahlejšie), sa spoliehajú na metódy väčšinového hlasovania antivírusových riešení (služba je dostupná aj v rámci VirusTotal). Autori spolu s datasetom zverejnili aj štúdiu, kde ukázali, že úspešnosť detekcie takýchto vzoriek, pomocou vyššie spomínanej metódy (a taktiež pomocou najznámejších modelov strojového učenia) je nižšia ako pri iných datasetoch.

Malpedia: jedná sa taktiež o menší dataset, ktorý obsahuje približne 8 tisíc vzoriek [68]. Svojou charakteristikou je podobný ako vyššie spomínaný MOTIF, avšak iba časť vzoriek bola expertmi označená ako škodlivá. Na označenie zvyšných vzoriek boli použité iné metódy (napr. YARA pravidlá).

MalDozer: jedná sa o dataset cielený na platformu *Android* [69]. Obsahuje približne 71 tisíc vzoriek so statickými dátami. Jedná sa o podobný dataset ako DREBIN.

1.8 Bezpečnosť klasifikátorov

Dôležitou témou v oblasti strojového učenia sa v posledných rokoch stala ich bezpečnosť. Prvá práca v tejto oblasti poukázala, že rôzne klasifikátory, či už sa jedná o doménu detekcie malvéru alebo rozpoznávania objektov, sú zraniteľné na tzv. Adversarial Examples (AE) [74]. AE sú špeciálne upravené vstupy (zvyčajne útočníkmi), ktoré pôsobia navonok rovnako ako pôvodný vstup, avšak spomínané úpravy spôsobia chybnú klasifikáciu. Formálne, majme klasifikátor $D(x)$, ktorého výstupom sú dve triedy: 1 (malvér) a 0 (legitímny súbor). Pôvodný vstupný objekt x je klasifikovaný ako $D(x) = 1$. Cieľom útočníka je následne vytvoriť takú modifikáciu x', aby zachoval pôvodné vlastnosti vstupu x tak, aby spôsobil chybnú klasifikáciu $D(x') = 0$. Typickým útokom v oblasti počítačového videnia bol príklad rozpoznávania dopravných značiek v práci [75]. Autori ukázali, že nato, aby klasifikátor chybne označil dopravnú značku, stačí k obrázku pripočítať špeciálny šum, ktorý nijakým spôsobom vizuálne nemení výsledný obraz. Taktiež predstavili jednoduchý útok, pri ktorom stačí na dopravnú značku nalepiť presne vypočítané štítky, ktoré sú však dostatočne malé nato, aby vizuálne nemenili význam značky, avšak spôsobili chybnú klasifikáciu. Podobné útoky postupne vznikali aj v rámci škodlivého kódu, kde cieľom útočníkov je takým spôsobom pozmeniť spustiteľný súbor aby pomýlili klasifikátor a zároveň zachovali pôvodnú funkcionalitu.

Všeobecne môžeme útoky rozdeliť do dvoch kategórií podľa [76]. Sem môžeme zaradiť útoky kategorizované podľa toho, čo je cieľom útočníka a podľa toho aké vedomosti daný útočník o cieli má. Podľa útočníkovho cieľa:

- *Evasion attack*. Jedná sa o základný typ útoku, kde cieľom útočníka je jednoducho zmeniť výstup klasifikátora. Takéto typy útokov sa väčšinou realizujú už na hotové modely s vysokou úspešnosťou, kde útočník nemôže nijakým spôsobom meniť parametre klasifikátora. Môže však vytvárať falošné vzorky, posielať ich do klasifikátora a sledovať výstupy. Typ útoku môže byť ešte špecifickejší, podľa toho či je jeho cieľom zmeniť výstup klasifikátora na ľubovoľnú alebo na špecifickú hodnotu.

- *Poisoning attack*. Cieľom tohto typu útoku je priamo ovplyvniť proces trénovania modelu a to injektovaním falošných vzoriek do trénovacej množiny. Takéto vzorky môžu následne spôsobiť nižšiu úspešnosť modelu alebo chybnú klasifikáciu niektorých vstupov. Akokoľvek sa môže tento typ útoku zdať ako hypotetický, scenárov, kde útočník môže ovplyvňovať trénovaciu množinu je viacero. Typickým príkladom sú tzv. honeypoty, ktoré využívajú antivírusové firmy na zber dát.

Podľa vedomostí útočníka, môžeme útoky charakterizovať nasledovne:

- *White-box útok*. Jedná sa o kategóriu útokov, pri ktorom má útočník úplnú znalosť o cieľovom modeli ako napr. architektúra, hodnoty váh a pod. Tieto znalosti môže následne využiť pri vytváraní falošných vstupov. Odolnosť voči *white-box* útokom je jedna z dôležitých vlastností, ktoré musí klasifikátor spĺňať, aby ho bolo možné považovať za bezpečný.

- *Black-box útok*. Pri tomto type útoku nemá útočník žiadne znalosti o cieľovom modeli. Útočník tak môže iba posielať vstupy do klasifikátora a pozorovať výstupy. V porovnaní s *white-box* útokmi sa jedná o praktickejší scenár, keďže väčšina autorov, či už akademických prác alebo komerčných riešení nezverejňuje detailné parametre klasifikátorov. Príklad útoku demonštrovali v práci [77], kde bolo ukázané, že stačí približne 800 až 6000 dopytov na verejný online klasifikátor škodlivého kódu, aby si dokázali vytvoriť lokálny model, podobný originálnemu a dosiahli tak vysokú úspešnosť chybnej klasifikácie.

Vytváranie AE pre spustiteľné súbory sa ukázalo ako komplikovanejšie v porovnaní s obrázkami alebo zvukom. Zatiaľ, čo pri obrázkoch sa väčšinou pridáva šum, pre spustiteľné súbory to nie je možné, keďže tomu zabraňuje striktná sémantika PE súboru. Pridanie šumu by mohlo narušiť funkcionalitu pôvodného kódu alebo v horšom prípade aj narušiť samotný formát, čím by súbor nebolo možné spustiť. V práci [78] predstavili tri útoky voči klasifikátorom, založených na hlbokom učení. Základom útokov bolo pripojenie sekvencie bajtov k pôvodnej vzorke. Autori predstavili tri stratégie: pripojenie náhodných bajtov, pripojenie sekvencie z regulérnych súborov (ktoré sú klasifikované ako neškodné) a taktiež metódu založenú na Fast Gradient Method (FGM), v ktorej sa pripojené bajty iteratívne menia na základe vstupného gradientu, až pokiaľ nie je vstupná vzorka chybne klasifikovaná. V samotnej práci ukázali, že ich útoky dosahujú úspešnosť 60%. Suciu a spol. [79] predstavili metódu nazvanú *Slack FGM*. V tejto metóde sa na rozdiel od predchádzajúcich útokov nepripájajú nové bajty (t.j. samotný súbor sa nezväčšuje), ale modifikujú sa už existujúce sekvencie bajtov, ktoré sa nepoužívajú. Na tento účel využili tzv. *slack* bajty, ktoré sa nachádzajú medzi sekciami v súbore a kompilátor ich automaticky vkladá na zarovnanie. Hu a Tan prezentovali systém *MalGAN*, ktorý je založený na Generative Adversarial Network (GAN) [80]. GAN je založený na princípe hry dvoch neurónových sietí a bežne sa používa na generovanie nových trénovacích vzoriek. V práci ukázali, že generovanie AE prostredníctvom GAN sietí je značne úspešnejšie v porovnaní s gradientovými metódami. Park a spol. ukázali prístup založený na obfuskácii [81]. V prezentovanej metóde špecifikovali 23 sekvencií inštrukcií, ktoré nijakým spôsobom nemenia funkcionalitu programu a väčšinou

sa používajú práve na obfuskáciu zdrojového kódu [82]. Na základe týchto operácií implementovali algoritmus Adversarial Malware Alignment Obfuscation (AMAO), ktorý injektuje spomínané sekvencie do pôvodnej vzorky tak, aby minimalizoval počet novo vložených inštrukcií a zároveň spôsobil chybnú klasifikáciu. Autori deklarovali úspešnosť až 98%, pričom testovali rôzne hlboké siete pri *white-box* a *black-box* scenári. Zatiaľ čo predchádzajúce práce boli zamerané najmä na hlboké siete, Rosenberg a spol. [83] otestovali generovanie AE pre klasifikátory, ktoré používajú ako vstup sekvenciu API volaní spolu s tradičnejšími algoritmami strojového učenia ako napr. SVM. Autori implementovali systém Generative API Adversarial Generic Example by Transferability (GADGET), ktorý najprv vytvorí aproximáciu modelu na ktorý sa snažíme útočiť a následne vytvorí AE, pridaním API volaní bez zmeny pôvodnej funkcionality.

Jedna zo zaujímavých vlastností, ktorá sa ukázala pri výskume útokov na rôzne klasifikátory, je ich prenositeľnosť [84]. Táto vlastnosť definuje, že ak vytvoríme AE, ktorý spôsobí chybnú klasifikáciu v jednom modeli, existuje nezanedbateľná pravdepodobnosť, že ju zároveň spôsobí aj v inom, odlišnom modeli. Taktiež bolo ukázané, že prenositeľnosť platí aj v prípadoch, kedy sú dané modely založené na rôznych algoritmoch, majú rozličnú architektúru alebo boli trénované na rozličných trénovacích datasetoch [85]. Príkladom môže byť, že ak máme cieľový klasifikátor implementovaný pomocou SVM a útočník si vytvorí lokálny model pomocou iného algoritmu, napríklad ANN, tak existuje vysoká šanca, že generovaný AE pre ANN bude taktiež AE pre SVM klasifikátor. Tento samotný fakt spôsobil vysokú úspešnosť spomínaných *black-box* útokov. Útočník si tak môže natrénovať lokálny a jednoduchší model klasifikátora bez toho, aby poznal aký algoritmus bol použitý v pôvodnom klasifikátore.

Spôsobov ako sa brániť voči AE bolo preskúmaných viacero. Szegedy a spol. prezentovali metódu, v ktorej vygenerovali veľké množstvo AE a zahrnuli ich priamo do trénovacej množiny, čím zvýšili robustnosť modelu [74]. V práci [86] vyskúšali proces tzv. destilácie, kde prvotný natrénovaný model využili na označenie novej trénovacej množiny, pomocou ktorej vytvorili nový model s rovnakou architektúrou. Úspešnosť AE klesla, avšak rovnako klesla aj úspešnosť korektnej detekcie.

1.9 Sémantické technológie v bezpečnosti

Ontológie a sémantické technológie našli svoje využitie aj v rámci domény počítačovej bezpečnosti. V tejto podkapitole si uvedieme stručný prehľad existujúcich aplikácií týchto technológií v oblasti detekcie malvéru, ale aj v ďalších príbuzných oblastiach v rámci bezpečnosti.

Xia a spol. [87] predstavili detekciu malvéru pomocou ontológií. V práci navrhli ontológiu, v ktorej definovali rôzne systémové komponenty ako triedy, napr. súbor, proces alebo register, spolu s binárnymi vzťahmi ako vytvorenie procesu, vlákna a pod. Spolu definovali 38 rôznych správaní založených na dynamických vlastnostiach. Pomocou *Apriori* algoritmu následne získali asociačné pravidlá, ktoré možno použiť na detekciu. V práci však chýba viacero informácií ohľadom použitého datasetu či detailnejšia evaluácia. Ding a spol. v práci [88] navrhli ontológiu pre rôzne rodiny malvéru a rovnako využili *Apriori* algoritmus na získanie pravidiel. Samotná ontológia bola zameraná na dynamické vlastnosti a napriek tomu že dosiahli relatívne vysokú úspešnosť (správnosť až 96%), samotná trénovacia vzorka bola relatívne malá (približne 500 vzoriek). Podobný prístup zvolili aj v práci [89], kde však využili predpripravené Semantic Web Rule Language (SWRL) pravidlá. Fasano a spol. [90] použili v práci temporálnu logiku na detekciu spajvéru (typ škodlivého kódu, ktorého cieľom je preposielať citlivé údaje tretím stranám) pre operačný systém *Android*. Pomocou temporálnej logiky špecifikovali rôzne pravidlá správania, typické pre spajvér, ktoré následne porovnávali s reálnymi vzorkami. Autori síce deklarovali vysokú úspešnosť, avšak spomínaný systém otestovali iba na relatívne malom datasete. Razzaq a spol. [91] naopak predstavili sémantický prístup k detekcii webových útokov ako sú SQLi alebo XSS. V ich práci definovali ontológiu pre komunikačný protokol HTTP a ontológiu pre webový útok. Taktiež definovali šablóny pre SWRL pravidlá na spomínané útoky pre ktoré nasledne vytvárali inštancie, priamo použiteľné na validáciu vstupov a detekciu webového útoku. Ich práca ukázala vysokú úspešnosť, najmä v porovnaní so signatúrami.

Sémantické technológie našli svoje využitie aj pri vyšetrovaní bezpečnostných incidentov [92]. Carvalho a spol. predstavili riešenie, pomocou ktorého je možné pomôcť vyšetrovateľom rôznych incidentov prostredníctvom definovania ontológie a hľadaním súvislostí pomocou SPARQL dotazov. Riešenie demonštrovali na analýze incidentu z roku 2015, kedy uniklo väčšie množstvo exploitov, ktoré zneužívali *zero day* zraniteľnosti. Chu a Lisitsa taktiež navrhli ontológiu pre rôzne formy útokov, spolu so systémom, ktorý pomocou SWRL pravidiel pomáha automatizovať penetračné testovanie [93].

Ontológie napokon našli svoje využitie taktiež v oblasti zdieľania vedomostí z oblasti počítačovej bezpečnosti, ako sú bázy znalostí rôznych foriem

útokov, škodlivého kódu alebo incidentov [94, 95, 96]. Z niektorých známych znalostných báz môžeme spomenúť Unified Cybersecurity Ontology (UCO), ktorá integruje viacero ontológií a zdieľacích štandardov z oblasti bezpečnosti [97] alebo ontológiu Malicious Behavior Ontology (MBO) [98], ktorá je zameraná na malvér, konkrétne na rôzne typy správania (ktoré je možné extrahovať dynamickou analýzou). Medzi ontológie zamerané na malvér patrí aj ontológia MALOnt [99]. Táto ontológia je však zameraná najmä na taktiky útočníkov a rôzne indikátory kompromitácie, tzv. Indicators of Compromise (IoC). Zaujímavú prácu prezentovali taktiež Ulicny a spol. [100]. V ich práci predstavili myšlienku automatického generovania ontológie zo známych zdieľacích štandardov, ktoré slúžia na výmenu informácií ohľadom bezpečnostných incidentov. V samotnom výskume sa zamerali na štandard Structured Threat Information Expression (STIX), kde spolu s automatickým generovaním ontológie prezentovali aj zefektívnenie analytického procesu pri reakcii na bezpečnostné incidenty prostredníctvom automatického usudzovania.

Kapitola 2

Ontologická reprezentácia

Táto časť práce sa venuje teoretickým základom ontológií a všeobecnému úvodu do konceptové učenia, ktoré tvorí dôležitú súčasť práce. V úvodnej časti 2.1 si definujeme pojem *ontológie* spolu s popisom ich výhod a spôsobov ako ich možno aplikovať v praxi. Druhá podkapitola 2.2 sa venuje popisu a definíciam deskripčných logík, ktoré sú jedným z najpopulárnejších jazykov na ich reprezentáciu. Keďže existuje viacero variantov deskripčných logík, podľa ponúkanej expresivity a výpočtovej náročnosti, ich opisu sa venujeme v podkapitole 2.3. V podkapitole 2.4 si uvedieme, akým spôsobom je možné počítačovo modelovať deskripčné logiky a znalostné bázy, pomocou rôznych variantov jazyka OWL. Podkapitola 2.5 sa venuje úvodu do konceptového učenia. V rámci podkapitoly si uvedieme jednotlivé varianty konceptového učenia spolu s popisom ako môžno skonštruovať všeobecný model algoritmu. Posledná podkapitola 2.6 sa venuje najdôležitejšej časti konceptového učenia, tzv. *spresňujúcim* operátorom, s dôrazom na všeobecnejší popis.

2.1 Ontológie

Všeobecne, ontológie vieme definovať ako *formálnu konceptualizáciu* určitej konkrétnej domény [101]. V rámci tejto definície môžeme chápať konceptualizáciu ako definíciu rôznych tried, objektov, faktov, vlastností a vzťahov medzi nimi. Ontológie označujeme ako formálnu reprezentáciu z toho dôvodu, že sú založené na formálných jazykoch ako sú napr. deskripčné logiky. Pomocou ontológií vieme formálne reprezentovať znalosti a koncepty z rôznych domén ako sú napr. medicína, história alebo aj informačná bezpečnosť. Ako príklady si môžeme uviesť ontológiu ľudskej anatómie [102], ontológiu génov [103] alebo hudby [104].

Podobným spôsobom ako v ontológiach vieme teoreticky znalosti reprezentovať aj inými metodológiami. Ako príklady môžeme spomenúť entitno-relačné diagramy, ktoré sa využívajú pri návrhu databáz alebo pomocou UML diagramov, ktoré našli svoje využitie najmä pri objektovo-orientovanom návrhu softvéru. Ontológie však v porovnaní so spomínanými metodológiami prinášajú viacero výhod. Ako prvú výhodu si môžeme uviesť fakt, že ontológie prinášajú možnosť jednoduchého zdieľania a integrácie, keďže pôvodne boli navrhnuté s cieľom implementácie *sémantického webu* (ktorý na rozdiel od klasického webu zahŕňa aj strojovo spracovateľné sémantické dáta). Ontológie teda umožňujú zdieľať dáta v jednotnej schéme, čím vedia zabezpečiť aj sémantickú interoperabilitu [105]. Jednoduchá možnosť zdieľania umožňuje doménovým expertom využívať alebo rozširovať znalostnú bázu.

Ako druhú významnú výhodu ontológií môžeme spomenúť samotný fakt, že sú založené na jazykoch formálnej logiky. Tento fakt so sebou prináša viacero výhod. Mnohé znalosti alebo fakty nemusia byť v ontológii explicitne uvedené, avšak implicitne vyplývajú z existujúcich znalostí [106]. Taktiež existujú rôzne algoritmy pre automatizované odvodzovanie nových znalostí z explicitných faktov v ontológii. Takýto proces sa zvyčajne označuje ako *usudzovanie* (z angl. *reasoning*). Okrem rozširovania ontológie je možné automatické *usudzovanie* použiť aj na zisťovanie, či sú rôzne logické formuly splniteľné alebo či je samotná ontológia konzistentná (t.j. či sa v nej nenachádzajú kontradikcie). Jedná sa o dôležitú úlohu z hľadiska tvorby korektných ontológií pre rôzne aplikácie.

Ontológie sa však neobmedzujú iba na využitie v podobe sémantických databáz. Ich využite môžeme nájsť v rôznych aplikáciach ako bioinformatika, agentové systémy, odporúčacie systéme a najmä, ako si ukážeme v kapitole 2.5, je možné aplikovať aj strojové učenie nad ontológiami a riešiť rôzne klasifikačné problémy [105].

2.2 Deskripčné logiky

Jedným z formálnych jazykov pri budovaní ontológií a znalostných báz tvoria deskripčné logiky, ktoré patria do rodiny formálnych jazykov využívaných pri reprezentácii znalostí. Všeobecne, deskripčné logiky môžeme charakterizovať ako podmnožinu logiky prvého rádu, ktorá je zvyčajne rozhodnuteľná [107]. V porovnaní s logikou prvého rádu ponúkajú menšiu expresivitu a jednoduchšiu syntax bez premenných. V rámci tejto kapitoly si predstavíme syntax a sémantiku deskripčných logík.

2.2.1 Syntax

Deskripčné logiky sú tvorené tromi množinami primárnych elementov, ktoré sú navzájom disjunktné:

- **Koncepty** označujú názvy, ktorými reprezentujeme rôzne triedy v doméne. Ekvivalentom konceptov v logike prvého rádu sú unárne predikáty. V kontexte informačnej bezpečnosti môžu do tejto množiny patriť napr. triedy `Process` (reprezentujúca spustený proces), `File` (trieda reprezentujúca súbor na disku) alebo trieda `Action`, ktorá reprezentuje konkrétnu akciu (vykonanú napr. triedou `Process`).

- **Roly** pomenúvajú binárne vzťahy, ktoré môžu existovať medzi individuálmi v konkrétnej znalostnej báze. V logike prvého rádu sú ekvivalentom rolí binárne predikáty. Ako príklad si môžeme uviesť role `create_process` (binárny vzťah reprezentujúci vytvorenie nového procesu) alebo `has_section` (binárny súbor obsahuje konkrétnu sekciu).

- **Individuály** reprezentujú samotné entity v danej doméne, resp. sa jedná o inštancie konceptov v znalostnej báze. V logike prvého rádu sú individuály charakterizované ako konštanty. Príkladom môžu byť individuály *chrome.exe* (inštancia triedy `Process`) alebo *key.dat* (inštancia triedy `File`).

Samotná báza znalostí (ktorú budeme označovať ako $\mathcal{K}$) je zložená z troch množín axióm:

- **TBox** (terminologický box): do tejto množiny patria všetky axiómy, ktoré definujú hierarchiu tried a vzťahy medzi nimi. V rámci TBoxu sa definuje samotná terminológia konkrétnej aplikačnej domény. V tejto množine môžeme definovať rozličné vzťahy medzi triedami ako napr. subsumpciu ($C_1 \sqsubseteq C_2$), ktorá definuje že trieda C_1 je podmnožinou triedy C_2, ekvivalenciu tried ($C_1 \equiv C_2$), disjunkciu tried ($C_1 \equiv \neg C_2$),

zlúčenie konceptov ($C_1 \sqcup C_2$) a pod. Príklad (2.1) reprezentuje znalosť, že koncept `GenerateKey` (akcia pre generovanie šifrovacieho kľúča) je podtriedou (resp. subsumpcia) konceptu `Action` (všeobecná trieda pre akciu). Druhý príklad (2.2) naopak reprezentuje fakt že `File` a `Process` sú navzájom disjunktné koncepty.

$$\texttt{GenerateKey} \sqsubseteq \texttt{Action} \tag{2.1}$$

$$\texttt{File} \equiv \neg\texttt{Process} \tag{2.2}$$

- **ABox** (aserčný box): v tejto množine sa nachádzajú tvrdenia o individuáloch vzhľadom na slovník znalostnej bázy. Priamo v rámci ABoxu sa špecifikujú už konkrétne znalosti a fakty, medzi ktoré môžu patriť napr. fakty o príslušnosti individuálov k určitej triede alebo vzťah medzi dvoma individuálmi. Príklad (2.3) definuje, že individuál *chrome.exe* patrí do triedy `Process` (špecifikujeme teda konkrétny proces). Druhý príklad (2.4) naopak definuje binárny vzťah `create_process` medzi individuálmi *chrome.exe* a *svrc.exe* (napr. patriacimi do triedy `Process`), čím definujeme fakt, že proces s názvom *chrome.exe* vytvára nový proces s názvom *svrc.exe*.

$$\texttt{Process}(\textit{chrome.exe}) \tag{2.3}$$

$$\texttt{create\_process}(\textit{chrome.exe}, \textit{svrc.exe}) \tag{2.4}$$

- **RBox** (relačný box): označuje vlastnosti rolí, ktoré sa nachádzajú v znalostnej báze. V rámci RBoxu môžeme definovať komplexné vlastnosti ako hierarchia rolí, tranzitivita, reflexivita a pod. V príklade (2.5) môžeme vidieť, že rola `download_executable` je podrolou `download_file`, čím vyjadríme fakt, že pre všetky páry individuálov, medzi ktorými platí vzťah `download_executable`, zároveň platí aj vzťah `download_file`.

$$\texttt{download\_executable} \sqsubseteq \texttt{download\_file} \tag{2.5}$$

Definícia RBoxu (resp. oddelenie od TBoxu) hrá významnú úlohu iba pri niektorých veľmi expresívnych deskripčných logikách (viď. kapitola 2.3). V rámci práce budeme teda uvažovať RBox ako súčasť TBoxu a nebudeme medzi nimi explicitne rozlišovať.

Znalostnú bázu $\mathcal{K}$ teda môžeme definovať ako $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, kde $\mathcal{T}$ označuje terminologický box a $\mathcal{A}$ označuje aserčný box.

V znalostnej báze môžeme tvoriť *zložené koncepty* (v práci ich budeme nazývať jednoducho ako koncepty alebo konceptové výrazy), kombináciou konceptov, individuálov, rolí a operátorov (v závislosti od expresivity konkrétnej logiky) [108]. Majme množinu N_R, ktorá zahŕňa atomické role, množinu N_C, ktorá zahŕňa atomické koncepty ($N_R \cap N_C = \emptyset$) a množinu N_I, obsahujúcu individuály. Zložené koncepty (pre deskripčnú logiku $\mathcal{SROIQ}$ - viď. podkapitola 2.3) potom vieme induktívne definovať:

1. Každý atomický koncept $C \in N_C$ je koncept.

2. Ak C a D sú koncepty, $r \in N_R$ je rola a $n \in N$, tak koncepty sú taktiež:

   - $\top$ (vrchný koncept, ktorý zahŕňa všetky individuály v znalostnej báze), $\bot$ (spodný koncept, zachytávajúci prázdnu množinu individuálov)
   - $C \sqcup D$ (disjunkcia konceptov), $C \sqcap D$ (konjunkcia konceptov), $\neg C$ (negácia konceptu)
   - $\forall r.C$ (všeobecný kvantifikátor), $\exists r.C$ (existenčný kvantifikátor)
   - $\exists r.\texttt{Self}$ (označenie lokálnej reflexivity)
   - $\geq n\, r.C$ (minimálny stupeň kardinality; *platí aspoň pre* n), $\leq n\, r.C$ (maximálny stupeň kardinality; *platí najviac pre* n), $= n\, r.C$ (*platí práve pre* n)
   - každá konečná množina $\{a_1, a_2, ..., a_n\} \in N_I$, resp. vymenovaná množina individuálov (nazývané ako nominály)

Príklady konceptových výrazov môžeme vidieť v (2.6), (2.7) a (2.8). Ako vidíme v príkladoch, koncepty môžu byť značne expresívne v závislosti od použitej logiky (resp. dostupných operátorov).

$$\exists \texttt{has\_action.GenerateKey} \sqcap \exists \texttt{create\_file.}\{key.dat\} \tag{2.6}$$

$$\geq 2.(\exists \texttt{create\_file}.\top) \tag{2.7}$$

$$\texttt{ExecutableFile} \sqcap \exists \texttt{has\_action.CreateMutex} \tag{2.8}$$

2.2.2 Sémantika

Sémantika nám špecifikuje, aké sú logické dôsledky ontológie a jej hlavnou úlohou je povedať, či je určitá axióma (resp. konceptový výraz) logickým dôsledkom znalostnej bázy. Samotná sémantika konceptových výrazov je definovaná prostredníctvom *interpretácií* [109]. Interpretácia $\mathcal{I}$ je definovaná ako dvojica $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, kde $\Delta^{\mathcal{I}}$ je neprázdna množina označovaná ako doména (obsahujúca všetky individuály) a $\cdot^{\mathcal{I}}$ je interpretačná funkcia, ktorá mapuje slovník znalostnej bázy na elementy z množiny $\Delta^{\mathcal{I}}$. Funkcia $\cdot^{\mathcal{I}}$ mapuje:

- každý koncept $C \in N_C$ na podmnožinu $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
- každú rolu $r \in N_R$ na podmnožinu $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
- každý individuál $a \in N_I$ na element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$

Konštrukt	**Syntax**	**Sémantika**
`individuál`	`a`	$a^{\mathcal{I}}$
`atomický koncept`	C	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
`rola`	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
`nominály`	$\{a\}$	$\{a\}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \lvert\{a\}\rvert^{\mathcal{I}} = 1$
`vrchný koncept`	$\top$	$\Delta^{\mathcal{I}}$
`spodný koncept`	$\bot$	$\emptyset$
`konjunkcia`	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
`disjunkcia`	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
`negácia`	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
`všeobecný kvantifikátor`	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \{a \mid \forall b.(a,b) \in r^{\mathcal{I}} \Rightarrow b \in C^{\mathcal{I}}\}$
`existenčný kvantifikátor`	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{a \mid \exists b.(a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
`max. stupeň kardinality`	$\leq n\, r.C$	$(\leq nr)^{\mathcal{I}} = \{a \mid \lvert\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \leq n\}$
`min. stupeň kardinality`	$\geq n\, r.C$	$(\geq nr)^{\mathcal{I}} = \{a \mid \lvert\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \geq n\}$
`lokálna reflexivita`	$\exists r.\texttt{Self}$	$(\exists r.\texttt{Self})^{\mathcal{I}} = \{a \mid (a,a) \in r^{\mathcal{I}}\}$
`inverzná rola`	r^-	$(r^-)^{\mathcal{I}} = \{(a,b) \mid (b,a) \in r^{\mathcal{I}}\}$

Tabuľka 2.1: Syntax a sémantika konceptov v jazyku $\mathcal{SROIQ}$.

Interpretácia zložených konceptov je založená na interpretácii základných výrazov. Sémantiku jednotlivých logických konštruktov môžeme vidieť v tabuľke 2.1. Majme znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. Ak interpretácia $\mathcal{I}$ spĺňa axiómu (resp. množinu axióm) z $\mathcal{T}$, hovoríme, že daná interpretácia je *modelom* axiómy (resp. modelom celej množiny axióm). Axiómy sú *ekvivalentné*, ak majú rovnaký model. Interpretácia $\mathcal{I}$ je modelom $\mathcal{A}$ (označované ako $\mathcal{I} \models \mathcal{A}$), práve vtedy, keď platí $a^{\mathcal{I}} \in C^{\mathcal{I}}$ pre všetky $C(a) \in \mathcal{A}$ a zároveň platí $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ pre všetky $r(a,b) \in \mathcal{A}$. Interpretácia $\mathcal{I}$ je následne modelom znalostnej

bázy $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (označované ako $\mathcal{I} \models \mathcal{K}$), ak je modelom $\mathcal{T}$ a $\mathcal{A}$. Znalostná báza $\mathcal{K}$ je *konzistentná* vtedy a len vtedy ak má model.

Majme opäť znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, koncept C a individuál $a \in N_I$. Hovoríme, že individuál a je inštanciou konceptu C vzhľadom na $\mathcal{A}$ (označované ako $\mathcal{A} \models C(a)$), práve vtedy ak v ľubovoľnom modeli $\mathcal{A}$ máme $a^{\mathcal{I}} \in C^{\mathcal{I}}$. Individuál a je inštanciou konceptu C, vzhľadom na znalostnú bázu $\mathcal{K}$ (označované ako $\mathcal{K} \models C(a)$), práve vtedy ak v ľubovoľnom modeli $\mathcal{K}$ máme $a^{\mathcal{I}} \in C^{\mathcal{I}}$. Ak individuál a nie je inštanciou C vzhľadom na $\mathcal{K}$, tak tento fakt označujeme ako $\mathcal{K} \not\models C(a)$.

Definujme si taktiež operáciu získavania inštancií zo znalostnej bázy. Majme znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a koncept C. Výraz $R_{\mathcal{A}}(C)$ vráti všetky inštancie konceptu C vzhľadom na $\mathcal{A}$, t.j. $R_{\mathcal{A}}(C) = \{a \mid a \in N_I \wedge \mathcal{A} \models C(a)\}$. Podobne je možné definovať operáciu aj vzhľadom na znalostnú bázu $\mathcal{K}$, t.j. $R_{\mathcal{K}}(C) = \{a \mid a \in N_I \wedge \mathcal{K} \models C(a)\}$.

Dôležitými pojmami sú taktiež *subsumpcia*, *ekvivalencia* konceptov a ich *splniteľnosť*. Majme koncepty C a D. Hovoríme že C je podtriedou D ($C \sqsubseteq D$), vtedy a len vtedy, ak pre ľubovoľný model $\mathcal{I}$ platí $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Koncepty C a D sú ekvivalentné ($C \equiv D$), vtedy a len vtedy, ak $C \sqsubseteq D$ a $D \sqsubseteq C$. Koncept C je striktnou podtriedou konceptu D ($C \sqsubset D$), vtedy a len vtedy, ak $C \sqsubseteq D$ a zároveň neplatí $C \equiv D$. O koncepte C hovoríme že je splniteľný vtedy a len vtedy, ak existuje interpretácia $\mathcal{I}$, pre ktorú platí $C^{\mathcal{I}} \neq \emptyset$.

2.3 Expresivita deskripčných logík

Tak ako bolo spomenuté v predchádzajúcej kapitole, deskripčných logík existuje viacero a primárne sa označujú podľa toho, akú expresivitu ponúkajú. Medzi základnú deskripčnú logiku, ktorá sa často používa pri výskume a taktiež slúži ako základ pre viac expresívne logiky patrí deskripčná logika $\mathcal{ALC}$ (z angl. *attributive language with complement*) [110]. Logika $\mathcal{ALC}$ ponúka základné operátory ako priesek, komplement, zlúčenie, existenčné a hodnotové obmedzenie. Jednotlivé písmená v názvoch deskripčných logík vyjadrujú vlastnosť, ktorú daná logika ponúka. Zoznam vlastností:

- $\mathcal{S}$: označuje operátory, ktoré ponúka logika $\mathcal{ALC}$ spolu s tranzitivitou pre roly. Označenie $\mathcal{S}$ sa používa podľa modálnej logiky S4, pri ktorej bolo dokázané, že je homomorfná s jazykom $\mathcal{ALC}$, obohateným o tranzitivitu rolí [111]. Pre tranzitívnu rolu r platí, že ak máme $r(a, b)$ a $r(b, c)$, tak z toho vyplýva že $r(a, c)$.

- $\mathcal{H}$: označuje podrole t.j. $r \sqsubseteq s$ (viď. (2.5) v kapitole 2.2).

- $\mathcal{I}$: označuje inverzné role. r^- označuje inverznú rolu k r, t.j. $r^-(a, b)$ platí vtedy a len vtedy keď $r(b, a)$.

- $\mathcal{O}$: označuje podporu pre nominály, t.j. skonštruovanie konceptového výrazu vymenovaním individuálov (viď. kapitola 2.2).

- $\mathcal{N}$: označuje podporu pre definovanie maximálneho (minimálneho) stupňa kardinality vo forme $\geq n\, r$ alebo $\leq n\, r$.

- $\mathcal{Q}$: označuje podporu definovanie stupňa kardinality, podobne ako v predchádzajúcom prípade, avšak s podmienením vo forme $\geq n\, r.C$ a $\leq n\, r.C$ (v prípade ak C je $\top$ koncept, jedná sa o totožný prípad ako v $\mathcal{N}$).

- $\mathcal{F}$: označuje podporu pre funkcionálne role, ktoré umožňujú definovať najviac jednu rolu pre konkrétny individuál.

- $\mathcal{R}$: označuje podporu pre komplexné reťazenie rolí vo forme axióm $r \circ s \sqsubseteq r$, ktoré označujú že ak platí $r(a, b)$ a $s(b, c)$, tak zároveň platí aj $r(a, c)$. Okrem reťazenia rolí sem patrí aj lokálna reflexivita ($\exists r.Self$), univerzálna rola alebo disjunkcia rolí.

- $(\mathcal{D})$: označuje podporu pre dátové typy. Umožňujú využívať rôzne typy dát ako sú celé čísla, čísla s pohyblivou rádovou čiarkou alebo reťazce.

Medzi často používané deskripčné logiky patria napr. $\mathcal{SROIQ}(\mathcal{D})$ alebo $\mathcal{SHOIN}(\mathcal{D})$. Je však nutné poznamenať, že zatiaľ čo $\mathcal{ALC}$ slúži ako základný jazyk pre viac expresívne logiky, existujú aj jednoduchšie jazyky, ktorých cieľom je poskytnúť práve nižšiu expresivitu. Všeobecne totiž platí, že čím je jazyk expresívnejší, tým aj výpočtovo náročnejší. Medzi takéto logiky patria $\mathcal{AL}$ a $\mathcal{EL}$. Logika $\mathcal{AL}$ poskytuje existenčný/všeobecný kvantifikátor, konjunkciu a negáciu. Naopak, logika $\mathcal{EL}$ je ešte menej expresívna, poskytujúca iba dva operátory a to konjunkciu a existenčný kvantifikátor.

2.4 Modelovanie ontológií

V tejto podkapitole si vysvetlíme vzťah medzi deskripčnými logikami a jazykom Web Ontology Language (OWL) [112]. Jedná sa o jazyk, ktorý primárne slúži na modelovanie ontológií a používa sa v sémantickom webe. Taktiež sa jedná o rozšírenie všeobecnejšieho značkovacieho jazyka RDF, ktoré ponúka ďalšie elementy potrebné pre definície tried a ich vlastností. Všeobecnejšie môžeme povedať, že jazyk OWL je založený na deskripčných logikách s tým, že ponúka ďalšie vlastnosti ako použitie identifikátorov v podobe URI/IRI, ktoré sú nevyhnutné pri sémantickom webe. Založenie jazyka OWL na matematických základoch deskripčných logík so sebou pochopiteľne prináša výhody v podobe možnosti využitia rôznych komplexných zdôvodňovacích algoritmov.

V modelovacom jazyku OWL sú taktiež zaužívané iné pomenovacie konvencie v porovnaní s deskripčnými logikami. Triedy v jazyku OWL korešpondujú s konceptmi v deskripčných logikách a vlastnosti korešpondujú s rolami (pomenovanie individuálov zostáva). Samotná syntax sa taktiež odlišuje v porovnaní s klasickou deskripčnou logikou (čo je pochopiteľné z nutnosti, aby bol jazyk strojovo spracovateľný). V tabuľke 2.2 uvádzame porovnanie syntaxe niektorých matematických konštruktov v štandardnej syntaxy deskripčných logík, jazyku OWL a syntaxe *Manchester* [108]. Syntax *Manchester* je obzvlášť populárna v rôznych editoroch na modelovanie ontológií ako napr. *Protégé*[1]. V práci budeme štandardne využívať klasickú syntax deskripčných logík.

Samotný jazyk OWL ponúka tri historické verzie: OWL Lite, OWL DL a OWL Full . Okrem toho existuje aj novšia verzia OWL 2, ktorá ponúka verzie OWL 2 DL, OWL 2 Full a tri špeciálne profily EL, QL a RL (rôzne formy reštrikcie základného jazyka primárne zamerané na efektívnejšie usudzovanie) [113]. Z praktického hľadiska má zmysel však v dnešnej dobe rozlišovať medzi OWL 2 DL a OWL 2 EL/QL/RL. Popis jednotlivých profilov:

- **OWL Lite**: korešponduje s deskripčnou logikou $\mathcal{SHIF}^{\mathcal{T}}(\mathcal{D})$ (výrazom $(\mathcal{D})$ označujeme podporu pre dátové typy, viď. kapitola 2.3). Zaraďujeme ho medzi základné jazyky a obsahuje konštruktory deskripčnej logiky ako sú konjunkcia/disjunkcia, negácia, existenčný/všeobecný kvantifikátor a obmedzený operátor pre definíciu stupňa kardinality (podporuje iba hodnoty 0 a 1).

- **OWL DL**: je ekvivalentný s deskripčnou logikou $\mathcal{SHOIN}(\mathcal{D})$. Okrem toho, že podporuje rovnaké matematické operácie ako vyššie spomínaný OWL Lite, obsahuje taktiež neobmedzený stupeň kardinality. Všeobecne je jazyk považovaný za expresívnejší v porovnaní s OWL Lite, zatiaľ čo

[1] https://protege.stanford.edu/

nárast z hľadiska výpočtového výkonu je zanedbateľný [114]. Jazyk patrí medzi veľmi rozšírený, keďže postačuje základným nárokom deskripčných logík (preto názov DL).

- **OWL Full**: táto verzia jazyka obsahuje vlastnosti, ktoré nie sú reprezentovateľné v deskripčnej logike. Ponúka maximálnu expresivitu spolu s neobmedzeným použitím konštruktov z RDF (OWL Full teda môžeme vnímať ako zjednotenie OWL DL a RDF). Jazyk OWL Full však na rozdiel od verzie DL negarantuje vypočítateľnosť.

- **OWL 2 DL**: korešponduje s deskripčnou logikou $\mathcal{SROIQ}(\mathcal{D})$. Jedná sa o espresívnejší jazyk v porovnaní s OWL DL (a logikou $\mathcal{SHOIN}(\mathcal{D})$). Hlavné prínosy jazyka spočívajú najmä v oblasti modelovania samotnej ontológie t.j. zlepšené možnosti anotácií, väčšie množstvo dátových typov (s možnosťou definovať ich rozsah) a rozšírené možnosti definovania kardinality či reťazenia rolí.

- **OWL 2 Full**: podobne ako v prípade jazyka OWL Full sa jedná o expresívne zjednotenie (bez záruky vypočítateľnosti) OWL 2 DL a RDF.

- **OWL 2 EL**: korešponduje s deskričnou logikou $\mathcal{EL}^{++}$, ktorá je určená pre jednoduchšie ontológie (viď. kapitola 2.3). Obsahuje iba základné konštruktory ako sú existenčný kvantifikátor, konjunkcia a a v porovnaní s logikou $\mathcal{EL}$ pridáva inverzné roly a nominály (limitované na jeden individuál). Hlavnou výhodou je nižšia výpočtová náročnosť v porovnaní s jazykom OWL 2 DL.

- **OWL 2 QL**: obsahuje ako základ deskripčnú logiku podobnú ako OWL 2 Lite. Jazyk je zameraný najmä na ontológie, ktoré obsahujú veľké množstvo individuálov. Aj keď samotný jazyk OWL 2 QL (Query Logic) je stále relatívne expresívny, stále je dostatočne limitovaný takým spôsobom, že dotaz v OWL 2 QL možno prepísať do jazyka SQL, čím sa ontológia stáva vhodná pre prípady, kedy potrebujeme časté dotazovanie.

- **OWL 2 RL**: zatiaľ čo profil QL je zameraný na dáta v podobe klasických relačných databáz, variant RL (Rules Logic) je zameraný na škálovateľné aplikácie, ktoré potrebujú byť stále dostatočne expresívne. Samotné usudzovanie býva implementované prostredníctvom pravidlových systémov (ktoré podobne ako profil EL bežia v polynomiálnom čase). Zatiaľ čo varianty QL a DL sú podmnožinou profilu OWL 2 Full, variant RL ponúka ďalšie dva špecifickejšie varianty, pričom jeden je podmnožinou OWL 2 Full a druhý OWL 2 DL.

```
<owl:Class rdf:about="...#DynamicLinkLibrary">
    <rdfs:subClassOf rdf::resource="...#PEFile"/>
</owl:Class>

<owl:ObjectProperty rdf::about="...#has_action">
    <rdfs:domain rdf::resource="...#PEFile"/>
    <rdfs:range rdf::resource="...#Action"/>
</owl:ObjectProperty>
```

Výpis 2.1: Ukážka RDF/XML syntaxe.

Jazyk OWL ponúka viacero syntaktických formátov, pomocou ktorých môžeme ukladať znalostné bázy. Keďže samotný OWL je založený na RDF, medzi najpoužívanejšie spôsoby ukladania ontológií patria formáty RDF/XML alebo *Turtle*[2]. Taktiež existujú špeciálne varianty XML syntaxe nazývané OWL/XML a Manchester OWL syntax (spomínaná vyššie). Ukážku RDF/XML syntaxe môžeme vidieť vo výpise 2.1. V spomínanom výpise môžeme vidieť definíciu triedy (konceptu) `DynamicLinkLibrary`, ktorá je podtriedou `PEFile` a definíciu role `has_action`.

Deskripčné logiky	**OWL syntax**	**Manchester syntax**
$\top$	`Thing`	`Thing`
$\bot$	`Nothing`	`Nothing`
$C_1 \sqcap ... \sqcap C_n$	`intersectionOf`	C_1 `and ... and` C_n
$C_1 \sqcup ... \sqcup C_n$	`unionOf`	C_1 `or ... or` C_n
$\neg C$	`complementOf`	`not` C
$\{x_1\} \sqcup ... \sqcup \{x_n\}$	`oneOf`	$\{x_1, ..., x_n\}$
$\forall r.C$	`allValuesFrom`	r `only` C
$\exists r.C$	`someValuesFrom`	r `some` C
$\leq n\, r$	`maxCardinality`	r `max` n
$\geq n\, r$	`minCardinality`	r `min` n
$C_1 \sqsubseteq C_2$	`subClassOf`	C_1 `SubClassOff:` C_2
$C_1 \equiv C_2$	`equivalentClass`	C_1 `EquivalentTo:` C_2
$C_1 \equiv \neg C_2$	`disjointWith`	C_1 `DisjointWith:` C_2
$\forall r.\top \sqsubseteq C$	`domain`	r `Domain:` C
$\top \sqsubseteq \forall r.C$	`range`	r `Range:` C
$r_1 \sqsubseteq r_2$	`subPropertyOf`	r_1 `SubPropertyOf:` r_2
$r_1 \equiv r_2$	`equivalentProperty`	r_1 `EquivalentTo:` r_2

Tabuľka 2.2: Porovnanie syntaxe deskripčných logík v rôznych jazykoch.

Dôležitým aspektom jednotlivých jazykov je ich výpočtová zložitosť vzhľadom na rôzne úlohy ako napr. kontrola konzistentnosti ontológie, splniteľnosť

[2]https://www.w3.org/TR/turtle/

konceptových výrazov alebo overovanie inštancií [113]. Pri despkripčných logikách nás zaujímajú nasledujúce zložitosti:

- **Dátová zložitosť** - zložitosť, ktorá je meraná vzhľadom na celkový počet tvrdení v ontológii.
- **Taxonomická zložitosť** - zložitosť meraná vzhľadom na počet axióm v ontológii (bez tvrdení).
- **Kombinovaná zložitosť** - kombinácia predchádzajúcich zložitostí (počet tvrdení a axióm).

V tabuľke 2.3 môžeme vidieť zložitosti niektorých vybraných profilov. Samotné zložitosti sú definované podľa [115]. Význam jednotlivých položiek je nasledovný:

- **Rozhodnuteľná, otvorená zložitosť** znamená, že rozhodnuteľnosť danej deskripčnej logiky je známa, avšak nie jej presná výpočtová zložitosť. V zátvorke uvedený výraz (NP-ťažký) označuje, že problém je minimálne tak ťažký, ako ľubovoľný problém z triedy NP.
- **X-úplný**, kde **X** predstavuje niektorú z nasledujúcich tried zložitostí (zoradené od najzložitejších až po najjednoduchšie):
  - `N2EXPTIME` - trieda problémov, ktoré sú riešiteľné nedeterministickým algoritmom, ktorý je najviac dvojnásobne exponenciálny vzhľadom na veľkosť vstupu, t.j. 2^{2^n}, kde n je veľkosť vstupu.
  - `NEXPTIME` - jedná sa o triedu problémov, ktoré sú riešiteľné nedeterministickým algoritmom v čase, ktorý je najviac exponenciálny vzhľadom na veľkosť vstupu, t.j. 2^n, kde n je veľkosť vstupu.
  - `NP` - trieda problémov, ktoré sú riešiteľné nedeterministickým algoritmom v čase, ktorý je najviac polynomiálny vzhľadom na veľkosť vstupu, t.j. n^c, kde n je veľkosť vstupu a c je konštanta.
  - `PTIME` - jedná sa o triedu problémov, riešiteľných deterministickým algoritmom v čase, ktorý je najviac polynomiálny vzhľadom na veľkosť vstupu, t.j. n^c, kde n je veľkosť vstupu a c je konštanta.
  - `LOGSPACE` - trieda problémov, ktoré sú riešiteľné deterministickým algoritmom s použitím priestoru, ktorý je najviac logaritmický vzhľadom na veľkosť vstupu, t.j. $log(n)$, kde n je veľkosť vstupu. Nedeterministická verzia tejto triedy je označovaná ako `NLOGSPACE`.

– AC^0 - jedná sa o podtriedu triedy LOGSPACE, ktorá je definovaná prostredníctvom obvodov. Táto trieda označuje problémy, ktoré možno definovať prostredníctvom obvodov konštantnej hĺbky a polynomiálnej veľkosti. Patria sem problémy, ktorú sú riešiteľné pomocou polynomiálne veľa procesorov v konštantnom čase.

Profil	**Dátová zl.**	**Taxonomická zl.**	**Kombinovaná zl.**
OWL 2 EL	PTIME-úplný	PTIME-úplný	PTIME-úplný
OWL 2 RL	PTIME-úplný	PTIME-úplný	NP-úplný (PTIME-úplný pre atomické konceptové výrazy)
OWL 2 QL	AC^0	NLOGSPACE-úplný	NLOGSPACE-úplný
OWL 2 DL	Rozhodnuteľná, otvorená zložitosť (NP-ťažký)	N2EXPTIME-úplný*	N2EXPTIME-úplný*
OWL DL	Rozhodnuteľná, otvorená zložitosť (NP-ťažký)	NEXPTIME-úplný	NEXPTIME-úplný

* NEXPTIME-úplný, v prípade ak môžeme hierarchiu rolí preložiť na nedeterministický konečný automat o polynomiálnej veľkosti

Tabuľka 2.3: Výpočtová zložitosť niektorých deskripčných logík.

2.5 Konceptové učenie

V predchádzajúcich kapitolách sme si postupne predstavili základy sémantických technológií ako sú ontológie, ich matematický základ a spôsob akým ich dokážeme modelovať. Hlavnou výskumnou časťou práce je práve strojové učenie nad ontológiami, tzv. *konceptové učenie*, predstavené v práci [116].

Cieľom konceptového učenia je nájsť logický popis konceptu (vo forme konceptového výrazu definovaného v kapitole 2.2) vzhľadom na existujúce (alebo neexistujúce) inštancie konceptu v znalostnej báze [117]. Počas učenia sa postupne generujú nové hypotézy vo forme konceptových výrazov (spôsob, akým možno generovať konceptové výrazy bude predstavený v kapitole 2.6). Individuály, ktoré sú inštanciou konceptového výrazu budeme nazývať ako *pozitívne príklady* a individuály, ktoré nie sú inštanciou konceptového výrazu, budeme nazývať ako *negatívne príklady*. Ak určitý individuál vieme popísať konceptovým výrazom (je teda jeho inštanciou), vravíme, že výraz *pokrýva* daný príklad. Konceptový výraz, ktorý pokrýva všetky pozitívne príklady, budeme označovať ako *úplný*. Naopak výraz, ktorý nepokrýva žiadny negatívny príklad, označujeme ako *konzistentný*. Konceptový výraz, ktorý je zároveň *úplny* a *konzistentný*, teda pokrýva všetky pozitívne a žiadny negatívny príklad, budeme označovať ako *korektný*. Majme teda znalostnú bázu $\mathcal{K}$, definovanú v konkrétnej deskripčnej logike. Pozitívne a negatívne príklady sú individuály, ktoré sa nachádzajú v znalostnej báze a cieľom je násť hypotézu vo forme komplexného konceptového výrazu.

Samotné konceptové učenie má svoj pôvod v *induktívnom logickom programovaní* (ILP) [118, 119]. ILP metódy používajú logický program ako bázu znalostí a snažia sa nájsť logický program popisujúci pozitívne príklady, ktorý zároveň nepopisuje žiadne negatívne príklady. Hlavným rozdielom je, že ILP metódy využívajú logické programy ako znalostné bázy, zatiaľ čo konceptové učenie sa spolieha na deskripčné logiky a OWL.

Konceptové učenie má v praxi viacero využití. Prvotným zámerom konceptového učenia bolo strojové rozširovanie ontológie. Ak teda máme nejakú základnú verziu ontológie, aplikovaním konceptové učenia môžeme získať nové konceptové výrazy, ktoré vieme do danej ontológie doplniť a získať tak expresívnejšiu a komplexnejšiu bázu znalostí. Ďalším využitím konceptového učenia (čo je aj hlavným cieľom práce), je aplikácia naučených výrazov na riešenie klasifikačných problémov. Z vyššie uvedeného textu vyplýva, že konceptové učenie je vhodné najmä na binárnu klasifikáciu (t.j. rozlišovanie medzi dvoma triedami). Existujú však aj iné variácie konceptového učenia.

Všeobecne existujú tri rôzne varianty konceptového učenia [108], ktoré si definujeme nasledovne:

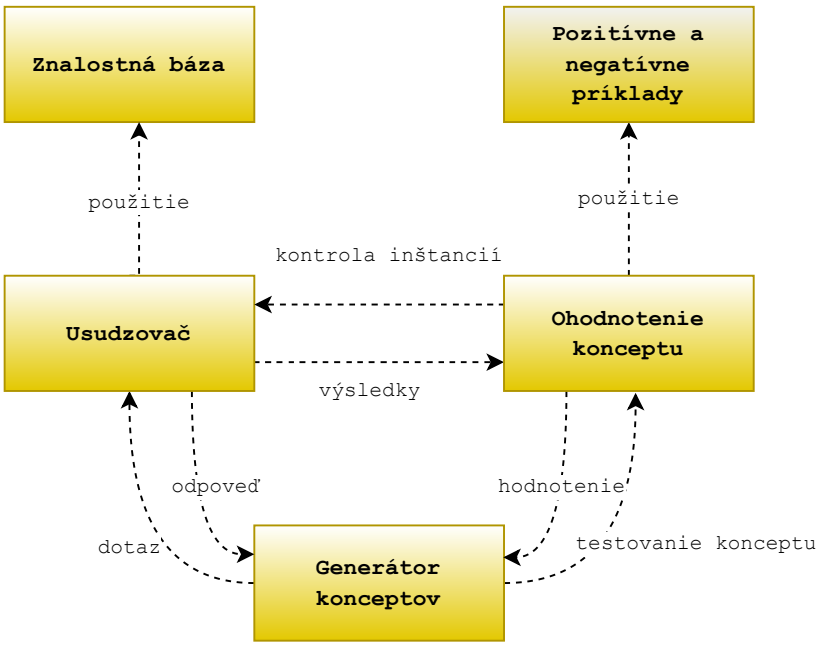
Učenie s pozitívnymi/negatívnymi príkladmi: Jedná sa o základný typ učenia, ktorý už bol z časti popísaný aj vyššie. Majme znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. Majme dve navzájom disjunktné množiny E^+ (pozitívne príklady) a E^- (negatívne príklady). Platí, že $E \subseteq N_I$ (N_I je množina všetkých individuálov v znalostnej báze), kde $E = E^+ \cup E^-$. Cieľom učenia je následne nájsť taký konceptový výraz C, pre ktorý platí $\forall e \in E^+ : \mathcal{K} \models C(e)$ a zároveň $\forall e \in E^- : \mathcal{K} \not\models C(e)$; t.j. výraz, ktorý pokrýva všetky pozitívne príklady a žiadne z negatívnych.

Učenie iba s pozitívnymi príkladmi: Pri tomto variante učenia, na rozdiel od predchádzajúceho, poskytujeme algoritmu iba pozitívne vzorky. Cieľom učenia je teda nájsť konceptový výraz, ktorý pokrýva čo najviac pozitívnych príkladov. Technicky je možné učenie iba s pozitívnymi príkladmi pretransformovať na učenie s pozitívnymi a negatívnymi príkladmi, ak si zadefinujeme množinu $E^- = N_I \setminus E^+$, kde ako negatívne príklady použijeme všetky individuály v ontológii, okrem pozitívnych príkladov. Formálne majme znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a množinu E^+, kde platí že $E^+ \subseteq N_I$. Cieľom učenia je následne nájsť taký konceptový výraz C, pre ktorý platí $R_{\mathcal{K}}(C) = R_K(E^+)$.

Učenie konceptu: V poslednom variante učenia je cieľom nájsť popis konceptu D v znalostnej báze. Majme znalostnú bázu $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a majme koncept $D \in N_C$. Cieľom učenia je následne nájsť taký koncept C, pre ktorý platí že $R_{\mathcal{K}}(C) = R_{\mathcal{K}}(D)$. Pri tomto variante učenia, na rozdiel od predchádzajúcich variantov, nepoužívame žiadne explicitné pozitívne alebo negatívne príklady a naučený koncept C je popisom inštancií konceptu D (samozrejme $C \neq D$). Využitie takého konceptu je následne napr. v podobe rozšírenie znalostnej bázy definovaním axiómy $D \equiv C$ alebo $D \sqsubseteq C$.

Je nutné poznamenať, že deskričpné logiky všeobecne fungujú na príncípe *otvoreného sveta* OWA (Open World Assumption), kde samotnú znalostnú bázu považujeme za nekompletnú a tvrdenia, o ktorých nevieme rozhodnúť či sú pravdivé, nepovažujeme automaticky za nepravdivé. Z toho dôvodu niektoré algoritmy konceptového učenia môžu vyžadovať, aby negatívne príklady boli inštancie $\neg C$, resp. $\mathcal{K} \models \neg C(e) : \forall e \in E^-$ (viď. kapitola 3.5). Algoritmy používané v tejto práci, však využívajú opačný prístup *uzavretého sveta* CWA (Closed World Assumption), podobný prístup ako v relačných databázach, kde považujeme znalostnú bázu za kompletnú a ak niektoré fakty nepoznáme, považujeme ich za nepravdivé.

Taktiež je nutné poznamenať, že algoritmy konceptového učenia s najväčšou pravdepodobnosťou nikdy nenájdu *korektný* výraz, ale iba jeho aproximá-



Obrázok 2.1: Schematické znázornenie konceptového učenia.

ciu (z dôvodu šumu v ontológii). Výrazy s vysokou mierou korektnosti však nie sú vhodné z hľadiska možného pretrénovania, kedy naučené výrazy síce korektne popisujú trénovaciu časť znalostnej bázy, avšak nie sú dostatočne všeobecné a nepredikujú dáta v testovacej množine.

Všeobecne môžeme konceptové učenie definovať taktiež ako prehľadávací proces v množine všetkých konceptov (ktorých je samozrejme nekonečne veľa). Schematické znázornenie konceptového učenia môžeme vidieť na obrázku 2.1. Jedným zo základných elementov je generátor konceptov, ktorý postupne generuje nové hypotézy (vo forme komplexných konceptových výrazov). Generátor konceptov pri vytváraní nových hypotéz používa automatický usudzovač, ktorý pracuje nad znalostnou bázou (softvér, ktorý dokáže automaticky odvodzovať nové fakty v ontológii). Kvalita vygenerovaných konceptov je následne ohodnocovaná podľa vybranej metriky. Jednou z metrík môže byť napríklad *pokrytie* pozitívnych a negatívnych príkladov. Na kontrolu *pokrytia* býva zvyčajne opäť využívaný automatický usudzovač. Medzi ďalšie metriky môže patriť napr. informačný zisk, dĺžka konceptu a pod.

2.6 Spresňujúci operátor

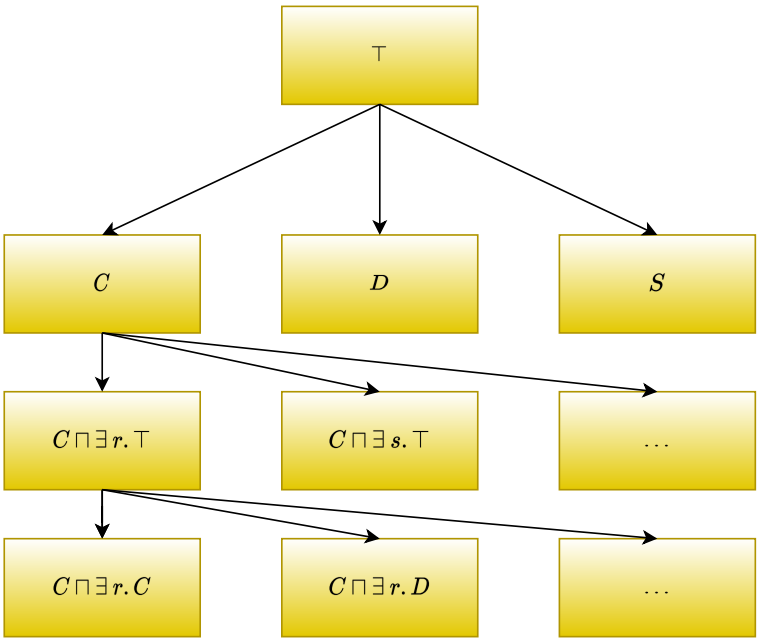
Ako sme si popísali v kapitole 2.5, jedným zo základným komponentov konceptové učenia je generátor nových konceptov. Na tento účel sa zvyčajne v konceptovom učení využíva tzv. spresňujúci operátor (z angl. *refinement operator*) [116]. Operátor si môžeme v jednoduchosti predstaviť ako funkciu, do ktorej vstupuje koncept C a výstupom je špecifickejší koncept C', ktorý môže byť rozšírený o ďalší koncept, logický operátor a pod. (ako si ukážeme neskôr, existuje aj opačný spresňujúci, resp. generalizujúci operátor). V konceptovom učení môžeme nádejný koncept, ktorý dosahuje dobré hodnoty metriky, spresniť pomocou operátora, získať tak nový koncept, ktorý môžeme znova ohodnotiť. Takýmto spôsobom vieme vytvárať prehľadávací priestor.

Majme množinu konceptov $N_C = \{C, D, S\}$ a množinu rolí $N_R = \{r, s\}$. Schematické znázornenie, ako môže prebiehať vytváranie prehľadávacieho stromu, môžeme vidieť na obrázku 2.2. Na danom obrázku môžeme vidieť tzv. *top-down* prístup (resp. špecializácia), kedy začíname vytvárať prehľadávajúci strom od vrchného konceptu $\top$. V prvej iterácii môže byť vstupom $\top$ a výstupom množina atomických konceptov C, D a S. Následne spresnením konceptu C môže byť napr. pridanie konjunkcie a začlenenie všetkých rolí v znalostnej báze, vygenerovaním konceptov $C \sqcap \exists\, r.\top$ a $C \sqcap \exists\, s.\top$. Takýmto spôsobom vieme vygenerovať nekonečne veľký strom. Existujú aj opačné prístupy, tzv. *bottom-up* (resp. zovšeobecňovanie), ktoré začínajú od spodného konceptu $\bot$ a postupne sa snažia koncepty generalizovať.

Formálnejšie, spresňujúci operátor a prehľadávací priestor vieme definovať predstavením tzv. *kváziusporiadania*, reflexívnej a tranzitívnej relácie. V *kváziusporiadanom* priestore $(S, \preceq)$ je spresňujúci operátor ρ mapovaním $\rho : S \mapsto 2^S$, kde pre každý koncept $C \in S$ a $C' \in \rho(C)$ (teda C' je výstupom spresnenia C pomocou ρ), platí, že $C' \preceq C$. Koncept C' nazývame špecializáciou konceptu C (pri *top-down* prístupe). Podobne vieme definovať aj opačný, zovšeobecňujúci princíp, kde mapovanie je opačné, $\rho : 2^S \mapsto S$ a platí $C \preceq C'$.

Pri budovaní prehľadávacieho priestoru v deskripčnej logike môžeme ako operátor, ktorý určuje usporiadanie priestoru, definovať operátor subsumpcie ($\sqsubseteq$). Ak koncept C zahŕňa koncept D ($C \sqsubseteq D$), koncept C bude pokrývať taktiež všetky príklady, ktoré pokrýva aj koncept D.

V nasledujúcej časti si definujeme niektoré dôležité pojmy týkajúce sa spresňujúcich operátorov. Majme ľubovoľnú deskripčnú logiku $\mathcal{L}$. Spresňujúci operátor v *kváziusporiadanom* priestore $(\mathcal{L}, \sqsubseteq)$ budeme nazývať $\mathcal{L}$ *spresňujúci operátor*. Následne, spresňujúcou reťazou (z angl. *refinement chain*) $\mathcal{L}$ spresňujúceho operátora ρ dĺžky n z konceptu C na koncept D budeme označovať konečnú postupnosť konceptov $C_0, C_1, C_2, ...C_n$, pre ktoré platí $C = C_0, C_1 \in \rho(C_0), C_2 \in \rho(C_1), ..., C_n \in \rho(C_{n-1}), D = C_n$. Takáto reťaz



Obrázok 2.2: Znázornenie vytvárania prehľadávajúceho stromu pomocou spresňujúceho operátora.

môže prechádzať cez koncept E, ak existuje také i $(1 \leq i \leq n)$, pre ktoré platí $E = C_i$. Taktiež, dokážeme dosiahnuť spresnenie na koncept D z konceptu C pomocou operátora ρ, ak existuje spresňujúca reťaz z C na D.

Dôležitým pojmom je taktiež tzv. *pokrytie smerom nadol* (z angl. *downward cover*). Koncept C pokrýva koncept D smerom nadol, ak platí $C \sqsubset D$ a zároveň neexistuje koncept E, pre ktorý platí $C \sqsubset E \sqsubset D$ (podobne je možné definovať aj pokrytie smerom nahor, resp. *upward cover*). Taktiež si zavedieme pojem tzv. *slabej ekvivalentnosti* dvoch konceptov (z angl. *weak equality*), ktorá na rozdiel od syntaktickej rovnosti neberie do úvahy poradie argumentov pri konjunkcii a disjunkcii (tj. koncept $C \sqcap D$ je rovný konceptu $D \sqcap C$). Majme dva koncepty C a D. Slabo rovné koncepty potom budeme označovať ako $C \simeq D$.

Spresňujúce operátory môžu mať rôzne vlastnosti, podľa ktorých môžeme definovať vhodnosť daného operátora pre konkrétny typ učenia. Majme $\mathcal{L}$ spresňujúci operátor ρ. Operátor ρ môže spĺňať nasledujúce vlastnosti:

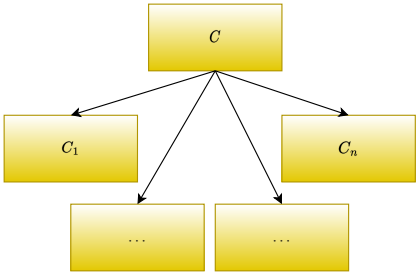
- **Konečnosť**: operátor ρ označujeme ako *konečný*, ak pre všetky koncepty C je výstup z operátora $\rho(C)$ konečný. Výstupom konečného operátora ρ je vždy konečná množina spresnených konceptov (viď. obrázok 2.3a).
- **Redundantnosť**: operátor ρ označujeme ako *redundantný*, ak exis-

tuje spresňujúca reťaz z konceptu C na koncept D, ktorá neprechádza nejakým konceptom E (t.j. koncept E nedosiahneme postupným spresňovaním konceptu C) a zároveň existuje spresňujúca reťaz z konceptu C na koncept slabo ekvivalentný s konceptom D, ktorá naopak prechádza konceptom E. Ak je operátor redundantný, vieme sa k ekvivalentnému (alebo slabo ekvivalentnému) konceptu dostať viacerými cestami (viď. obrázok 2.3b).

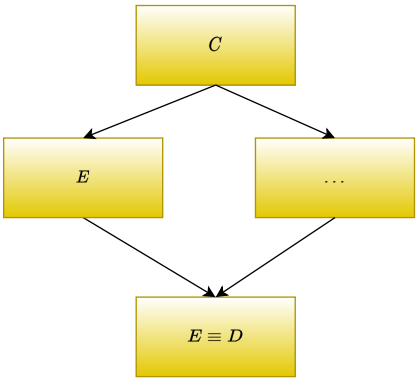
- **Správnosť**: hovoríme, že operátor ρ je *správny*, ak pre všetky koncepty C a D, kde $D \in \rho(C)$ platí $C \not\equiv D$ (t.j. výstup spresnenia nikdy nie je ekvivalentný vstupnému konceptu, viď. obrázok 2.3c).
- **Úplnosť**: operátor ρ označujeme ako *úplný*, ak pre všetky koncepty C a D, pre ktoré platí $C \sqsubset D$, vieme dosiahnuť koncept E z konceptu D, pre ktorý platí $E \equiv C$ (viď. obrázok 2.3d).
- **Ideálnosť**: ako *ideálny* spresňujúci operátor označujeme taký operátor ρ, ktorý je zároveň *konečný*, *úplný* a *správny*.
- **Slabá úplnosť**: platí, ak pre všetky koncepty $C \sqsubset \top$ vieme dosiahnuť koncept E z $\top$ taký že $E \equiv C$.
- **Minimálnosť**: operátor ρ označujeme ako *minimálny*, ak pre všetky koncepty C obsahuje výsledok spresnenia $\rho(C)$ iba pokrytia smerom nadol (pre generalizujúci operátor pokrytia smerom nahor).

Podľa dôkazu, uvedeného v práci [108], ak berieme do úvahy vlastnosti ako *úplnosť*, *konečnosť*, *slabá úplnosť*, *správnosť* a *neredundancia*, môže $\mathcal{L}$ *spresňujúci operátor*, kde $\mathcal{L} \in \{\mathcal{ALC}, \mathcal{ALCQ}, \mathcal{SHOIN}, \mathcal{SROIQ}\}$ (resp. expresívnejšie jazyky) spĺňať iba nasledujúce maximálne množiny vlastností (v zmysle, že žiadnu ďalšiu vlastnosť nevieme pridať bez toho, aby sme inú odobrali):

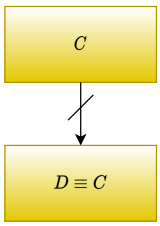
- *slabá úplnosť*, *úplnosť*, *konečnosť*
- *slabá úplnosť*, *úplnosť*, *správnosť*
- *slabá úplnosť*, *neredundancia*, *konečnosť*
- *slabá úplnosť*, *neredundancia*, *správnosť*
- *neredundancia*, *konečnosť*, *správnosť*



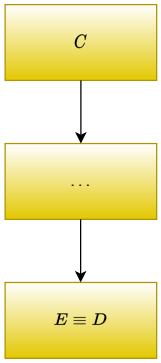
(a) Konečnosť



(b) Redundancia



(c) Správnosť



(d) Úplnosť

Obrázok 2.3: Schematické znázornenie niektorých vlastností spresňujúceho operátora.

Kapitola 3

Algoritmy konceptového učenia

V tejto časti práce sa hlbšie venujeme konceptovému učeniu a jednotlivým algoritmom. Úvodná podkapitola 3.1 je venovaná nástroju DL-Learner, ktorý bol použitý v rámci práce a zároveň implementuje všetky analyzované algoritmy. Časť 3.2 sa venuje teoretickému popisu spresňujúceho operátora ρ (vrátane detailného popisu jeho fungovania), ktorý tvorí základ analyzovaných algoritmov, predstavených v kapitole 3.3. V rámci tejto kapitoly sú postupne predstavené štyri základné algoritmy, kde každý pracuje na odlišných princípoch: OCEL, CELOE, PARCEL a SPACEL. Podkapitola 3.4 sa venuje niektorým úpravám v nástroji DL-Learner, ktoré pomohli zefektívniť učiaci proces. Keďže existujú aj ďalšie algoritmy konceptového učenia (pričom ich vývoj stále pokračuje), ich popisu sa venujeme v podkapitole 3.5. Posledná podkapitola 3.6 sa venuje ďalším metódam, ktoré poskytujú interpretovateľné výsledky. Napriek tomu, že metódy predstavené v tejto podkapitole nesúvisia priamo s konceptovým učením, považovali sme za dôležité ich uviesť, keďže samotná interpretovateľnosť patrí medzi dôležité aspekty tejto práce.

3.1 Softvér DL-Learner

V tejto kapitole si popíšeme softvér DL-Learner, ktorý patrí v súčasnosti medzi najpoužívanejší *framework* pre štrukturované strojové učenie v deskripčných logikách [120]. Všetky experimenty uvádzané v tejto práci boli vykonávané prostredníctvom nástroja DL-Learner. Samotný nástroj je implementovaný v jazyku Java, prostredníctvom aplikačného rámca Java Spring. DL-Learner taktiež patrí medzi nástroje s otvoreným zdrojovým kódom[1]. Celkovú architektúru nástroja môžeme vidieť na obrázku 3.1. Ako môžeme vidieť, architektúru tvorí päť hlavných komponentov:

- *Zdroj znalostí*: tento komponent nástroja definuje, kde sa nachádzajú samotné znalosti a akým spôsobom ich vieme získavať. DL-Learner podporuje všetky známe serializačné formáty pre RDF a OWL. Okrem lokálnych znalostných báz, podporuje aj vzdialené získavanie znalostí prostredníctvom SPARQL. V rámci učenia je taktiež možné použiť viacero zdrojov znalostí, pričom je možné kombinovať lokálne a vzdialené zdroje.

- *Usudzovač*: ako bolo spomínané v kapitole 2.5, usudzovač je nástroj, ktorý robí automatickú inferenciu v znalostných bázach a je dôležitou súčasťou konceptového učenia. DL-Learner umožňuje pripojiť sa k usudzovaču prostredníctvom štandardného OWL API, protokolu *OWLLink* alebo taktiež umožňuje aj priamy prístup v prípade, ak sú potrebné určité pokročilé vlastnosti, ktoré neponúka API rozhranie. Okrem známych usudzovačov, ako sú *Pellet*, *HermiT* alebo *FaCT++*, ponúka DL-Learner aj vlastný tzv. *closed world reasoner*, ktorý pracuje na báze uzavretého sveta (CWA). Tento usudzovač poskytuje aj určité optimalizácie pre overovanie inštancií (jedna z výkonovo najnáročnejších častí konceptového učenia), predpočítaním inferencií a ich uložením v pamäti.

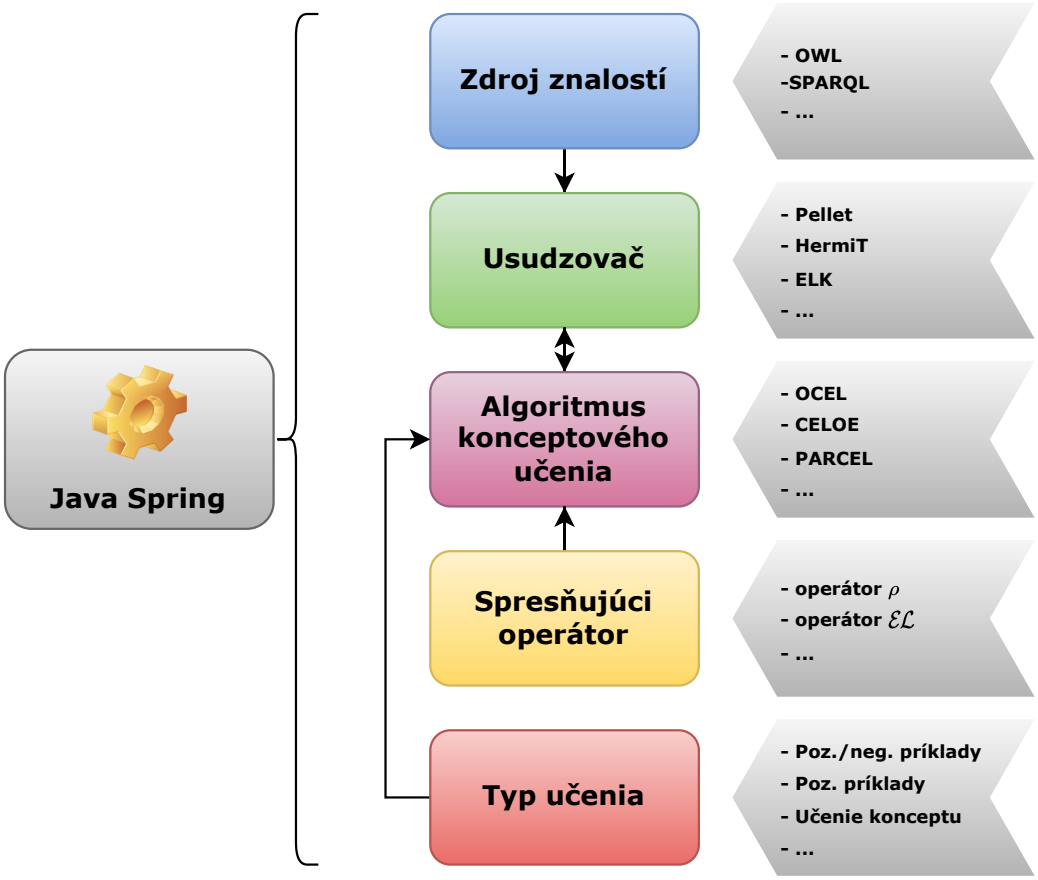
- *Typ učenia*: tento komponent slúži na definíciu problému, ktorý sa snažíme pomocou konceptového učenia riešiť. Učiace algoritmy používajú tento komponent následne na testovanie novo vygenerovaných hypotéz. Ako sme spomínali v kapitole 2.5, existujú tri základné typy (ktoré zároveň aj podporuje DL-Learner): učenie s pozitívnymi a negatívnymi prvkami, učenie iba s pozitívnymi prvkami a učenie konceptu.

- *Spresňujúci operátor*: tento komponent definuje spresňujúci operátor, ktorý sa používa na generovanie konceptov (viac v kapitole 2.6). DL-Learner ponúka dva základné operátory: ρ (viac v kapitole 3.2) a špe-

[1] https://github.com/SmartDataAnalytics/DL-Learner

ciálny operátor pre logiku $\mathcal{EL}$. DL-Learner umožňuje taktiež konfigurovať spresňujúci operátor tak, aby zodpovedal konkrétnemu fragmentu OWL (napr. vypnutie/zapnutie negácie, nominálov, operátora kardinality, atď.).

- *Algoritmy konceptového učenia*: sem patria algoritmy, ktoré implementujú konkrétnu učiacu stratégiu. Patria sem algoritmy ako OCEL, CELOE, PARCEL, ELTL atď. Niektoré vybrané algoritmy, ktoré sme skúmali v rámci našej práce sú detailnejšie popísané v rámci kapitoly 3.3.



Obrázok 3.1: Architektúra softvéru DL-Learner.

3.2 Spresňujúci operátor ρ

V tejto podkapitole si popíšeme spresňujúci operátor ρ [116], ktorý tvorí základ všetkých algoritmov konceptového učenia použitých v tejto práci (kapitola 3.3) v rámci softvéru DL-Learner (kapitola 3.1). V kapitole 2.6 sme spomínali, že spresňujúci operátor môže spĺňať maximálne tri vlastnosti. Operátor ρ spĺňa vlastnosti *slabá úplnosť*, *úplnosť* a *správnosť*. Vlastnosti ako *slabá úplnosť* a *úplnosť* sú pre operátor dôležité z toho hľadiska, aby bol operátor schopný konvergovať k správnemu riešeniu, ak nejaké riešenie existuje. To, že operátor spĺňa spomínané vlastnosti zároveň znamená, že spĺňa aj negatívne vlastnosti v podobe *nekonečnosti* a *redundancie*. Takéto problémy spresňujúceho operátora je však možné riešiť algoritmicky (viac v kapitole 3.3).

Pre potreby definície spresňujúceho operátora ρ je nutné definovať niektoré ďalšie funkcie. Funkcia *sh* (z angl. *subsumption hierarchy*) vracia ďalší koncept (resp. rolu), ktorá nasleduje v hierarchii subsumpcie (podľa ktorej máme usporiadaný prehľadávací priestor). Funkcia $sh_{\downarrow}(A)$ platí pre všetky koncepty $A \in N_C$ (viď. (3.1)). Podobne platí aj funkcia $sh_{\downarrow}(r)$ pre všetky $r \in N_R$ (viď. (3.2)). Analogicky je taktiež možné definovať funkcie $sh_{\uparrow}$, v prípade ak sa posúvame v hierarchií subsumpcie smerom nahor.

$$sh_{\downarrow}(A) = \{A' \in N_C \mid A' \sqsubset A \wedge \nexists A'' \in N_C \Rightarrow A' \sqsubset A'' \sqsubset A\} \tag{3.1}$$

$$sh_{\downarrow}(r) = \{r' \in N_R \mid r' \sqsubset r \wedge \nexists r'' \in N_R \Rightarrow r' \sqsubset r'' \sqsubset r\} \tag{3.2}$$

Funkcia $domain(r)$ označuje doménu role r, t.j. definuje inštancie akého konceptu C sa môžu nachádzať v predikáte $r(C, D)$ (t.j. na ľavej strane). Obdobne funkcia $range(r)$ definuje, aké inštancie sa môžu nachádzať na pravej strane (teda D). Potom môžeme definovať funkciu $ad(r)$ (viď. (3.3)), ktorá vracia atomické koncepty podľa domény (z angl. *atomic domain*). Obdobne je možné definovať funkciu $ar(r)$ (z angl. *atomic range*).

$$ad(r) = \{A \in \{\top\} \cup N_C \wedge domain(r) \sqsubseteq A \wedge \nexists A' \Rightarrow domain(r) \sqsubseteq A' \sqsubset A\} \tag{3.3}$$

Množina app_B (z angl. *applicable*) obsahuje všetky aplikovateľné role, vzhľadom na atomický koncept B (viď. (3.4)). Cieľom tejto množiny v definícii operátora je zmenšiť prehľadávací priestor a vylúčiť tak koncepty, ktoré sú nesplniteľné.

$$app_B = \{r \mid r \in N_R, ad(r) = A, A \sqcap B \not\equiv \bot\} \tag{3.4}$$

Množinu mgr_B (z angl. *most general applicable roles*), vzhľadom na atomický koncept B, možeme definovať nasledovne:

$$mgr_B = \{r \mid r \in app_B, \nexists r' \Rightarrow r \sqsubset r', r' \in app_B\} \tag{3.5}$$

Potom množinu M_B, kde $B \in N_C \cup \{\top\}$, môžeme definovať ako zjednotenie nasledujúcich množín:

- $\{A \mid A \in N_C, A \sqcap B \not\equiv \bot, A \sqcap B \not\equiv B, \nexists A' \in N_C \Rightarrow A \sqsubset A'\}$
- $\{\neg A \mid A \in N_C, \neg A \sqcap B \not\equiv \bot, \neg A \sqcap B \not\equiv B, \nexists A' \in N_C \Rightarrow A' \sqsubset A\}$
- $\{\exists r.\top \mid r \in mgr_B\}$
- $\{\forall r.\top \mid r \in mgr_B\}$

Operátor $\rho_B(C)$ môžeme následne definovať pomocou spresňujúcich pravidiel, kde index B (ktorý je na začiatku vždy $B = \top$) je nastavený na atomický rozsah rolí, ktoré obsahuje vstupný koncept C a používa sa na vylúčenie konceptov, ktoré sú disjunktné s konceptom B. Spresňujúce pravidlá pre operátor $\rho_B(C)$ potom môžeme definovať nasledovne:

1. ak $C = \bot$, tak $\rho_B(C) = \emptyset$
2. ak $C = \top$, tak $\rho_B(C) = \{C_1 \sqcup C_2 \sqcup ...C_n \mid C_i \in M_B, 1 \leq i \leq n\}$
3. ak $C = A$, tak $\rho_B(C) = \{A' \mid A' \in sh_{\downarrow}(A)\} \cup \{A \sqcap D \mid D \in \rho_B(\top)\}$
4. ak $C = \neg A$, tak $\rho_B(C) = \{\neg A' \mid A' \in sh_{\uparrow}(A)\} \cup \{\neg A \sqcap D \mid D \in \rho_B(\top)\}$
5. ak $C = \exists r.D$, tak $\rho_B(C) = \{\exists r.E \mid A = ar(r), E \in \rho_A(D)\} \cup \{\exists r.D \sqcap E \mid E \in \rho_B(\top)\} \cup \{\exists s.D \mid s \in sh_{\downarrow}(r)\}$
6. ak $C = \forall r.D$, tak $\rho_B(C) = \{\forall r.E \mid A = ar(r), E \in \rho_A(D)\} \cup \{\forall r.D \sqcap E \mid E \in \rho_B(\top)\} \cup \{\forall r.\bot \mid D = A\ a\ sh_{\downarrow}(A) = \emptyset\} \cup \{\forall s.D \mid s \in sh_{\downarrow}(r)\}$
7. ak $C = C_1 \sqcap ... \sqcap C_n$, tak $\rho_B(C) = \{C_1 \sqcap ... \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap ... \sqcap C_n \mid D \in \rho_B(C_i), 1 \leq i \leq n\}$
8. ak $C = C_1 \sqcup ... \sqcup C_n$, tak $\rho_B(C) = \{C_1 \sqcup ... \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup ... \sqcup C_n \mid D \in \rho_B(C_i), 1 \leq i \leq n\} \cup \{(C_1 \sqcup ... \sqcup C_n) \sqcap D \mid D \in \rho_B(\top)\}$

Ako môžeme vidieť, pravidlo č. 2 sa používa pri spresňovaní konceptu $\top$ (t.j. na začiatku tvorby prehľadávacieho priestoru), kde sa využíva množina M_B. Pravidlá 3 a 4 naopak definujú spresňovanie pre atomický koncept (resp. negáciu atomického konceptu). Pravidlá 5 a 6 naopak definujú spresňovanie pre role.

Pre jednoduchosť si uveďme nasledujúci príklad. Majme znalostnú bázu $\mathcal{K} = \{B \sqsubset A; C \sqsubset A; Y \sqsubset X; Z \sqsubset X; A \sqcap X \equiv \bot; domain(r) = X; range(r) = A\}$. Aplikáciou pravidla 2 a spresnením konceptu $\top$ v znalostnej báze $\mathcal{K}$ môžeme dostať množinu $\rho(\top) = \{X, A, \neg Y, \neg Z, \neg B, \neg C, \exists r.\top, \forall r.\top, X \sqcup X, X \sqcup A, ...\}$. V danom príklade môžeme vidieť, akým spôsobom sa skonštruuje množina $M_\top$ pri počiatočnom spresnení. Koncept $X \sqcap \exists r.A$, potom môže mať nasledujúce spresnenia $\rho(X \sqcap \exists r.A) = \{X \sqcap \exists r.B, X \sqcap \exists r.C, Y \sqcap \exists r.A, Z \sqcap \exists r.A, ...\}$. V príklade môžeme vidieť aplikáciu pravidla 3 (spresnenie atomického konceptu) najprv na koncept A, kde spresnením sú ďalšie koncepty v hierarchii subsumpcie (t.j. koncepty B a C) a potom na koncept X (t.j. koncepty Y a Z). Ako ďalšie spresnenie môže nasledovať pravidlo 7, ktoré definuje akým spôsobom spresniť konjunkciu konceptov a pravidlo 5, ktoré definuje spresnenie role s existenčným kvantifikátorom. Je opätovne nutné spomenúť, že operátor ρ je nekonečný.

V predchádzajúcej časti sme si definovali základné pravidlá pre spresňujúci operátor ρ, ktoré dokážu pokryť aj expresívnejšie logiky. Operátor ρ však ponúka aj ďalšie rozšírenia, ktoré dokážu spresňovať koncepty s definíciou kardinality vzťahu alebo dátových vlastností. Pomocou takéhoto rozšírenia vieme následne tvoriť koncepty ako (3.6) (spustiteľný súbor, ktorý vykonáva aspoň tri akcie) alebo (3.7) (sekcia, ktorá má výšku entropie viac ako 7.253).

$$ExecutableFile \sqcap \geq 3\, has\_action.\top \tag{3.6}$$

$$Section \sqcap entropy \geq 7.253 \tag{3.7}$$

Jedným z problémov tohoto rozšírenia je, že ak povolíme napr. dátový typ `double`, koncept (3.7), môže byť spresňovaný na koncepty v podobe $Section \sqcap entropy \geq x$, kde x môže byť ľubovoľná hodnota získaná z analýzy dostupnej znalostnej bázy (ako si ukážeme ďalej). Pochopiteľne, množina všetkých reálnych čísel je nekonečná, čím nemôžeme dosiahnuť všetky koncepty pri spresňovaní. Operátor ρ sa tak stáva neúplným.

Majme definovanú množinu mgd_B (z angl. *most general double*) definovanú analogicky k množine mgr_B (viď. (3.5)). Rovnako môžeme definovať aj množiny pre ďalšie dátové typy ako mgb_B, pre dátový typ `boolean`. Ďalšie číselné dátové typy, ako napr. `integer`, možno definovať podobným spôsobom ako pre `double` a z toho dôvodu ich nebudeme explicitne uvádzať. Majme

definovanú množinu N_{DCR}, kde sú uvedené všetky hodnoty `double` nachádzajúce sa v konkrétnej znalostnej báze. Ďalej, máme definovanú množinu $values_d$ (kde $d \in N_{DCR}$), v ktorej sa nachádzajú všetky konkrétne hodnoty typu `double`, usporiadané od najmenšej hodnoty po najväčšiu. Označenie $values_d[i]$ bude definovať i-tý prvok v množine. Parameter $\#splits_d \in N$, bude označovať používateľom zadávaný parameter, ktorý definuje koľko spresňujúcich krokov má vykonať operátor, aby prehľadal priestor všetkých `double` hodnôt (keďže nechceme prehľadávať nekonečný priestor). Množina $splits_d$ potom bude obsahovať nasledujúce hodnoty:

$$\{t_j \mid i = \frac{\#values_d}{\#splits_d + 1}, \\ t = \frac{1}{2}(values_d[\lfloor i \cdot j \rfloor] + values_d[\lfloor i \cdot j \rfloor + 1])\ pre\ 1 \leq j \leq \#splits_d\} \quad (3.8)$$

Taktiež, na spresňovanie kardinality pre role je nutné definovať hodnotu $mf_r = max_{a \in N_I} \mid \{b \mid \mathcal{K} \models r(a,b)\} \mid$, ktorá slúži ako maximálny limit pri spresňovaní.

Vyššie spomínanú množinu M_B je potom možné doplniť nasledujúcimi množinami (pre spresňovanie kardinality vzťahu a dátových typov `boolean` a `double`):

- $\{\leq mf_r\ r.\top \mid r \in mgr_B\}$
- $\{b = true \mid b \in mgb_B\}$
- $\{b = false \mid b \in mgb_B\}$
- $\{d \geq v \mid d \in mgd_B, v = splits_d[\#splits_d]\}$
- $\{d \leq v \mid d \in mgd_B, v = splits_d[1]\}$

Potom, spresňujúci operátor $\rho_B(C)$ môžeme rozšíriť o nasledujúce pravidlá:

1. ak $C = \exists r.D$, tak $\rho_B(C) =\geq 2\ r.D$
2. ak $C =\geq n\ r.D$, tak $\rho_B(C) = \{\geq n+1\ r.D \mid n < mf_r\} \cup \{\geq n\ r.E \mid E \in \rho_B(D)\}$
3. ak $C =\leq n\ r.D$, tak $\rho_B(C) = \{\geq n-1\ r.D \mid n > 1\} \cup \{\leq n\ r.E \mid E \in \rho_B(D)\}$
4. ak $C = (b = true)$, tak $\rho_B(C) = \emptyset$

5. ak $C = (b = false)$, tak $\rho_B(C) = \emptyset$
6. ak $C = (d \geq v)$, tak $\rho_B(C) = \{d \geq w \mid v = splits_d[i], i > 1, w = splits_d[i-1]\}$
7. ak $C = (d \leq v)$, tak $\rho_B(C) = \{d \leq w \mid v = splits_d[i], i < \#splits_d, w = splits_d[i+1]\}$

Ak chceme použiť spresňujúci operátor ρ na klasifikačnú úlohu, tak jedným z najdôležitejších faktorov, ktorý rozhoduje o tom, či daná úloha bude úspešná, je fakt do akej miery je cieľový jazyk (resp. jeho expresivita) vhodný na nájdenie riešenia. V softvéri DL-Learner (kapitola 3.1) je možné špecifikovať expresivitu výrazov ignorovaním konkrétnych konceptov/rolí alebo zapnutím/vypnutím operátorov, využívaných pri spresňovaní. Nastavovať je možné všeobecný/existenčný kvantifikátor, negáciu, nominály, dátové typy atď. Nastavenie správnej expresivity spresňujúceho operátora zohráva dôležitú úlohu, keďže správnym nastavením dokážeme zmenšiť prehľadávací priestor a teoreticky tak rýchlejšie nájsť riešenie.

3.3 Algoritmy

V tejto podkapitole si postupne detailnejšie popíšeme jednotlivé algoritmy, ktoré sme skúmali v rámci našej práce. Patria sem štyri algoritmy implementované v rámci softvéru DL-Learner: OCEL, CELOE, PARCEL a SPACEL. Každý z týchto algoritmov využíva spresňujúci operátor ρ, ktorý bol bližšie popísaný v kapitole 3.2. Taktiež, ako sme spomínali v kapitole 2.5, každý algoritmus konceptového učenia môže byť definovaný ako dvojica: generátor konceptov a heuristika. Generátor konceptov je v našom prípade spresňujúci operátor ρ. Heuristika naopak rozhoduje o kvalite konceptov, o tom ktorý koncept je vhodné ďalej spresňovať a celkovo o stratégii prehľadávania priestoru konceptov.

3.3.1 OCEL

OWL Class Expression Learner (OCEL) patrí medzi najzákladnejší algoritmus konceptového učenia [108]. V kapitole 2.6 sme spomínali, že operátor ρ, ktorý využíva aj algoritmus OCEL, je redundantný a nekonečný. Z toho dôvodu sú tieto problémy riešené algoritmicky. Jedným zo spôsobov ako riešiť redundanciu je prechádzať jednotlivé koncepty v prehľadávacom strome a pokiaľ nájdeme *slabo rovný* koncept, tak sa zahodí a ďalej sa nebude spresňovať. Ako príklad si môžeme uviesť koncept $A_1 \sqcap A_2$, ktorý je *slabo rovný* s konceptom $A_2 \sqcap A_1$ (nie syntakticky rovný). Takýto spôsob je však relatívne výpočtovo náročný. V algoritme OCEL je redundancia riešená takým spôsobom, že autori zaviedli aj usporiadanie jednotlivých konceptov v množine konjunkcií a disjunkcií, pomocou ktorého je možné skontrolovať redundanciu pre koncept v lineárnom čase. Jedná sa o relatívne dôležitú súčasť algoritmu, keďže medzi najviac výpočtovo náročný element konceptového učenia patrí overovanie inštancií vygenerovaného konceptu (pri ohodnocovaní). Zredukovaním prehľadávacieho priestoru o redundantné koncepty tak vieme zrýchliť učiaci proces.

Ďalším negatívom spresňujúceho operátora ρ, ktorý je potrebné riešiť v rámci učiaceho algoritmu je nekonečnosť. Algoritmus OCEL rieši nekonečnosť operátora takým spôsobom, že v jednom čase spresňuje konkrétny koncept iba po určitú dĺžku n. Parameter n sa nazýva aj ako *horizontálna expanzia* konceptu v prehľadávacom strome. Tento parameter sa môže následne počas učenia dynamicky zväčšovať, ak sme už prehľadali všetky koncepty určitej dĺžky.

Formálne, každý jeden element v prehľadávacom strome je definovaný ako trojica $N = (C, n, b)$, kde C označuje konkrétny konceptový výraz, $n \in N$ označuje *horizontálnu expanziu* konceptu C a $b \in \{true, false\}$ označuje, či je daný koncept C redundantný alebo nie.

Pre definíciu heuristiky algoritmu OCEL (teda spôsobu, pomocou ktorého rozhodujeme, ktoré koncepty budeme ďalej spresňovať) si potrebujeme uviesť parameter *noise*, resp. šum. Jedná sa o podobný parameter aký sa používa v rôznych ILP systémoch. Parameter obvykle označuje minimálnu správnosť pri trénovaní $(1 - noise)$, ktorú musí algoritmus dosiahnuť, aby sme považovali riešenie za akceptovateľné a ukončili proces učenia. Ako $N = (C, n, b)$ budeme označovať jeden element v prehľadávacom strome.

Ako *up* (z angl. *uncovered positives*) budeme označovať všetky nepokryté pozitívne príklady (viď. (3.9)), kde E^+ je množina všetkých pozitívnych príkladov a $R(C)$ označuje všetky pokryté inštancie pomocou konceptového výrazu C.

$$up = |E^+ \backslash R(C)| \tag{3.9}$$

Ako *cn* (z angl. *covered negatives*) budeme označovať všetky pokryté negatívne príklady, kde množina E^- označuje všetky negatívne príklady (viď. (3.10)).

$$cn = |R(C) \cap E^-| \tag{3.10}$$

Správnosť konceptového výrazu C je následne definovaná ako podiel nepokrytých pozitívnych príkladov a pokrytých negatívnych príkladov voči všetkým príkladom (viď. kapitola (5.1)).

$$accuracy(C) = 1 - \frac{up + cn}{|E|} \tag{3.11}$$

Funkciou *acc_gain* následne označujeme rozdiel správnosti medzi konceptom C a konceptom C', kde C' je koncept v rodičovskom elemente N v prehľadávacom strome (viď. (3.12)).

$$acc\_gain(N) = accuracy(C) - accuracy(C') \tag{3.12}$$

Následne, funkcia pre ohodnotenie elementu N v strome $score(N)$ je definovaná ako (3.13). Samotná heuristika je primárne založená na prediktívnej správnosti, pričom váha pri zlepšení správnosti je kontrolovaná parametrom α a dĺžka horizontálnej expanzie pomocou parametra β (parametre sú štandardne nastavené ako $\alpha = 0.5$ a $\beta = 0.02$).

$$score(N) = accuracy(C) + \alpha \cdot acc\_gain(N) - \beta \cdot n \tag{3.13}$$

Samotný algoritmus následne funguje relatívne jednoducho. Prehľadávací strom začína od elementu $N = (\top, 0, false)$ a beží pokiaľ nenájdeme koncept s dostatočnou správnosťou (kontrolovanou parametrom *noise*). Na začiatku

každej iterácie sa vyberie z prehľadávajúceho stromu element s najvyššou hodnotou funkcie $score(N)$. Daný element sa následne spresní s inkrementovanou hodnotou n a zároveň sa odstránia všetky redundantné koncepty. Zároveň koncepty, ktoré pokrývajú málo pozitívnych príkladov t.j. $up > |E| \cdot noise$, sú automaticky vyhadzované z prehľadávacieho stromu a ďalej sa nespresňujú.

3.3.2 CELOE

Algoritmus Class Expression Learner for Ontology Engineering (CELOE) je priamo postavený nad algoritmom OCEL [121]. Hlavným cieľom autorov bolo navrhnúť algoritmus, ktorý by bol použiteľný priamo na rozširovanie ontológie. Cieľom je naučiť sa koncept C, aby bolo následne možné obohatiť ontológiu o axiómy v podobe výrazov ako $A \equiv C$ alebo $A \sqsubseteq C$. Samotný algoritmus je založený na rovnakých princípoch ako OCEL. Jediným rozdielom je použitá heuristika, ktorej cieľom je preferencia kratších konceptov, keďže kratšie konceptové výrazy sú jednoduchšie na pochopenie (čo je dôležitý faktor, keďže hlavné použitie algoritmu je pre manuálne vylepšovanie ontológie). Samotný algoritmus bol primárne navrhnutý na variant **učenia konceptu** (viď. kapitola 2.5), kde je cieľom popísať koncept A, iným konceptovým výrazom C. V takom prípade sa ako pozitívne príklady použijú inštancie konceptu A a ako negatívne príklady všetky ostatné individuály. Pochopiteľne je možné aplikovať aj klasický variant s pozitívnymi a negatívnymi príkladmi.

Keďže hlavnou zmenou voči algoritmu OCEL je heuristika, je potrebné si definovať niektoré funkcie. Samotná správnosť konceptu sa počíta nasledujúcim spôsobom:

$$acc_c(C,t) = \frac{1}{t+1} \cdot \left(t \cdot \frac{|R(A) \cap R(C)|}{|R(A)|} + \sqrt{\frac{|R(A) \cap (C)|}{|R(C)|}} \right) \tag{3.14}$$

Ako môžeme vidieť v (3.14), samotná miera správnosti sa v CELOE počíta podobne ako F1 miera, t.j. kombinácia presnosti a senzitivity (viď. kapitola 5.2.2). Ako $R(A)$ označujeme inštancie cieľového konceptu A a ako $R(C)$ označujeme inštancie aktuálne testovaného konceptu C. Parameter t sa používa na penalizovanie falošne negatívnych výsledkov.

Podobne ako v OCEL aj v CELOE sa počíta rozdiel v správnosti. Rozdielom je použitie funkcie definovanej v (3.14). C' opätovne označuje koncept v rodičovskom elemente v strome. Funkcia *acc_gain* vyzerá následovne:

$$acc\_gain(C) = acc_c(C,t) - acc_c(C',t) \tag{3.15}$$

Funkcia na ohodnotenie konceptu C potom vyzerá nasledovne:

$$score(C) = acc_c(C,t) + \alpha \cdot acc\_gain(N) - \beta \cdot |C| \tag{3.16}$$

Samotné skóre je počíta veľmi podobným spôsobom ako v prípade OCEL. Hlavným kritériom pri ohodnocovaní je správnosť acc_c spolu s vylepšením správnosti v podobe funkcie *acc_gain*. Funkcia *acc_gain* je kontrolovaná parametrom α. Kontrolovaná je taktiež dĺžka konceptu $|C|$, pomocou parametra β. Štandardné nastavenie parametrov je $\alpha = 0.3$ a $\beta = 0.05$.

3.3.3 PARCEL

Algoritmus Parallel Class Expression Learner (PARCEL) [122] je podobný ako algoritmus DL-FOIL (viac v kapitole 3.5), kde je oddelené hľadanie čiastkových riešení od výpočtu finálneho konceptu. Jednou z hlavných výhod algoritmu PARCEL v porovnaní s vyššie spomínanými algoritmami OCEL a CELOE, je, že PARCEL je plne paralelizovateľný, čím dokáže prehľadať väčší priestor.

Formálne, majme znalostnú bázu $\mathcal{K}$ a množinu pozitívnych (E^+) a negatívnych príkladov (E^-). Koncept C budeme označovať ako *korektný* ak nepokrýva žiadny z negatívnych príkladov. Ak koncept C nepokrýva žiadny pozitívny príklad, budeme ho označovať ako *slabý*. V prípade, ak koncept C pokrýva aspoň jeden a menej ako všetky pozitívne príklady, budeme ho považovať za *čiastkové riešenie*. V ideálnom prípade, ak pokrýva všetky pozitívne príklady, koncept nazývame ako *úplný*.

Schematické znázornenie algoritmu PARCEL môžeme vidieť na obrázku 3.2. Hlavnou časťou algoritmu sú pracovné vlákna, ktorých cieľom je hľadanie *čiastkových riešení* (t.j. nepokrývajú žiadny negatívny príklad a zároveň pokrývajú aspoň jeden pozitívny príklad). Samotné vlákna si vyberajú koncept zo **zoznamu konceptov**, ktorý následne spresnia pomocou spresňujúceho operátora. Ako všetky spomínané algoritmy, aj PARCEL využíva spresňujúci operátor ρ. Jedinou zmenou je, že operátor má vypnutú možnosť spresňovania pomocou operátora disjunkcie, keďže tento operátor sa používa na záver pri redukcii a tvorbe finálneho konceptu. Samotný **zoznam konceptov** je zoradený podľa heuristiky, kde na začiatku zoznamu sa nachádza najsľubnejší koncept. Heuristika je nastavená podobným spôsobom ako pri OCEL a CELOE. Samotná funkcia na ohodnocovanie kvality konceptu C je definovaná nasledovne:

$$score(C) = correctness(C) + \alpha \cdot acc\_gain(C) + \beta \cdot accuracy(C) - \gamma \cdot n - \delta \cdot m \tag{3.17}$$

Skóre je kombináciou *korektnosti*, rozdielu správnosti v porovnaní s rodičovským konceptom (podobne ako v (3.12), pričom je kontrolovaný parametrom α) a samotnou správnosťou (kontrolovaná parametrom β). Dlhé koncepty sú kontrolované parametrom *gamma* (kde n je dĺžka konceptu C). Heuristika taktiež penalizuje koncepty, ktoré boli už veľakrát spresňované, čím sa stávajú výpočtovo náročnejšie. Tento fakt je v heuristike zohľadňovaný parametrom δ, pričom m definuje počet spresnení konceptu C. Štandardné nastavenie parametrov je $\alpha = 0.2$, $\beta = 0.01$, $\gamma = 0.05$ a $\delta = 0.0001$.

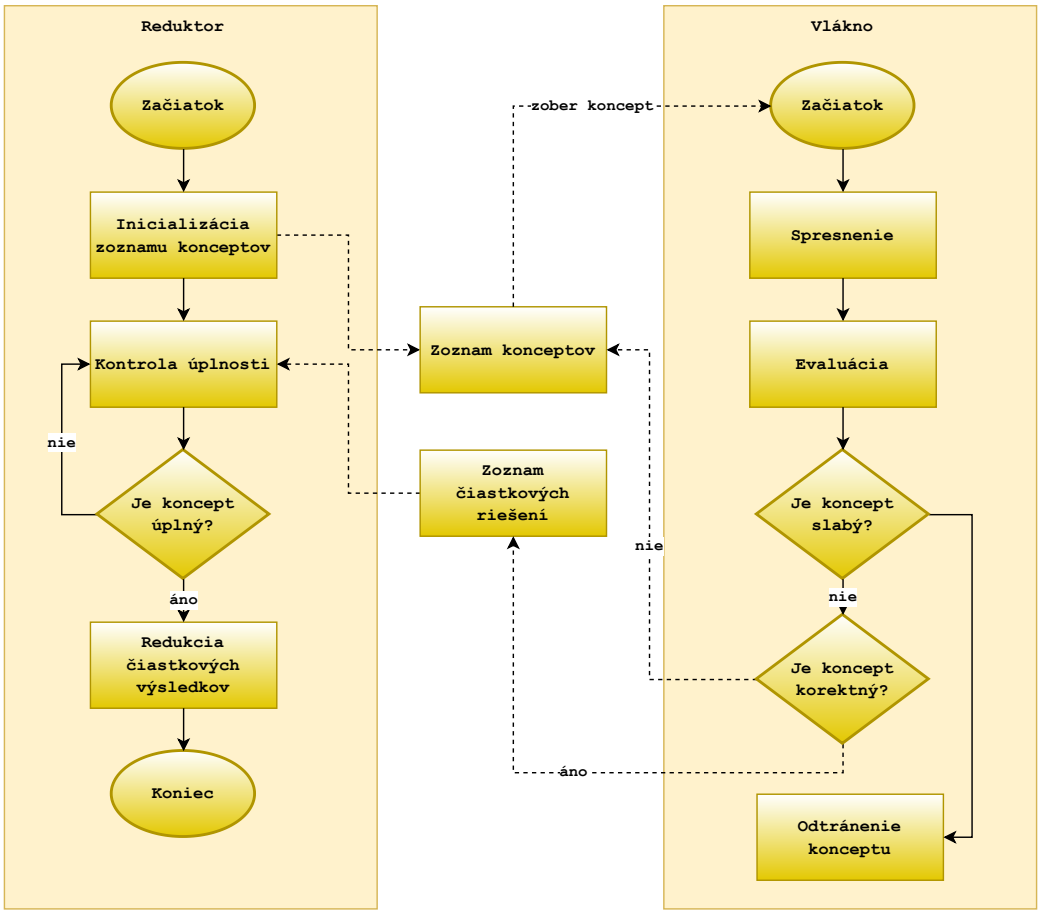
Vlákno po spresnení konceptu prejde na jeho evaluáciu (pokrytie pozitívnych a negatívnych príkladov). V prípade ak je spresnený koncept *slabý*, tak sa automaticky zahadzuje. Ak prejde koncept aj testom na korektnosť, uloží sa do **zoznamu čiastkových riešení**. Ak však koncept nie je *slabý* ale neprejde testom na korektnosť, vráti sa späť do **zoznamu konceptov**, kde môže byť následne spresnený v inom vlákne. Druhá časť algoritmu sa nazýva *reduktor* (ktorá taktiež beží v samostatnom vlákne). V tejto časti sa inicializuje samotný **zoznam konceptov** a následne kombinuje koncepty zo **zoznamu čiastkových riešení** pokiaľ nenájde *úplny* koncept (resp. koncept s dostačujúcou *úplnosťou*). Po kombinácii čiastkových riešení (pomocou disjunkcie) prichádza redukcia, keďže jednotlivé čiastkové riešenia môžu byť redundantné v zmysle, že pokrývajú rovnakú množinu pozitívnych príkladov.

Pre samotnú redukciu, implementuje PARCEL *greedy* optimalizačné algoritmy, založené na usporiadaní čiastkových výsledkov. Po zoradení čiastkových výsledkov sa vytvorí nová redukčná množina, do ktorej sa postupne pridávajú čiastkové riešenia, pokiaľ dané riešenie pokrýva aspoň jeden pozitívny príklad, ktorý nebol dovtedy pokrytý iným riešením v množine. V rámci PARCEL sú dostupné tri redukčné *greedy* algoritmy, na základe rôznych stratégií usporiadania čiastkových riešení:

- GMPC - v tomto algoritme sú čiastkové riešenia usporiadané podľa počtu pozitívnych príkladov, ktoré pokrývajú. V prípade ak dve čiastkové riešenia pokrývajú rovnaký počet pozitívnych príkladov, algoritmus preferuje lexikografické usporiadanie podľa samotného reťazca reprezentujúceho koncept (kvôli opakovateľnosti experimentov).

- GMPL - v tomto algoritme sú čiastkové riešenia usporiadané podľa dĺžky konceptu, pričom sa preferujú krátke koncepty. V prípade rovnakej dĺžky je použité lexikografické usporiadanie, podobne ako pri GMPC.

- GOLR - pri tomto prípade sa používa na usporiadanie množiny čas, kedy bolo nájdené čiastkové riešenie. Preferované riešenia sú teda tie, ktoré boli nájdené skôr. Výhoda tohto algoritmu je, že zatiaľ čo predchádzajúce dva bolo možné aplikovať až potom ako sme našli už istú množinu



Obrázok 3.2: Schematické znázornenie algoritmu PARCEL.

čiastkových riešení, pri GOLR je možné aplikovať redukciu ihneď po nájdení nového čiastkového riešenia. Nevýhodou je, že algoritmus je nedeterministický, keďže čas kedy je čiastkové riešenie nájdené závisí od plánovača vláken v operačnom systéme.

Ako najväčšiu výhodu, najmä v porovnaní s neparalelnými algoritmami OCEL a CELOE môžeme spomenúť jeho škálovateľnosť. Neparalelné algoritmy pracujú sériovo a hľadajú jeden špecifický konceptový výraz, ktorý je riešením problému. Takýto prístup môže byť značne náročný v prípade, ak učenie prebieha na veľkej ontológii s problémom, ktorý si vyžaduje relatívne dlhý a zložitejší konceptový výraz. PARCEL na druhú stranu hľadá čiastkové riešenia, ktoré následne skombinuje pomocou disjunkcie do jedného veľkého výrazu. V prípade, ak by bolo riešením daného problému skutočne výraz zložený z viacerých disjunkcií, napr. výraz dĺžky 16 (4 čiastkové riešenia dĺžky 3, spojené 4 operátormi disjunkcie), algoritmy ako OCEL a CELOE, by sa potrebovali vnoriť v prehľadávacom strome až po hĺbku 16. V prípade algoritmu PARCEL by však stačilo vnorenie do hĺbky 4. Samotný charakter konceptového učenia hovorí taktiež v prospech paralelných prístupov, keďže sa jedná o prehľadávací proces a možnosť prehľadať viac konceptov za rovnaký čas, môže byť výhodné pri náročnejších klasifikačných úlohách.

3.3.4 SPACEL

Algoritmus Symmetric Parallel Class Expression Learner (SPACEL) je priamym nasledovníkom algoritmu PARCEL, predstaveného v kapitole 3.3.3 [123]. Hlavným rozdielom je, že SPACEL, na rozdiel od ostatných algoritmov, berie do úvahy aj negatívne príklady. Algoritmy ako OCEL, CELOE a PARCEL berú do úvahy negatívne prvky iba implicitne, kde sú zahrnuté v rámci definícií pozitívnych príkladov. Z toho dôvodu je SPACEL invariantný ak by sme prepínali medzi pozitívnymi a negatívnymi príkladmi. Algoritmus bol inšpirovaný prípadmi, kedy dobre zovšeobecňujúci konceptový výraz nepredikuje správne relatívne malé množstvo špeciálnych prípadov. Z toho dôvodu sa autori rozhodli hľadať *čiastkové riešenia* nielen pre pozitívne príklady (ako v prípade algoritmu PARCEL), ale aj pre negatívne príklady. Negácie čiastkových riešení (ktoré definujú výnimky, resp. špeciálne prípady pri učení) je potom možné kombinovať s čiastkovými riešeniami pre pozitívne príklady pomocou rôznych stratégií.

Fungovanie algoritmu môžeme zhrnúť do troch hlavných krokov:

- Hľadanie čiastkových riešení (pre pozitívne a negatívne príklady) tak, aby spoločne tieto riešenia pokrývali čo najviac pozitívnych a negatívnych príkladov.

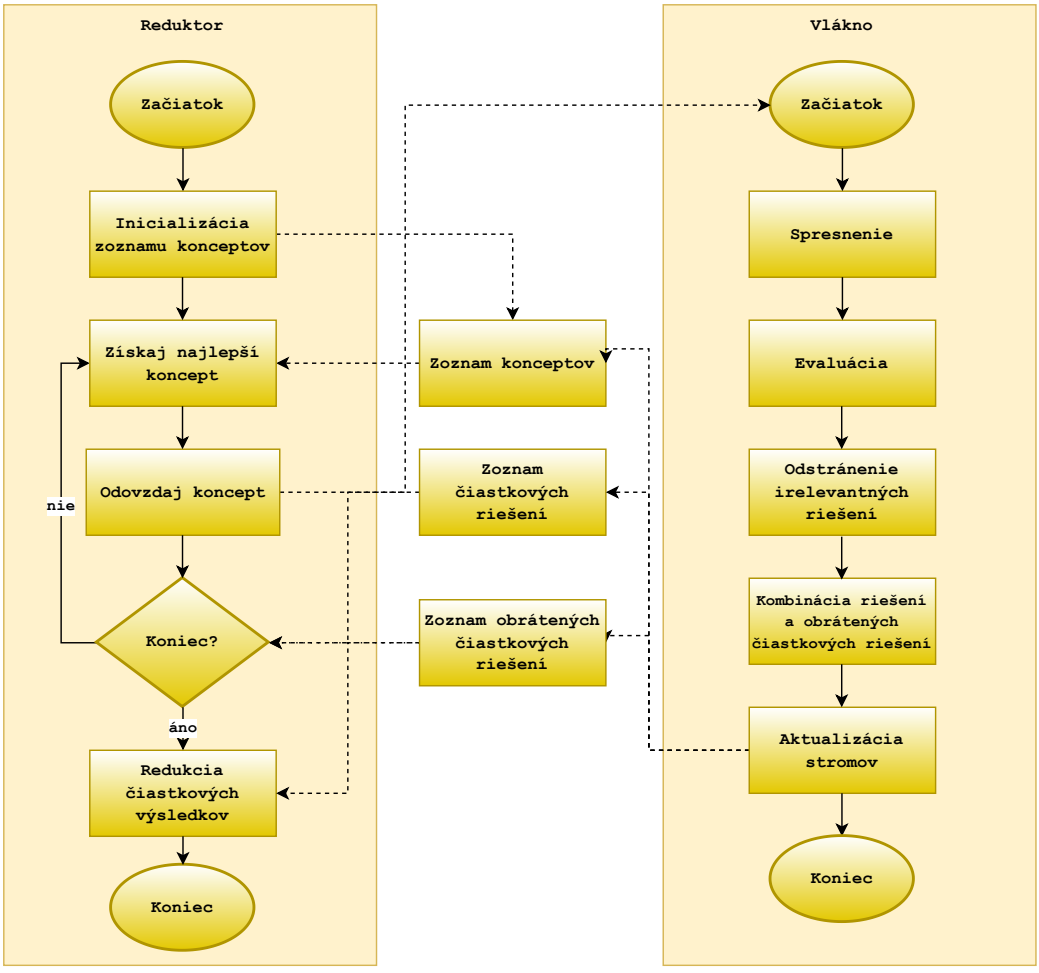
- Odstránenie redundancií v čiastkových riešeniach (t.j. odstránenie takých čiastkových riešení, ktoré pokrývajú podmnožinu príkladov, ktoré pokrýva iné čiastkové riešenie) a výber najlepších kandidátov na skonštruovanie finálneho úplneho riešenia.
- Agregácia najlepších kandidátov na tvorbu úplného riešenia pomocou disjunkcie.

Definícia *čiastkových riešení*, *úplnosti* a *koreknosti* pre konceptové výrazy je rovnaká ako v prípade algoritmu PARCEL. V prípade SPACEL však špecificky označujeme čiastkové riešenia, ktoré popisujú negatívne príklady, ako *obrátené čiastkové riešenia* (z angl. *counter partial definitions*). Podobne ako PARCEL, aj SPACEL využíva spresňujúci operátor ρ s určitými obmedzeniami. Prvým obmedzením je nemožnosť spresňovať koncepty pomocou operátora disjunkcie, keďže tento operátor sa využíva na kombinovanie čiastkových riešení rovnakým spôsobom ako pri PARCEL. Druhým obmedzením je nemožnosť spresňovania pomocou operátora negácie, ktorý sa používa pri redukcii. Operátor tak využíva iba 6 z 8 pravidiel pre spresňujúci operátor ρ, definovaných v kapitole 3.2.

Schematické zobrazenie algoritmu SPACEL môžeme vidieť na obrázku 3.3. Ako môžeme vidieť na obrázku, samotná architektúra je veľmi podobná algoritmu PARCEL. Hlavné vlákno algoritmu, nazývané **reduktor**, po inicializácii **zoznamu konceptov** z neho vyberá najlepší koncept a odovzdáva ho pracovným vláknam. Skóre, ktoré ohodnocuje koncepty, je opätovne veľmi podobné funkcii, ktorú používa PARCEL (viď. (3.17)). Funkcia $score(C)$ na ohodnotenie konceptu C je definovaná nasledovne:

$$score(C) = correctness(C) + \alpha \cdot acc\_gain(C) + \beta \cdot completeness(C) - \gamma \cdot n \tag{3.18}$$

Skóre je kombináciou korektnosti, úplnosti a rozdielu správnosti voči rodičovskému konceptu. Štandardné nastavenie parametrov je $\alpha = 0.2$, $\beta = 0.01$ a $\gamma = 0.05$. Hlavné vlákno vyberá najlepšie čiastkové riešenia a odovzdáva ich pracovným vláknam, až pokiaľ nenastane ukončovacia podmienka (t.j. pokiaľ čiastkové riešenia nedosiahnu požadovanú úplnosť alebo neubehne stanovený čas). Po splnení podmienky prejde hlavné vlákno k redukcii čiastkových riešení a tvorbe finálneho konceptu. Samotné pracovné vlákna si vyberajú koncepty zo **zoznamu konceptov**, na ktoré použijú modifikovaný spresňujúci operátor ρ. Po spresnení sa daný koncept vyhodnotí z hľadiska pokrytia pozitívnych a negatívnych príkladov. Po evaluácii sa odstránia tzv. *irelevantné koncepty*, ktoré nepokrývajú pozitívne ani negatívne príklady,



Obrázok 3.3: Schematické znázornenie algoritmu SPACEL.

t.j. spresnením nedokážeme získať čiastkové riešenia. Nové čiastkové riešenia a obrátené čiastkové riešenia sú potom uložené do korešpondujúcich dátových štruktúr, t.j. **zoznamu konceptov**, **zoznamu čiastkových riešení** a **zoznamu obrátených čiastkových riešení**. Novo spresnené koncepty môžu byť taktiež skombinované s obrátenými čiastkovými riešeniami, čím môžu vzniknúť nové čiastkové riešenia. Samotný kombinačný algoritmus je založený na riešení optimalizačného problému *set cover* [124], kde je cieľom nájsť množiny, ktoré možno vyhodiť (t.j. negatívne príklady, ktoré pokrýva konkrétne obrátené čiastkové riešenie) tak, aby zjednotenie zvyšných množín pokrývalo všetky negatívne príklady.

Obrátené čiastkové riešenia sú kombinované s výrazmi zo **zoznamu konceptov** (t.j. prehľadávací strom) tak, aby vytvárali nové čiastkové riešenia vo forme $C \sqcap \neg(D_1 \sqcup D_2 \sqcup ... D_N)$, kde C je koncept z prehľadávacieho stromu a D_n sú obrátené čiastkové riešenia. V rámci algoritmu SPACEL sú implementované tri rôzne stratégie pre kombináciu obrátených čiastkových riešení:

- Neskorá kombinácia (z angl. *late combination*); pri tejto stratégii si algoritmus počas celého behu udržuje množiny čiastkových riešení a množiny obrátených čiastkových riešení. Samotná kombinácia nastáva až keď sa dosiahne ukončovacia podmienka v podobe času alebo dostatočného pokrytia príkladov, t.j. obrátené čiastkové riešenia sa kombinujú až na konci.

- Kombinácia za behu (z angl. *on-the-fly combination*); pri tejto stratégii (ktorá je znázornená aj v rámci obrázku 3.3), sú obrátené čiastkové riešenia kombinované za behu, priamo v pracovných vláknach.

- Odložená kombinácia (z angl. *delayed combination*); jedná sa o kombináciu predchádzajúcich dvoch stratégií. Pri tejto stratégii sa skúma možnosť kombinácie priamo za behu, avšak namiesto kombinácie sú koncepty vložené do množiny potenciálnych čiastkových riešení. Kombinácia následne prebieha až po ukončení algoritmu.

Autori v práci uvádzali, že na základe evaluácie sa ako najlepšia možnosť javila kombinácia obrátených čiastkových riešení za behu.

3.4 Úpravy v DL-Learner

V tejto podkapitole si v stručnosti popíšeme niektoré zmeny (opravené chyby, resp. optimalizácie) implementované v softvéri DL-Learner, ktoré pomohli zlepšiť samotný priebeh učenia. Zmeny boli detailnejšie popísané v článku [8].

Maximálna kardinalita. Jednalo sa o chybu v spresňujúcom operátori ρ, pri spresňovaní maximálnej kardinality (t.j. $\leq nr.D$, viď. kapitola 3.2). Konkrétne sa jednalo o chybu v spresnení z $\leq nr.D$ na $\leq nr.E$, pre ľubovoľný koncept D, rolu r a $n \in N$. V takomto prípade (keď $E \sqsubseteq D$) sa nejednalo o spresnenie, ale generalizáciu.

Negácie a všeobecný kvantifikátor. Jedná sa o dve menšie optimalizácie. Prvou optimalizáciou je obmedzenie spresňujúceho operátora ρ, aby generoval koncepty vo forme $\neg C$, kde C je superkoncept v doméne spresňovania (t.j. vo výraze $C \sqsubseteq D$, je C je subkoncept a D je superkoncept). Týmto spôsobom je možné redukovať prehľadávací strom. Druhou optimalizáciou je skrátenie spresňovania z vrchného konceptu $\top$ na univerzálnu reštrikciu vo forme $\forall r.\bot$. V pôvodnej verzii bolo potrebné dosiahnuť koncept,$\forall r.C$, (kde C je taký koncept, že neexistuje žiadny ďalší koncept D, pre ktorý platí $D \sqsubseteq C$), aby mohol byť koncept následne spresnený na výraz $\forall r.\bot$. V niektorých prípadoch sa však spresňovanie nemusí dostať až do takej hĺbky (kvôli tomu, že algoritmus nevyhodnotí dané koncepty ako dostatočne sľubné na ďalšie spresňovanie). Z toho dôvodu sa zaviedol výraz $\forall r.\bot$ ako priame spresnenie výrazu $\forall r.\top$.

Heuristika v OCEL. Jeden z nezdokumentovaných komponentov heuristickej funkcie algoritmu OCEL, je pridávanie bonusu ku skóre pre koncepty, ktoré obsahujú $\forall r.\top$ alebo $\leq nr.\top$. Táto funkcia však bola nedeterministická a zároveň nezahŕňala, koľkokrát sa dané výrazy vyskytli v konceptovom výraze. Samotné zlepšenie tejto funkcie spočívalo v oprave jej nedeterminizmu a zároveň aj v lineárnom náraste skóre pre koncept, ak obsahuje dané výrazy.

Spresnenie rovnakej dĺžky. Jedna z chýb, identifikovaných v paralelných algoritmoch PARCEL a SPACEL bola, že spresňujúci operátor neumožňoval spresnenie na koncept rovnakej dĺžky ako jeho predchodca (napr. nebolo možné spresnenie z $\forall r.\top$ na $\forall r.C$).

Výpočet správnosti a pokrytia v PARCEL. Pri výpočte správnosti konceptového výrazu a jeho pokrytia pozitívnych príkladov používal

PARCEL vždy iba doposiaľ nepokryté pozitívne vzorky. Tento aspekt algoritmu bol reimplementovaný tak, aby používal štandardnú definíciu správnosti a pokrytia s dvoma doplňujúcimi pravidlami. Prvé pravidlo hovorí, že nové čiastkové riešenie, ktoré pokrýva aspoň jeden z doposiaľ nepokrytých pozitívnych príkladov, môže byť ďalej spresňované. Druhé pravidlo naopak hovorí, že konceptový výraz môže byť považovaný za čiastkové riešenie iba vtedy, ak pokrýva viac doposiaľ nepokrytých pozitívnych príkladov ako doposiaľ nepokrytých negatívnych príkladov.

Zrýchlenie výpočtu správnosti a pokrytia. Keďže operátor ρ generuje subkoncepty, na výpočet správnosti a pokrytia vygenerovaného konceptu stačí poznať, ktoré z pozitívnych a negatívnych príkladov, ktoré pokrýva pôvodny koncept, zároveň pokrýva aj nové spresnenie. Takáto optimalizácia bola následne zavedená pre algoritmy PARCEL a SPACEL, kde pre každý element v strome bola pridaná informácia ohľadom pokrytých pozitívnych a negatívnych príkladov.

Optimalizácia prehľadávacieho stromu. Paralelné algoritmy PARCEL a SPACEL používaju na reprezentáciu prehľadávacieho stromu dátovú štruktúru `ConcurrentSkipListSet`, ktorá je dostupná v rámci knižníc jazyka Java. Štandardne boli koncepty v tejto štruktúre zoradené podľa skóre, získaného z heuristiky (t.j. najlepšie výrazy boli uložené na konci). Podľa oficiálnej dokumentácie je však efektívnejšie získavať prvky zo začiatku štruktúry ako z jej konca. Obrátenie elementov v tejto štruktúre prinieslo takmer dvojnásobné zrýchlenie.

3.5 Ďalšie algoritmy

V tejto kapitole si v krátkosti predstavíme ďalšie algoritmy konceptového učenia, ktoré sme neskúmali v rámci našej práce a ktoré zároveň nie sú súčasťou softvéru DL-Learner.

Algoritmus DL-FOIL (inšpirovaný pôvodným FOIL algoritmom [125]) kombinuje špecializáciu a zovšeobecňovanie (teda dva operátory), pričom hľadá čiastkové riešienia, ktoré kombinuje prostredníctvom disjunkcie podobne ako pri algoritme PARCEL [126, 127]. Špecifikom algoritmu je fakt, že pracuje na báze *otvoreného sveta* (OWA) a taktiež platia prísnejšie obmedzenia, ktoré musia platiť pre negatívne príklady pri konceptovom učení v podobe: $K \models \neg C(e)$ kde $e \in E^-$ (v porovnaní s $K \not\models C(e)$ pri uzavretom svete). Tento model možno považovať za viac koherentný so štandardnými znalostnými bázami, ktoré bývajú považované za neúplné, t.j. ako negatívne príklady môžeme považovať iba také individuály, ktoré explicitne patria do negácie danej triedy. Samotný algoritmus je do istej miery podobný algoritmu PARCEL v tom, že hľadá čiastkové riešenia, ktoré pokrývajú pozitívne príklady a zároveň pokrývajú čo najmenej negatívnych príkladov. Implementácia však nie je paralelizovaná. Samotný algoritmus využíva spresňujúci operátor ρ (nejedná sa však o rovnaký operátor ako v DL-Learner algoritmoch), ktorý obsahuje 10 pravidiel, pričom spresňovanie je z časti založené aj na náhodnom výbere pravidiel, ktoré sa aplikujú. Rovnakí autori predstavili aj tri varianty vylepšeného DL-FOIL algoritmu, nazvaného DL-FOCL, ktorý využíva rôzne metaheuristiky na redukciu prehľadávacieho priestoru [128]. Cieľom týchto vylepšení bolo zlepšiť prehľadávanie priestoru konceptov, keďže samotné konceptové učenie môže byť považované za určitú formu horolezeckého prehľadávania, o ktorom je známe že trpí tzv. *krátkozrakosťou* a nemusí tak produkovať najoptimálnejšie riešenia [129]. Autori tak na riešenie spomínaného problému skúšali aplikovať rôzne stratégie v podobe opakovaného horolezeckého algoritmu, *lookahead* stratégie (v podobe aplikovania viacero pravidiel spresňujúceho operátora v jednom kroku) alebo využitím lokálnej pamäte v podobe *tabu* prehľadávania [130] tak, aby sa nemuseli znova skúmať suboptimálne riešenia.

V štandardných deskripčných logikách je problematické reprezentovať koncepty, ktoré prejavujú určitú formu neurčitosti (napr. miera príslušnosti k danej triede). Na tento účel slúžia tzv. fuzzy deskripčné logiky [131]. Vo fuzzy logike sú tvrdenia pravdivé vždy do určitej miery z intervalu $[0, 1]$, na rozdiel od klasickej výrokovej logiky, kde výrazy môžu mať pravdivostnú hodnotu iba 0 alebo 1. Na interpretáciu takýchto výrazov sa používajú rôzne fuzzy funkcie (napr. trojuholníkové alebo lichobežníkové), ktoré mapujú vstupnú hodnotu do oboru hodnôt z intervalu $[0, 1]$. Algoritmus pFOIL-DL predstavuje

riešenie konceptového učenia, ktoré pracuje práve nad fuzzy ontológiami [132]. Autori v práci navrhli algoritmus, ktorý sa dokáže učiť výrazy v menej expresívnej logike $\mathcal{EL}(D)$ (viď. kapitola 2.3). Samotné riešenie je založené na podobnom princípe ako iné FOIL algoritmy a taktiež pracuje nad uzavretým svetom (CWA). V rámci algoritmu sa taktiež konštruujú fuzzy dátové typy, na základe analýzy dátových vlastností v ontológii a následne sa využívajú počas spresňovania. Spresňujúci operátor ρ, ktorý sa využíva v algoritme bol navrhnutý podľa [133]. Rovnakí autori predstavili aj novšiu verziu algoritmu, nazvanú Fuzzy OWL-Boost [134]. Táto verzia využíva známu techniku z oblasti strojového učenia, tzv. AdaBoost [135], ktorá sa využíva pri kombinovaní slabších modelov do jedného silnejšieho. Samotný *boosting* je adaptovaný na prípad fuzzy ontológií. Poslednou verziou, ktorú predstavili rovnakí autori je dvojfázový algoritmus na učenie vo fuzzy ontológiách, tzv. PN-OWL [136]. Tento algoritmus je založený na princípoch PN-rule metódy navrhnutej v [137]. Fungovanie PN-OWL algoritmu je zložené z dvoch fáz. V prvej fáze (tzv. P fáza), je cieľom vygenerovať množinu fuzzy konceptov (pričom sa stále pohybujeme v $\mathcal{EL}(D)$ logike), ktoré pokrývajú čo najviac pozitívnych vzoriek, pričom pokrývajú čo najmenej negatívnych príkladov (t.j. cieľom je zvýšiť senzitivitu modelu). V druhej fáze (tzv. N fáza) sa naopak hľadá fuzzy koncept, ktorý pokrýva čo najviac negatívnych príkladov (t.j. cieľom je zvýšiť presnosť). Následne sa tieto naučené konceptové výrazy kombinujú pomocou rôznych agregačných funkcií tak, aby finálne výrazy dosahovali čo najvyššiu efektivitu v klasifikácii v podobe dostatočnej F1 miery.

EvoLearner je algoritmus konceptového učenia, ktorý je založený na princípoch evolučných algoritmov [138]. Jedná sa o najodlišnejší algoritmus od vyššie spomínaných. Zaujímavosťou algoritmu je jeho inicializačná metóda. Prvotná populácia je tvorená náhodne vytvorenými konceptmi, pričom ako základ sa berú pozitívne príklady, t.j. zoberie sa trieda, ktorej inštancia je daný pozitívny príklad a konceptový výraz sa vytvorí z n náhodných trojíc zo znalostnej bázy, ktoré sú napojené na triedu (hodnota n je použitá ako hyperparameter). Na populáciu sú následne aplikované štandardné evolučné operácie ako selekcia, kríženie alebo mutácia. Základom fitness funkcie je presnosť konceptu a informačný zisk. Podobne, ako vyššie spomínané fuzzy algoritmy, pracuje nad zatvoreným svetom. Autori v práci uvádzali, že efektivita a presnosť značne prevyšuje výsledky dosiahnuté pomocou štandardných algoritmov v rámci DL-Learner.

3.6 Iné vysvetliteľné metódy

Jednou z hlavných výhod algoritmov konceptového učenia je, že poskytujú výsledky, ktoré možno priamo interpretovať. V tejto podkapitole si v stručnosti predstavíme niektoré ďalšie metódy (mimo konceptového učenia), ktoré dokážu poskytovať intepretovatelné výsledky.

Napriek všeobecnému tvrdeniu, že algoritmy strojového učenia fungujú ako čierne skrinky a nie sú interpretovateľné, tak aj v rámci množiny algoritmov, ktoré patria medzi tradičné algoritmy strojového učenia, môžeme nájsť také, ktoré potvrdzujú opak. Medzi takéto algoritmy môžeme zaradiť rozhodovacie stromy, pri ktorých je výstupným modelom stromová štruktúra, ktorá sa vetví na základe vstupných hodnôt [139]. Podobným prípadom je aj algoritmus *Random forest*, ktorý kombinuje viacero rozhodovacích stromov do jedného finálneho modelu [140].

Medzi ďalšie metódy, ktoré sú v súčasnosti na vzostupe patrí aj tzv. *knowledge base embedding* [141]. Táto metóda taktiež využíva znalostné bázy, pričom sa snaží vkladať znalosti priamo do vektorového priestoru tak, aby bola zachovaná ich sémantika. Ďalšia metóda, využívajúca znalostné bázy bola predstavená v práci [142], kde sa autori snažili vysvetliť rozhodnutia neurónových sietí hľadaním vzorov a ich mapovaním priamo na koncepty v ontológii.

V posledných rokoch, s nárastom popularity neurónových sietí a ich aplikáciou v praxi, bola vo výskumnej oblasti identifikovaná dôležitosť vedieť interpretovať rozhodnutia rôznych klasifikátorov [143]. Tento trend priniesol množstvo metód, ktoré sa práve snažia priniesť určitú formu vysvetliteľnosti do tzv. *black-box* modelov, keďže práve tieto modely dosahujú vysokú úspešnosť klasifikácie. Dané metódy sa líšia v závislosti od toho, či ich cieľom je priniesť lokálnu (jedna konkrétna klasifikácia) alebo globálnu interpretáciu (náhľad do celého modelu), či fungujú iba na konkrétny model alebo všeobecne na všetky algoritmy a v poslednom rade, aké vstupné dáta dokážu interpretovať (obrázky, text, atď.). Z najviac citovaných metód môžeme spomenúť napr. algoritmy LIME [144] a SHAP [145]. Je však nutné poznamenať, že dané metódy vysvetlenia pre *black-box* modely určitým spôsobom aproximujú (správnosť tejto aproximácie sa označuje v literatúre ako *fidelity*) a nejedná sa o úplne exaktné vysvetlenie (ako v prípade konceptového učenia).

Kapitola 4

Návrh ontológie

V tejto časti práce si bližšie predstavíme ontológiu *PE Malware*, ktorú sme navrhli v rámci práce. Návrh a tvorba ontológie, ktorá by mohla slúžiť pre detekciu malvéru bol jeden z hlavných vedeckých cieľov práce. V kapitole 4.1 si povieme o motivácii, ktorá stála za tvorbou ontológie. Zhrnieme si hlavné problémy, ktoré sme sa snažili pomocou ontológie vyriešiť a ktoré dúfame vyrieši naša ontológia v budúcnosti. V kapitole 4.2 si v krátkosti predstavíme, akým spôsobom je možné aplikovať nami vytvorenú ontológiu a datasety v praxi. Kapitola 4.3 sa venuje predspracovaniu dát z datasetu EMBER, ktorý sme sa rozhodli použiť ako základ našich ontologických datasetov. V ďalšej kapitole 4.4 sa už venujeme technickým detailom navrhovanej ontológie *PE Malware*. Táto kapitola zahŕňa detailný popis tried a rolí, resp. ozrejmuje niektoré myšlienky, ktoré stáli za návrhom ontológie. Posledná kapitola 4.5 popisuje jednotlivé čiastkové ontologické datasety, ktoré boli vytvorené v rámci práce.

4.1 Motivácia

V súčasnosti existuje viacero rôznych datasetov, ktoré je možné využiť pre potreby strojového učenia. Z najznámejších a najpoužívanejších môžeme spomenúť EMBER a SoReL (viď. kapitoly 1.6 a 1.7). Oba datasety sú primárne určené pre klasické algoritmy strojového učenia. Dataset EMBER dokonca štandardne ponúka aj skripty, ktoré transformujú vzorky do vektorovej reprezentácie. Samotné dáta sú taktiež reprezentované vo forme JSON objektov, ktoré môžu byť priamo transformované do zmysluplnej znalostnej bázy. Existencia datasetu vo forme ontológie by mohla nájsť využitie v rôznych interpretovateľných prístupoch. Sem môžeme zaradiť okrem samotného konceptového učenia (kapitola 2.5) aj ďalšie metódy uvedené v kapitole 3.6.

Jedným z ďalších problémov, s ktorými sa potýkajú datasety ako EMBER a SoReL, je ich rozsiahlosť. Z dôvodu, že obsahujú veľké množstvo vzoriek, existuje mnoho vedeckých prác, ktoré používajú iba konkrétnu menšiu podmnožinu datasetu (bez špecifikácie, ktoré vzorky boli v danej práci použité). Z prác, ktoré redukovali pôvodný dataset EMBER (najmä kvôli výpočtovej náročnosti) môžeme spomenúť [146, 147, 148]. Zmenšovanie datasetov samozrejme vedie k zníženej možnosti reprodukovať výsledky, ktoré autori uvádzajú v prácach.

V oblasti symbolického strojového učenia, inak označovaného aj ako Structured Machine Learning (SML), kam patria aj rôzne induktívne prístupy či konceptové učenie, existuje množstvo rôznych datasetov, ktoré sa používajú pri testovaní nových algoritmov [149]. V rámci známych datasetov existujú aplikačné domény ako finančníctvo a geografické dáta [127], medicína [149], klasifikácia scény na obrázkoch [150] alebo rôzne časti z *DBpedia*[1] [128]. Napriek väčšiemu množstvu datasetov je väčšina z nich relatívne menšieho rozsahu. Medzi najväčšie datasety patria finančné dáta (17k príkladov) a geografické dáta (2k príkladov).

Celkovo môžeme zhrnúť, že naša ontológia spĺňa nasledujúce ciele:

Jednotná sémantická reprezentácia: našim cieľom bolo navrhnúť všeobecnú ontológiu, ktorá by dokázala popísať PE škodlivé vzorky. Do takejto ontológie je následne možné namapovať ľubovoľný zdroj dát, ktorý ponúka škodlivé vzorky pre operačný systém *Windows*. V čase písania práce ontológia zameraná na statické vlastnosti neexistovala. Samotnú terminológiu v ontológii sme taktiež prispôsobili aj štandardu Malware Attribute Enumeration and Characterization (MAEC) [151], ktorý je v praxi bežne zaužívaný pri opise rôznych akcií, ktoré môže

[1] https://www.dbpedia.org/

malvér vykonávať. Využitie štandardu taktiež prispieva k lepšej interpretovateľnosti.

Interpretovateľnosť: našim cieľom bolo navrhnúť ontológiu s takým slovníkom, ktorý by bol zmysluplný pre používateľov. Konceptové výrazy alebo rôzne pravidlá generované z našej ontológie, by mohli byť následne ľahko pochopiteľné a interpretovateľné. Je nutné poznamenať, že aj keď naša ontológia je založená na datasete EMBER, nejedná sa o jeho priame mapovanie. Pri návrhu sme dbali nato, aby vlastnosti, ktoré sme použili aj v ontológii, boli zmysluplné pre popis malvéru a zároveň boli zmysluplné aj z expertného hľadiska. Z toho dôvodu sme vynechali vlastnosti ako napr. veľkosť súboru, či čas kompilácie, ktoré z expertného hľadiska nezohrávajú rolu pri rozlišovaní malvéru a zároveň môžu viesť k útokom (viď. kapitola 1.8). Taktiež je nutné poznamenať, že niektoré koncepty v našej ontológii, označované ako tzv. *odvodené* vlastnosti, sú zložené z viacerých nízkoúrovňových vlastností z EMBER datasetu tak, aby spolu vytvárali zmysluplnejší koncept. Odvodené vlastnosti, ktoré pracujú napr. s prahovou číselnou hodnotou, tak ponúkajú lepšiu interpretovateľnosť a zároveň uľahčujú prácu algoritmom, ktoré majú problémy s numerickými hodnotami (viac v kapitole 4.4.2).

Reprodukovateľnosť experimentov: našim cieľom bolo taktiež zlepšiť reprodukciu výsledkov experimentov, ktoré môžu byť vykonávané na ontológii. Z toho dôvodu sme sa zamerali aj na vytvorenie viacerých datasetov o rôznych veľkostiach, čím odpadá potreba redukovať pôvodný dataset a je možné si vybrať dataset zodpovedajúci náročnosti experimentov. Takýmto spôsobom môže náš dataset slúžiť ako štandardný *benchmark* a pomôcť tak napredovaniu strojového učenia v oblasti detekcie malvéru. Viac v kapitole 4.5.

Robustnosť datasetov: našim cieľom bolo poskytnúť robustnejšie datasety pre rôzne SML prístupy. Niektoré naše datasety (veľkosti 100k a 800k) sú značne rozsiahlejšie ako vyššie spomínané datasety, ktoré sú najčastejšie používané v rámci SML algoritmov. Napriek tomu, že v súčasnej dobe prístupy, ako napr. konceptové učenie, nie sú dostatočne efektívne, aby zvládali obrovské množstvá dát, tento trend sa môže v budúcnosti zmeniť. Z toho dôvodu sme považovali za potrebné vytvoriť aj značne robustnejšie datasety, ktoré obsahujú veľké množstvo dát z aplikačnej domény, ktorá rieši aktuálne vedecké problémy (detekcia malvéru). Takéto datasety následne môžu prispieť k vývoju efektívnejších algoritmov.

4.2 Aplikácia datasetu

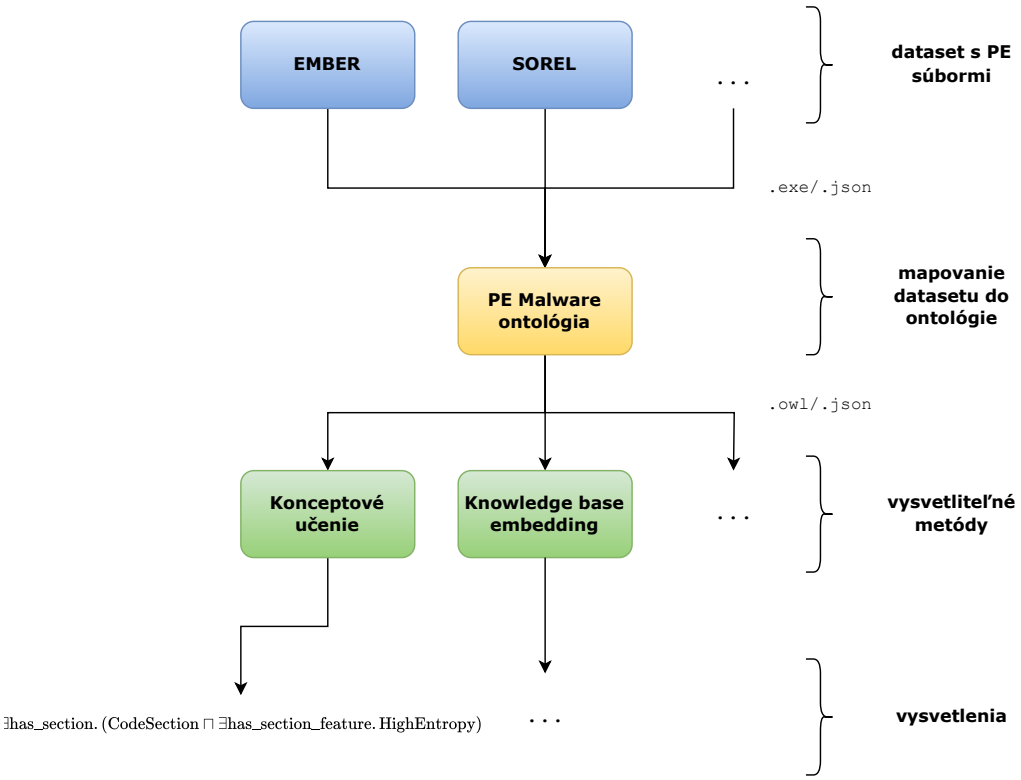
Medzi dôležité aspekty nami vytvorenej ontológie patrí fakt, že nie je limitovaná iba na použitie s datasetom EMBER, resp. pre konceptové učenie. Na obrázku 4.1 môžeme vidieť schému aplikácie našej ontológie. Schéma sa skladá z nasledujúcich častí:

Dataset. Základ celej aplikácie tvorí dataset s PE súbormi. Zatiaľ čo v našej práci sme si vybrali dataset EMBER, ontológiu je možné použiť s ľubovoľným datasetom, ktorý obsahuje priamo spustiteľné PE súbory alebo vyextrahované vlastnosti (ako v prípade datasetu EMBER).

Ontológia. Druhým krokom je transformácia datasetu do nami navrhnutej ontológie. Nami implementovaný skript dokáže namapovať dataset reprezentovaný v špecifickom JSON formáte do ontológie (viď. kapitola 1.6). Spomínaný formát používajú datasety EMBER a SoReL. Autori datasetu EMBER však spoločne so samotnými dátami zverejnili aj skript, ktorý dokáže pretransformovať ľubovoľný PE súbor do spomínaného JSON formátu. Našu ontológiu je teda taktiež možné aplikovať na ľubovoľný dataset obsahujúci binárne PE súbory (viď. tabuľka 1.1).

Učenie. Výsledkom predchádzajúceho kroku je ontológia, spolu so všetkými znalosťami o konkrétnom datasete (spolu so špecifikáciou pozitívnych a negatívnych príkladov v JSON súbore). Takýto ontologický dataset je následne možné použiť v rámci učiaceho procesu. V našej práci sme sa zamerali na konceptové učenie (konkrétne na algoritmy dostupné v softvéri DL-Learner), avšak vytvorené datasety možno použiť aj s inými metódami, ktoré dokážu pracovať so znalostnými bázami. Sem môžeme zaradiť okrem ďalších algoritmov konceptového učenia (viď. kapitola 3.5) aj iné metódy, ako napr. *knowledge base embedding* (viď. kapitola 3.6). Samotnú ontológiu je taktiež možné použiť aj pri tradičných algoritmoch strojového učenia, kde je však potrebné vstupné dáta určitým spôsobom vektorizovať (viď. rozhodovacie stromy v [152]).

Vysvetlenia. Výstupom celého procesu sú konceptové výrazy/pravidlá (v závislosti od použitej metódy), ktoré okrem samotnej identifikácie škodlivých vzoriek, ponúkajú aj vysvetlenia, ktorým možno priamo porozumieť.



Obrázok 4.1: Schéma aplikácie ontologického datasetu.

4.3 Predspracovanie dát

V tejto podkapitole si popíšeme niektoré základné informácie ohľadom predspracovania datasetu EMBER spolu s jeho štandardizáciou.

4.3.1 Štandardizácia datasetu

Samotné PE vzorky v datasete môžu potenciálne obsahovať veľké množstvo rôznych importovaných funkcií, ktoré sú načítané z DLL súborov. Informácie o tom aké funkcie daná vzorka importuje sú dostupné priamo v rámci metadát (kapitola 1.5) a taktiež ich máme dostupné v rámci EMBER datasetu (kapitola 1.6). Importované funkcie môžu zohrávať dôležitú úlohu pri rozlišovaní škodlivého kódu, keďže naznačujú potenciálne správanie danej vzorky. Keďže existuje veľké množstvo rôznych API volaní, potrebovali sme určitým spôsobom zredukovať funkcie, ktoré budú zahrnuté v rámci ontológie a zároveň vybrať také, ktoré sú najviac relevantné z hľadiska správania škodlivého kódu. Pre tento účel sme využili štandard Malware Attribute Enumeration and Characterization (MAEC) [151]. MAEC môžeme charakterizovať ako komunitou udržiavaný štrukturovaný slovník, ktorý sa bežne používa na popis rôznych informácií o škodlivom kóde, pričom obsahuje statické aj dynamické dáta. Môžeme sem zaradiť rôzne formy správania, interakcie medzi procesmi malvéru a pod. Pre naše účely sme sa rozhodli využiť časť slovníka, nazývanú *malware actions* [153]. Využitie MAEC slovníka so sebou prináša dve výhody. Prvou je samotná štandardizácia a využitie bežne používaných pojmov v praxi. Druhou výhodou je redukcia priestoru importovaných funkcií (podobne ako pri tradičnom strojovom učení), keďže v praxi ich existuje veľké množstvo a z praktického hľadiska je vhodné zaviesť do ontológie iba tie, ktoré sú relevantné.

Jeden z problémov, na ktorý sme narazili bol, že akcie definované v rámci MAEC slovníka, definujú správanie škodlivého kódu vo všeobecnosti, t.j. nezávisle od metódy akou bolo dané správanie získané (staticky alebo dynamicky). Keďže dataset EMBER používa statické dáta, niektoré akcie sme nedokázali priamo previesť do ontológie. Ako príklad si môžeme uviesť dve API volania: `CreateFile` a `HttpSendRequest`. Prvé API volanie `CreateFile` (ktoré slúži na vytvorenie nového súboru) vieme priamo namapovať na akciu `create-file` z MAEC slovníka. Druhé API volanie, `HttpSendRequest` (ktoré slúži na odosielanie HTTP požiadaviek), môže byť teoreticky mapované na ľubovolnú `send-http-`*`method`*`-request` akciu, kde *method* je ľubovoľná HTTP požiadavka (GET, POST, atď.). O tom, aká požiadavka sa odosiela, rozhodujú vstupné parametre daného API volania, ktoré sa vyskytujú častejšie v rámci dynamicky extrahovaných dát a ktoré nemáme k dispozícii v rámci EMBER datasetu.

Z dôvodu aby sme vedeli namapovať viac akcií na API volania, rozhodli sme sa rozšíriť MAEC slovník o ďalšie akcie (viď. tabuľka 4.1). Špecifické akcie `send-http-`*`method`*`-request` sme sa rozhodli nahradiť jednou všeobecnou akciou `send-http-request`. Taktiež sme pridali rôzne akcie pracujúce s kryptografickými API volaniami, keďže sa nenachádzali v MAEC slovníku a môže sa jednať o dôležité akcie z hľadiska detekcie malvéru.

Ďalšie akcie v rámci našej ontológie budú detailnejšie predstavené v kapitole 4.4.7.

Akcia	**Popis**
`send-http-request`	Akcia odosielania HTTP požiadavky na server
`encrypt`	Akcia definujúca šifrovanie
`decrypt`	Akcia definujúca dešifrovanie
`generate-key`	Akcia generovania kryptografického kľúča

Tabuľka 4.1: Rozšírenie MAEC slovníka.

4.3.2 Vlastnosti súborov a sekcií

V rámci predspracovania dát z datasetu EMBER, sme sa rozhodli vyberať najdôležitejšie vlastnosti, ktoré sú zmysluplné z expertného hľadiska pre detekciu malvéru [154, 155]. Tieto vlastnosti sme následne namapovali do ontológie a budeme ich nazývať ako vlastnosti PE súboru alebo vlastnosti sekcií. Všeobecne sa v našej ontológii nachádzajú tri typy vlastností (súborov alebo sekcií):

1. priama reprezentácia binárnych vlastností v pôvodnom datasete (buď 0 alebo 1)

2. vlastnosti, ktoré sme získali predspracovaním dát a neboli obsiahnuté v pôvodnom datasete

3. vlastnosti, ktoré sme získali predspracovaním dát a boli priamo reprezentované v pôvodnom datasete

Vlastnosti prvého typu sme získali priamo z datasetu (viď. 1.1). Môžeme sem zaradiť vlastnosti ako `general.has_tls` alebo `general.has_signature` (binárne vlastnosti). Viac detailov o vlastnostiach súboru bude uvedených v kapitole 4.4.2.

Vlastnosti, ktoré spĺňajú charakteristiku druhého typu vlastností máme v ontológii dve. Prvá reprezentuje prítomnosť neprázdneho dátového priečinku

Common Language Runtime (CLR), ktorý sa v datasete EMBER nachádza v časti `datadirectories`. Takáto vlastnosť môže indikovať .NET binárny súbor. Druhá vlastnosť naopak označuje vstupný bod programu, ktorý sa nenachádza v sekcii s právami na spustenie. Takúto vlastnosť vieme vyčítať z datasetu EMBER kontrolou časti `section.entry`, kde sa nachádza názov sekcie so vstupným bodom a následne preskúmaním oprávnení danej sekcie v časti `section.sections.props`.

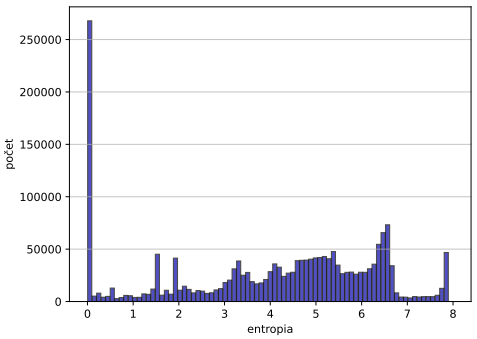
Tretí typ vlastností budeme nazývať ako *odvodené* vlastnosti. Sem môžeme zaradiť vlastnosti, ktoré by sme síce vedeli štandardne reprezentovať v rámci OWL 2 jazyka, avšak jedná sa o dôležité indikátory škodlivého kódu, dôsledkom čoho sme sa ich rozhodli explicitne pomenovať. Explicitné pomenovanie takýchto vlastností môže byť užitočné najmä pre algoritmy strojového učenia, ktoré nedokážu pracovať s expresívnejšími konštruktormi (dátové vlastnosti, kardinalita vzťahu, nominály a pod.), resp. ich povolenie môže priniesť zvýšenie výpočtovej zložitosti. V opačnom prípade, ak je algoritmus schopný s danými konštuktormi efektívne pracovať, je možné dané odvodené vlastnosti odstrániť (resp. ich ignorovať v rámci nastavenia algoritmu) a umožniť tak efektívnejšie prehľadávanie priestoru.

V rámci ontológie máme definované dve *odvodené* vlastnosti, ktoré zohrávajú dôležitú úlohu pri detekcii malvéru. Tieto vlastnosti môžeme označiť ako *nízky počet importovaných funkcií* a *vysoká entropia sekcie*. Spomínané vlastnosti sú dôležité z toho hľadiska, že môžu indikovať zbalenú vzorku, napr. pomocou nástroja UPX[2]. Pri návrhu odvodených numerických hodnôt zohráva hodnotu hraničná hodnota, ktorá definuje či konkrétna vzorka bude obsahovať danú vlastnosť. Pri návrhu hraničných hodnôt sme sa inšpirovali nástrojom *pestudio* [156]. Nástroj je bežne používaný v rámci bezpečnostných tímov pri počiatočnom skúmaní potenciálne škodlivého programu. Samotný nástroj okrem načítania metadát PE súboru, obsahuje aj rôzne zaujímavé indikátory pre škodlivé programy. Medzi takéto indikátory patrí aj podozrivo nízky počet importovaných funkcií, pričom *pestudio* má ako hraničnú hodnotu nastavené číslo 10. Taktiež obsahuje hraničnú hodnotu pre sekciu s vysokou entropiou, pričom hodnota je nastavená na 7.0.

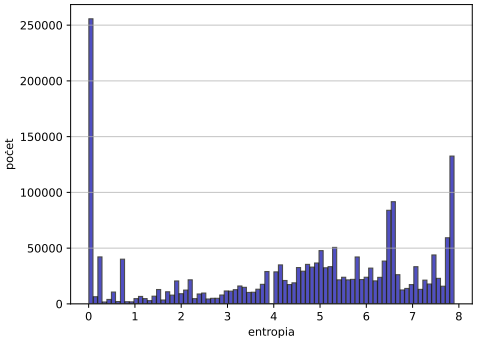
Spomínané hraničné hodnoty sme sa pokúsili overiť aj v rámci celého datasetu EMBER. Konkrétne nás zaujímali histogramy reprezentujúce počty importovaných funkcií pre všetky vzorky hodnoty a entropie pre všetky sekcie v datasete. Výsledky možno vidieť na obrázku 4.2. V rámci histogramov pre entropie sekcií môžeme vidieť, že hraničná hodnota pre entropiu bola potvrdená. Regulérne obsahujú značne menej sekcií s hodnotou entropie väčšou ako 7.0 v porovnaní so škodlivými vzorkami. Čo sa týka histogramov

[2]https://upx.github.io/

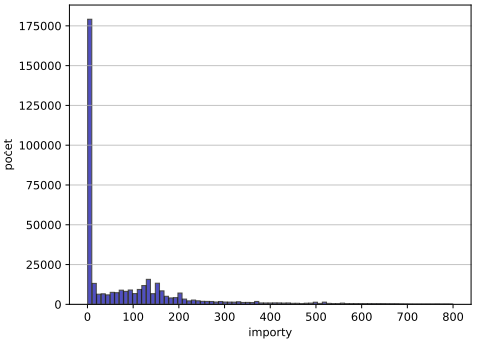
s počtom importovaných funkcií, z výsledkov sme nevideli žiadnu hraničnú hodnotu, ktorá by oddeľovala regulérne a škodlivé vzorky. V oboch prípadoch existovalo veľké množstvo vzoriek, ktoré importovalo 10 a menej funkcií. Táto vlastnosť teda sama o sebe nedokáže správne rozdeliť vzorky, môže sa však ukázať ako užitočná v kombinácii s inými vlastnosťami. Z toho dôvodu sme sa rozhodli ponechať hraničnú hodnotu na základe hodnoty z nástroja *pestudio*.



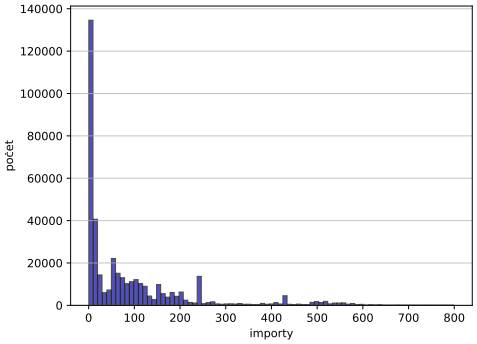
(a) Entropia sekcií pre regulárne vzorky



(b) Entropia sekcií pre škodlivé vzorky



(c) Počet importov pre regulárne vzorky



(d) Počet importov pre škodlivé vzorky

Obrázok 4.2: Zobrazenie histogramu pre entropiu sekcií a importovaných funkcií z datasetu EMBER.

Ďalšou netriviálne *odvodenou* vlastnosťou je sekcia s *neštandardným menom*. Túto vlastnosť sme odvodili takým spôsobom, že sme získali zoznam názvov sekcií, ktoré štandardne generujú kompilátory a do ontológie sa daná vlastnosť mapuje, ak sekcia danej vzorky neobsahuje meno zo zoznamu. Danú vlastnosť by tiež bolo možné reprezentovať v rámci OWL 2 jazyka, pomocou dátových vlastností, avšak mohlo by to byť značne náročné pre učiace algoritmy.

Viac *odvodených* vlastností bude taktiež spomenutých v rámci kapitoly 4.4.2.

4.4 *PE Malware* ontológia

V tejto podkapitole si hlbšie predstavíme našu ontológiu, ktorú sme nazvali ako *PE Malware*. Ako sme spomínali v predchádzajúcich kapitolách, samotná ontológia nie je priamym mapovaním datasetu EMBER. Jedným z hlavných cieľov ontológie bolo poskytnúť interpretovateľnosť. Z toho dôvodu sme nepoužili vlastnosti, ktoré neposkytujú dôležité informácie z expertného hľadiska alebo nie sú interpretovateľné. Sem môžeme zaradiť rôzne časové značky (čas kompilácie), verzia linkera, veľkosť súboru a jeho jednotlivých sekcií alebo histogramy bajtov či reťazcov. Z teoretického hľadiska, v takýchto (primárne numerických) vlastnostiach môžu existovať rôzne vzory, ktoré sa dokážu tradičné algoritmy strojového učenia naučiť a dokázať tak rozlišovať škodlivý kód (napr. škodlivý PE súbor je taký, ktorého veľkosť je menšia ako nejaká konkrétna hodnota). Takéto naučené vzory však nie sú užitočné z expertného hľadiska a ako sme spomínali v predchádzajúcich kapitolách, môžu viesť k triviálnym útokom na natrénované modely (napr. umelým zväčšením súboru).

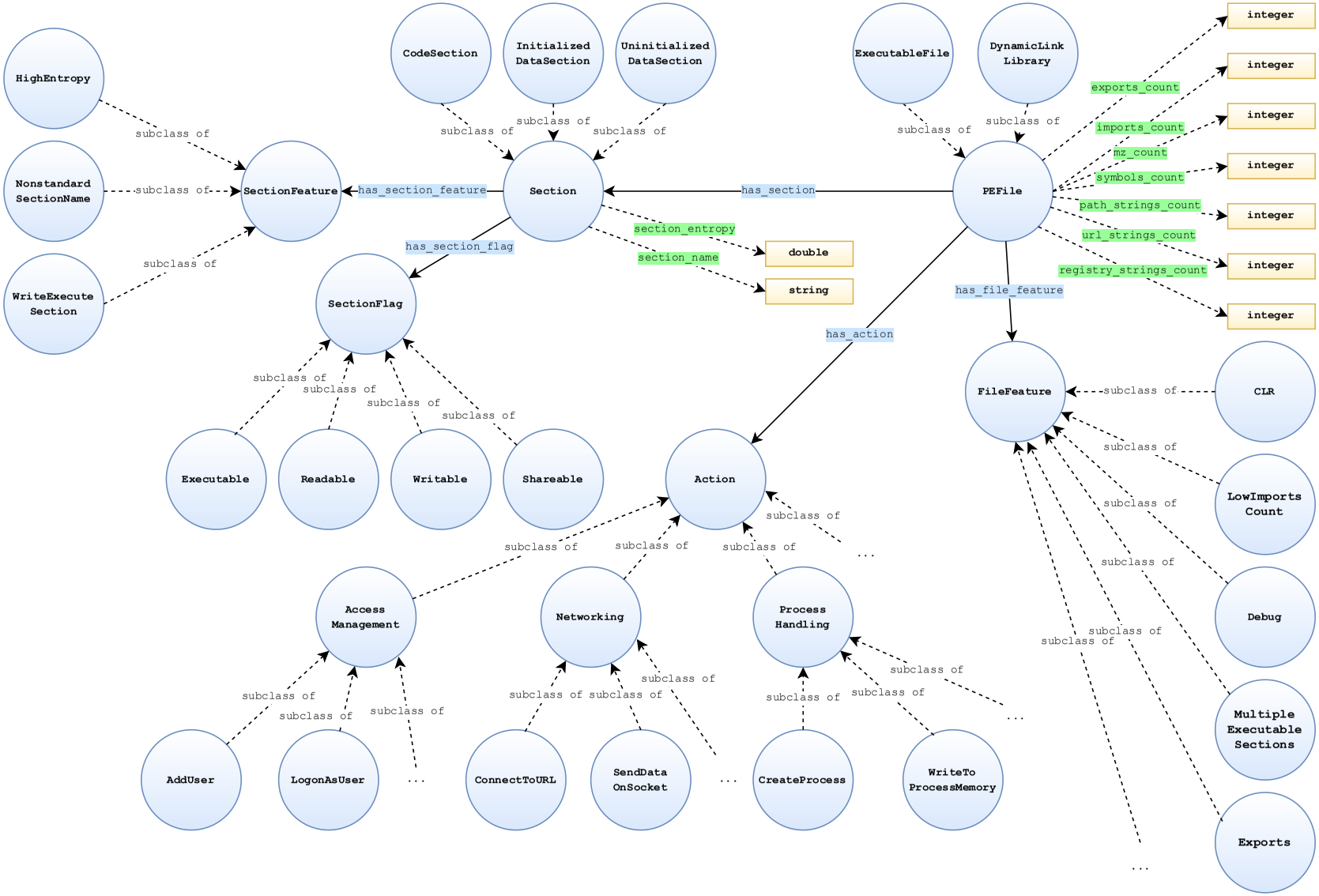
Samotnú ontológiu (resp. znázornenie niektorých základných tried) môžeme vidieť na obrázku 4.3. Celková ontológia pozostáva zo 195 tried, 6 rolí a 9 dátových vlastností. Čo sa týka ponúkanej expresivity a výpočtovej náročnosti, ontológiu môžeme zaradiť do profilu OWL 2 QL (viď. kapitola 2.4). Tento profil ponúka efektívne overovanie inštancií, čím je vhodným kandidátom pre prípady, keď ontológia obsahuje veľké množstvo individuálov (ako v našom prípade).

4.4.1 PE súbory

Každá vzorka v datasete EMBER je reprezentovaná centrálnou triedou `PEFile`. Každá inštancia je ďalej klasifikovaná ako podtrieda `ExecutableFile`, ak sa jedná o štandardný spustiteľný súbor alebo ako `DynamicLinkLibrary`, ak sa jedná o DLL knižnicu. Informáciu o type súboru možno nájsť v COFF hlavičke v EMBER datasete.

Trieda `PEFile` je doménou, resp. ponúka 7 dátových vlastností, ktoré zahŕňajú informácie o počte importovaných/exportovaných funkcií, počte symbolov, počte `MZ` hlavičiek alebo počte reťazcov, ktoré obsahujú cesty k súborom, registrom alebo URL (viď. tabuľka 4.2).

Štrukturálne PE súbory obsahujú sekcie, môžu vykazovať rôzne vlastnosti, ktoré sú relevantné z hľadiska detekcie malvéru a taktiež môžu vykonávať rôzne akcie (na základe importovaných funkcií). Tieto objekty sú v ontológií reprezentované inštanciami tried `Section`, `FileFeature` a `Action`. Inštancie tých tried sú prepojené s inštanciou triedy `PEFile` vlastnosťami `has_section`, `has_file_feature` a `has_section`.



Obrázok 4.3: Znázornenie základných tried v ontológii.

Dátová vlastnosť	Rozsah	Vlastnosť	Rozsah
exports_count	xsd:integer	has_action	Action
imports_count	xsd:integer	has_file_feature	FileFeature
mz_count	xsd:integer	has_section	Section
path_strings_count	xsd:integer		
symbols_count	xsd:integer		
registry_strings_count	xsd:integer		
url_strings_count	xsd:integer		

Tabuľka 4.2: Vlastnosti triedy PEFile.

4.4.2 Vlastnosti súborov

Trieda FileFeature obsahuje spoločne 15 podtried, ktoré reprezentujú rôzne vlastnosti PE súborov, ktoré sú relevantné z hľadiska detekcie malvéru. Každá podtrieda FileFeature obsahuje jednu tzv. *prototypickú inštanciu*. Všetky inštancie triedy PEFile, ktoré obsahujú konkrétnu vlastnosť sú spojené prostredníctvom role has_file_feature s protypickou inštanciou danej triedy. Ako príklad si môžeme uviesť triedu MultipleExecutableSections a jej prototypickú inštanciu multiple_executable_sections (na tento indvididuál sú napojené všetky inštancie PEFile s danou vlastnosťou). Takýmto spôsobom je potom možné generovať nasledujúce výrazy (v Manchester syntaxy): ExecutableFile has_file_feature some {low_imports_count} - ktorý sa dá priamo čítať ako *"spustiteľný súbor, ktorý má vlastnosť nízky počet importovaných funkcií"*. Ako sme spomínali v kapitole 4.4.2, definované vlastnosti môžeme deliť do troch kategórií: priamo reprezentované vlastnosti, predspracované vlastnosti a odvodené vlastnosti.

V našej ontológii máme nasledujúce priamo reprezentované vlastnosti (v zátvorke je uvedená korešpondujúca vlastnosť z EMBER datasetu):

- Debug - definuje, či vzorka obsahuje sekciu s ladiacimi informáciami (general.has_debug).
- Relocations - definuje, či vzorka obsahuje sekciu s relokáciami (general.has_relocations).
- Resources - označuje, či vzorka obsahuje zdroje v podobe obrázkov, písma a pod. (general.has_resources).
- Signature - špecifikuje, či je daná vzorka digitálne podpísaná (general.has_signature).
- TLS - označuje, či daná vzorka obsahuje TLS sekciu (t.j. potenciálne ukrytý vstupný bod programu; general.has_tls).

Zoznam predspracovaných vlastností súboru:

- `CLR` - označuje prítomnosť CLR dátovej sekcie (ktorá je používaná pri .NET binárnych súboroch).
- `NonexecutableEntryPoint` - definuje, že daná vzorka má vstupný bod programu v sekcii, ktorá nie je spustiteľná.

Zoznam odvodených vlastností súboru (v zátvorke je ekvivalentný OWL 2 výraz v Manchester syntaxy):

- `Exports` - označuje prítomnosť exportovaných funkcií vo vzorke (najčastejšie sa vyskytujú v DLL súboroch; `exports_count some xsd:integer [> 0]`).
- `MultipleExecutableSections` - definuje, že vzorka obsahuje viac sekcií, ktoré sú spustiteľné (`has_section min 2 (has_section_flag some Executable)`).
- `LowImportsCount` - počet importovaných funkcií je menší ako nami definovaná prahová hodnota (`imports_count some xsd:integer [<` *`imports_threshold`*`]`).
- `NonstandardMZ` - vzorka obsahuje viac ako jednu `MZ` hlavičku (t.j. teoreticky obsahuje vložený ďalší PE súbor; `mz_count some xsd:integer [> 1]`).
- `PathStrings` - vzorka obsahuje reťazce, ktoré definujú cesty k súborom alebo priečinkom (`path_strings_count some xsd:integer [> 0]`).
- `RegistryStrings` - vzorka obsahuje reťazce, ktoré definujú registre (`registry_strings_count some xsd:integer [> 0]`).
- `Symbols` - vzorka obsahuje ladiace symboly (`symbols_count some xsd:integer [> 0]`).
- `URLStrings` - vzorka obsahuje reťazce, ktoré definujú URL (`url_strings_count some xsd:integer [> 0]`).

4.4.3 Sekcie

Inštancie triedy `Section` reprezentujú priamo sekcie, ktoré sa nachádzajú v PE súbore. Každá inštancia triedy `PEFile` je prepojená so všetkými svojimi sekciami prostredníctvom role `has_section`. Sekcie sú ďalej klasifikované podtriedami, podľa dát, ktoré obsahujú. Podtriedy `Section` sú: `CodeSection` (sekcia obsahujúca spustiteľný kód), `InitializedDataSection` (sekcia s inicializovanými dátami) a `UninitializedDataSection` (sekcia s neinicializovanými dátami). Inštancie triedy `Section` obsahujú aj dátové vlastnosti definujúce názov samotnej sekcie a výšku entropie (viď. tabuľka 4.3). Okrem toho sú prepojené aj na ďalšie triedy definujúce vlastnosti danej sekcie (`SectionFeature`) a príznaky s oprávneniami (`SectionFlag`).

Dátová vlastnosť	**Rozsah**	**Vlastnosť**	**Rozsah**
`section_entropy`	`xsd:double`	`has_section_feature`	`SectionFeature`
`section_name`	`xsd:string`	`has_section_flag`	`SectionFlag`

Tabuľka 4.3: Vlastnosti triedy `Section`.

4.4.4 Príznaky sekcií

Trieda `SectionFlag` sa používa na klasifikovanie jednotlivých oprávnení, ktoré obsahujú sekcie v PE súbore. V rámci EMBER sú tieto oprávnenia zahrnuté v časti `prop` pri jednotlivých sekciách. Z hľadiska detekcie malvéru, ako najzaujímavejšie príznaky môžeme označiť tie, ktoré definujú, akým spôsobom môže proces spúšťajúci PE súbor pristupovať k sekciám (čítanie, zápis, spustenie) a či môže povoliť ich zdieľanie. Do našej ontológie sme zahrnuli 4 triedy reprezentujúce príznaky: `Executable`, `Readable`, `Writeable` a `Shareable`, pričom všetky sú podtriedou `SectionFlag`. Podobne ako v prípade vlastností súborov, sú jednotlivé triedy reprezentujúce príznaky, implementované ako prototypické inštancie. Inštancie triedy `Section` sú prepojené s inštanciami `SectionFlag` prostredníctvom vlastnosti `has_section_flag`.

4.4.5 Vlastnosti sekcií

Vlastnosti sekcií sú podobné ako vlastnosti súborov, pričom sú založené na relevantných vlastnostiach z pohľadu detekcie malvéru. Vlastnosti sekcií sú reprezentované triedou `SectionFeature`, pričom všetky tieto vlastnosti môžeme považovať za odvodené. Inštancie triedy `Section` sú prepojené s prototypickými inštanciami triedy `SectionFeature` prostredníctvom role `has_section_feature`. Do našej ontológie sme zahrnuli nasledujúce tri vlastnosti, podtriedy `SectionFeature`:

- `HighEntropy` - reprezentácia vlastnosti, že hodnota entropie pre konkrétnu sekciu je vyššia ako nami definovaná prahová hodnota (`section_entropy some xsd:double [> entropy_threshold]`).
- `NonstandardSectionName` - názov konkrétnej sekcie sa nenachádza v zozname názvov, ktoré štandardne generujú kompilátory (`section_name some not {".text", ".data", ".rsrc", ...}`).
- `WriteExecuteSection` - sekcia má práva na spustenie a zároveň má aj práva na zapisovanie (`(has_section_flag some Writeable) and (has_section_flag some Executable)`).

4.4.6 Anotácie

Ako sme spomínali v kapitole 4.4.2, niektoré vlastnosti PE súborov a sekcií môžu byť v určitých aplikáciách redundantné. Vlastnosti ako napr. `MultipleExecutableSections` sa teoreticky dokáže učiaci algoritmus naučiť aj samostatne. V takých prípadoch sú potom niektoré odvodené vlastnosti nežiadúce a dokážu tak mať negatívny vplyv pri učení. Z toho dôvodu sme takéto vlastnosti zdokumentovali priamo v ontológii pomocou anotačnej vlastnosti `derived_as`, čím sa dajú automaticky identifikovať. Všetky takéto odvodené vlastnosti sú anotované v ontológii spolu s ich definíciou v OWL 2 (v Manchester syntaxy).

4.4.7 Akcie

Inštancie triedy `Action` reprezentujú akcie, ktoré môže teoreticky proces vykonať ak spustí daný PE súbor. Celkovo sa v našej ontológii nachádza 139 listových podtried triedy `Action` (listových v zmysle, že daná trieda nemá žiadne ďalšie podtriedy), ktoré sú navrhnuté podľa MAEC štandardu (kapitola 4.3.1). Aby sme zlepšili celkové zovšeobecňovanie počas učiaceho procesu, do ontológie sme taktiež pridali ďalších 17 tried, ktoré reprezentujú jednotlivé kategórie akcií. Každá kategória je reprezentovaná triedou, napr. `ProcessHandling`, ktorá je podtriedou `Action`. Trieda ďalej obsahuje ďalšie podtriedy, ktoré priamo mapujú MAEC akcie. Kategória `ProcessHandling` obsahuje napr. ďalšie podtriedy `CreateProcess`, `WriteToProcessMemory` atď. Zoznam samotných kategórií je nasledovný:

- `AccessManagement` - akcie, ktoré súvisia s riadením používateľov v systéme (pridávanie používateľov, vymazanie používateľa, enumerácia existujúcich používateľov atď.
- `AntiDebugging` - akcie, ktoré sa využívajú na detekciu ladenia programu (napr. API volanie `IsDebuggerPresent`).

- `Cryptography` - akcie súvisiace so šifrovaním/dešifrovaním a generovaním kryptografických kľúčov.
- `DirectoryHandling` - manipulácia s priečinkami (vytváranie, vymazanie, atď.).
- `DiskManagement` - pripájanie/odpájanie diskov, enemurácia existujúcich diskov, atď.
- `FileHandling` - manipulácia so súbormi (vytváranie, vymazanie, atď.).
- `InterProcessCommunication` - akcie súvisiace s medziprocesovou komunikáciou (napr. pomenované rúry).
- `LibraryHandling` - načítavanie knižníc do bežiacich procesov (`LoadLibrary`) alebo enumerácia už načítaných knižníc.
- `Networking` - rôzne akcie súvisiace so sieťou ako napr. pripájanie sa na soket, posielanie DNS požiadaviek, posielanie HTTP požiadaviek atď.
- `ProcessHandling` - akcie, ktoré sa využívajú na manipuláciu s procesmi ako napr. vytváranie nových procesov, modifikácia pamäti bežiaceho procesu atď.
- `RegistryHandling` - enumerácia kľúčov v registroch, získavanie ich hodnôt, tvorba nových kľúčov atď.
- `ResourceSharing` - manipulácia so zdrojmi zdieľanými v rámci siete.
- `ServiceHandling` - manipulácia so servismi, ktoré bežia v operačnom systéme.
- `SynchronizationPrimitivesHandling` - správa rôznych synchronizačných primitív ako sú mutexy a semafory.
- `SystemManipulation` - rôzne API volania, ktoré sa používajú na získavanie informácií o systéme ako napr. meno používateľa, aktuálny čas, cesta k dátam operačného systému atď.
- `ThreadHandling` - práca s vláknami, napr. tvorba nového vlákna v procese, enumerácia bežiacich vláken atď.
- `WindowHandling` - manipulácia s *Windows* oknami - tvorba nového okna, dialógového boxu, zatvorenie okna a pod.

Všetky listové podtriedy triedy `Action` sú taktiež implementované ako prototypické inštancie. Každá inštancia triedy `PEFile` je napojená na konkrétnu akciu prostredníctvom vlastnosti `has_action`. Samotné prepojenie je vytvorené mapovaním importovaných funkcií z EMBER datasetu (sekcia `imports`) na konkrétnu MAEC akciu (v prípade, ak je to možné). Ako príklad si môžeme uviesť tvorbu ontologickej reprezentácie API volania, ktoré vytvára nový proces. V takom prípade je prototypická inštancia `create-process` (triedy `CreateProcess`) napojená na inštanciu triedy `PEFile` prostredníctvom role `has_action`. Opäť je nutné poznamenať, že importované API funkcie automaticky neznamenajú, že konkrétna vzorka danú akciu aj reálne vykoná, ak ju spustíme. V našom prípade sa spoliehame iba na určitý odhad správania, keďže sme limitovaní statickými dátami.

4.5 Čiastkové datasety

Ako sme spomínali v úvode kapitoly, jedným z dôležitých cieľov našej ontológie bolo zabezpečiť jednoznačnú reprodukciu experimentálnych výsledkov. Z toho dôvodu sme vygenerovali viacero datasetov o rôznych veľkostiach od 1000 vzoriek až po 800 000 vzoriek (t.j. všetky označené vzorky z EMBER datasetu). Všetky datasety ponúkajú rovnakú distribúciu škodlivých a legitímnych vzoriek (t.j. každý dataset obsahuje 50% malvéru a 50% legitímneho softvéru). Hlavným dôvodom vytvorenia datasetov o rôznych veľkostiach bolo, že algoritmy konceptového učenia (t.j. algoritmy, na ktoré primárne cielime) sú relatívne výpočtovo náročné, najmä v porovnaní s tradičnými algoritmami z oblasti strojového učenia. Z toho dôvodu môže byť užitočné mať k dispozícii aj menej robustné datasety, ktoré sú vhodnejšie pre náročné algoritmy a zároveň umožnia jednoduchú reprodukciu a porovnanie výsledkov. Taktiež, pre každú veľkosť datasetu (okrem celého datasetu) sme vygenerovali viacero variantov, kde vzorky boli náhodne vyberané z celého EMBER datasetu. Viac detailov je možné vidieť v tabuľke 4.4.

Názov	**Počet vzoriek**	**Pozitívne príklady**	**Negatívne príklady**	**Varianty**
`dataset_N_1000.owl`	1000	500	500	10
`dataset_N_10000.owl`	10 000	5000	5000	10
`dataset_N_100000.owl`	100 000	50 000	50 000	10
`dataset_N_800000.owl`	800 000	400 000	400 000	1

Tabuľka 4.4: Všeobecné vlastnosti čiastkových datasetov.

Každý jeden variant datasetu obsahuje nasledujúce súbory (kde `N` je poradové číslo variantu a `M` je konkrétna veľkosť datasetu):

`dataset_N_M.owl`: jedná sa o súbor so samotnou ontológiou, ktorá je naplnená individuálmi a obsahuje `M` škodlivých/legitímnych vzoriek.

`dataset_N_M_raw.json`: jedná sa o JSON súbor, ktorý obsahuje surové dáta z EMBER datasetu použité na generovanie konkrétneho variantu (vrátane nepoužitých vlastností).

`dataset_N_M_examples.json`: zoznam v JSON formáte so špecifikáciou, ktoré vzorky patria do pozitívnej triedy a ktoré do negatívnej.

Pôvodným zámerom bolo taktiež zverejniť aj preddefinované rozdelenie datasetov na trénovaciu a testovaciu časť z dôvodu aby bola možná úplne exaktná replikácia výsledkov. Takéto rozdelenie však môže spôsobiť príliš

dobré (resp. zlé) výsledky pre niektoré algoritmy, v závislosti od náhody, prípadne môže viesť aj k pretrénovaniu. Z toho dôvodu sme sa rozhodli nedefinovať špecifické rozdelenie na trénovaciu/testovaciu časť a odporučiť využitie metodológie *krížovej validácie* [157] (viac v kapitole 5.2.1).

	Veľkosť			
Vlastnosť	**1000**	**10 000**	**100 000**	**800 000**
Axiómy	63 k	621 k	6,189 k	49,506 k
Triedy tvrdenia	5.8 k	57 k	568 k	4,545 k
Role tvrdenia	34 k	343 k	3,418 k	27,348 k
Dátové vl. tvrdenia	16 k	164 k	1,634 k	13,075 k
Individuály	5.8 k	57 k	567 k	4,537 k
Pozitívne príklady	500	5 k	50 k	400 k
Negatívne príklady	500	5 k	50 k	400 k

Tabuľka 4.5: Približné vlastnosti čiastkových datasetov o rôznych veľkostiach.

Základné metriky pre jednotlivé veľkosti datasetov možno vidieť v tabuľke 4.5. Je nutné poznamenať, že sa jedná o aproximované priemery pre jednotlivé varianty, ktorých metriky sa čiastočne líšia (aproximované sú z dôvodu vysokej výpočtovej náročnosti). Niektoré čiastkové datasety sa môžu javiť ako relatívne menšie v porovnaní so známymi datasetmi, bežne používanými v strojovom učení. Avšak v porovnaní s bežne používanými SML datasetmi, sú veľkosti 1k a 10k porovnateľné s tými najrobustnejšími, pričom veľkosti 100k a 800k sú neporovnateľne rozsiahlejšie.

Naša *PE Malware* ontológia, vrátane všetkých čiastkových datasetov, skriptov a zoznamu akcií je dostupná v repozitári[3].

[3]https://github.com/orbis-security/pe-malware-ontology

Kapitola 5

Experimenty

Táto časť práce sa venuje samotným experimentom, ktoré sme vykonali v rámci výskumu. V kapitole 5.1 sa venujeme popisu a celkovému prehľadu všetkých experimentov, spolu s vedeckými hypotézami, ktoré sme chceli overiť. Kapitola 5.2 sa venuje popisu spôsobu, akým sme trénovali a validovali naše detekčné modely. Následne, kapitoly 5.3 až 5.8 sa postupne venujú výsledkom jednotlivých experimentálnych častí, od hľadania optimálnych hyper-parametrov, trénovania rôznych typov modelov až po samotné útoky na klasifikátory.

5.1 Popis experimentov

V rámci tejto podkapitoly si uvedieme sumarizáciu experimentov, ktoré sme vykonali v rámci výskumu. Všetky experimenty uvádzané v práci boli spúšťané na počítači s 18 jadrovým procesorom Intel Core i9-10980XE, 256 GB pamäte RAM a operačným systémom Debian 5.10.140-1. Skúmali sme štyri algoritmy konceptového učenia, ktoré môžeme všeobecne rozdeliť do dvoch kategórií: neparalelné (OCEL a CELOE) a paralelné (PARCEL a SPACEL). Na meranie jednotlivých trénovacích cyklov sme používali *čistý čas* (nazývaný aj *user time* alebo CPU čas), ktorý môžeme charakterizovať ako čas ktorý strávil procesor vykonávaním programu, čím dokážeme efektívnejšie porovnávať algoritmy, keďže máme istotu, že všetky strávili na procesore rovnaký čas [158]. Jeden trénovací cyklus pre neparalelné algoritmy bol nastavený na 2 hodiny čistého času, ktoré zodpovedajú 2 hodinám reálneho času, keďže tieto algoritmy bežia v jednom vlákne. Pre paralelné algoritmy sme zvolili 24 hodín čistého času. Paralelné algoritmy boli spúšťané s 12 pracovnými vláknami, t.j. reálny čas jedného cyklu trval taktiež približne 2 hodiny reálneho času. Samotné časy boli nastavené experimentálne počas prvotných experimentov, pričom cieľom bolo poskytnúť algoritmom dostatok času. Taktiež je nutné poznamenať, že vyššie spomínané časy platia pre jeden trénovací cyklus z piatich pri použití krížovej validácie (viď. kapitola 5.2). Celkovo, jeden kompletný trénovací cyklus trval 10 hodín (pre neparalelné algoritmy), resp. 5 dní čistého času (pre paralelné algoritmy). V rámci experimentov sme používali vždy 80% vzoriek na trénovanie a 20% na testovanie (kde pomer medzi triedami bol zachovaný: 50% malvér a 50% legitímny softvér).

Našim hlavným cieľom bolo overiť nasledujúce vedecké hypotézy:

1. Konceptové učenie je vhodné pre riešenie problému detekcie malvéru, pričom samotný charakter metodiky poskytuje možnosť interpretovať nielen výsledky jednotlivých klasifikácií, ale aj celý model (v podobe vysvetliteľných konceptových výrazov).

2. Konceptové učenie je výpočtovo náročná metóda. Použitím rozsiahlejšieho datasetu vieme získať úspešnejší model (za cenu dlhšieho času trénovania).

3. Zameraním sa na jednotlivé rodiny malvéru vieme zúžiť prehľadávací priestor pre konceptové učenie a kombináciou jednotlivých modelov vieme skonštruovať efektívnejší klasifikátor.

4. Konceptové učenie je z hľadiska bezpečnosti a rôznych útokov porovnateľné s bežnými metódami strojového učenia.

Na overenie vyššie spomínaných hypotéz sme navrhli nasledovnú množinu experimentov:

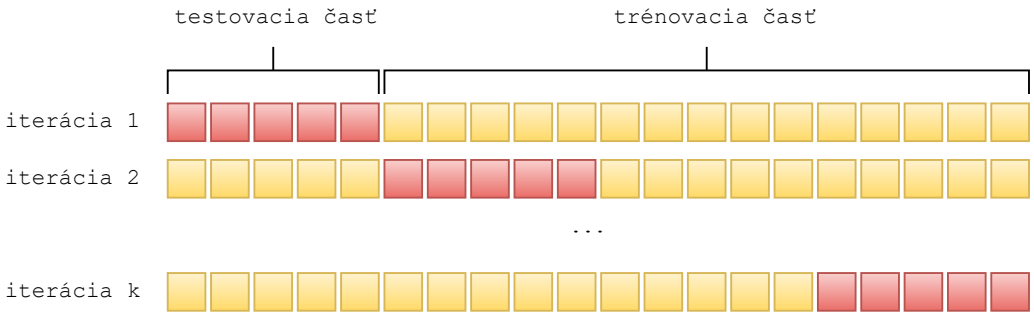
- **Časť 1**: prvá experimentálna časť sa venuje hľadaniu optimálnych hyper-parametrov pre jednotlivé algoritmy konceptového učenia. V rámci tejto časti sme sa venovali kalibrácii parametrov na jednom testovacom čiastkovom datasete s cieľom nájsť také hodnoty parametrov, ktoré nám dávali najlepšie čísla z hľadiska samotnej detekcie. Táto časť je popísaná v kapitole 5.3.
- **Časť 2**: druhá časť sa venuje validácii hyper-parametrov, získaných z predchádzajúcich experimentov. V rámci tejto časti sme použili nájdené parametre a natrénovali jednotlivé modely na ďalších piatich rôznych čiastkových datasetoch bez ďalšej kalibrácie. Cieľom bolo preskúmať, do akej miery je možné preniesť nájdené parametre na ďalšie datasety. Táto časť je popísaná v kapitole 5.4.
- **Časť 3**: z dôvodu výpočtovej náročnosti konceptového učenia boli predchádzajúce experimenty aplikované na relatívne malých datasetoch. Cieľom tejto časti experimentov bolo natrénovať jednotlivé modely na väčšom datasete a overiť tak, do akej miery je možné aplikovať konceptové učenie na robustnejšie datasety. Táto časť je popísaná v kapitole 5.5.
- **Časť 4**: štvrtá časť sa venuje skúmaniu a kalibrácii hyper-parametrov, pomocou ktorých vieme počas učenia využiť aj dátové typy. V nami navrhnutej ontológii sa nachádza relatívne veľké množstvo dátových parametrov (reťazce a numerické dátové typy), ktoré sme však v predchádzajúcich experimentoch nepoužívali (t.j. spresňovanie dátových parametrov bolo vypnuté). Cieľom tejto experimentálnej časti bolo preveriť, do akej miery je užitočné využiť tieto dátové typy pri konceptovom učení. Táto časť experimentov je popísaná v kapitole 5.6.
- **Časť 5**: piata časť experimentov sa zameriava na jednotlivé rodiny malvéru. Cieľom týchto experimentov bolo preskúmať, či je možné zefektívniť konceptové učenie, ak sa pri trénovaní zameriame na jednotlivé rodiny malvéru namiesto trénovania jedného všeobecného modelu (ktorý zahŕňa viacero rodín). Táto časť práce je popísaná v kapitole 5.7.
- **Časť 6**: posledná experimentálna časť sa venuje bezpečnosti samotných naučených konceptových výrazov. Cieľom tejto práce bolo preskúmať, do akej miery sú konceptové výrazy odolné voči útokom. Táto časť práce je popísaná v kapitole 5.8.

5.2 Metódy validácie modelu

V rámci tejto podkapitoly si uvedieme, akým spôsobom a pomocou akých metrík sme validovali natrénované modely v našej práci.

5.2.1 Krížová validácia

Pre všetky experimenty (v kapitolách 5.3 až 5.8) sme použili techniku *krížovej validácie* (z angl. *cross-validation*) [159]. Všeobecne, pri strojovom učení je nutné dataset, ktorý máme k dispozícii, rozdeliť na dve časti: trénovaciu a testovaciu časť. Trénovacia časť datasetu sa použije na učenie modelu a testovacia časť sa použije na overenie, do akej miery je natrénovaný model všeobecný a dokáže správne klasifikovať dáta, ktoré predtým nevidel. V prípade jednoduchého rozdelenia datasetu na dve časti však hrozí, že môžeme mať šťastie pri výbere (resp. smolu) a vyhodnotenie modelu nebude odrážať jeho skutočné možnosti. Pri *krížovej validácii* sa naopak dataset rozdelí nie na dve časti, ale na k častí, kde každá z nich je približne rovnakej veľkosti. Vtedy hovoríme aj o tzv. *k-násobnej krížovej validácii*.



Obrázok 5.1: Diagram k násobnej krížovej validácie.

Ako môžeme vidieť na obrázku 5.1, z k častí sa vždy jedna použije na testovanie modelu a zvyšné sa použijú na trénovanie, až kým sa nevystriedajú všetky časti. Takýmto spôsobom sa vytvorí k dvojíc (každá dvojica tvorí trénovaciu a testovaciu časť). Výsledky sa následne priemerujú. Takýmto spôsobom vieme lepšie validovať zovšeobecnenie natrénovaného modelu aj na relatívne menších datasetoch. Zrejmou nevýhodou takéhoto prístupu je, že model musíme trénovať k krát, čo môže byť relatívne náročné z výpočtového hľadiska.

Medzi najpoužívanejšie hodnoty parametra k patria 3, 5 a 10. Z dôvodu vysokej výpočtovej náročnosti experimentov v našej práci sme sa rozhodli zvoliť ako kompromis nastavenie $\mathbf{k = 5}$.

5.2.2 Metriky na vyhodnocovanie úspešnosti

Keďže jedným z cieľov práce je vytvoriť binárny klasifikátor, pri predikcii prvkov do jednej z tried (pozitívnej alebo negatívnej) môžu nastať nasledujúce štyri stavy:

- **skutočne pozitívny**: prvok patrí do pozitívnej triedy a zároveň ho správne klasifikujeme ako pozitívny. Takéto prvky budeme označovať ako TP (z angl. *True Positive*).
- **falošne pozitívny**: prvok patrí do negatívnej triedy, ale klasifikujeme ho ako pozitívny. Takéto prvky budeme označovať ako FP (z angl. *False Positive*).
- **skutočne negatívny**: prvok patrí do negatívnej triedy a zároveň ho správne klasifikujeme ako negatívny. Takéto prvky budeme označovať ako TN (z angl. *True Negative*).
- **falošne negatívny**: prvok patrí do pozitívnej triedy, ale klasifikujeme ho ako negatívny. Takéto prvky budeme označovať ako FN (z angl. *False Negative*).

Taktiež si definujme množinu P ako celkový počet pozitívnych prvkov a množinu N ako celkový počet negatívnych prvkov. Na základe definícií štyroch stavov si uvedieme metriky [160], ktoré sa používajú na vyhodnocovanie úspešnosti binárnych klasifikátorov a ktoré zároveň aj používame v našej práci:

Správnosť: z angl. *accuracy*, môžeme chápať ako pomer všetkých správnych výstupov modelu ku všetkým výstupom. Definíciu správnosti môžeme vidieť vo vzorci 5.1, kde v čitateli vidíme všetky správne výsledky klasifikátora, t.j. TP a TN a menovateľ P + N predstavuje celkový dataset.

$$\text{správnosť} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} \tag{5.1}$$

Presnosť: z angl. *precision*, vyjadruje podiel medzi správne predikovanými pozitívnymi prvkami voči všetkým prvkom, ktoré klasifikátor označil ako pozitívne t.j. TP + FP (vzorec 5.2).

$$\text{presnosť} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5.2}$$

Senzitivita: z angl. *recall*, vyjadruje citlivosť natrénovaného modelu na pozitívne prvky. Ako môžeme vidieť vo vzorci 5.3, jedná sa o podiel všetkých správne označených pozitívnych prvkov TP voči všetkým pozitívnym prvkom P, resp. TP + FN.

$$\text{senzitivita} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.3}$$

FN miera: z angl. *FN rate*, vyjadruje pomer nesprávne klasifikovaných pozitívnych prvkov (t.j. FN) voči všetkým pozitívnym prvkom v datasete (viď. 5.4).

$$\text{FN miera} = \frac{\text{FN}}{\text{FP} + \text{FN}} \tag{5.4}$$

FP miera: z angl. *FP rate*, označuje pomer nesprávne klasifikovaných negatívnych prvkov, t.j. FP, voči všetkým negatívnym prvkom v datasete (viď. 5.5).

$$\text{FP miera} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{5.5}$$

F1 miera: z angl. *F1-measure* kombinuje dve metriky: presnosť a senzitivitu. Formálne sa jedná o ich geometrický priemer, viď. 5.6.

$$\text{F1 miera} = 2\frac{\text{presnosť} \cdot \text{senzitivita}}{\text{presnosť} + \text{senzitivita}} \tag{5.6}$$

5.3 Experimentálna časť 1

Cieľom prvej experimentálnej časti bolo nájsť optimálne hyper-parametre a natrénovať tak klasifikačný model pre všetky štyri algoritmy konceptového učenia. Z dôvodu vysokej výpočtovej náročnosti sme sa zamerali na menší dataset o veľkosti 1000 vzoriek. Samotný dataset bol náhodne vybraný z množiny čiastkových datasetov veľkosti 1000 (viď. 4.5). Použitý bol konkrétne dataset `dataset_8_1000.owl`. Z dôvodu vysokého množstva rôznych parametrov a ich kombinácií sme sa rozhodli kalibráciu hyper-parametrov vykonať nasledovne. Na začiatku sme identifikovali množinu najdôležitejších parametrov, ktoré sme sa rozhodli testovať (viď. kapitola 5.3.1). Samotná kalibrácia následne prebiehala vo fázach, kde v jednej fáze sa vždy kalibroval konkrétny parameter, pričom najúspešnejšie nastavenie sa použilo v ďalšej fáze, pri kalibrácii ďalšieho parametra.

Všetky tabuľky uvedené v rámci tejto kapitoly dodržujú nasledujúci formát. Konkrétne kalibračné nastavenie je vo formáte:

`algoritmus(šum/nominály/negácia/some-only/kardinalita)`

kde `algoritmus` predstavuje konkrétny algoritmus konceptového učenia (t.j. OCEL, CELOE, PARCEL alebo SPACEL) a `šum`, `nominály`, `some-only` a `kardinalita` predstavujú jednotlivé hyper-parametre. Keďže niektoré hyper-parametre predstavujú binárnu hodnotu, sú znázornené ako ✓(zapnutý parameter) a ✗(vypnutý parameter). Na začiatku kalibrácie sme začínali so zapnutými všetkými parametrami (resp. s implicitnými hodnotami), t.j. s najvyššou expresivitou použitej logiky, v ktorej môžu byť generované konceptové výrazy. Samotné výsledky v rámci tabuliek sú zobrazené ako *priemer* ± *štandardná odchýlka* (v rámci piatich iterácií pri krížovej validácii). Podčiarknuté konkrétne nastavenie v tabuľkách značí výber daného nastavenia v rámci kalibračnej fázy.

5.3.1 Hyper-parametre

V rámci tejto experimentálnej časti sme kalibrovali nasledujúce hyper-parametre.

Šum. Šum (známy ako *noise*) patrí medzi jeden z najdôležitejších hyper-parametrov, pričom jeho význam pri učení závisí od konkrétneho algoritmu. Pre algoritmy OCEL, CELOE a PARCEL reprezentuje šum minimálne kvalitatívne kritériá, ktoré kladieme na generované riešenia, zatiaľ čo SPACEL používa šum ako ukončovaciu podmienku pri učení (parameter je možné použiť ako ukončovaciu podmienku aj pri ostatných algoritmoch, avšak v našom prípade sme to nevyužili). Pre neparalelné

algoritmy šum charakterizuje maximálny počet falošne negatívnych prvkov, ktoré povoľujeme, aby pokrýval vygenerovaný konceptový výraz, aby sme ho uvažovali na ďalšie spresňovanie. Pre OCEL musí konceptový výraz spĺňať $FN\,miera < 2 \cdot (šum/100)$, zatiaľ čo pre CELOE musí platiť $FN\,miera < šum/100$. Paralelný algoritmus PARCEL používa šum ako charakteristiku maximálnej možnej FP miery, ktorú môže spĺňať čiastkové riešenie, t.j. $FP\,miera < šum/100$. Ako sme spomínali vyššie, SPACEL používa šum ako ukončovaciu podmienku, t.j. keď správnosť finálneho riešenia dosiahne $100 - šum$. Implicitná hodnota v rámci DL-Learner je nastavená na 5.

Nominály. Tento parameter umožňuje algoritmom (resp. spresňujúcemu operátoru ρ) generovať konceptové výrazy vo forme $\exists r.\{a\}$, kde r je rola a je individuál. Nominály patria do expresívnejších logík a taktiež zväčšujú prehľadávací priestor pri učení. Primárnou motiváciou pre kalibráciu tohto hyper-parametra, bolo otestovať ich efekt pri učení, keďže značné množstvo tried v našej ontológii bolo modelovaných ako prototypické inštancie (viď. kapitola 4.4.2). Tento parameter je štandardne vypnutý.

Negácie. Zapnuté negácie umožňujú spresňujúcemu operátoru generovať konceptové výrazy vo forme $\neg C$ (t.j. komplement triedy C; implicitne zapnutý parameter). Cieľom bolo opäť skúmať efekt zapnutej negácie na učenie, keďže podobne ako nominály, zväčšujú prehľadávací priestor a ich prítomnosť môže taktiež viesť k náročnejšiemu overovaniu inštancií (jedna z výpočtovo najnáročnejších častí konceptového učenia). Taktiež je nutné poznamenať, že algoritmus SPACEL bol navrhnutý tak, aby pracoval bez negácie (kvôli obráteným čiastkovým riešeniam, viď. kapitola 3.3.4). Z dôvodu zachovania celkovej konzistencie kalibračného procesu sme však tento algoritmus testovali aj so zapnutými negáciami.

Pravidlo *some-only*. Zapnuté pravidlo *some-only* zabezpečuje, že spresňujúci operátor môže generovať konceptové výrazy so všeobecným kvantifikátorom nad rolou r iba v prípade, ak je v danom výraze zároveň existenčný kvantifikátor alebo/a obmedzenie kardinality vzťahu (nad tou istou rolou r). To znamená, že napr. výraz $\forall r.C$ je neplatný a naopak výraz $\forall r.C \sqcap \exists r.D$ je platný (kde C a D nie sú navzájom disjunktné). Účelom nastavenia (ktoré je implicitne zapnuté) je predísť pretrénovaniu. Primárnym cieľom kalibrácie *some-only* pravidla bolo preveriť, či v našom prípade nedokážu aj konceptové výrazy vo forme $\forall r.C$ dobre generalizovať a či so zapnutým pravidlom nestrácame možnosť preskúmať potenciálne vhodné konceptové výrazy.

Kardinalita. Toto nastavenie špecifikuje maximálny limit n pre kardinalitu vzťahu. Tento limit n následne používa spresňujúci operátor na generovanie výrazov vo forme $\geq n\ r.C$, $\leq n\ r.C$ alebo $= n\ r.C$, kde r je rola a C je trieda. V prípade, ak je n nastavené na 0, tak spresňujúci operátor negeneruje konceptové výrazy s obmedzením kardinality vzťahu. Zvýšenie maximálneho limitu n zväčšuje expresívnu silu generovaných konceptov, avšak za cenu zväčšenia prehľadávacieho priestoru. Cieľom bolo preskúmať vplyv tohoto parametra na úspešnosť generovaných výrazov, pričom štandardne je hodnota n nastavená na 5.

5.3.2 Šum

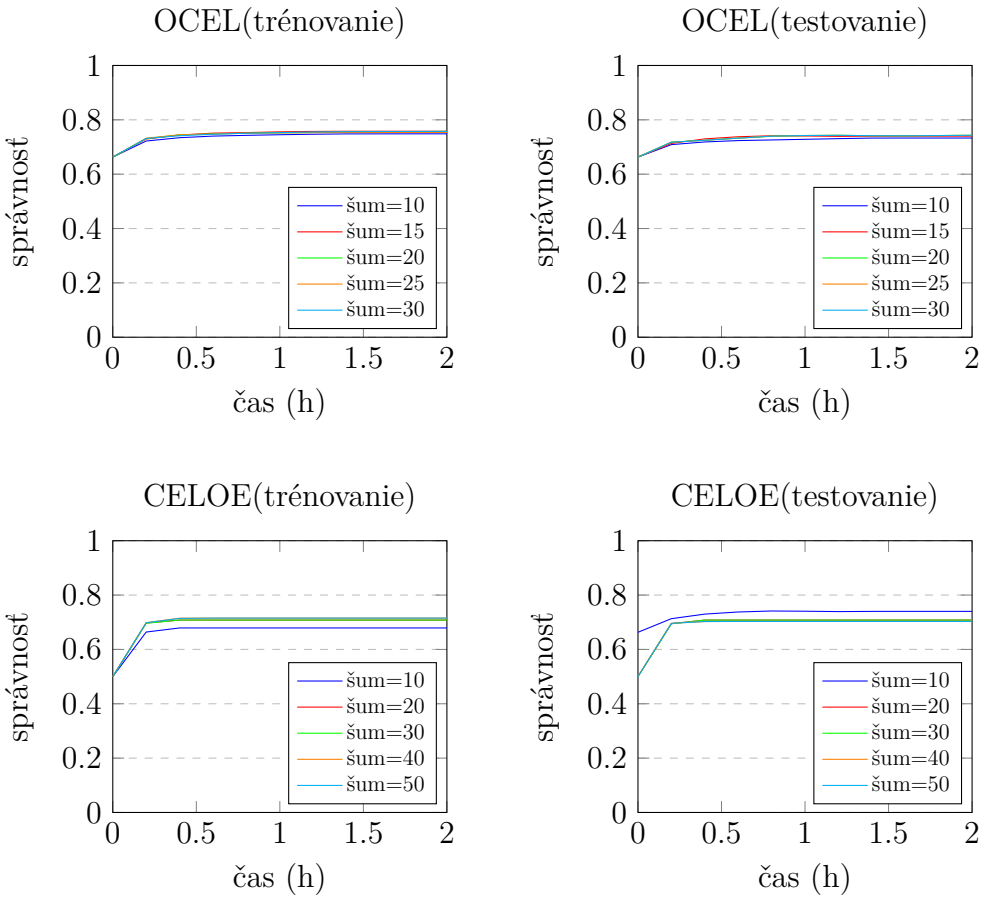
Pri hľadaní optimálnych parametrov šumu pre jednotlivé algoritmy sme postupovali nasledovne. Pre algoritmus OCEL sme začali na počiatočnej hodnote 10 a postupne ju inkrementovali po 5 až na hodnotu 30 (t.j. minimálna prípustná výška senzitivity pre trénovaciu množinu môže dosahovať hodnoty od 0.4 až po 0.8, viď. kapitola 5.3.1). Podobným spôsobom sme testovali hodnoty pre CELOE, kde sme začali na hodnote 10 a inkrementovali po 10 až na maximálnu hodnotu 50. Pre algoritmus PARCEL špecifikuje šum maximálnu možnú FP mieru, ktorú môže dosiahnuť čiastkové riešenie. Z toho dôvodu sme testovali hodnoty 0 až 3 (t.j. hodnota FP miery pre čiastkový výraz môže dosahovať hodnoty od 0.00 až po 0.03). Je nutné poznamenať, že kombináciou viacerých čiastkových riešení pomocou disjunkcie môže rýchlo narastať hodnota FP miery a z toho hľadiska bolo potrebné testovať hlavne malé hodnoty šumu. Ako sme spomínali vyššie, SPACEL používa šum iba ako ukončovaciu podmienku. Z toho dôvodu sme tento parameter nekalibrovali a zvolili si fixnú hodnotu 1.

Výsledky kalibrácie šumu pre neparalelné algoritmy môžeme vidieť v tabuľke 5.1. Z výsledkov si môžeme všimnúť, že zvyšovaním šumu postupne klesá hodnota FP miery, avšak zároveň klesá aj senzitivita (keďže postupne znižujeme množstvo pozitívnych príkladov, ktoré môže algoritmus pokryť). Samotný priebeh kalibrácie v čase môžeme vidieť na obrázku 5.2 (je dôležité poznamenať, že zobrazené krivky predstavujú priemer z piatich behov pri krížovej validácii). Ako môžeme vidieť, algoritmus OCEL dosiahol vysokú mieru správnosti približne za 30 minút CPU času a následne sa postupne zlepšoval (aj keď je nutné poznamenať, že relatívne pomaly a po veľmi malých prírastkoch na správnosti). Algoritmus CELOE naopak dosiahol svoj vrchol približne za 15 minút následne iba stagnoval, bez akéhokoľvek zlepšenia. Tento fakt môžeme pripísať tomu, že CELOE sa snaží hľadať krátke riešenia, čo môže byť značne problematické pri neskorších fázach učenia.

Výsledky kalibrácie šumu pre paralelné algoritmy môžeme vidieť v tabuľke 5.2. Z dát vidíme, že zvyšovaním hodnoty šumu zároveň rastie hodnota FP

Nastavenie	trénovanie	testovanie				
	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(10/✓/✓/✓/5)	0.74 ± 0.01	0.73 ± 0.03	0.70 ± 0.03	0.79 ± 0.04	0.32 ± 0.05	0.74 ± 0.03
OCEL(15/✓/✓/✓/5)	0.75 ± 0.01	0.74 ± 0.03	0.73 ± 0.03	0.74 ± 0.04	0.26 ± 0.04	0.74 ± 0.03
OCEL(20/✓/✓/✓/5)	0.75 ± 0.00	0.74 ± 0.02	0.77 ± 0.04	0.69 ± 0.05	0.20 ± 0.05	0.72 ± 0.03
OCEL(25/✓/✓/✓/5)	0.75 ± 0.01	0.73 ± 0.02	0.76 ± 0.03	0.69 ± 0.07	0.22 ± 0.05	0.72 ± 0.03
OCEL(30/✓/✓/✓/5)	0.75 ± 0.00	0.72 ± 0.02	0.79 ± 0.05	0.62 ± 0.09	0.17 ± 0.08	0.69 ± 0.04
CELOE(10/✓/✓/✓/5)	0.67 ± 0.00	0.67 ± 0.03	0.62 ± 0.02	0.92 ± 0.01	0.57 ± 0.05	0.74 ± 0.01
CELOE(20/✓/✓/✓/5)	0.70 ± 0.00	0.70 ± 0.02	0.65 ± 0.02	0.86 ± 0.01	0.44 ± 0.03	0.74 ± 0.01
CELOE(30/✓/✓/✓/5)	0.70 ± 0.00	0.70 ± 0.02	0.65 ± 0.02	0.86 ± 0.01	0.44 ± 0.03	0.74 ± 0.01
CELOE(40/✓/✓/✓/5)	0.71 ± 0.01	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
CELOE(50/✓/✓/✓/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03

Tabuľka 5.1: Výsledky kalibrácie šumu pre neparalelné algoritmy OCEL a CELOE.



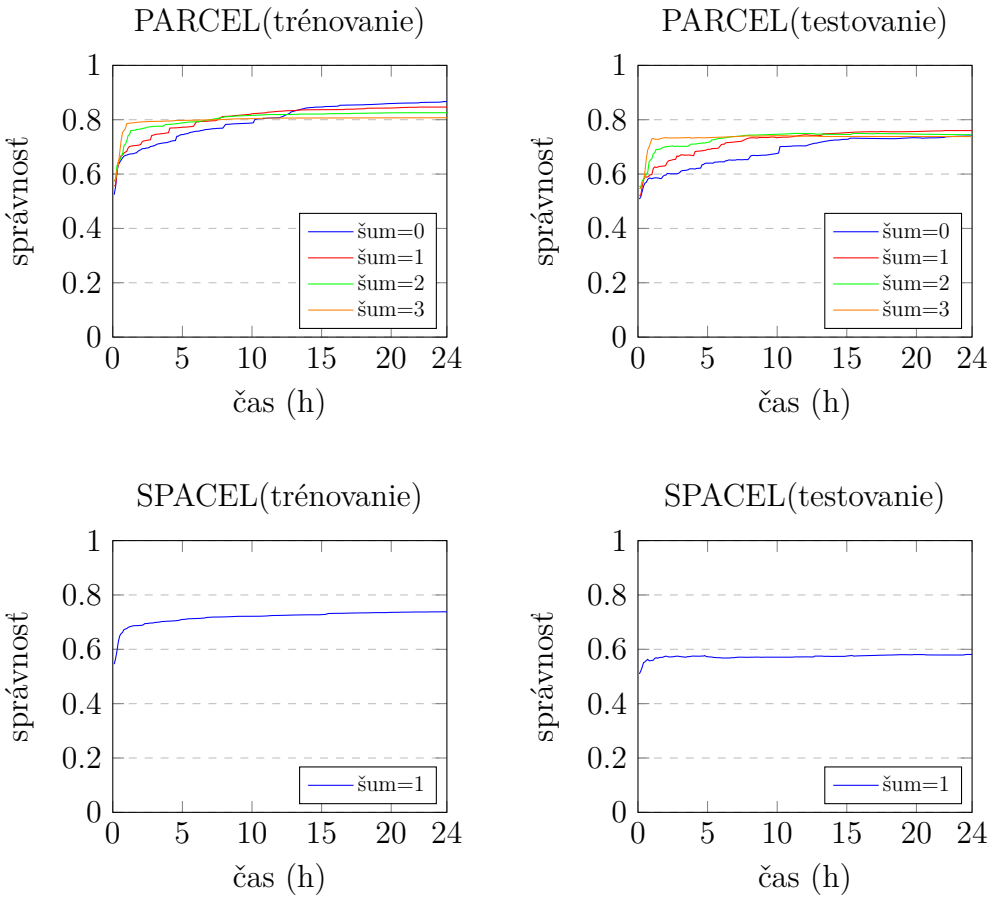
Obrázok 5.2: Priebeh kalibrácie šumu v čase pre neparalelné algoritmy OCEL a CELOE.

miery spolu so senzitivitou. Taktiež si môžeme všimnúť, že pri nastavení šumu na 0 (t.j. žiadne čiastkové riešenie nesmie obsahovať falošne pozitívne prvky) dosahujeme FP mieru 0.15 na testovacej množine. Tento fakt je pravdepodobne spôsobený relatívne malým datasetom, ktorý nie je dostatočne reprezentatívny. Zaujímavým pozorovaním boli aj výsledky algoritmu SPACEL, ktorý v tejto fáze obstál najhoršie napriek očakávaniam, že výsledky budú porovnateľné s algoritmom PARCEL. Tento fakt bol pravdepodobne spôsobený zapnutými negáciami, s ktorými algoritmus štandardne nepracuje v rámci spresňujúceho operátora (viď. kapitola 3.3.4). Priebeh kalibrácie šumu pre paralelné algoritmy môžeme vidieť na obrázku 5.3. Z kriviek môžeme vidieť, že paralelné algoritmy sa zlepšujú v čase neporovnateľne lepšie voči neparalelným algoritmom, naviac aj v neskorších fázach učenia (jednotlivé skoky na krivke, obzvlášť viditeľné pri algoritme PARCEL, znázorňujú objavenie nového čiastkové riešenia). Tento jav je spôsobený tým, že je značne náročnejšie hľadať jeden konceptový výraz, popisujúci celú trénovaciu vzorku ako výraz, ktorý popisuje iba jej časť.

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
PARCEL(0/✓/✓/✓/✓/5)	0.86 ± 0.02	0.74 ± 0.01	0.80 ± 0.03	0.63 ± 0.00	0.15 ± 0.03	0.70 ± 0.01
PARCEL(1/✓/✓/✓/✓/5)	0.84 ± 0.00	0.76 ± 0.03	0.75 ± 0.04	0.76 ± 0.05	0.24 ± 0.06	0.76 ± 0.03
PARCEL(2/✓/✓/✓/✓/5)	0.82 ± 0.00	0.74 ± 0.04	0.72 ± 0.05	0.78 ± 0.05	0.29 ± 0.08	0.75 ± 0.04
PARCEL(3/✓/✓/✓/✓/5)	0.80 ± 0.00	0.73 ± 0.04	0.71 ± 0.04	0.80 ± 0.03	0.32 ± 0.07	0.75 ± 0.03
SPACEL(1/✓/✓/✓/✓/5)	0.73 ± 0.01	0.58 ± 0.03	0.64 ± 0.06	0.37 ± 0.04	0.20 ± 0.05	0.46 ± 0.04

Tabuľka 5.2: Výsledky kalibrácie šumu pre paralelné algoritmy PARCEL a SPACEL.

Pri výbere najlepších parametrov sme sa primárne zamerali na dve metriky. Prvou bola výška F1 miery, ktorá poukazuje na celkový výkon klasifikátora a FP miery, ktorá je dôležitým ukazovateľom v oblasti detekcie malvéru. Pre OCEL sme vybrali najlepšie nastavenie šumu hodnotu 20, ktorá sa javila ako najlepší kompromis medzi dostatočne vysokou F1 mierou a nízkou FP mierou. Podobným prípadom je aj algoritmus CELOE, kde sme vybrali hodnotu 40. Toto nastavenie má však stále relatívne vysokú falošnú pozitivitu a nižšiu F1 mieru v porovnaní s nižšími nastaveniami šumu alebo v porovnaní s OCEL. Jedná sa však o logický výsledok, keďže popísať malvér jedným konceptovým výrazom, ktorý navyše musí byť krátky, je náročná úloha. Pri algoritme PARCEL sme vybrali nastavenie 0, keďže pri zvyšných nastaveniach začala relatívne rýchlo rásť hodnota FP miery. Šum pri algoritme SPACEL sme nekalibrovali, ale vykonali sme základný experiment s nastavením 1, ktorý poslúžil ako základ na porovnanie pri ďalších hyper-parametroch.



Obrázok 5.3: Priebeh kalibrácie šumu v čase pre paralelné algoritmy PARCEL a SPACEL.

5.3.3 Nominály a negácia

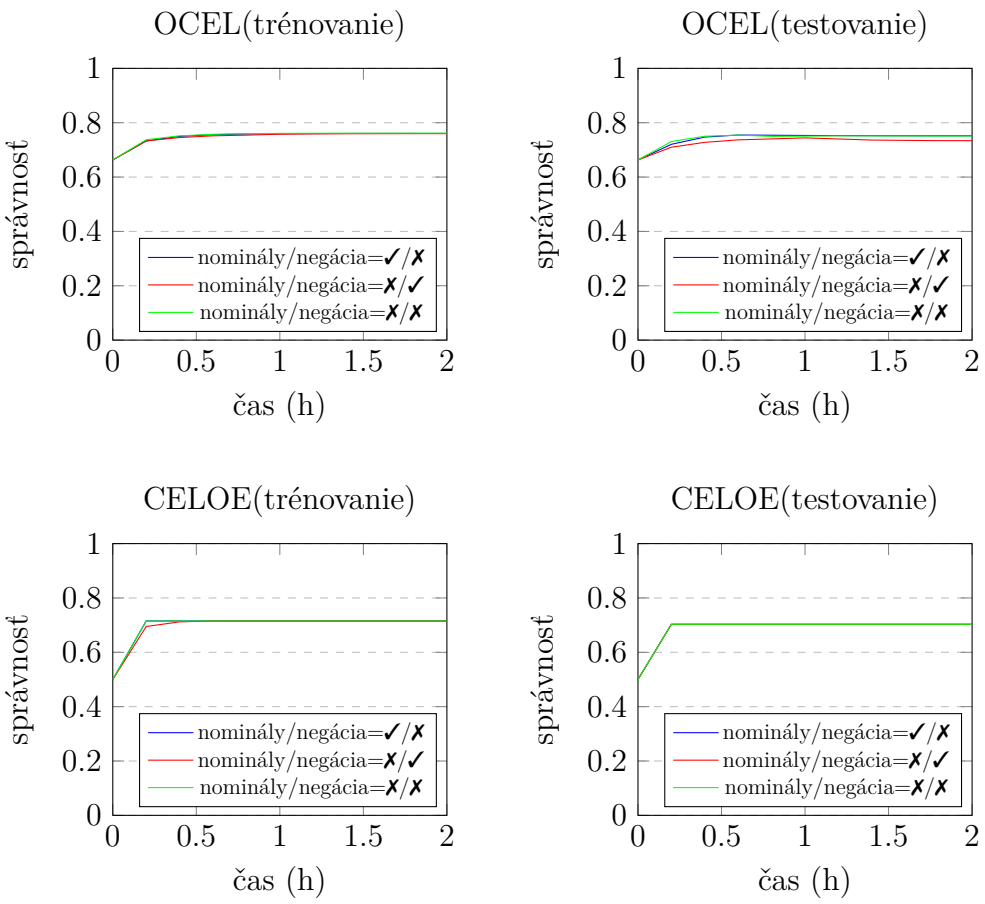
V druhej fáze kalibračného procesu sme sa rozhodli spojiť a otestovať dva binárne hyper-parametre: nominály a negácie. Zatiaľ čo v predchádzajúcej fáze sme mali oba parametre zapnuté, v tejto fáze sme zobrali najlepšie nastavenia šumu pre jednotlivé algoritmy a pre každé sme vyskúšali aplikovať tri ďalšie kombinácie spomínaných hyper-parametrov: zapnuté nominály a vypnuté negácie, vypnuté nominály a zapnuté negácie a vypnuté oba parametre.

Výsledky kalibrácie pre neparalelné algoritmy môžeme vidieť v tabuľke 5.3. Z výsledkov môžeme vidieť, že konfigurácia týchto parametrov príliš veľkú zmenu nepriniesla. Pri CELOE sme nezaznamenali dokonca žiadne zmeny. OCEL naopak zaznamenal čiastočné zlepšenie pri konfiguráciách s vypnutou negáciou, kde sme pozorovali nárast F1 miery z 0.72 na 0.74. Pôvodnou hypotézou bolo taktiež obmedzenie expresivity spresňujúceho operátora s cieľom zrýchliť generovanie konceptov a prehľadať tak väčšiu časť stromu za rovnaký čas. Tento trend sa prejavil pri OCEL iba čiastočne, keď s vypnutými hyper-parametrami dokázal algoritmus prehľadať priemerne o 20k konceptov viac za 2 hodiny CPU času. Tento trend bol výraznejší pri CELOE (približne o 800k konceptov viac s vypnutými negáciami a nominálmi), aj keď sa neprejavil priamo do zlepšenia samotných metrík. Priebeh kalibrácie v čase môžeme vidieť na obrázku 5.4. Trend učenia zostal približne rovnaký ako v predchádzajúcej fáze, aj keď si môžeme všimnúť, že najrýchlejšie prebiehalo učenie s vypnutými parametrami.

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(20/✓/✗/✓/5)	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.02	0.73 ± 0.04	0.23 ± 0.03	0.74 ± 0.03
OCEL(20/✗/✓/✓/5)	0.75 ± 0.00	0.73 ± 0.04	0.77 ± 0.07	0.67 ± 0.06	0.20 ± 0.09	0.71 ± 0.03
OCEL(20/✗/✗/✓/5)	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.04	0.72 ± 0.04	0.22 ± 0.05	0.74 ± 0.02
CELOE(40/✓/✗/✓/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
CELOE(40/✗/✓/✓/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
CELOE(40/✗/✗/✓/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03

Tabuľka 5.3: Výsledky kalibrácie nominálov a negácie pre neparalelné algoritmy OCEL a CELOE.

Výsledky kalibrácie pre paralelné algoritmy môžeme vidieť v tabuľke 5.4. Zo samotných dát môžeme vidieť, že pre PARCEL sme vypnutím oboch hyper-parametrov dokázali zlepšiť F1 mieru z 0.70 na 0.77, iba s minimálnym nárastom FP miery. Podobným prípadom bol aj algoritmus SPACEL, kde sme pozorovali nárast F1 miery z 0.46 na 0.76. Z dát taktiež vyplýva, že vysoký efekt na úspešnosť tohoto algoritmu malo najmä vypnutie negácií. Taktiež z priebehu kalibrácie v čase vyplýva, že učenie prebiehalo najrýchlejšie s



Obrázok 5.4: Priebeh kalibrácie nominálov a negácie v čase pre neparalelné algoritmy OCEL a CELOE.

vypnutými hyper-parametrami (viď. obrázok 5.5). Pri paralelných algoritmoch sme taktiež nepozorovali vyššie množstvo prehľadaných výrazov pri obmedzení expresivity spresňujúceho operátora, avšak pozorovali sme práve výraznejšie zlepšenie metrík.

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
PARCEL(0/✓/✗/✓/5)	0.93 ± 0.00	0.77 ± 0.02	0.80 ± 0.04	0.71 ± 0.04	0.17 ± 0.05	0.75 ± 0.02
PARCEL(0/✗/✓/✓/5)	0.77 ± 0.02	0.65 ± 0.01	0.76 ± 0.02	0.44 ± 0.05	0.14 ± 0.03	0.56 ± 0.04
PARCEL(0/✗/✗/✓/5)	0.93 ± 0.00	0.78 ± 0.02	0.80 ± 0.02	0.74 ± 0.05	0.17 ± 0.03	0.77 ± 0.03
SPACEL(1/✓/✗/✓/5)	0.89 ± 0.01	0.71 ± 0.02	0.73 ± 0.03	0.66 ± 0.05	0.24 ± 0.05	0.69 ± 0.02
SPACEL(1/✗/✓/✓/5)	0.72 ± 0.01	0.58 ± 0.03	0.64 ± 0.04	0.35 ± 0.06	0.19 ± 0.03	0.45 ± 0.06
SPACEL(1/✗/✗/✓/5)	0.93 ± 0.00	0.76 ± 0.03	0.76 ± 0.03	0.77 ± 0.04	0.24 ± 0.04	0.76 ± 0.03

Tabuľka 5.4: Výsledky kalibrácie nominálov a negácie pre paralelné algoritmy PARCEL a SPACEL.

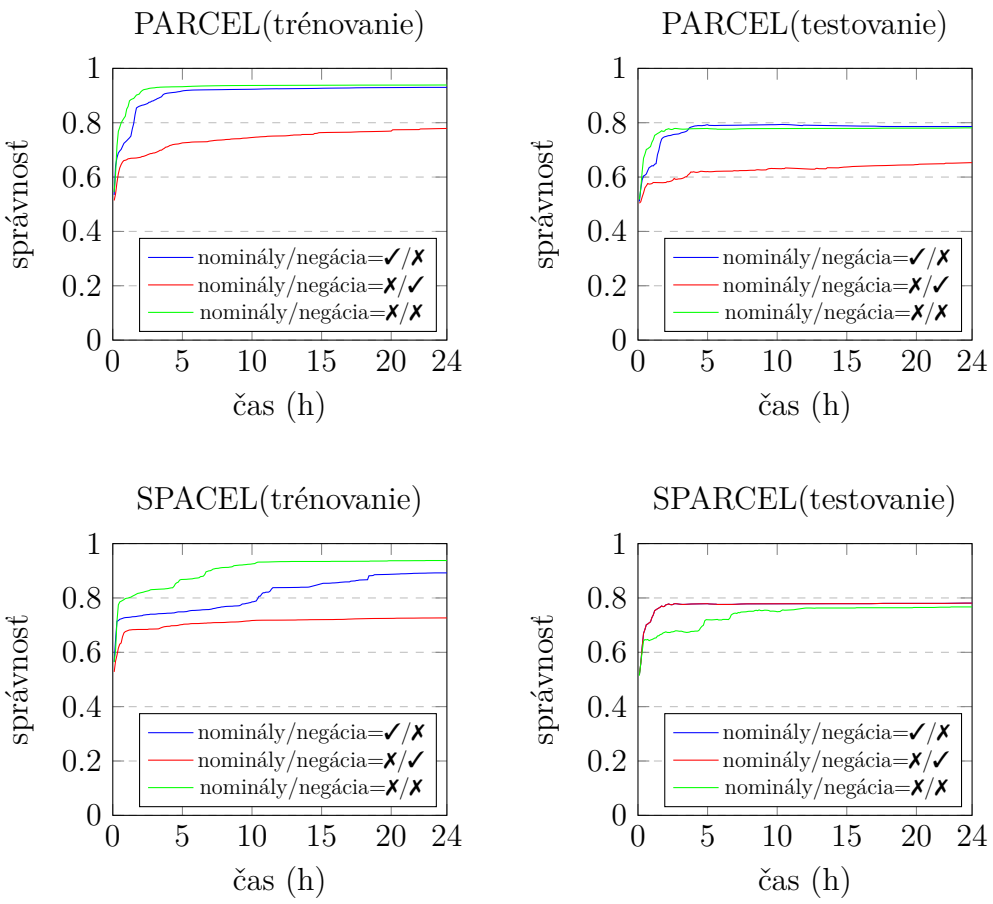
Pre všetky algoritmy sme vybrali do ďalšej fázy nastavenie s vypnutými nominálmi a negáciami. Zatiaľ čo pri OCEL, PARCEL a SPACEL sme sa rozhodli na základe zlepšenia pozorovaných metrík, pri algoritme CELOE sme sa rozhodli vybrať rovnaké nastavenie z dôvodu prehľadania väčšieho množstva konceptov, aj keď sme nedokázali získať vyššiu úspešnosť (avšak prehľadaním väčšieho množstva konceptov vieme pravdepodobnosť zlepšenia zvýšiť).

5.3.4 Pravidlo *some-only*

V tejto fáze sme testovali vplyv *some-only* pravidla na učenie. Samotný hyper-parameter sme testovali tak, že pre doposiaľ najúspešnejšie nastavenia sme daný parameter vypli (predchádzajúce testy mali tento parameter zapnutý).

Výsledky kalibrácie pre neparalelné a paralelné algoritmy spolu môžeme vidieť v tabuľke 5.5. Minimálne zlepšenie metrík sme pozorovali pri algoritme OCEL[1]. Samotný všeobecný kvantifikátor, bez existenčného, sme vo finálnych konceptových výrazoch nepozorovali, avšak je pravdepodobné, že vypnutím pravidla sme sa dokázali vnoriť do iných častí prehľadávacieho stromu a nájsť tak čiastočne lepšie riešenie (viac o samotných výstupoch v podobe konceptových výrazov v kapitole 5.3.7). Algoritmus CELOE nezaznamenal opäť žiadne zlepšenie v porovnaní s predchádzajúcou fázou. Tento efekt bol pravdepodobne opäť spôsobený zameraním algoritmu na krátke koncepty. Rozhodli sme sa však posunúť do ďalšej fázy nastavenie s vypnutým *some-only* pravidlom z dôvodu vyššej flexibility spresňujúceho operátora. Pre paralelné

[1]z dôvodu uvádzania metrík na dve desatinné miesta nemožno samotné zlepšenie vidieť v tabuľkách (napr. nárast F1 miery z 0.7433 na 0.7447)



Obrázok 5.5: Priebeh kalibrácie nominálov a negácie v čase pre paralelné algoritmy PARCEL a SPACEL.

algoritmy sme naopak pozorovali relatívne vysoký pokles úspešnosti v prípade vypnutia *some-only* pravidla. V prípade algoritmu PARCEL nastal pokles F1 miery z 0.77 na 0.71 a v prípade SPACEL z 0.76 na 0.68. Zhoršenie metrík môžeme pravdepodobne pripísať čiastkovým riešeniam, ktoré využívali samotný všeobecný kvantifikátor a zároveň pokrývali dostatok pozitívnych príkladov v trénovacej množine, čím sa dostali do zoznamu konceptov určených na ďalšie spresňovanie (na úkor lepšie zovšeobecňujúcim riešeniam). Takéto čiastkové riešenia následne nepreukázali dostatočne vysokú úspešnosť na testovacej množine. Pre paralelné algoritmy sme teda ponechali *some-only* pravidlo zapnuté.

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(20/✗/✗/✗/5)	0.76 ± 0.00	0.75 ± 0.02	0.77 ± 0.03	0.72 ± 0.04	0.22 ± 0.05	0.74 ± 0.02
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
PARCEL(0/✗/✗/✗/5)	0.88 ± 0.02	0.75 ± 0.02	0.83 ± 0.05	0.63 ± 0.08	0.13 ± 0.06	0.71 ± 0.04
SPACEL(1/✗/✗/✗/5)	0.85 ± 0.00	0.69 ± 0.01	0.74 ± 0.03	0.60 ± 0.02	0.20 ± 0.05	0.66 ± 0.01

Tabuľka 5.5: Výsledky kalibrácie pravidla *some-only* pre všetky algoritmy

5.3.5 Kardinalita

Posledný hyper-parameter, ktorý sme kalibrovali v rámci tejto experimentálnej časti bol limit na kardinalitu vzťahu. V rámci tejto časti sme testovali postupne štyri hodnoty limitu n: 0, 10, 20 a 30. Pri nastavení n na hodnotu 0 sme simulovali vypnutie samotných reštrikcií na kardinalitu vzťahu, zatiaľ čo postupným zvyšovaním n sme rozširovali potenciálny prehľadávací priestor a samotnú expresivitu generovaných výrazov.

Výsledky kalibrácie pre neparalelné algoritmy môžeme vidieť v tabuľke 5.6. Pri oboch algoritmoch môžeme vidieť rovnaký fenomén, kde pri vypnutí reštrikcií na kardinalitu vzťahu (t.j. nastavenie $n = 0$) sa výsledky metrík zhoršili v porovnaní s predchádzajúcou fázou, kde sme mali reštrikcie zapnuté s implicitnou hodnotou 5 (aj keď je nutné poznamenať, že zhoršenie výsledkov pri OCEL bolo oveľa významnejšie v porovnaní s CELOE). Rovnako pri zvyšovaní n nad hodnotu 5 sme nepozorovali žiadne zlepšenie a výsledky boli totožné s predchádzajúcou fázou. Vo finálnych konceptoch v predchádzajúcich fázach sme vždy pozorovali prítomnosť kardinality vzťahu. Zamedzením tohoto operátora a následným zhoršením výsledkov môžeme konštatovať, že v samotných dátach sa nachádzajú také vzory, ktoré možno efektívne charakterizovať iba pomocou obmedzenia kardinality vzťahu. Následná stagnácia pri zvyšovaní hodnoty n môže mať dva dôvody. Prvým je, že hodnota vyššia ako 5 nie je potrebná na popísanie dát (pomocou jedného konceptového výrazu).

Druhým dôvodom je efektivita prehľadávania, kedy zvyšovaním hodnoty n značne zväčšujeme prehľadávací priestor, ktorý následne algoritmy nemajú šancu prejsť v danom čase. Priebeh kalibrácie v čase môžeme vidieť na obrázku 5.6. Z kriviek môžeme jasne pozorovať, že pri vypnutí kardinality vzťahu pre OCEL sa spomalilo učenie, zatiaľ čo pri CELOE nenastal žiadny viditeľný pokles.

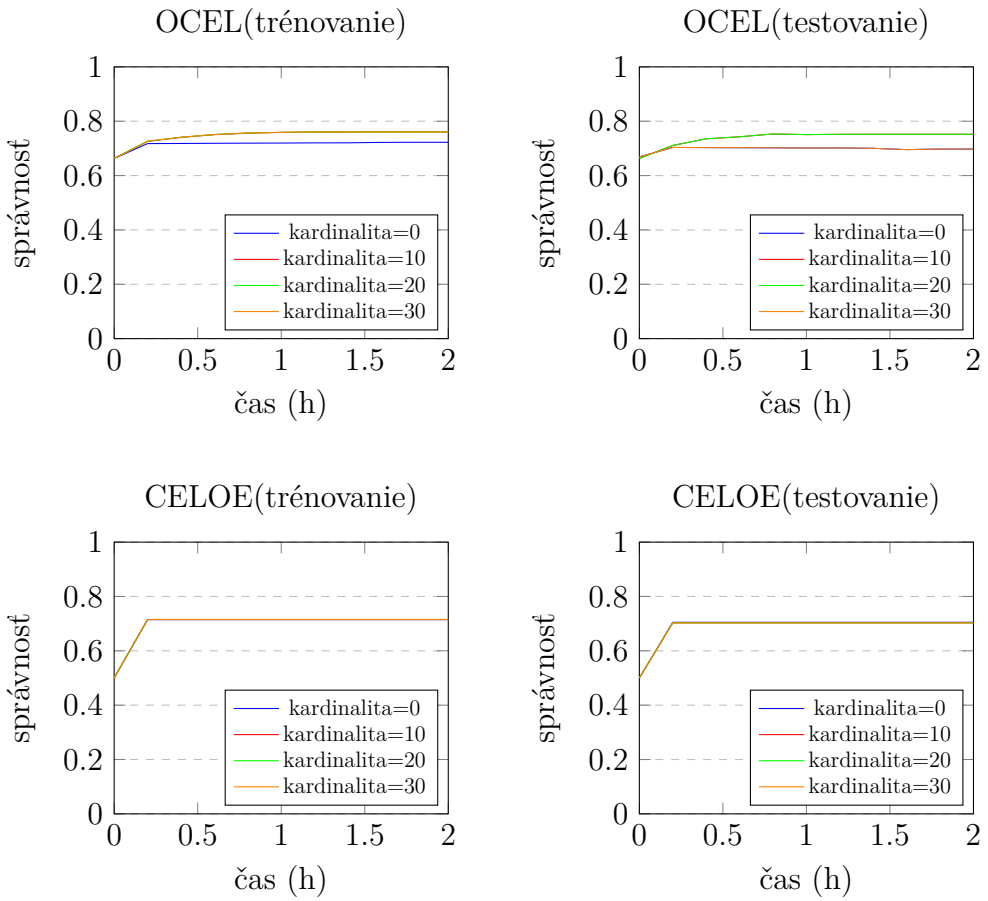
	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(20/✗/✗/✗/0)	0.72 ± 0.00	0.69 ± 0.01	0.71 ± 0.04	0.65 ± 0.10	0.27 ± 0.11	0.67 ± 0.03
OCEL(20/✗/✗/✗/10)	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.02	0.73 ± 0.04	0.23 ± 0.03	0.74 ± 0.03
OCEL(20/✗/✗/✗/20)	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.02	0.73 ± 0.04	0.23 ± 0.03	0.74 ± 0.03
OCEL(20/✗/✗/✗/30)	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.02	0.73 ± 0.04	0.23 ± 0.03	0.74 ± 0.03
CELOE(40/✗/✗/✗/0)	0.71 ± 0.00	0.70 ± 0.02	0.73 ± 0.03	0.64 ± 0.03	0.23 ± 0.04	0.68 ± 0.02
CELOE(40/✗/✗/✗/10)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
CELOE(40/✗/✗/✗/20)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
CELOE(40/✗/✗/✗/30)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03

Tabuľka 5.6: Výsledky kalibrácie maximálnej kardinality pre neparalelné algoritmy OCEL a CELOE.

Výsledky kalibrácie pre paralelné algoritmy môžeme vidieť v tabuľke 5.7. Podobne ako pri OCEL, tak aj pri PARCEL sme mohli pozorovať zhoršenie metrík pri nastavení parametra $n = 0$, pričom za poklesom môžeme opäť vidieť rovnaké dôvody (konkrétne pokles F1 miery z 0.77 na 0.71). Avšak pri vyšších nastaveniach, konkrétne $n = 10$, sme dokázali získať mierne zlepšenie[2] (aj keď za cenu mierneho nárastu FP miery). Zvyšné nastavenia ukázali iba stagnáciu metrík, resp. mierny pokles. Pri algoritme SPACEL sme pozorovali iba mierny pokles hodnôt pri všetkých nastaveniach, dokonca aj pri $n = 0$. V tomto prípade dokázal algoritmus, pomocou konceptov popisujúcich negatívne príklady, kompenzovať celkovú efektivitu modelu aj na úkor zákazu kardinality vzťahu. Priebeh učenia môžeme vidieť na obrázku 5.7, kde pri algoritme PARCEL môžeme vidieť evidentné spomalenie učenia pri $n = 0$, zatiaľ čo pri SPACEL vidíme konzistentný priebeh učenia pri všetkých nastaveniach.

Pre algoritmy OCEL, CELOE a SPACEL sme sa rozhodli ponechať originálne nastavenie $n = 5$, keďže v ostatných prípadoch sme dosahovali horšie metriky. Pre algoritmus PARCEL sme sa naopak rozhodli použiť nastavenie $n = 10$, ktoré prinieslo mierne zlepšenie a môže byť zaujímavé z hľadiska ďalších experimentov.

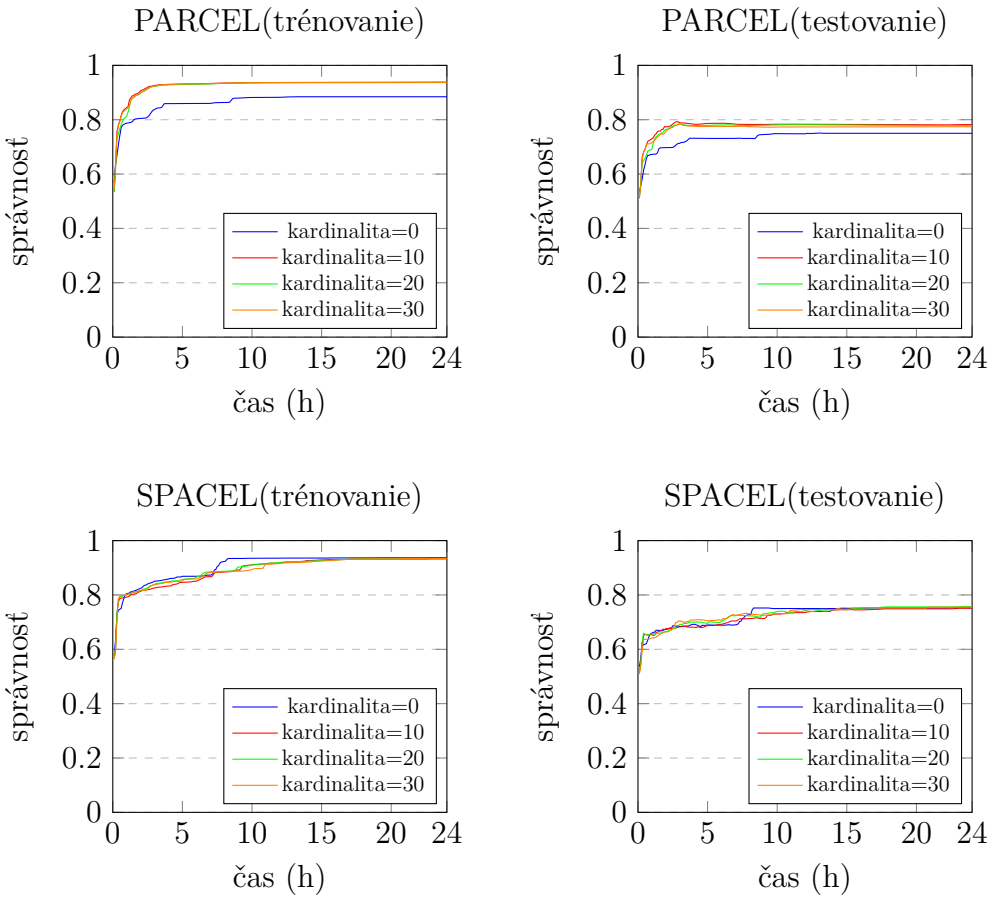
[2]napr. nárast F1 miery z 0.7708 na 0.7755



Obrázok 5.6: Priebeh kalibrácie maximálnej kardinality v čase pre neparalelné algoritmy OCEL a CELOE.

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
PARCEL(0/✗/✗/✓/0)	0.88 ± 0.00	0.75 ± 0.02	0.83 ± 0.05	0.62 ± 0.04	0.12 ± 0.06	0.71 ± 0.02
PARCEL(0/✗/✗/✓/10)	0.93 ± 0.00	0.78 ± 0.03	0.80 ± 0.03	0.74 ± 0.05	0.18 ± 0.04	0.77 ± 0.03
PARCEL(0/✗/✗/✓/20)	0.93 ± 0.00	0.77 ± 0.01	0.80 ± 0.02	0.73 ± 0.03	0.17 ± 0.02	0.76 ± 0.02
PARCEL(0/✗/✗/✓/30)	0.93 ± 0.00	0.77 ± 0.02	0.81 ± 0.04	0.71 ± 0.06	0.17 ± 0.05	0.75 ± 0.03
SPACEL(1/✗/✗/✓/0)	0.93 ± 0.00	0.75 ± 0.03	0.75 ± 0.03	0.74 ± 0.05	0.24 ± 0.04	0.76 ± 0.03
SPACEL(1/✗/✗/✓/10)	0.93 ± 0.00	0.75 ± 0.02	0.75 ± 0.02	0.75 ± 0.02	0.25 ± 0.03	0.75 ± 0.01
SPACEL(1/✗/✗/✓/20)	0.93 ± 0.00	0.75 ± 0.02	0.75 ± 0.02	0.75 ± 0.02	0.23 ± 0.03	0.75 ± 0.03
SPACEL(1/✗/✗/✓/30)	0.93 ± 0.00	0.75 ± 0.02	0.76 ± 0.03	0.75 ± 0.04	0.23 ± 0.04	0.75 ± 0.02

Tabuľka 5.7: Výsledky kalibrácie maximálnej kardinality pre paralelné algoritmy PARCEL a SPACEL.



Obrázok 5.7: Priebeh kalibrácie maximálnej kardinality v čase pre paralelné algoritmy PARCEL a SPACEL.

5.3.6 Diskusia

Finálne nakalibrované hyper-parametre pre jednotlivé algoritmy môžeme vidieť v tabuľke 5.8. Taktiež, sumarizáciu výsledkov pre všetky algoritmy (pre najlepšie nastavenia) môžeme vidieť v tabuľke 5.9. Ako môžeme vidieť, najlepší výsledok v rámci tejto experimentálnej časti sme dosiahli pomocou algoritmu PARCEL, s dosiahnutou F1 mierou 0.77 a najhorší výsledok naopak prostredníctvom algoritmu CELOE (kde sme dosiahli F1 mieru 0.69). Jedná sa o relatívne očakávaný výsledok, keďže detekcia malvéru je náročný problém a popísať dátovú množinu jedným krátkym konceptovým výrazom (CELOE) bude značne problematickejšie ako ju popísať skombinovanou množinou viacerých výrazov (PARCEL). Z celkového pohľadu sa aj najlepší výsledok javí ako relatívne slabý, najmä v porovnaní s niektorými riešeniami, ktoré využívajú strojové učenie (viď. kapitola 1.3). Všetky výsledky dosahujú najmä relatívne vysoké hodnoty FP miery (od 0.15 pre PARCEL až po 0.27 pre CELOE), čo môže byť značne problematické pri detekcii malvéru. V porovnaní so štandardnými metódami strojového učenia sme však boli limitovaní viacerými aspektami. Prvým bolo, že sme použili relatívne malú vzorku na trénovanie a to z dôvodu limitácie z hľadiska výpočtového výkonu (jednalo sa o 0.125% zo všetkých označených vzoriek z EMBER datasetu). Druhým aspektom bolo vynechanie niektorých vlastností z EMBER datasetu, ako sú napr. histogramy bajtov a reťazcov, z dôvodu, že sú inherentne nevysvetliteľné. Ako však bolo ukázané v práci [161], tieto vlastnosti sú významným prínosom pre úspešnosť algoritmov strojového učenia. Experimenty s niektorými dátovými vlastnosťami, ktoré sme sa rozhodli ponechať, budú popísané v kapitole 5.6.

algoritmus	šum	nominály	negácia	*some-only* pravidlo	kardinalita
`OCEL`	20	✗	✗	✗	5
`CELOE`	40	✗	✗	✗	5
`PARCEL`	0	✗	✗	✓	10
`SPARCEL`	1	✗	✗	✓	5

Tabuľka 5.8: Konečné nakalibrované hyper-parametre pre jednotlivé algoritmy.

5.3.7 Konceptové výrazy

Ako sme spomínali v predchádzajúcich kapitolách, jednou z najväčších výhod algoritmov konceptového učenia je ich implicitná vysvetliteľnosť v podobe konceptových výrazov. Zároveň sa môže jednať o určitý *trade-off* medzi úspešnosťou modelu a jeho interpretabilitou. V tejto podkapitole sa zameriame práve na konceptové výrazy, ktoré sme sa naučili v rámci tejto časti.

Nastavenie	trénovanie	testovanie				
	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(20/✗/✗/✗/5)	0.76 ± 0.00	0.75 ± 0.02	0.77 ± 0.03	0.72 ± 0.04	0.22 ± 0.05	0.74 ± 0.02
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.10	0.27 ± 0.11	0.69 ± 0.03
PARCEL(0/✗/✗/✓/10	0.93 ± 0.00	0.79 ± 0.01	0.82 ± 0.02	0.73 ± 0.05	0.15 ± 0.03	0.77 ± 0.02
SPACEL(1/✗/✗/✓/5)	0.93 ± 0.00	0.76 ± 0.03	0.76 ± 0.03	0.77 ± 0.04	0.24 ± 0.04	0.76 ± 0.03

Tabuľka 5.9: Sumarizácia výsledkov najlepších nastavení hyper-parametrov pre jednotlivé algoritmy.

Konceptový výraz, naučený pomocou algoritmu OCEL, môžeme vidieť v (5.3) (pozn. jedná sa o výraz získaný z jednej iterácie). Tento výraz dosiahol správnosť 0.77 na testovacej množine a môže byť interpretovaný nasledovne: *Vzorka je škodlivá, ak sa jedná o spustiteľný súbor, ktorý má viacero sekcií s právami na spustenie alebo obsahuje viac ako jednu MZ hlavičku, zároveň má sekciu, ktorá má práva na zápis a zároveň najviac jednu z nasledujúcich akcií: prijímanie spojenia, manipulácia s adresármi, enumerácie vláken v rámci procesu, získanie aktuálneho pracovného adresára pre proces alebo otváranie mutexu.* Tu je nutné poznamenať, že zatiaľ čo väčšina akcií, ktoré sa algoritmus naučil, zodpovedá jednotlivým API volaniam (napr. `OpenMutex` alebo `EnumerateThreads`), v tomto prípade sa taktiež naučil aj celkovú kategóriu akcií v podobe triedy `DirectoryHandling` (zahŕňajúca funkcie na prácu s adresármi ako sú vytváranie, prehľadávanie alebo mazanie). Prvá časť výrazu, popisujúca sekcie, môže značiť vzorku, ktorá je zbalená alebo používa nejakú formu ukrývania ďalšieho spustiteľného súboru, zatiaľ čo popis jednotlivých akcií môžeme označiť za relatívne všeobecný. Celkovo však môžeme konceptový výraz označiť ako interpretovateľný.

$$\begin{aligned}&\text{ExecutableFile} \sqcap \exists \text{has\_file\_feature}.(\text{MultipleExecutableSections} \sqcup \text{NonstandardMZ}) \\ &\sqcap \exists \text{has\_section}.\exists \text{has\_section\_flag}.\text{Writable} \\ &\sqcap \leqslant 1\, \text{has\_action}.(\text{AcceptSocketConnection} \sqcup \text{DirectoryHandling} \\ &\sqcup \text{EnumerateThreads} \sqcup \text{GetProcessCurrentDirectory} \sqcup \text{OpenMutex})\end{aligned} \tag{5.7}$$

Výraz (5.8) bol získaný z algoritmu CELOE (správnosť 0.75 na testovacej množine). Tento výraz môžeme interpretovať ako: *Vzorka je škodlivá, ak obsahuje sekciu, ktorá má vysokú entropiu alebo má zároveň práva na zápis a spustenie.* Môžeme vidieť, že tento konceptový výraz je oveľa kratší v porovnaní s výrazom (5.7). Taktiež, celkom jednoznačne označuje zbalené vzorky (napr. pomocou UPX). Takéto vzorky sa vyznačujú viacerými skomprimovanými/zašifrovanými sekciami (t.j. s vysokou hodnotou entropie) a zároveň môžu obsahovať sekciu, do ktorých sa za behu uloží dekomprimovaný/dešifrovaný kód (práva na zápis) a následne spustí (práva na spustenie).

$$\exists \text{has\_section}.\exists \text{has\_file\_feature}.(\text{HighEntropy} \sqcup \text{WriteExecuteSection}) \tag{5.8}$$

Konceptové výrazy (5.9) a (5.10) boli získané z algoritmu PARCEL. V tomto prípade je nutné poznamenať, že sa jedná o dve ukážkové čiastkové riešenia. Počas učenia konečných hyper-parametrov našiel PARCEL priemerne 80 podobných čiastkových riešení, v rámci jednej iterácie, ktoré sú kombinované do jedného finálneho konceptového výrazu (z dôvodu rozsiahlosti dané finálne výrazy nebudeme uvádzať). Výraz (5.9) dosiahol správnosť 0.56 na testovacej množine a výraz (5.10) dosiahol 0.54. Môžeme vidieť, že dosiahnuté metriky sú značne nižšie v porovnaní s výsledkami z neparalelných algoritmov. Tento jav je však očakávaný, keďže dané čiastkové riešenia popisujú vždy iba určitú podmnožinu a algoritmus získava na sile práve kombinovaním takýchto čiastkových riešení. Výraz (5.9) môžeme interpretovať ako: *Vzorka je škodlivá, ak má aspoň tri sekcie, ktoré majú neštandardné meno a zároveň majú práva na zápis.* Keďže všetky algoritmy v rámci tejto časti sme trénovali na rovnakom čiastkovom datasete, môžeme konštatovať, že veľká časť škodlivých vzoriek bola pravdepodobne zbalená. Rovnako ako v predchádzajúcich výrazoch, aj vyššie spomínané čiastkové riešenie nám potvrdzuje tento fakt. Rôzne baliče sú práve známe aj tým, že pôvodný spustiteľný súbor skomprimujú/zašifrujú do viacerých sekcií, ktoré si práve označia špeciálnym menom (t.j. menom, ktoré štandardne rôzne kompilátory negenerujú). Zbalený súbor čiastočne popisuje aj výraz (5.10), ktorý môže byť interpretovaný ako: *Vzorka je škodlivá, ak vykonáva rôzne systémové akcie a zároveň má sekciu s vysokou entropiou, ktorá obsahuje kód.* Druhá časť výrazu (ktorá spomína entropiu), nám práve označuje zbalený súbor, zatiaľ čo začiatok výrazu je viac všeobecnejší, keďže zahŕňa akcie z kategórie `SystemManipulation` (získanie aktuálneho času, vypnutie systému, získanie adresára s operačným systémom a pod.). Zatiaľ čo jednotlivé čiastkové riešenia sú relatívne krátke a ľahko interpretovateľné, celkový finálny výraz (priemerne zložený z 80 čiastkových riešení) môže byť značne menej čitateľný. Zaujímavým pozorovaním je taktiež relatívne vysoký nárast komplexity finálneho konceptového výrazu, najmä v porovnaní s algoritmom OCEL, kde sme získali výraz o dĺžke 26 s F1 mierou 0.74. Pomocou PARCEL sme získali výraz o dĺžke 756 (t.j. súčet všetkých čiastkových riešení), pričom nárast F1 miery bol iba o 0.03 na 0.77 (pri tej istej iterácií).

$$\geqslant 3\,\text{has\_section}.(\exists \text{has\_section\_feature} \,.\, \text{NonstandardSectionName} \sqcap \exists \text{has\_section\_flag}.\text{Writable}) \tag{5.9}$$

$$\exists \text{has\_action}.\text{SystemManipulation} \sqcap \exists \text{has\_section}.(\text{CodeSection} \sqcap \exists \text{has\_section\_feature}.\text{HighEntropy}) \tag{5.10}$$

Konceptové výrazy (5.11) a (5.12) boli získané prostredníctvom algoritmu SPACEL. Podobne, ako v predchádzajúcom prípade, jedná sa o dve ukážkové čiastkové riešenia. Priemerne našiel SPACEL 60 čiastkových riešení. Môžeme vidieť, že konceptové výrazy sú dlhšie ako v prípade PARCEL a to z dôvodu, že výrazy sa kombinujú aj s obrátenými čiastkovými riešeniami, ktoré popisujú negatívne príklady (časti výrazov, nasledujúce vždy za $\sqcap\neg$). Výraz (5.11) dosiahol na testovacej množine správnosť 0.57, zatiaľ čo výraz (5.12) získal 0.53. Výraz (5.11) môžeme interpretovať ako: *Vzorka je škodlivá, ak má viacero sekcií s právami na spustenie, zároveň má aspoň tri sekcie s neštandardným menom a zároveň však nemá viacero sekcií s právami na spustenie, ak obsahuje akciu mapovania soketu na konkrétnu IP adresu.* Daný výraz môžeme označiť ako najťažšie čitateľný, v porovnaní so všetkými ostatnými algoritmami, najmä vďaka negovanej časti, ktorá popisuje negatívne príklady. Prvá časť výrazu je relatívne ľahko pochopiteľná a popisuje zbalené vzorky (tieto časti výrazu sú podobné ako pri PARCEL, keďže SPACEL je jeho rozšírením). Prvou časťou výrazu, získame množinu vzoriek, ktoré majú viacero spustiteľných sekcií a aspoň tri sekcie s neštandardným menom. Následne, pomocou negovanej časti, odstránime z tejto množiny všetky vzorky, ktoré síce majú viacero spustiteľných sekcií, avšak zároveň obsahujú aj akciu `BindAddressToSocket` a získame tak finálnu množinu popisujúcu pozitívne príklady. Podobným prípadom je aj výraz (5.12), ktorý obsahuje až tri skombinované obrátené čiastkové riešenia. V niektorých prípadoch (závisiac od datasetu), môže obsahovať konceptový výraz relatívne veľké množstvo obrátených čiastkových riešení, pričom sa stále jedná iba o čiastkové riešenie, t.j. súčasť celkového riešenia. Celkovo tak môžeme zhodnotiť, že SPACEL, aj napriek relatívne dobrým metrikám v porovnaní s ostatnými algoritmami, produkuje najkomplexnejšie výrazy, ktoré sú relatívne ťažko čitateľné.

$$\begin{gathered}\exists \text{has\_file\_feature.MultipleExecutableSections} \sqcap \\ \geqslant 3\,\text{has\_section.}(\exists \text{has\_section\_feature.NonstandardSectionName}) \sqcap \\ \neg(\exists \text{has\_action.BindAddressToSocket} \sqcap \exists \text{has\_file\_feature.MultipleExecutableSections})\end{gathered} \tag{5.11}$$

$$\begin{gathered}\exists \text{has\_action.DeleteCriticalSection} \sqcap \exists \text{has\_section.}(\exists \text{has\_section\_feature.WriteExecuteSection}) \\ \sqcap\neg(\text{ExecutableFile} \sqcap \exists \text{has\_action.FlushProcessInstructionCache} \\ \sqcap \exists \text{has\_file\_feauture.MultipleExecutableSections}) \\ \sqcap\neg(\exists \text{has\_action.CreateSocket} \sqcap \exists \text{has\_file\_feature.Signature}) \\ \sqcap\neg(\exists \text{has\_action.DeleteRegistryKey} \sqcap \leqslant 1\,\text{has\_section.InitializedDataSection})\end{gathered} \tag{5.12}$$

5.4 Experimentálna časť 2

Hlavným cieľom tejto experimentálnej časti bolo validovať nájdene hyper-parametre (viď. kapitola 5.3) na ďalších čiastkových datasetoch. Zatiaľ čo samotná kalibrácia prebiehala na čiastkovom datasete 8 o veľkosti 1000 (t.j. `dataset_8_1000.owl`), na validáciu sme použili datasety 1-5 rovnakej veľkosti. Trénovanie modelov prebiehalo totožným spôsobom ako pri kalibrácii (t.j. čas trénovania, počet iterácií, pomer medzi škodlivými a legitímnymi vzorkami a pod.)

5.4.1 Validácia

Výsledky validácie pre jednotlivé čiastkové datasety možno vidieť v tabuľke 5.10. V tabuľke 5.11 uvádzame taktiež priemerné metriky v rámci všetkých validačných datasetov pre jednotlivé nastavenia. Z výsledkov môžeme pozorovať, že v porovnaní s kalibračným datasetom nastal mierny pokles (resp. nárast) hlavných metrík, ktoré primárne sledujeme, t.j. F1 miera a FP miera. Pre najlepšie nastavenie z kalibračnej fázy (t.j. nastavenie s algoritmom PARCEL) nastal pokles F1 miery z 0.77 na 0.71 a nárast FP miery z 0.15 na 0.20. Podobné pozorovanie sme si mohli všimnúť aj pri algoritme SPACEL. Konzistentnejšie výsledky sme si mohli všimnúť skôr pri neparalelných algoritmoch. Pri oboch algoritmoch síce nastal nárast FP miery, avšak čo sa týka F1 miery, tak pri algoritme OCEL nastal iba veľmi mierny pokles, zatiaľ čo pri CELOE sa hodnota F1 miery zvýšila z 0.69 na 0.72.

5.4.2 Diskusia

Napriek všeobecnému poklesu úspešnosti metrík v porovnaní s kalibračnou fázou, môžeme výsledky označiť za uspokojivé. Je nutné poznamenať, že sme stále pracovali s relatívne malými datasetmi, v ktorých môže byť ťažšie nájsť určité vzory (v závislosti od toho, aké vzorky sa náhodne do daného datasetu dostali). V niektorých iteráciách krížovej validácie sme tak narazili aj na veľmi priaznivé kombinácie vzoriek a dosiahli úspešnosť F1 miery nad 0.80 (t.j. lepšie hodnoty ako pri kalibrácii). Celkovo sme tak mohli pozorovať relatívne väčší rozptyl štandardnej odchýlky pri F1 miere. Z validačných výsledkov tak môžeme odvodiť dve pozorovania. Prvým pozorovaním je, že sa nám potvrdil potenciál konceptového učenia pre detekciu malvéru, keďže sme dokázali získať uspokojivé výsledky priamočiarou aplikáciou metódy aj na ďalšie datasety. Druhým pozorovaním je, že pravdepodobne je potrebné vykonať kalibráciu hyper-parametrov špecificky na konkrétny dataset. Výsledky však môžu značiť, že hyper-parametre, pomocou ktorých sa nastavuje optimálna expresivita deskripčnej logiky pre konceptové výrazy (t.j. nominály, negácie a pod.) sa dokázali preniesť z kalibračného datasetu aj na iné datasety a dodatočná

Nastavenie	trénovanie	testovanie				
	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
			dataset_1_1000.owl			
OCEL(20/✗/✗/✗/5)	0.76 ± 0.00	0.75 ± 0.01	0.73 ± 0.01	0.78 ± 0.02	0.27 ± 0.03	0.76 ± 0.01
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.71 ± 0.03	0.67 ± 0.02	0.85 ± 0.03	0.42 ± 0.03	0.75 ± 0.03
PARCEL(0/✗/✗/✓/10)	0.89 ± 0.02	0.71 ± 0.05	0.75 ± 0.03	0.64 ± 0.11	0.20 ± 0.01	0.69 ± 0.07
SPACEL(1/✗/✗/✓/5)	0.89 ± 0.02	0.67 ± 0.07	0.67 ± 0.05	0.66 ± 0.07	0.32 ± 0.05	0.69 ± 0.07
			dataset_2_1000.owl			
OCEL(20/✗/✗/✗/5)	0.75 ± 0.00	0.72 ± 0.02	0.71 ± 0.03	0.74 ± 0.02	0.29 ± 0.05	0.72 ± 0.01
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.71 ± 0.02	0.67 ± 0.02	0.84 ± 0.02	0.41 ± 0.04	0.74 ± 0.01
PARCEL(0/✗/✗/✓/10)	0.92 ± 0.02	0.75 ± 0.05	0.77 ± 0.05	0.69 ± 0.08	0.19 ± 0.04	0.73 ± 0.07
SPACEL(1/✗/✗/✓/5)	0.93 ± 0.00	0.74 ± 0.03	0.74 ± 0.03	0.75 ± 0.07	0.26 ± 0.04	0.74 ± 0.04
			dataset_3_1000.owl			
OCEL(20/✗/✗/✗/5)	0.74 ± 0.01	0.70 ± 0.06	0.72 ± 0.06	0.67 ± 0.01	0.25 ± 0.10	0.69 ± 0.09
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.68 ± 0.03	0.67 ± 0.04	0.72 ± 0.15	0.35 ± 0.12	0.69 ± 0.07
PARCEL(0/✗/✗/✓/10)	0.92 ± 0.00	0.75 ± 0.05	0.77 ± 0.04	0.70 ± 0.07	0.20 ± 0.03	0.73 ± 0.06
SPACEL(1/✗/✗/✓/5)	0.93 ± 0.00	0.75 ± 0.04	0.74 ± 0.05	0.77 ± 0.02	0.27 ± 0.06	0.76 ± 0.03
			dataset_4_1000.owl			
OCEL(20/✗/✗/✗/5)	0.73 ± 0.01	0.72 ± 0.04	0.70 ± 0.04	0.77 ± 0.04	0.32 ± 0.05	0.74 ± 0.04
CELOE(40/✗/✗/✗/5)	0.70 ± 0.01	0.70 ± 0.05	0.65 ± 0.04	0.84 ± 0.03	0.44 ± 0.06	0.73 ± 0.04
PARCEL(0/✗/✗/✓/10)	0.89 ± 0.03	0.69 ± 0.03	0.72 ± 0.02	0.61 ± 0.08	0.23 ± 0.02	0.66 ± 0.05
SPACEL(1/✗/✗/✓/5)	0.92 ± 0.00	0.71 ± 0.02	0.71 ± 0.01	0.73 ± 0.03	0.29 ± 0.01	0.72 ± 0.02
			dataset_5_1000.owl			
OCEL(20/✗/✗/✗/5)	0.76 ± 0.00	0.74 ± 0.01	0.75 ± 0.04	0.73 ± 0.06	0.24 ± 0.07	0.73 ± 0.02
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.71 ± 0.01	0.73 ± 0.06	0.72 ± 0.15	0.27 ± 0.15	0.70 ± 0.03
PARCEL(0/✗/✗/✓/10)	0.93 ± 0.00	0.74 ± 0.02	0.78 ± 0.03	0.68 ± 0.02	0.18 ± 0.03	0.73 ± 0.02
SPACEL(1/✗/✗/✓/5)	0.93 ± 0.00	0.74 ± 0.05	0.73 ± 0.05	0.76 ± 0.05	0.27 ± 0.07	0.75 ± 0.05

Tabuľka 5.10: Výsledky nakalibrovaných hyper-parametrov pre validačné datasety.

Nastavenie	trénovanie	testovanie				
	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
OCEL(20/✗/✗/✗/5)	0.75 ± 0.01	0.73 ± 0.01	0.72 ± 0.01	0.77 ± 0.04	0.27 ± 0.02	0.73 ± 0.02
CELOE(40/✗/✗/✗/5)	0.71 ± 0.00	0.70 ± 0.01	0.68 ± 0.02	0.79 ± 0.06	0.38 ± 0.06	0.72 ± 0.02
PARCEL(0/✗/✗/✓/10)	0.91 ± 0.01	0.73 ± 0.02	0.76 ± 0.02	0.66 ± 0.03	0.20 ± 0.01	0.71 ± 0.03
SPACEL(1/✗/✗/✓/5)	0.92 ± 0.01	0.72 ± 0.03	0.72 ± 0.02	0.73 ± 0.04	0.28 ± 0.02	0.72 ± 0.03

Tabuľka 5.11: Priemerné validačné výsledky pre jednotlivé nastavenia.

kalibrácia je potrebná len pre šum (čím špecifikujeme kvalitatívne kritériá na samotné koncepty v závislosti od datasetu). Z dôvodu vysokej časovej náročnosti sme kalibráciu pre validačné datasety nevykonali.

Pri samotných konceptových výrazoch, ktoré sme získali v rámci validačnej fázy, sme mohli pozorovať podobný trend ako pri kalibrácii, kde väčšina konceptov popisovala práve zbalené vzorky. Tento trend je jasne zdôvodniteľný faktom, že na rozlišovanie škodlivého kódu používame iba statické dáta. Pomocou takýchto dát sa dá relatívne jednoducho rozlišovať či je daná vzorka zbalená alebo nie (obzvlášť ak používa na tento účel komerčné a známe nástroje), avšak už nič nám nehovorí o samotnom správaní. Z toho dôvodu môžeme aj hypotetizovať, že úspešnosť daného datasetu mohla závisieť aj od počtu zbalených vzoriek, ktoré sa v ňom nachádzali (samotný dataset nám o tom však nehovorí). V rámci vedeckej komunity, venujúcej sa detekcii malvéru, existuje aj názor, že statické dáta možno použiť iba na rozlišovanie, či je daná vzorka zbalená a nie na rozlišovanie či je zároveň aj škodlivá [162].

$$\begin{aligned}&\text{ExecutableFile} \sqcap \exists \text{has\_section\_flag.Writable} \\ &\sqcap \leqslant 1\ \text{has\_action.(CreateFileMapping} \sqcup \text{CreateMutex} \\ &\sqcup \text{DirectoryHandling} \sqcup \text{EnumerateDisks} \sqcup \text{EnumerateThreads} \sqcup \text{ExecuteFile} \\ &\sqcup \text{GetProcessCurrentDirectory} \sqcup \text{KillWindow} \sqcup \text{ListenOnSocket} \sqcup \text{SendIcmpRequest} \\ &\sqcup \text{UnlockFile} \sqcup \text{UnmapFileFromProcess)}\end{aligned} \tag{5.13}$$

Napriek všeobecnému smerovaniu konceptových výrazov smerom k zbaleným vzorkách, v niektorých datasetoch sme mohli pozorovať aj iné trendy. Konceptový výraz (5.13) bol získaný z datasetu `dataset_5_1000.owl`, prostredníctvom algorimu OCEL so správnosťou 0.72 na testovacej množine. Ako môžeme vidieť, samotný výraz je v tomto prípade viac zameraný na rôzne akcie, najmä v porovnaní s predchádzajúcimi výrazmi.

5.5 Experimentálna časť 3

Cieľom v rámci tejto experimentálnej časti bolo overiť jednotlivé algoritmy konceptového učenia na väčšom datasete. Konkrétne sme použili prvý dataset o veľkosti 10 000 (t.j. `dataset_1_10000.owl`, viď. kapitola 4.5). Samotné trénovanie opätovne prebiehalo ako v predchádzajúcich experimentoch. Ako základ sme využili hyper-parametre z kalibračnej fázy.

5.5.1 Doplňujúca kalibrácia

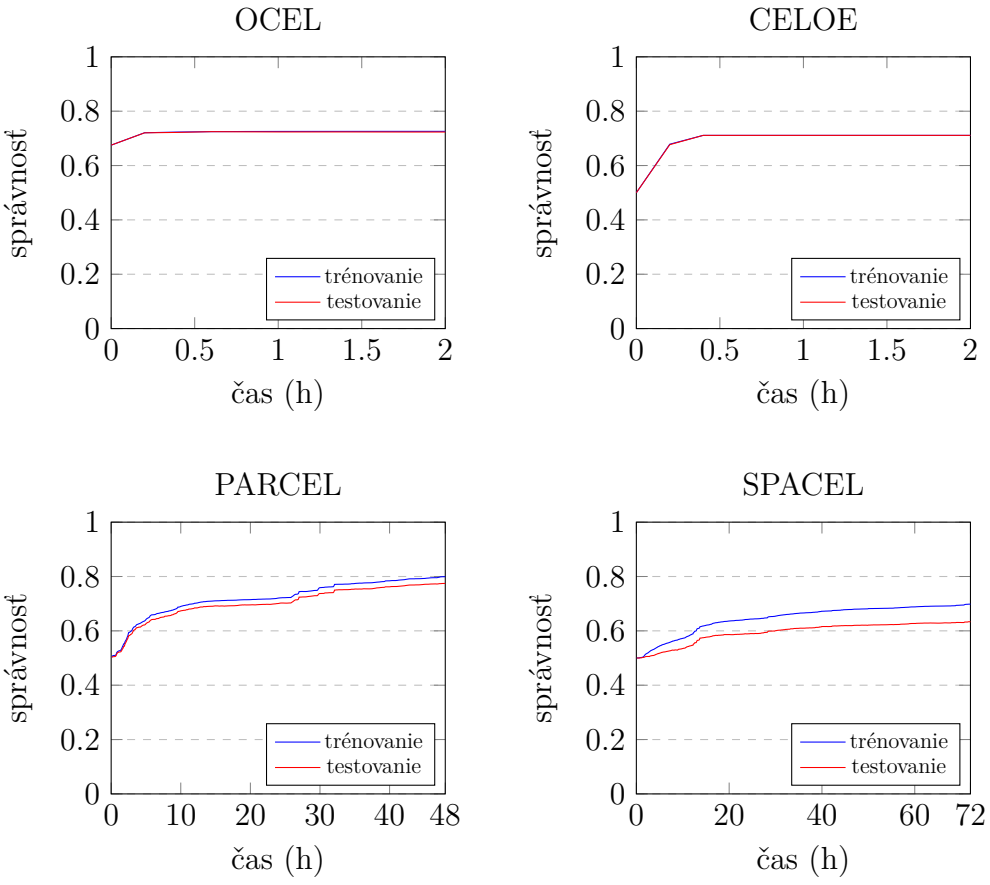
Pri prvotnom experimentovaní s väčším datasetom sme priamo vyskúšali aplikovať nakalibrované hyper-parametre, vrátane nastaveného času trénovania (ktorý sa pri menších datasetoch javil ako dostatočný). Ako môžeme vidieť v tabuľke 5.12, neparalelné algoritmy OCEL a CELOE dosiahli porovnateľnú úspešnosť čo sa týka FP miery a F1 miery ako pri validácii. V rámci experimentov sme testovali aj dodatočne upraviť šum, avšak pozorovali sme podobný efekt ako pri samotnej kalibrácii (viď. kapitola 5.3.2) a hodnoty 20 pre OCEL a 40 pre CELOE sa javili ako najoptimálnejšie aj pri väčšom množstve vzoriek. Zaujímavým pozorovaním bolo, že aj pri rozsiahlejšom datasete priebeh učenia kopíroval priebeh pri kalibrácii, kde najvýraznejšie zlepšenie riešenia nastalo v priebehu prvých 30 minút a následné zlepšenia boli už minimálne a takmer nepozorovateľné na krivke (viď. 5.8). Väčší dataset (t.j. zároveň aj väčšia ontológia) taktiež značne zvýšil časovú náročnosť overovania inštancií pri konceptovom učení. Zatiaľ čo pre dataset o veľkosti 1000 sme boli schopný otestovať približne 400k konceptových výrazov, pre algoritmus OCEL, pri datasete o veľkosti 10k to bolo približne 50k výrazov za rovnaký čas.

Paralelné algoritmy sa pri väčšom datasete správali komplikovanejšie ako v prípade OCEL a CELOE. Priamočiarym aplikovaním hyper-parametrov sme získali pre algoritmus PARCEL hodnotu F1 miery 0.26 a 0.36 pre SPACEL. Pri algoritme PARCEL sme skúšali postupne zvyšovať čas učenia, avšak nepozorovali sme takmer žiadne zlepšenie. Na základe tohto pozorovania sme usúdili, že šum o veľkosti 0 (t.j. žiadne falošne pozitívne prvky na trénovacej množine) je príliš striktné obmedzenie na väčší dataset. Zvýšením hodnoty šumu z 0 na 1 sme získali zlepšenie F1 miery z 0.26 na 0.67. Z výsledkov však bolo evidentné, že 24 hodín na jednu trénovaciu iteráciu nebolo dostatočných, nasledkom čoho sme sa rozhodli daný čas zdvojnásobiť. Ako môžeme vidieť v tabuľke 5.12, pre PARCEL sme dosiahli F1 mieru 0.78 a hodnotu FP miery 0.25 (hlavný nárast FP miery nastal zvýšením šumu, keď hodnota narástla z 0.03 na 0.23 a dodatočným trénovaním narástla až na finálnych 0.25).

Pri algoritme SPACEL sme postupovali podobným spôsobom, avšak v tomto prípade sme mali možnosť postupne iba zvyšovať trénovací čas (keďže

Nastavenie	trénovanie	testovanie				
	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`OCEL(20/✗/✗/✗/5)`	0.72 ± 0.00	0.72 ± 0.00	0.70 ± 0.00	0.76 ± 0.01	0.31 ± 0.01	0.73 ± 0.00
`CELOE(40/✗/✗/✗/5)`	0.71 ± 0.00	0.71 ± 0.00	0.66 ± 0.01	0.84 ± 0.00	0.42 ± 0.00	0.74 ± 0.00
`PARCEL(1/✗/✗/✓/10)`	0.79 ± 0.00	0.77 ± 0.00	0.75 ± 0.01	0.81 ± 0.01	0.25 ± 0.01	0.78 ± 0.00
`SPACEL(1/✗/✗/✓/5)`	0.69 ± 0.00	0.63 ± 0.00	0.79 ± 0.00	0.36 ± 0.01	0.09 ± 0.00	0.49 ± 0.01

Tabuľka 5.12: Výsledky pre dataset `dataset_1_10000.owl`.



Obrázok 5.8: Priebeh učenia pre dataset `dataset_1_10000.owl`.

šum sa pri tomto algoritme nekalibruje). Po zdvojnásobení času narástla hodnota F1 miery z 0.36 na 0.46. Zvýšením času o ďalších 24 hodín na 72 nastal nárast z 0.46 na 0.49. Ako môžeme vidieť aj na obrázku 5.8, postupne sme síce získavali lepšiu úspešnosť, avšak za cenu dlhého výpočtového času, ktorý už môžeme označiť za relatívne neúnosný z hľadiska výpočtového výkonu (je opätovne nutné poznamenať, že 72 hodín trvala jedna trénovacia iterácia). Ďalšie zvyšovanie času sme už neaplikovali.

5.5.2 Diskusia

Celkovo môžeme zhodnotiť, že na základe výsledných metrík si najlepšie s väčším datasetom poradil algoritmus PARCEL. Neparalelné algoritmy OCEL a CELOE obstáli podobne ako pri validácii s menšími datasetmi, iba s menším poklesom metrík. Suverénne najhoršie však v tomto porovnaní obstál algoritmus SPACEL, ktorý si nedokázal poradiť s väčším datasetom v rozumnom čase. Napriek tomu že sme prostredníctvom algoritmu PARCEL získali doposiaľ najlepší model v zmysle priemernej F1 miery, stále sme dosahovali relatívne vysokú mieru falošnej pozitivity. Môžeme však skonštatovať, že do istej miery bola druhá hypotéza splnená (viď. kapitola 5.1). Keďže sme pre veľký dataset nevykonali úplnú kalibráciu hyper-parametrov, v porovnaní s validačnými výsledkami, sme pomocou algoritmu PARCEL dokázali získať úspešnejší model. Pre úplné overenie tejto hypotézy by bolo vhodné natrénovať aj ďalšie datasety podobnej veľkosti. Z časového hľadiska sme však ďalšie experimenty už nevykonali.

$$\begin{gathered}\exists \text{has\_file\_feature.MultipleExecutableSections} \sqcap \\ \geqslant 2\,\text{has\_section.}(\exists \text{has\_section\_feature.HighEntropy})\end{gathered} \tag{5.14}$$

V rámci výsledných konceptových výrazov sme opäť mohli pozorovať rovnaké trendy ako v predchádzajúcich experimentoch. Konceptový výraz (5.14) zobrazuje najúspešnejšie čiastkové riešenie z algoritmu PARCEL, ktoré dosiahlo správnosť 0.571 na trénovacej množine (599 pokrytých pozitívnych príkladov) a 0.55 na testovacej množine (128 pokrytých príkladov). Opätovne môžeme vidieť trend smerujúci k zbaleným škodlivým vzorkám.

5.6 Experimentálna časť 4

Cieľom tejto experimentálnej časti bolo preveriť možnosť využitia dátových vlastností pri konceptovom učení. V našej ontológii (viď. kapitola 4.4) sa nachádza väčšie množstvo dátových parametrov, ktoré boli prevzaté z datasetu EMBER a zároveň sme ich v doterajších experimentoch nepoužívali (t.j. spresňovanie bolo explicitne vypnuté). Konkrétne sa jedná o vlastnosti, ktoré explicitne určujú napr. veľkosť entropie pre sekciu, počet importovaných API volaní alebo názov sekcie. V rámci týchto experimentov sme využili rovnaký čiastkový dataset ako v kapitole 5.3, t.j. `dataset_8_1000.owl`, kde hlavným cieľom bolo prevziať najlepšie hyper-parametre pre jednotlivé algoritmy a pokračovať v kalibrácii dátových vlastností.

Všetky tabuľky uvedené v rámci tejto kapitoly dodržujú nasledujúci formát:

`algoritmus(reťazcové typy/numerické typy/počet rozdelení)`

kde algoritmus predstavuje jeden z algoritmov OCEL, CELOE, PARCEL a SPACEL s finálnymi nakalibrovanými hyper-parametrami, uvedenými v kapitole 5.3 (z toho dôvodu už nebudú explicitne uvádzané). Parametre `reťazcové typy` a `numerické typy` sú binárne parametre a budeme ich opätovne označovať symbolmi ✓/✗.

5.6.1 Dátové hyper-parametre

V rámci tejto experimentálnej časti sme kalibrovali nasledujúce hyper-parametre.

Dátový typ `xsd:string`. Tento parameter umožňuje spresňujúcemu operátoru ρ generovať výrazy vo forme $\exists r.\{"test"\}$, kde r je rola a $"test"$ je reťazec typu `xsd:string`. V nami navrhnutej ontológii sa nachádza iba jedna vlastnosť tohto typu, ktorou je `section_name`. Táto vlastnosť sa v ontológii nachádza pre každú jednu sekciu danej vzorky a označuje jej názov. Primárnym cieľom kalibrácie tohto hyper-parametra bolo vyskúšať, či nám dokáže priniesť určité zlepšenie voči triede, ktorá obsahuje predspracované reťazcové dáta ohľadom sekcií (t.j. `NonstandardSectionName`).

Numerické dátové typy. Hyper-parameter slúži na povolenie spresňovania numerických dát (v našej ontológii `xsd:int` a `xsd:double`). Následne operátor ρ dokáže generovať výrazy vo forme $\exists r.\{\geq n\}$, $\exists r.\{\leq n\}$ alebo $\exists r.\{n\}$, kde r je rola a n je hodnota typu `xsd:integer` alebo `xsd:double`. V rámci softvéru DL-Learner nedokážeme presnejšie špecifikovať, či chceme povoliť iba celočíselné typy alebo aj dátové typy s

desatinnou čiarkou. Z toho dôvodu sa tento hyper-parameter kalibroval celkovo ako numerický dátový typ. Priamo s týmto hyper-parametrom súvisí aj ďalší nastaviteľný parameter: počet rozdelení. Ako bolo uvedené v kapitole 3.2, tento parameter určuje, maximálne koľko spresnení môže operátor vykonať pre konkrétnu rolu s dátovým typom tak, aby prehľadal celý priestor výrazov. Tento parameter je štandardne nastavený na hodnotu 12. Cieľom kalibrácie bolo opätovne preveriť, či dokážeme získať presnejšie výrazy voči triedam v ontológii, ktoré predspracúvajú numerické dátové vlastnosti.

5.6.2 Dátový typ `xsd:string`

Hyper-parameter, ktorý povoľuje spresňovanie dátového typu `xsd:string` sme kalibrovali takým spôsobom, že pre jednotlivé algoritmy sme prevzali najlepšie parametre z kalibračnej fázy (viď. kapitola 5.3) a daný parameter sme zapli.

Výsledky pre neparalelné a paralelné algoritmy môžeme vidieť v tabuľke 5.13. Výsledky je možné porovnať s finálnymi kalibračnými výsledkami v tabuľke 5.9. Ako môžeme vidieť, pre neparalelné algoritmy OCEL a CELOE sme nezískali žiadne zlepšenie a samotné metriky zostali rovnaké. V rámci týchto algoritmov sme nepozorovali výskyt reťazcov v postupne sa zlepšujúcich riešeniach (v samotnom prehľadávacom strome sa však nachádzali). Tento fakt je pravdepodobne spôsobený zložením vzoriek v datasete. Aby sa samotný reťazec dostal do úspešného konceptového výrazu, musel by popisovať značnú časť vzoriek (keďže hľadáme jeden výraz). V prípade, ak by bol dataset zložený iba zo vzoriek zbalených pomocou UPX, pravdepodobne by sa v konečnom výraze nachádzal reťazec `.upx` (tento názov používa UPX pre zbalené sekcie). Je taktiež nutné poznamenať, že v konceptových výrazoch pre neparalelné algoritmy sa nenachádzala ani trieda, ktorá predspracúva názvy sekcií (t.j. `NonstandardSectionName`).

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`OCEL(✓/✗/-)`	0.76 ± 0.00	0.75 ± 0.02	0.76 ± 0.04	0.72 ± 0.02	0.22 ± 0.05	0.74 ± 0.02
`CELOE(✓/✗/-)`	0.71 ± 0.00	0.70 ± 0.02	0.71 ± 0.05	0.68 ± 0.05	0.27 ± 0.01	0.69 ± 0.03
`PARCEL(✓/✗/-)`	0.94 ± 0.00	0.79 ± 0.02	0.82 ± 0.02	0.74 ± 0.05	0.16 ± 0.03	0.78 ± 0.02
`SPACEL(✓/✗/-)`	0.93 ± 0.00	0.75 ± 0.02	0.74 ± 0.02	0.76 ± 0.04	0.25 ± 0.03	0.75 ± 0.02

Tabuľka 5.13: Výsledky kalibrácie dátového typu `xsd:string`.

V prípade paralelných algoritmov PARCEL a SPACEL určité zmeny v metrikách nastali, aj keď minimálne. Čo sa týka algoritmu PARCEL, získali sme nárast F1 miery z 0.77 na 0.78 a zároveň aj nárast FP miery z

0.15 na 0.16. V prípade druhého paralelného algoritmu SPACEL nastal pokles F1 miery z 0.76 na 0.75, spolu s nárastom FP miery z 0.24 na 0.25. V porovnaní s neparalelnými algoritmami, v čiastkových riešeniach sme už mohli pozorovať výskyt reťazcových dátových typov (keďže v tomto prípade stačí pokryť značne menej príkladov nachádzajúcich sa v datasete). Príklady konceptových výrazov môžeme vidieť v (5.15) a (5.16). Výraz (5.15) patrí do množiny čiastkových riešení z algoritmu PARCEL. Ako môžeme vidieť, popisuje vzorky s konkrétnym menom sekcie `.upx0`. Konceptový výraz (5.16) bol naopak získaný z algoritmu SPACEL. V tomto výraze môžeme vidieť opätovne konkrétne názvy sekcií, avšak už v negovanej časti výrazu (t.j. popisujúcu negatívne, resp. legitímne vzorky). Vidíme, že vo výraze sa nachádzajú názvy sekcií `pdata` a `data`, ktoré patria medzi štandardné názvy generované kompilátormi.

$$\exists \text{has\_file\_feature.RegistryStrings} \sqcap \text{has\_section.}(\exists \text{section\_name.}\{\text{".}upx0\text{"}\}) \tag{5.15}$$

$$\begin{aligned} &\exists \text{has\_file\_feature.MultipleExecutableSections} \sqcap \\ &\geqslant 2\,\text{has\_section.}(\exists \text{has\_section\_feature.HighEntropy}) \sqcap \\ &\neg(\exists \text{has\_action.ConnectToSocket} \sqcap \exists \text{has\_section.}(\exists \text{section\_name.}\{\text{"}pdata\text{"}\})) \sqcap \\ &\neg(\exists \text{has\_file\_feature.MultipleExecutableSections} \sqcap \exists \text{has\_section.}(\exists \text{section\_name.}\{\text{"}data\text{"}\})) \end{aligned} \tag{5.16}$$

Napriek tomu, že v prípade algoritmu PARCEL sme získali menšie zlepšenie, čo sa týka F1 miery (za cenu nárastu FP miery), z celkového hľadiska sa reťazcový dátový typ ukázal ako neužitočný. Z toho dôvodu sme ponechali tento hyper-parameter vypnutý.

5.6.3 Numerické dátové typy

Druhý hyper-parameter, ktorý sme testovali v rámci tejto experimentálnej časti, bolo povolenie numerických dátových typov spolu s počtom rozdelení. Začínali sme s implicitnou hodnotou parametra 12 a postupne zvyšovali na 24, 48 až na konečných 96. Je nutné poznamenať, že povolenie tohto parametra, spolu s počtom rozdelení priestoru, ovplyvňuje všetky numerické dátové typy, ktoré sa nachádzajú v ontológii.

Výsledky pre neparalelné algoritmy môžeme vidieť v tabuľke 5.14. Zo samotných výsledkov vidíme, že numerické dáta majú výraznejší vplyv na generované koncepty ako pri dátovom type `xsd:string`. Pre OCEL sme mohli pozorovať pokles F1 miery spolu s poklesom FP miery. Postupným zvyšovaním počtu rozdelení zároveň zvyšujeme aj prehľadávací priestor pre algoritmus. Z toho dôvodu sa pravdepodobne spomalilo celkové učenie (t.j. postupný pokles F1 miery pri vyššom počte rozdelení priestoru). Pri algoritme CELOE sme naopak dosahovali podobnú F1 mieru, avšak s vyššou falošnou pozitivitou. V

tomto prípade sa našli krátke koncepty, využívajúce numerické dáta, ktoré síce popisujú dostatočný počet pozitívnych prvkov, avšak nezovšeobecňujú dostatočne dobre na testovacej množine (vyššia FP miera). Príklad konceptového výrazu z algoritmu OCEL môžeme vidieť v (5.17). V samotnom výraze vidíme využitie dvoch numerických dátových typov. Jednotlivé časti môžeme interpretovať nasledovne: *vzorka je škodlivá, ak obsahuje sekciu, ktorá má hodnotu entropie vyššiu ako 5.4483*, resp. *vzorka je škodlivá, ak obsahuje 11 a menej reťazcov, ktoré reprezentujú URL adresy.* Druhý konceptový výraz, získaný z algoritmu CELOE (viď. (5.18)) môžeme interpretovať nasledovne: *vzorka je škodlivá, ak sa jedná o spustiteľný súbor, ktorý neobsahuje žiadne reťazce reprezentujúce cesty na súborovom systéme.*

$$\begin{aligned}\text{ExecutableFile} \sqcap \exists \text{has\_section}.\exists \text{has\_section\_flag}.\text{Writable} \sqcap \\ \exists \text{has\_section}.\exists \text{section\_entropy}.\{\geq 5.4483\} \sqcap \leqslant 1\, \text{has\_action}.(\text{AcceptSocketConnection} \sqcup \text{CreateSocket} \\ \sqcup \text{DirectoryHandling} \sqcup \text{SetThreadContext} \sqcup \text{StopService}) \sqcap \exists \text{url\_strings\_count}.\{\leq 11\}\end{aligned} \tag{5.17}$$

$$\text{ExecutableFile} \sqcap \exists \text{path\_strings\_count}.\{\leq 0\} \tag{5.18}$$

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`OCEL(✗/✓/12)`	0.75 ± 0.01	0.73 ± 0.01	0.72 ± 0.04	0.65 ± 0.07	0.19 ± 0.05	0.70 ± 0.03
`OCEL(✗/✓/24)`	0.75 ± 0.02	0.73 ± 0.01	0.77 ± 0.03	0.66 ± 0.06	0.20 ± 0.05	0.71 ± 0.03
`OCEL(✗/✓/48)`	0.76 ± 0.01	0.72 ± 0.02	0.79 ± 0.02	0.62 ± 0.08	0.16 ± 0.03	0.69 ± 0.04
`OCEL(✗/✓/96)`	0.75 ± 0.01	0.72 ± 0.01	0.79 ± 0.02	0.61 ± 0.08	0.16 ± 0.04	0.69 ± 0.04
`CELOE(✗/✓/12)`	0.71 ± 0.00	0.69 ± 0.00	0.69 ± 0.03	0.70 ± 0.01	0.32 ± 0.01	0.69 ± 0.03
`CELOE(✗/✓/12)`	0.71 ± 0.00	0.69 ± 0.00	0.69 ± 0.03	0.70 ± 0.01	0.32 ± 0.01	0.69 ± 0.03
`CELOE(✗/✓/12)`	0.71 ± 0.00	0.69 ± 0.00	0.69 ± 0.03	0.70 ± 0.01	0.32 ± 0.01	0.69 ± 0.03
`CELOE(✗/✓/24)`	0.71 ± 0.00	0.69 ± 0.00	0.68 ± 0.03	0.71 ± 0.12	0.33 ± 0.12	0.69 ± 0.03

Tabuľka 5.14: Výsledky kalibrácie numerických hyper-parametrov pre neparalelné algoritmy OCEL a CELOE.

$$\exists \text{has\_action}.\text{ReadFromFile} \sqcap \exists \text{has\_file\_feature}.\text{MultipleExecutableSections} \sqcap \exists \text{imports\_count}.\{\leq 106\} \tag{5.19}$$

$$\exists \text{has\_action}.\text{CreateWindow} \sqcap \exists \text{imports\_count}.\{\leq 54\} \tag{5.20}$$

Výsledky kalibrácie numerických dátových typov pre paralelné algoritmy môžeme vidieť v tabuľke 5.15. Z výsledkov môžeme pozorovať rovnaký trend ako pri algoritme CELOE, kde hodnota F1 miery klesala v priebehu kalibrácie iba minimálne, zatiaľ čo miera falošnej pozitivity narástla. Taktiež si môžeme všimnúť, že hodnota správnosti pri trénovaní narástla v porovnaní s pôvodnými

výsledkami kalibrácie. Tento fakt, spolu s horšími výsledkami na testovacej množine nám pravdepodobne ukazuje, že nastalo mierne pretrénovanie, kde čiastkové riešenia až príliš presne popisujú dáta v trénovacej množine (taktiež narástol aj počet výsledných čiastkových riešení). Ukážky čiastkových riešení získaných prostredníctvom algoritmu PARCEL môžeme vidieť v (5.19) a (5.20).

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`PARCEL(✗/✓/12)`	0.96 ± 0.00	0.77 ± 0.02	0.78 ± 0.03	0.76 ± 0.04	0.21 ± 0.04	0.77 ± 0.02
`PARCEL(✗/✓/24)`	0.98 ± 0.01	0.76 ± 0.02	0.76 ± 0.03	0.77 ± 0.04	0.24 ± 0.04	0.76 ± 0.02
`PARCEL(✗/✓/48)`	0.98 ± 0.01	0.77 ± 0.04	0.77 ± 0.04	0.76 ± 0.05	0.22 ± 0.06	0.76 ± 0.04
`PARCEL(✗/✓/96)`	0.97 ± 0.01	0.76 ± 0.03	0.75 ± 0.04	0.77 ± 0.03	0.24 ± 0.06	0.76 ± 0.03
`SPACEL(✗/✓/12)`	0.95 ± 0.00	0.74 ± 0.03	0.71 ± 0.03	0.79 ± 0.03	0.31 ± 0.05	0.75 ± 0.02
`SPACEL(✗/✓/12)`	0.96 ± 0.00	0.73 ± 0.04	0.70 ± 0.04	0.80 ± 0.04	0.33 ± 0.07	0.75 ± 0.03
`SPACEL(✗/✓/12)`	0.95 ± 0.00	0.75 ± 0.03	0.72 ± 0.03	0.82 ± 0.03	0.34 ± 0.05	0.77 ± 0.02
`SPACEL(✗/✓/24)`	0.95 ± 0.00	0.74 ± 0.03	0.71 ± 0.04	0.82 ± 0.04	0.33 ± 0.06	0.76 ± 0.02

Tabuľka 5.15: Výsledky kalibrácie numerických hyper-parametrov pre paralelné algoritmy PARCEL a SPACEL.

5.6.4 Diskusia

Cieľom tejto experimentálnej časti bolo preveriť vhodnosť dátových vlastností pri konceptovom učení. Z vyššie uvedených výsledkov sme dospeli k záveru, že použitie týchto vlastností nie je ideálne pre skúmané algoritmy, resp. by mohli byť vhodnejšie pri iných podmienkach. Použitie dátového typu `xsd:string`, by mohli algoritmy využiť efektívnejšie, ak by sme použili značne robustnejší dataset, v ktorom by sa mohlo nájsť viac vzorov ohľadom sekcií (napr. ak by sa v datasete vyskytovalo dostatočne veľké množstvo vzoriek zbalených pomocou nástroja UPX). Využitie numerických vlastností pri daných algoritmoch v súčasnej podobe sa taktiež javí ako problematické. V kapitole 3.2 sme uvádzali, akým spôsobom funguje rozdeľovanie priestoru numerických hodnôt pre operátor ρ. Je nutné poznamenať, že tento spôsob je relatívne naivný, keďže neberie do úvahy početnosť jednotlivých hodnôt v ontológii, resp. nepoužíva žiadne iné štatistické metódy. Z hľadiska využitia dátových vlastností sa javia ako efektívnejšie algoritmy konceptového učenia uvedené v kapitole 3.5.

5.7 Experimentálna časť 5

V rámci tejto experimentálnej časti sme sa venovali možnosti zefektívnenia konceptového učenia, zameraním sa na jednotlivé rodiny škodlivého kódu. Výsledky v predchádzajúcich kapitolách ukázali, že konceptové učenie výrazne zaostáva za štandardnými algoritmami strojového učenia. V rámci týchto experimentov sme dospeli k záveru, že pre konceptové učenie je značne náročné naučiť sa rozpoznávať škodlivý kód vo všeobecnosti, keďže sa jedná o rozsiahly prehľadávací priestor a pre lepšie výsledky by bolo vhodné určitým spôsobom tento priestor zúžiť. Z toho dôvodu sme sa rozhodli natrénovať modely pre samostatné rodiny malvéru (informácia, ku ktorej rodine patrí konkrétna vzorka, sa nachádza v datasete EMBER), keďže práve množina škodlivých vzoriek z určitej rodiny by mala obsahovať viacero spoločných znakov (t.j. menší prehľadávací priestor) v porovnaní so všetkými vzorkami malvéru v datasete. Hlavným cieľom bolo overiť hypotézu, že natrénovaním klasifikátorov pre jednotlivé rodiny a ich následnou kombináciou vieme získať lepšie výsledky, ako keby sme trénovali jeden klasifikátor s viacerými rodinami.

5.7.1 Rodiny malvéru

V datasete EMBER sa celkovo nachádza 400k škodlivých vzoriek, z 3227 rôznych rodín. Je však nutné poznamenať, že viac ako polovica rodín obsahuje iba jednu vzorku, pričom 5000 a viac vzoriek obsahuje iba približne 20 rodín. Pre naše účely sme sa rozhodli vybrať 5 rôznych rodín, ktoré obsahovali aspoň 5000 vzoriek. Rodiny, ktoré sme vybrali sú:

- *WannaCry*: jedná sa o jeden z najznámejších ransomvérov [12].
- *Zusy*: trójsky kôň používaný prevažne pre účely kradnutia informácií ohľadom internet bankingu obeti [163].
- *Sivis*: trójsky kôň, ktorý sa dokáže pripájať k existujúcim súborom s cieľom poskytnúť útočníkom zadné vrátka [164].
- *Fareit*: malvér, ktorého cieľom je krádež citlivých údajov obete v podobe prístupových údajov k emailovým kontám [165].
- *Vtflooder*: trójsky kôň, ktorého cieľom je, okrem poskytnutia zadných vrátok, aj krádež citlivých údajov obeti [166].

Samotné datasety, spolu s ich vlastnosťami, ktoré sme vytvorili špecificky kvôli tejto experimentálnej časti, môžeme vidieť v tabuľke 5.16. Po prvotných experimentoch s jednotlivými rodinami sme zistili, že konceptové učenie si s nimi dokáže poradiť značne efektívnejšie ako s datasetmi, kde sa nachádzalo

viacero rozličných rodín. Z toho dôvodu sme dokázali zvýšiť veľkosť datasetu pre jednu rodinu až na 20 000 vzoriek. Z tabuľky je možné taktiež vidieť, že sme zvolili iný pomer medzi pozitívnymi a negatívnymi príkladmi. Dôvodom iného pomeru bol fakt, že pri kombinácii klasifikátorov (samotná kombinácia klasifikátorov funguje ako disjunkcia konceptových výrazov zo všetkých rodín), sme dosahovali až príliš vysokú mieru falošnej pozitivity. Tento efekt sme dokázali zmierniť vyšším počtom negatívnych príkladov, voči ktorým sa učili pozitívne príklady. Algoritmy konceptového učenia tak dokázali presnejšie popísať malvér voči legitímnym súborom aj v rámci iných rodín. Taktiež je nutné poznamenať, že pre všetky rodiny sme použili totožnú množinu negatívnych príkladov. Dôvodom bol fakt, že ak sme mali pri každej rodine náhodnú množinu 15 000 legitímnych vzoriek, po kombinácii klasifikátorov sme získavali relatívne vysoký počet falošne negatívnych prvkov, ktoré značne znižovali konečnú hodnotu F1 miery.

Okrem datasetov pre konkrétne rodiny sme vytvorili aj dataset s názvom `baseline_1_20000.owl`, kde sme medzi pozitívne príklady zahrnuli náhodné vzorky z vyššie spomínaných piatich rodín. Primárnym účelom datasetu bolo porovnať výsledky s kombinovaným klasifikátorom a overiť tak našu hypotézu.

Názov	**Rodina**	**Početnosť EMBER**	**Pozitívne príklady**	**Negatívne príklady**
`wannacry_1_20000.owl`	*WannaCry*	5885	5000	15 000
`zusy_1_20000.owl`	*Zusy*	18 766	5000	15 000
`sivis_1_20000.owl`	*Sivis*	8598	5000	15 000
`fareit_1_20000.owl`	*Fareit*	14 382	5000	15 000
`vtflooder_1_20000.owl`	*Vtflooder*	16 164	5000	15 000
`baseline_1_20000.owl`	-	63 795	5000	15 000

Tabuľka 5.16: Vlastnosti novovytvorených datasetov pre trénovanie konkrétnych rodín.

5.7.2 Výsledky rodín

Výsledky trénovania pre jednotlivé rodiny môžeme vidieť v tabuľke 5.17. Trénovanie prebiehalo rovnakým spôsobom ako pri predchádzajúcich experimentoch (krížová validácia, 80% vzoriek použitých na trénovanie a 20% na testovanie a rovnaký čas trénovania). V tabuľke pri jednotlivých algoritmoch uvádzame v zátvorke hodnotu šumu (ostatné hyper-parametre zostali rovnaké ako v predchádzajúcich experimentoch), ktorá bola použitá pri trénovaní. V rámci týchto experimentov sa nám potvrdila hypotéza, že hyper-parametre, ktoré definujú cieľovú logiku generovaných konceptových výrazov sa dokázali preniesť aj na iné datasety, ktoré používajú rovnakú ontológiu. Naopak bolo

nutné dokalibrovať hodnotu šumu pre jednotlivé datasety. Z dôvodu vysokej časovej náročnosti sme vykonali iba zrýchlenú kalibráciu hyper-parametra, ktorá sa však ukázala ako dostatočná (t.j. pre samotnú kalibráciu sme nevykonali plnú krížovú validáciu a taktiež sme trénovali model iba kratší čas).

	trénovanie	testovanie				
Nastavenie	Správnosť	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
			WannaCry			
`OCEL(10)`	0.94 ± 0.00	0.94 ± 0.00	0.99 ± 0.00	0.76 ± 0.01	0.00 ± 0.00	0.86 ± 0.00
`CELOE(20)`	0.83 ± 0.00	0.83 ± 0.00	0.62 ± 0.02	0.83 ± 0.07	0.17 ± 0.03	0.71 ± 0.00
`PARCEL(1)`	0.97 ± 0.00	0.97 ± 0.00	0.90 ± 0.01	0.98 ± 0.00	0.03 ± 0.00	0.94 ± 0.00
`SPACEL(1)`	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.95 ± 0.01	0.00 ± 0.00	0.96 ± 0.00
			Zusy			
`OCEL(10)`	0.95 ± 0.00	0.95 ± 0.00	0.99 ± 0.00	0.80 ± 0.01	0.00 ± 0.00	0.89 ± 0.00
`CELOE(20)`	0.94 ± 0.00	0.94 ± 0.00	0.98 ± 0.00	0.80 ± 0.01	0.00 ± 0.00	0.88 ± 0.00
`PARCEL(1)`	0.95 ± 0.00	0.95 ± 0.00	0.91 ± 0.01	0.88 ± 0.01	0.02 ± 0.00	0.89 ± 0.00
`SPACEL(1)`	0.96 ± 0.00	0.95 ± 0.00	0.94 ± 0.01	0.86 ± 0.02	0.01 ± 0.00	0.91 ± 0.04
			Sivis			
`OCEL(10)`	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.97 ± 0.00	0.00 ± 0.00	0.98 ± 0.00
`CELOE(20)`	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.97 ± 0.00	0.00 ± 0.00	0.98 ± 0.00
`PARCEL(1)`	0.99 ± 0.00	0.95 ± 0.00	0.91 ± 0.01	0.88 ± 0.01	0.02 ± 0.00	0.89 ± 0.00
`SPACEL(1)`	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.00 ± 0.00	0.99 ± 0.00
			Fareit			
`OCEL(10)`	0.95 ± 0.00	0.95 ± 0.00	0.99 ± 0.00	0.81 ± 0.01	0.00 ± 0.00	0.89 ± 0.00
`CELOE(20)`	0.94 ± 0.00	0.94 ± 0.00	0.96 ± 0.00	0.81 ± 0.01	0.01 ± 0.00	0.88 ± 0.00
`PARCEL(5)`	0.92 ± 0.00	0.92 ± 0.00	0.85 ± 0.01	0.82 ± 0.00	0.04 ± 0.00	0.83 ± 0.00
`SPACEL(1)`	0.74 ± 0.00	0.74 ± 0.00	0.40 ± 0.06	0.01 ± 0.00	0.00 ± 0.00	0.02 ± 0.00
			Vtflooder			
`OCEL(20)`	0.97 ± 0.00	0.97 ± 0.00	0.99 ± 0.00	0.90 ± 0.00	0.00 ± 0.00	0.94 ± 0.00
`CELOE(40)`	0.98 ± 0.00	0.98 ± 0.00	0.95 ± 0.00	0.94 ± 0.00	0.00 ± 0.00	0.96 ± 0.00
`PARCEL(1)`	0.98 ± 0.00	0.98 ± 0.00	0.95 ± 0.00	0.99 ± 0.00	0.01 ± 0.00	0.97 ± 0.00
`SPACEL(1)`	0.98 ± 0.00	0.98 ± 0.00	0.99 ± 0.00	0.93 ± 0.03	0.00 ± 0.00	0.96 ± 0.01

Tabuľka 5.17: Výsledky pre jednotlivé rodiny malvéru.

Ako môžeme vidieť v tabuľke, pri obmedzení datasetov na jednotlivé rodiny malvéru, sme dokázali získať neporovnateľne lepšie výsledky (aj na robustnejších datasetoch), ktoré sa dokážu porovnávať so štandardným strojovým učením. Slabšie výsledky sme dosiahli iba v dvoch prípadoch. Prvým bola rodina *WannaCry* a algoritmus CELOE, kde sme dosiahli F1 mieru 0.71 a mieru falošnej pozitivity 0.17 (výsledok je však porovnateľný s predchádzajúcimi experimentami). Druhým prípadom bola rodina *Fareit* a algoritmus SPACEL, kde sa dosiahla hodnota F1 miery iba 0.02. V tomto prípade sme mohli predpokladať, že sa algoritmus zastavil v nejakej časti prehľadávacieho stromu, našiel niekoľko čiastkových riešení (popisujúcich iba pár

pozitívnych príkladov), ktoré sa mu už nepodarilo ďalej spresniť. Pre tento prípad nepomohol ani dlhší čas trénovania, resp. ďalšie experimentovanie s hyper-parametrami (hodnota šumu sa pre SPACEL nekalibruje). V ostatných prípadoch sme však dosiahli relatívne presné výsledky s veľmi malou FP mierou. Ako najkonzistentnejší môžeme označiť algoritmus OCEL. Algoritmus PARCEL, aj keď dosahoval pravidelne vysokú F1 mieru, vykazoval aj známky zvýšenej FP miery (aj keď neporovnateľne menšej ako pri predchádzajúcich experimentoch), ktorá sa mohla negatívne prejaviť po kombinácii klasifikátorov.

5.7.3 Kombinácia klasifikátorov

Výsledky kombinácie jednotlivých klasifikátorov rodín môžeme vidieť v tabuľke 5.18. Konceptové výrazy zo všetkých rodín (pre konkrétny algoritmus), boli spojené disjunkciou a aplikované na kombinovanú testovaciu množinu (kde boli zahrnuté všetky rodiny, synchronizované v rámci konkrétnej trénovacej iterácie pri krížovej validácii). Z výsledkov môžeme vidieť, že najlepšiu kombinovanú úspešnosť dosiahol algoritmus OCEL s F1 mierou 0.91 a FP mierou 0.00 (presnejšie sa jedná o 0.0024). Tento výsledok môžeme aj z hľadiska interpretovateľnosti a čitateľnosti označiť za najúspešnejší, keďže výsledkom je disjunkcia 5 konceptových výrazov (jeden konceptový výraz na rodinu), najmä v porovnaní s paralelnými algoritmami, kde počet konceptových výrazov bol priemerne 75 pre PARCEL a 102 pre SPACEL. Algoritmus CELOE dosiahol značne vyššiu mieru falošnej pozitivity 0.18 (ktorá bola spôsobená najmä rodinou *WannaCry*). V opačnom prípade by sa mohlo jednať o úspešnejší výsledok, keďže výsledkom je taktiež 5 konceptových výrazov, ktoré sú kratšie ako pri OCEL. PARCEL naopak dosiahol najvyššiu hodnotu F1 miery medzi kombinovanými klasifikátormi, avšak podľa očakávania sme dosiahli relatívnu vysokú FP mieru (jedná sa takmer o súčet FP miery jednotlivých rodín). Algoritmus SPACEL naopak utrpel rodinou *Fareit*, ktorú sme nedokázali natrénovať. V opačnom prípade sa mohlo jednať o najúspešnejší klasifikátor z hľadiska sledovaných metrík.

Algoritmus	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`OCEL`	0.90 ± 0.00	0.99 ± 0.00	0.85 ± 0.00	0.00 ± 0.00	0.91 ± 0.00
`CELOE`	0.85 ± 0.00	0.88 ± 0.02	0.88 ± 0.01	0.18 ± 0.03	0.88 ± 0.00
`PARCEL`	0.92 ± 0.00	0.93 ± 0.00	0.93 ± 0.00	0.11 ± 0.00	0.93 ± 0.00
`SPACEL`	0.83 ± 0.00	0.97 ± 0.00	0.74 ± 0.02	0.03 ± 0.00	0.84 ± 0.01

Tabuľka 5.18: Výsledky pre kombinovaný klasifikátor.

V tabuľke 5.19 môžeme naopak vidieť výsledky pre porovnávací data-

set `baseline_1_20000.owl`. Ako môžeme vidieť, výsledky sú značne horšie ako pri kombinovanom klasifikátore, čím sa potvrdila naša hypotéza. Pre neparalelné algoritmy OCEL a CELOE, ktoré dosiahli medzi sledovanými algoritmami najnižšiu F1 mieru, sa zdá byť takmer nemožné, aby dokázali vygenerovať koncepty podobné kombinovanému klasifikátoru. Pri paralelných algoritmoch (ktoré naopak implicitne hľadajú disjunkcie) sa javí, že teoretická možnosť existuje, keďže dosiahli relatívne dobré výsledky, avšak za cenu neúmerne vysokého výpočtového výkonu. Pri trénovaní sme mohli pozorovať podobné trendy ako pri kalibračných experimentoch, kde sme dosiahli vysokú hodnotu správnosti približne v polovici trénovacieho času a následné zlepšovanie bolo veľmi pomalé.

Jednotlivé datasety s konkrétnymi rodinami a výsledné konceptové výrazy sú dostupné v repozitári[3].

Algoritmus	Správnosť	Presnosť	Senzitivita	FP miera	F1 miera
`OCEL`	0.84 ± 0.00	0.90 ± 0.09	0.45 ± 0.10	0.02 ± 0.02	0.59 ± 0.06
`CELOE`	0.75 ± 0.00	0.51 ± 0.00	0.61 ± 0.00	0.19 ± 0.00	0.55 ± 0.00
`PARCEL`	0.88 ± 0.00	0.81 ± 0.02	0.68 ± 0.03	0.05 ± 0.00	0.74 ± 0.01
`SPACEL`	0.90 ± 0.00	0.94 ± 0.00	0.67 ± 0.01	0.01 ± 0.00	0.78 ± 0.00

Tabuľka 5.19: Výsledky pre dataset `baseline_1_20000.owl`.

5.7.4 Diskusia

V rámci tejto experimentálnej časti sa nám potvrdila hypotéza, že ak si zoberieme istú množinu rodín a vytvoríme pre každú rodinu samostatný klasifikátor, ktorý spojíme prostredníctvom disjunkcie, tak vieme natrénovať efektívnejší model, ako keby sme sa snažili vytvoriť klasifikátor daných rodín v jednom samostatnom modeli. Ak by sme však chceli vytvoriť klasifikátor, vhodný do reálnej produkcie, bolo by potrebné natrénovať značne väčšie množstvo rodín. V takom prípade by sme však dosahovali limitácie datasetu EMBER, keďže viac ako polovica dostupných rodín obsahuje iba jednu vzorku. Ako možné riešenie sa javí použiť zhlukovanie na menšie rodiny a vytvoriť tak väčšie čiastkové datasety. Môžeme hypotetizovať, že výsledky pre takéto čiastkové datasety by mohli byť veľmi podobné výsledkom z predchádzajúcich experimentov (keďže dané datasety obsahovali taktiež vzorky z rozličných rodín). Takéto čiastočne slabšie výsledky by sa následne mohli prejaviť v menšej miere pri kombinácii klasifikátorov, ak by sme trénovali značne viac ako 5 rodín.

[3]https://github.com/orbis-security/malware-families

5.8 Experimentálna časť 6

V rámci tejto experimentálnej časti, sme sa venovali bezpečnostným aspektom klasifikátorov škodlivého kódu, založených na báze konceptových výrazov. Bezpečnosti klasifikátorov, vo všeobecnosti, sme sa venovali v kapitole 1.8. Ako sme uvádzali v danej kapitole, cieľom útočníkov je pozmeniť škodlivú vzorku takým spôsobom, aby dokázala pomýliť klasifikátor (t.j., aby klasifikátor označil vzorku za legitímny softvér) a zároveň zabezpečiť, aby bola zachovaná pôvodná škodlivá funkcionalita. Aplikácia existujúcich riešení na generovanie AE sa ukázala ako problematická pre náš prípad. *White-box* útoky bolo nemožné aplikovať zo samotnej podstaty útokov, keďže sú použiteľné iba pre klasifikátory založené na tradičnom strojovom učení (t.j. pri neurónových sieťach treba poznať samotnú architektúru, hodnoty váh a pod.). *Black-box* útoky sa napokon ukázali taktiež ako problematické, keďže väčšina riešení (ktoré majú taktiež dostupný zdrojový kód) pracuje priamo nad spustiteľnými PE súbormi, resp. používa nejakú alternatívnu formu ich reprezentácie (PE súbory nemáme k dispozícii v rámci EMBER datasetu). Z toho dôvodu sme sa rozhodli navrhnúť vlastnú sadu teoreticky možných útokov a sledovať ich úspešnosť. Samotné útoky sme overovali na dvoch klasifikátoroch vytvorených v rámci tejto práce. Prvým bol klasifikátor natrénovaný na datasete `dataset_8_1000.owl` (viď. kapitola 5.3) a druhým bol kombinovaný klasifikátor založený na jednotlivých rodinách (viď. kapitola 5.8).

Testované útoky prebiehali v dvoch scenároch:

- ***Black-box***: pri tomto scenári útočník nepozná konceptové výrazy, ktoré sa používajú na klasifikáciu škodlivého kódu. V tomto scenári však predpokladáme, že útočník vie, že klasifikácia je založená na statických dátach (keďže sme do istej miery limitovaní samotným datasetom). Z toho hľadiska je možné polemizovať, že sa jedná o tzv. *grey-box* útok.
- ***White-box***: pri tomto scenári predpokladáme, že útočník pozná konceptové výrazy, používané na klasifikáciu. V tomto scenári sme sa zamerali skôr na teoretické možnosti, ktoré má útočník, ak chce spôsobiť nesprávnu klasifikáciu.

Samotnú úspešnosť útokov sme vyhodnocovali pomocou vzorca (5.21), kde Adversarial Examples (AE) je počet chybne označených škodlivých súborov a TP je počet správne klasifikovaných škodlivých súborov, ktoré klasifikátor korektne označil pred útokom.

$$\text{úspešnosť} = \frac{\text{AE}}{\text{TP}} \tag{5.21}$$

5.8.1 Popis *Black-box* útokov

Hlavnou myšlienkou našich útokov bolo navrhnúť také modifikácie PE súborov, ktoré dokáže potenciálny útočník reálne vytvoriť a zároveň nezmenia samotnú funkcionalitu.

V rámci tejto časti sme navrhli nasledovnú množinu útokov:

Ladiace symboly. V rámci našej ontológie máme triedu, ktorá reprezentuje, či daná vzorka obsahuje alebo neobsahuje ladiace symboly (trieda `Debug`). V tomto scenári predpokladáme, že útočník má k dispozícii zdrojový kód a dokáže skompilovať škodlivú vzorku podľa potreby. V takom prípade dokáže ovplyvniť, či výsledný spustiteľný súbor bude alebo nebude obsahovať tieto symboly. Tento útok sme simulovali jednoduchým odstránením/pridaním tejto vlastnosti pre jednotlivé vzorky, ktoré túto vlastnosť obsahovali/neobsahovali.

Podpis. Okrem ladiacich symbolov sa v našej ontológii nachádza aj ďalšia binárna vlastnosť: prítomnosť digitálneho podpisu v PE súbore (trieda `Signature`). Spustiteľný kód, ktorý je digitálne podpísaný môže byť dobrým znakom toho, že sa jedná o legitímny softvér, avšak nemusí to byť pravda. Teoreticky dokáže útočník ukradnúť legitímny certifikát, prípadne narušiť samotnú verifikáciu vlastnoručne vytvoreným certifikátom. Odstrániť digitálny podpis z PE súboru je naopak triviálne. Tento útok sme simulovali rovnakým spôsobom ako pri ladiacich symboloch.

Entropia sekcií. Medzi úspešné útoky na modely založené na tradičnom strojovom učení, je pripájanie bajtov k PE súboru. V našom prípade sme podobný útok simulovali znížením entropie pre sekciu. Takúto modifikáciu vie útočník relatívne jednoducho vykonať, ak vezme určitú časť bajtov napr. z legitímneho súboru alebo použije iné štrukturované dáta (t.j. s nižšou entropiou) a pripojí ich k existujúcej sekcii. Týmto spôsobom útočník zníži entropiu sekcie, pričom pôvodná funkcionalita zostáva zachovaná. Tento útok sme simulovali znížením entropie o 1.5 pre konkrétnu sekciu.

Importy. Medzi ďalšie možnosti, ktoré má k dispozícii útočník, je zmeniť importované API volania. Samotné API volania môžu do istej miery naznačovať správanie škodlivého kódu a útočník môže predpokladať, že detekčný systém bude tieto volania využívať. Pri úprave importovaných funkcií (bez toho, aby sa zmenila škodlivá funkcionalita) má útočník niekoľko možností. Pri tomto útoku sme uvažovali dve možnosti. Pri prvej možnosti útočník pridá nové API volania (nemusia byť nutne aj použité, resp. je postačujúce, ak sa vyskytnú v zozname

importov). V tomto prípade sa útočník spolieha na fakt, že pridaním nových importovaných funkcií naruší vzťahy medzi funkciami, ktoré boli nájdené počas trénovania modelu. Tento útok sme simulovali pridaním náhodných API funkcií zo štandardnej knižnice `Kernel32.dll`. Druhou možnosťou, ktorú má útočník, je použiť kombináciu API volaní `LoadLibraryA` a `GetProcAddress` na dynamické načítanie API funkcií a následné použitie získaných smerníkov na funkcie. Tento spôsob však môže byť pre útočníka relatívne nepraktický (a taktiež si vyžaduje zdrojový kód). Tento útok sme simulovali nahradením štandardných API volaní z knižnice `Kernel32.dll` za vyššie spomínané volania.

5.8.2 Výsledky

Výsledky z jednotlivých útokov pre klasifikátor, ktorý bol trénovaný na datasete `dataset_8_1000.owl` môžeme vidieť v tabuľke 5.20. Keďže pôvodné modely boli validované prostredníctvom krížovej validácie, rovnaký postup sme zvolili aj pri týchto experimentoch. V samotnej tabuľke sme znázornili nasledujúce údaje:

- **Algoritmus**: konkrétny algoritmus konceptového učenia. Jednotlivé hodnoty hyper-parametrov sú totožné s finálnymi hodnotami (viď. kapitola 5.3, resp. 5.7).
- **Pôvodná F1 miera**: hodnota F1 miery, ktorú dosiahol konkrétny algoritmus na pôvodnej testovacej množine.
- **Nová F1 miera**: hodnota F1 miery, ktorú dosiahol algoritmus konceptového učenia po tom, ako sme na všetky vzorky v testovacej množine aplikovali konkrétny útok.
- **Pokles**: pokles novej F1 miery voči pôvodnej.
- **Úspešnosť útoku**: úspešnosť konkrétneho útoku (počítaná prostredníctvom vzorca (5.21)).

Ako môžeme vidieť v tabuľke, útok na sekcie PE súboru sme testovali pre rôzny počet sekcií (parameter n). Podobne, aj pri útoku s pridávaním API volaní, sme zvolili dve hodnoty n pre počet pridaných funkcií (100 a 500). Zmena ladiacich symbolov a podpisu príliš veľkú úspešnosť nepriniesla. Pre PARCEL sme dosiahli úspešnosť útoku 0.05 pri ladiacich symboloch a 0.08 pre SPACEL pri podpise. Neparalelné algoritmy sa ukázali ako odolné voči týmto útokom, keďže vlastnosti `Debug` a `Signature` (resp. vlastnosť

`FileFeature`) v konceptových výrazoch nepoužívali, zatiaľ čo paralelné algoritmy áno. Zníženie entropie sekcií sa ukázalo ako efektívnejšie a taktiež sme pozorovali nárast úspešnosti útoku pri zvyšovaní počtu sekcií, ktoré sme upravili. Tento efekt môžeme zdôvodniť dôležitosťou vlastnosti `HighEntropy` v konceptových výrazoch. Pri paralelných algoritmoch sa v modeli nachádzal určitý počet výrazov, ktoré zahŕňali kardinalitu vzťahu s rôznym n (napr. $\geqslant 3$ has_section_feature.HighEntropy) a postupným zvyšovaním upravených sekcií sme odstraňovali pokrytie takýmito čiastkovými výrazmi. Podobným prípadom boli aj konceptové výrazy z algoritmu CELOE, kde sa však nachádzal iba výraz vo forme $\exists$has_file_feature.HighEntropy (t.j. bez kardinality), kde bolo naopak potrebné odstrániť všetky sekcie s vysokou entropiou, aby sme odstránili pokrytie vzorky. Algoritmus OCEL naopak vlastnosť `HighEntropy` nevyužíval v žiadnom konceptovom výraze a tak bol odolný voči takémuto útoku. Útok pomocou nahradenia API volaní sa ukázal ako relatívne efektívny pre paralelné algoritmy, zatiaľ čo na neparalelné algoritmy nemal žiadny efekt. Úspešnosť útoku sme dosiahli 0.32 pre PARCEL a 0.35 pre SPACEL. Hlavným dôvodom bolo, že paralelné algoritmy obsahovali množstvo konceptových výrazov, kde boli rôzne importované API volania v konjunkcii a ich nahradením sme dokázali odstrániť ich pokrytie. Útok prostredníctvom pripájania náhodných API volaní sa ukázal pre algoritmus OCEL ako najefektívnejší (konkrétne pripojenie 500 volaní), kde sme dokázali získať úspešnosť až 0.95, zatiaľ čo paralelné algoritmy preukázali odolnosť (keďže pridaním ďalších volaní nenarušíme logické väz/by v čiastkových riešeniach). Vysoká úspešnosť útoku bola spôsobená tým, že väčšina konceptových výrazov z algoritmu OCEL obsahovala časť konceptového výrazu (v konjunkcii s ďalšími časťami), ktorá sa venovala výhradne importovaným volaniam, viď. príklad (5.7). Pridaním veľkého množstva API volaní, sa stala táto časť konceptového výrazu nepravdivou, čím sa zároveň stal nepravdivým celý výraz.

Rovnakú množinu útokov sme aplikovali aj na kombinovaný klasifikátor s cieľom zistiť, do akej miery sú dané útoky efektívne aj na kvalitnejších a robustnejších klasifikátoroch. Samotné výsledky a úspešnosť jednotlivých útokov môžeme vidieť v tabuľke 5.21. Čo sa týka prvých troch útokov (ladiace symboly, podpis a sekcie), tak výsledky boli podobné, resp. o niečo menej úspešné ako pri predchádzajúcom datasete (až na výnimku pre algoritmus SPACEL). Obzvlášť útoky na sekcie boli značne menej efektívne. Útoky na importované API volania preukázali podobnú úspešnosť, avšak v prípade kombinovaného klasifikátora sa ako najúspešnejší ukázal práve útok nahradením volaní. V tomto prípade sme nahradením dokázali získať aj vysokú úspešnosť útoku pri neparalelných algoritmoch (0.81 pre OCEL a 0.72 pre CELOE). Týmto útokom sme dokázali taktiež získať vyššiu úspešnosť pre paralelné algoritmy v porovnaní s prvým klasifikátorom (nárast z 0.32 na 0.41

Algoritmus	Pôvodná F1 miera	Nová F1 miera	Pokles	Úspešnosť útoku
		Ladiace symboly		
OCEL	0.74 ± 0.02	0.74 ± 0.02	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.69 ± 0.03	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.77 ± 0.02	0.74 ± 0.03	0.03 ± 0.01	**0.05 ± 0.01**
SPACEL	0.76 ± 0.03	0.75 ± 0.01	0.01 ± 0.02	0.02 ± 0.05
		Podpis		
OCEL	0.74 ± 0.02	0.74 ± 0.02	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.69 ± 0.03	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.77 ± 0.02	0.74 ± 0.02	0.03 ± 0.01	0.06 ± 0.02
SPACEL	0.76 ± 0.03	0.72 ± 0.02	0.04 ± 0.01	**0.08 ± 0.04**
		Sekcia ($n = 1$)		
OCEL	0.74 ± 0.02	0.74 ± 0.02	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.67 ± 0.04	0.02 ± 0.01	**0.06 ± 0.04**
PARCEL	0.77 ± 0.02	0.76 ± 0.01	0.01 ± 0.00	0.02 ± 0.01
SPACEL	0.76 ± 0.03	0.75 ± 0.02	0.01 ± 0.01	0.02 ± 0.03
		Sekcia ($n = 2$)		
OCEL	0.74 ± 0.02	0.74 ± 0.02	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.65 ± 0.06	0.04 ± 0.02	**0.12 ± 0.07**
PARCEL	0.77 ± 0.02	0.76 ± 0.01	0.01 ± 0.01	0.03 ± 0.02
SPACEL	0.76 ± 0.03	0.74 ± 0.02	0.01 ± 0.01	0.03 ± 0.03
		Sekcia ($n = 3$)		
OCEL	0.74 ± 0.02	0.74 ± 0.02	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.63 ± 0.06	0.06 ± 0.03	**0.16 ± 0.09**
PARCEL	0.77 ± 0.02	0.74 ± 0.01	0.03 ± 0.01	0.06 ± 0.02
SPACEL	0.76 ± 0.03	0.73 ± 0.02	0.03 ± 0.00	0.06 ± 0.01
		Importy (nahradenie)		
OCEL	0.74 ± 0.02	0.72 ± 0.03	0.01 ± 0.01	0.00 ± 0.00
CELOE	0.69 ± 0.03	0.69 ± 0.03	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.77 ± 0.02	0.59 ± 0.02	0.17 ± 0.03	0.32 ± 0.08
SPACEL	0.76 ± 0.03	0.60 ± 0.02	0.16 ± 0.03	**0.35 ± 0.07**
		Importy ($n = 100$)		
OCEL	0.74 ± 0.02	0.58 ± 0.09	0.16 ± 0.09	**0.33 ± 0.14**
CELOE	0.69 ± 0.03	0.68 ± 0.02	0.01 ± 0.03	0.03 ± 0.07
PARCEL	0.77 ± 0.02	0.71 ± 0.01	0.06 ± 0.01	0.00 ± 0.00
SPACEL	0.76 ± 0.03	0.71 ± 0.01	0.05 ± 0.02	0.00 ± 0.00
		Importy ($n = 500$)		
OCEL	0.74 ± 0.02	0.06 ± 0.02	0.68 ± 0.01	**0.95 ± 0.01**
CELOE	0.69 ± 0.03	0.58 ± 0.22	0.10 ± 0.24	0.17 ± 0.38
PARCEL	0.77 ± 0.02	0.70 ± 0.00	0.07 ± 0.02	0.00 ± 0.00
SPACEL	0.76 ± 0.03	0.70 ± 0.00	0.06 ± 0.03	0.00 ± 0.00

Tabuľka 5.20: Úspešnosť jednotlivých útokov na modeli získanom z datasetu `dataset_8_1000.owl`.

pre PARCEL a nárast z 0.35 na 0.60 pre SPACEL). Zaujímavým pozorovaním bolo (pri oboch klasifikátoroch), že v niektorých prípadoch nastal pokles F1 miery, aj keď samotná úspešnosť útoku bola 0.00. V týchto prípadoch (obzvlášť pri algoritme SPACEL) nastávali prípady, kedy sme samotným útokom naopak zvýšili počet správne označených pozitívnych príkladov (t.j. TP), ale zároveň nám vznikli nové, zle označené vzorky (t.j. FP a FN), čím nám klesla F1 miera.

5.8.3 *White-box* útoky

V predchádzajúcej kapitole sme sa venovali teoreticky možným útokom, ktoré dokáže útočník vykonať, ak nemá žiadne znalosti o klasifikátore a má k dispozícii zdrojový kód malvéru (aj keď nie je potrebný pre všetky vyššie spomínané útoky). Pri skúmaní bezpečnostných aspektov klasifikátora je však nutné brať do úvahy aj scenár, v ktorom má útočník všetky znalosti o použitom detekčnom modely. V našom prípade si môžeme predstaviť situáciu, kde útočník získa množinu konceptových výrazov, ktoré sa používajú na klasifikáciu, či sa jedná o škodlivý alebo legitímny softvér.

Ako prvý príklad si môžeme opätovne ukázať konceptový výraz získaný z OCEL (viď. (5.7)) a predpokladajme, že útočník má k dispozícii zdrojový kód a môže vykonávať ľubovoľné zmeny. Ak útočník bližšie zanalyzuje daný výraz, zistí, že sa skladá zo štyroch menších konceptov, ktoré sú spojené konjunkciou (t.j. $\sqcap$). V takom prípade stačí, aby zmenil ľubovoľný zo štyroch konceptov (t.j. zmenil jeho pravdivostnú hodnotu z 1 na 0, ak predpokladáme, že útočníkova vzorka bola pôvodne vyhodnotená na 1). Prvý menší výraz `ExecutableFile` útočník zmení relatívne ťažko. Alternatívne môže vzorku prerobiť na dynamickú knižnicu, avšak v takom prípade bude musieť zmeniť celkový vektor útoku. Druhá časť $\exists$`has_file_feature`.(`MultipleExecutableSections` $\sqcup$ `NonstandardMZ`) môže byť pre útočníka taktiež problematická. Keďže predpokladáme, že pôvodná vzorka bola správne označená ako malvér, tak aj táto časť konceptu musela byť pravdivá. Z toho môžeme usudzovať, že útočník použil určitú formu baliaceho nástroja na ukrytie pôvodnej funkcionality. V takom prípade má dve možnosti: nevyužiť baliaci nástroj (tento fakt môže opätovne viesť k nutnosti zmeniť vektor útoku) alebo využiť alternatívny, resp. vlastný baliaci nástroj, ktorý funguje na iných princípoch. V oboch prípadoch sa môže jednať o náročnú zmenu. Posledná časť konceptového výrazu, ktorá definuje akcie malvéru sa javí ako najslabší článok výrazu, keďže útočníkovi stačí pridať správne akcie (t.j. importovať ďalšie API volania), aby narušil túto väzbu. Útočník však potrebuje poznať, aké akcie daný model používa. Keďže uvažujeme *white-box* scenár, môžeme taktiež predpokladať, že útočník má taktiež k dispozícii samotnú ontológiu (t.j. pozná všetky potenciálne akcie). Táto zmena taktiež patrí medzi najtriviálnejšie, čo sa týka praktickej

Algoritmus	Pôvodná F1 miera	Nová F1 miera	Pokles	Úspešnosť útoku
		Ladiace symboly		
OCEL	0.91 ± 0.00	0.91 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.88 ± 0.00	0.78 ± 0.04	0.09 ± 0.05	0.17 ± 0.09
PARCEL	0.93 ± 0.00	0.93 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
SPACEL	0.84 ± 0.01	0.68 ± 0.11	0.15 ± 0.09	**0.27 ± 0.16**
		Podpis		
OCEL	0.91 ± 0.00	0.91 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.88 ± 0.00	0.78 ± 0.04	0.09 ± 0.05	0.17 ± 0.09
PARCEL	0.93 ± 0.00	0.93 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
SPACEL	0.84 ± 0.01	0.76 ± 0.07	0.07 ± 0.06	**0.14 ± 0.11**
		Sekcia ($n = 1$)		
OCEL	0.91 ± 0.00	0.91 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.88 ± 0.00	0.88 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.93 ± 0.00	0.93 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
SPACEL	0.84 ± 0.01	0.82 ± 0.01	0.02 ± 0.00	**0.04 ± 0.00**
		Sekcia ($n = 2$)		
OCEL	0.91 ± 0.00	0.91 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.88 ± 0.00	0.88 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.93 ± 0.00	0.91 ± 0.00	0.01 ± 0.00	0.03 ± 0.00
SPACEL	0.84 ± 0.01	0.82 ± 0.02	0.02 ± 0.00	**0.04 ± 0.00**
		Sekcia ($n = 3$)		
OCEL	0.91 ± 0.00	0.91 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
CELOE	0.88 ± 0.00	0.88 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
PARCEL	0.93 ± 0.00	0.91 ± 0.00	0.01 ± 0.00	0.03 ± 0.00
SPACEL	0.84 ± 0.01	0.81 ± 0.02	0.02 ± 0.00	**0.04 ± 0.00**
		Importy (nahradenie)		
OCEL	0.91 ± 0.00	0.26 ± 0.14	0.65 ± 0.14	**0.81 ± 0.10**
CELOE	0.88 ± 0.00	0.31 ± 0.18	0.56 ± 0.19	0.72 ± 0.12
PARCEL	0.93 ± 0.00	0.57 ± 0.16	0.35 ± 0.16	0.41 ± 0.32
SPACEL	0.84 ± 0.01	0.44 ± 0.09	0.39 ± 0.08	0.60 ± 0.09
		Importy ($n = 100$)		
OCEL	0.91 ± 0.00	0.45 ± 0.05	0.46 ± 0.05	**0.64 ± 0.06**
CELOE	0.88 ± 0.00	0.80 ± 0.05	0.07 ± 0.04	0.09 ± 0.10
PARCEL	0.93 ± 0.00	0.81 ± 0.02	0.11 ± 0.02	0.01 ± 0.05
SPACEL	0.84 ± 0.01	0.73 ± 0.03	0.11 ± 0.02	0.07 ± 0.05
		Importy ($n = 500$)		
OCEL	0.91 ± 0.00	0.31 ± 0.03	0.60 ± 0.03	**0.77 ± 0.02**
CELOE	0.88 ± 0.00	0.75 ± 0.02	0.12 ± 0.01	0.13 ± 0.05
PARCEL	0.93 ± 0.00	0.78 ± 0.02	0.14 ± 0.02	0.00 ± 0.00
SPACEL	0.84 ± 0.01	0.75 ± 0.03	0.08 ± 0.01	0.00 ± 0.00

Tabuľka 5.21: Úspešnosť jednotlivých útokov na kombinovanom klasifikátore.

realizácie, keďže nie je nutný zdrojový kód a pridávanie ďalších importov nenaruší funkcionalitu malvéru.

Ako druhý príklad si môžeme uviesť čiastkové riešenia (5.9) a (5.10) z algoritmu PARCEL. Prvý konceptový výraz označuje sekcie, ktoré sa vyskytujú pri zbalených vzorkách. Zaujímavosťou výrazu je, že vzorka musí obsahovať aspoň tri sekcie s neštandardným menom a právami na zápis. Odstrániť takéto sekcie, aby ich bolo menej ako tri, môže byť značne problematické, obzvlášť pri komerčných riešeniach. Jednou alternatívou pre útočníka teda môže byť využitie iného nástroja. Alternatívne môže zmeniť neštandardné meno sekcie (v tomto prípade však môže byť problematické zachovanie funkcionality s baliacim nástrojom) alebo odstrániť práva na zápis a tieto práva doplniť neskôr za behu prostredníctvom API volania. Pri druhom čiastkovom riešení musí útočník narušiť konjunkciu výrazov, kde má dve možnosti: vyhodiť API volania, ktoré patria do kategórie `SystemManipulation` (napr. podobným spôsobom, ako pri nahradení importov pri *black-box* útokoch) alebo znížiť entropiu sekcie, ktorá obsahuje kód programu, kde sa jedná o triviálnejšiu zmenu. Pri paralelných algoritmoch (resp. aj pri kombinovanom klasifikátore) zohráva dôležitú úlohu aj počet čiastkových výrazov, ktoré popisujú konkrétnu vzorku. Čím ich je viac, tým taššiu úlohu má útočník, keďže potrebuje odstrániť pokrytie všetkých výrazov (samotné výrazy sú v disjunkcii).

5.8.4 Diskusia

V rámci tejto experimentálnej časti sme skúmali bezpečnostné aspekty konceptových výrazov. Zo samotných výsledkov môžeme zhodnotiť, že bezpečnosť výrazov je porovnateľná so štandardným strojovým učením, keďže naše najúspešnejšie útoky dosahovali podobné čísla. Ako najúspešnejšie útoky môžeme označiť tie, ktoré manipulovali importované API volania. Ak však zoberieme do úvahy fakt, že útok nahradením API volaní je príliš náročný pre útočníka, tak ako najodolnejšie algoritmy sa ukázali práve paralelné algoritmy PARCEL a SPACEL. Útok nahradením API volaní by však bolo možné teoricky eliminovať, ak by sme do samotnej ontológie zahrnuli výraz popisujúci kombináciu API volaní, ktoré sa používajú na tento účel. Taktiež môžeme zhodnotiť, že napriek užitočnosti vysvetliteľných výrazov z hľadiska detekcie a dôveryhodnosti klasifikačných modelov, práve samotná možnosť interpretovať model poskytuje detailný náhľad pre útočníkov, ktorý im značne uľahčuje tvorbu AE.

5.9 Zhrnutie experimentálnych výsledkov

Celkovo môžeme výsledky z experimentov označiť za uspokojivé. Metódy konceptového učenia sa ukázali ako vhodné pri riešení detekcie škodlivého kódu. Napriek tomu, že sa jedná o výpočtovo náročný proces, ich hlavná výhoda spočíva v implicitnej vysvetliteľnosti, kde dokážeme pomerne jednoducho rozumieť rozhodnutiam klasifikátora (viď. kapitola 5.3.7). Možeme teda hovoriť, že existuje určitý *trade-off*, keďže natrénovať model je relatívne náročný proces z výpočtového hľadiska, avšak za cenu plnej vysvetliteľnosti.

Najúspešnejší model, z hľadiska schopnosti správne rozlišovať škodlivý kód, sme získali kombinovaním klasifikátorov pre jednotlivé rodiny (viď. kapitola 5.7). Takýmto spôsobom sme boli taktiež schopní použiť značne rozsiahlejšie datasety (s počtom vzoriek až 20k pre jednu rodinu). Ako najúspešnejšie algoritmy v rámci týchto experimentov sme označili OCEL, ktorý dosiahol F1 mieru 0.91 pri falošnej pozitivite iba 0.0024 a PARCEL s F1 mierou 0.93, avšak s relatívne vysokou FP mierou 0.11. Môžeme však skonštatovať, že OCEL vyprodukoval značne prehľadnejší a ľahšie čitateľnejší model (keďže kombinovaný klasifikátor obsahuje jeden konceptový výraz na rodinu). Napriek tomu, že v porovnaní s tradičným strojovým učením sme použili relatívne malý dataset (kvôli výpočtovej náročnosti), vyššie spomínané výsledky sú úspešnosťou porovnateľné (viď. kapitola 1.3, kde úspešnosť klasifikátorov, s použitím štandardného strojového učenia, dosahovala hodnoty od 0.90 až po 0.99).

V rámci kapitoly 5.8 sme taktiež skúmali odolnosť nami vytvorených klasifikátorov voči rôznym útokom. Samotná úspešnosť nami navrhnutých útokov sa líšila v závislosti od použitej modifikácie a dosiahli sme úspešnosti od 0.00 až po 0.95 (t.j. pre 95% škodlivých vzoriek sme dokázali modifikáciou zmeniť označenie vzorky zo škodlivého kódu na legitímny súbor). Celkovo sme tak skonštatovali, že úroveň bezpečnosti konceptových výrazov je porovnateľná so strojovým učením, kde autori prác dosahovali úspešnosť útokov od 0.60 až po 0.98 (viď. kapitola 1.8). Je však nutné poznamenať, že samotná vysvetliteľnosť modelu dokáže útočníkovi ponúknuť užitočný náhľad do fungovania klasifikátora a taktiež efektívnejšie možnosti ako obchádzať klasifikáciu bez nutnosti navrhovať komplikované algoritmy.

Záver

V našej práci sme sa venovali výskumu sémantických technológií, konkrétne ontológiám a konceptovému učeniu, a ich prínosu v oblasti informačnej bezpečnosti. Špecificky sme sa zaoberali využitím a aplikáciou týchto technológií pri rozpoznávaní škodlivého kódu.

Celkovo môžeme konštatovať, že sa nám podarilo potvrdiť jednotlivé vedecké hypotézy, ktoré sme si stanovili v rámci výskumu (viď. kapitola 5.1). V rámci práce sme navrhli a implementovali novú ontológiu, ktorá reprezentuje škodlivé súbory pre operačný systém *Windows*. Okrem samotného návrhu sme publikovali aj množinu datasetov, ktorá môže slúžiť na ďalšie napredovanie výskumu v tejto oblasti. Okrem samotného faktu, že podobná ontológia doposiaľ neexistovala, v porovnaní s inými ontológiami môžeme vyzdvihnúť hlavne jej zameranie na aktuálny vedecký problém spolu s robustnosťou datasetu, ktorá môže byť užitočná pri výskume nových algoritmov. Samotná ontológia nie je limitovaná iba pre konceptové učenie, ale môže byť využitá aj pri ďalších podobných prístupoch (viď. kapitola 3.6). V tejto práci sme taktiež skúmali existujúce algoritmy konceptového učenia (dostupné v rámci softvéru *DL-Learner*). Konkrétne sme sa venovali ich aplikácii na tvorbu vysvetliteľných modelov pre detekciu škodlivého kódu. Takáto aplikácia konceptového učenia doposiaľ nebola preskúmaná. Ako najlepšie algoritmy z hľadiska úspešnosti sme označili OCEL a PARCEL, kde najlepšie modely dosiahli F1 mieru 0.91, resp. 0.93 (pri kombinácii rodín, viď. kapitola 5.7). Celkovo sme tak zhodnotili, že konceptové učenie preukazuje vysoký potenciál pri riešení problému detekcie malvéru, keďže metriky pri niektorých prípadoch príliš nezaostávali za štandardným strojovým učením, pričom výsledné modely poskytovali plnú vysvetliteľnosť (viď. kapitola 5.3.7). Ako zrejmá nevýhoda algoritmov konceptového učenia, však stále zostáva výpočtová náročnosť. Okrem samotnej úspešnosti algoritmov a ich vysvetliteľnosti sme skúmali aj ich bezpečnosť (viď. kapitola 5.8). V rámci týchto experimentov sme zistili, že bezpečnosť konceptových výrazov je porovnateľná s modelmi strojového učenia.

Naša práca taktiež ponúka viacero smerovaní ďalšieho výskumu do budúcnosti. Môžeme si ich zhrnúť v nasledujúcich bodoch:

- Limitáciou nami navrhnutej ontológie a následných experimentov je, že používajú iba statické dáta (extrahované bez spustenia vzorky). Využitie dynamických vlastností by mohlo priniesť presnejšiu detekciu a potenciálne aj vyššiu odolnosť voči rôznym útokom. V rámci našej práce sme boli limitovaní samotným datasetom EMBER, ktorý ponúka iba statické dáta.

- Výskum ďalších algoritmov konceptového učenia. V kapitole 3.5 sme uvádzali ďalšie algoritmy konceptového učenia, ktoré však neboli skúmané v rámci našich experimentov. Niektoré z týchto algoritmov ako napr. *EvoLearner*, ktorý bol publikovaný počas písania tejto práce, sa javia, že by mohli byť efektívnejšie pri učení ako algoritmy z *DL-Learner*. Zaujímavým smerovaním sú aj algoritmy, ktoré pracujú nad fuzzy logikou, ako napr. PN-OWL (viď. kapitola 3.5), ktorý by potenciálne dokázal efektívne využiť rôzne dátové vlastnosti nachádzajúce sa v našej ontológii.

- V poslednom rade, potenciálny budúci výskum môže byť smerovaný aj k vývoju nových algoritmov konceptového učenia. Ako sme spomínali vyššie, najväčšia limitácia konceptového učenia je výpočtová náročnosť. Ďalší výskum teda môže dbať dôraz na vývoj efektívnejších algoritmov, ktoré dokážu prehľadať väčší priestor konceptov.

Na záver si môžeme zhrnúť celkové prínosy dizertačnej práce. Prvým prínosom je vytvorenie a publikovanie ontológie a datasetov pre malvér. Samotná ontológia môže slúžiť pri vývoji nových detekčných modelov alebo pri napredovaní algoritmov konceptového učenia. Druhým prínosom práce je analýza existujúcich algoritmov konceptového učenia a preukázanie vhodnosti ich aplikácie na riešenie problému rozpoznávania škodlivého kódu. V neposlednom rade môžeme za prínos označiť aj základnú bezpečnostnú analýzu konceptových výrazov, ktorou sme ukázali, že úroveň bezpečnosti príliš nezaostáva za tradičným strojovým učením. Celkovo sme tak ukázali, že konceptové učenie je vhodná metodika na riešenie problému a pokiaľ budú dané algoritmy napredovať, hlavne čo sa týka výpočtového hľadiska, môžu zohrávať dôležitú rolu pri vývoji dôveryhodných a vysvetlitelných modelov na detekciu škodlivého kódu.

Bibliografia

1. *Malware Statistics Trends Report*. [B.r.]. Dostupné tiež z: `https://www.av-test.org/en/statistics/malware/`. [Online; cit. 2023-10-21].
2. ŠVEC, Peter, BALOGH, Štefan, HOMOLA, Martin a KL'UKA, Ján. Ontological Representation of the EMBER Dataset. *Application of Knowledge Methods in Information Security*. 2022, s. 3.
3. ŠVEC, Peter, BALOGH, Štefan, HOMOLA, Martin a KL'UKA, Ján. Knowledge-Based Dataset for Training PE Malware Detection Models. *arXiv preprint arXiv:2301.00153*. 2022.
4. ŠVEC, Peter, BALOGH, Štefan, HOMOLA, Martin, KL'UKA, Ján a BISTÁK, Tomáš. Semantic Data Representation for Explainable Windows Malware Detection Models. *arXiv preprint arXiv:2403.11669*. 2024.
5. ŠVEC, Peter a BALOGH. Description Logics Concept Learning in Malware Detection. *Application of Knowledge Methods in Information Security*. 2021, s. 1.
6. ŠVEC, Peter, BALOGH, Štefan a HOMOLA, Martin. Experimental Evaluation of Description Logic Concept Learning Algorithms for Static Malware Detection. In: *ICISSP*. 2021, s. 792–799.
7. ŠVEC, Peter, BISTÁK, Tomáš, HOMOLA, Martin, BALOGH, Štefan, KĽUKA, Ján a ŠIMKO, Alexander. Towards Explainable Malware Detection with Structured Machine Learning. In: *Workshop on Explainable Logic-Based Knowledge Representation XLoKR 2023*. 2023.
8. BISTÁK, Tomáš, ŠVEC, Peter, KĽUKA, Ján, ŠIMKO, Alexander, BALOGH, Štefan a HOMOLA, Martin. Improving DL-Learner on a Malware Detection Use Case. In: *36th International Workshop on Description Logics, DL 2023*. CEUR-WS, 2023.

9. OORSCHOT, P.C. van. *Computer Security and the Internet: Tools and Jewels from Malware to Bitcoin*. Springer International Publishing, 2021. Information Security and Cryptography. ISBN 9783030834104. Dostupné tiež z: `https://books.google.sk/books?id=3IyKzgEACAAJ`.
10. *ESET Threat Report T3 2022*. [B.r.]. Dostupné tiež z: `https://www.welivesecurity.com/2023/02/08/eset-threat-report-t3-2022/`. [Online; cit. 2023-06-17].
11. TN, Nisha a SHAILENDRA KULKARNI, Mugdha. Zero click attacks–a new cyber threat for the e-banking sector. *Journal of Financial Crime*. 2022.
12. CHEN, Qian a BRIDGES, Robert A. Automated behavioral analysis of malware: A case study of wannacry ransomware. In: *2017 16th IEEE International Conference on machine learning and applications (ICMLA)*. IEEE, 2017, s. 454–460.
13. COHEN, Fred. *Computer viruses*. 1986. Diz. pr. University of Southern California Janvier.
14. ALZAROONI, KMA. *Malware variant detection*. 2012. Diz. pr. UCL (University College London).
15. SZOR, Peter a FERRIE, Peter. Hunting for metamorphic. In: *Virus bulletin conference*. Citeseer, 2001.
16. YAN, Wei, ZHANG, Zheng a ANSARI, Nirwan. Revealing packed malware. *ieee seCurity & PrivaCy*. 2008, roč. 6, č. 5, s. 65–69.
17. WANG, Xiaolu, WANG, Zhi, SHAO, Wei, JIA, Chunfu a LI, Xiang. Explaining concept drift of deep learning models. In: *Cyberspace Safety and Security: 11th International Symposium, CSS 2019, Guangzhou, China, December 1–3, 2019, Proceedings, Part II 11*. Springer, 2019, s. 524–534.
18. SAAD, Sherif, BRIGUGLIO, William a ELMILIGI, Haytham. The curious case of machine learning in malware detection. *arXiv preprint arXiv:1905.07573*. 2019.
19. HAHN, Katja. Robust static analysis of portable executable malware. *HTWK Leipzig*. 2014, s. 134.
20. BOROJERDI, Haniye Razeghi a ABADI, Mahdi. MalHunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection. In: *ICCKE 2013*. IEEE, 2013, s. 430–436.

21. ZHENG, Min, SUN, Mingshen a LUI, John CS. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, s. 163–171.

22. BALDANGOMBO, Usukhbayar, JAMBALJAV, Nyamjav a HORNG, Shi-Jinn. A static malware detection system using data mining methods. *arXiv preprint arXiv:1308.2831*. 2013.

23. MALHOTRA, Aashima a BAJAJ, Karan. A hybrid pattern based text mining approach for malware detection using DBScan. *CSI transactions on ICT*. 2016, roč. 4, č. 2-4, s. 141–149.

24. BAHADOR, Mohammad Bagher, ABADI, Mahdi a TAJODDIN, Asghar. HLMD: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*. 2019, roč. 75, č. 8, s. 5551–5582.

25. PARK, Younghee, REEVES, Douglas S a STAMP, Mark. Deriving common malware behavior through graph clustering. *Computers & Security*. 2013, roč. 39, s. 419–430.

26. DING, Yuxin, XIA, Xiaoling, CHEN, Sheng a LI, Ye. A malware detection method based on family behavior graph. *Computers & Security*. 2018, roč. 73, s. 73–86.

27. DAS, Sanjeev, LIU, Yang, ZHANG, Wei a CHANDRAMOHAN, Mahintham. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*. 2015, roč. 11, č. 2, s. 289–302.

28. NOROUZI, Monire, SOURI, Alireza a SAMAD ZAMINI, Majid. A data mining classification approach for behavioral malware detection. *Journal of Computer Networks and Communications*. 2016, roč. 2016.

29. CHANDRAMOHAN, Mahinthan, TAN, Hee Beng Kuan, BRIAND, Lionel C, SHAR, Lwin Khin a PADMANABHUNI, Bindu Madhavi. A scalable approach for malware detection through bounded feature space behavior modeling. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, s. 312–322.

30. PAJOUH, Hamed Haddad, DEHGHANTANHA, Ali, KHAYAMI, Raouf a CHOO, Kim-Kwang Raymond. Intelligent OS X malware threat detection with code inspection. *Journal of Computer Virology and Hacking Techniques*. 2018, roč. 14, č. 3, s. 213–223.

31. ESKANDARI, Mojtaba a HASHEMI, Sattar. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*. 2012, roč. 23, č. 3, s. 154–162.

32. MOSLI, Rayan, LI, Rui, YUAN, Bo a PAN, Yin. A behavior-based approach for malware detection. In: *IFIP International Conference on Digital Forensics*. Springer, 2017, s. 187–201.

33. ALSULAMI, Bander, SRINIVASAN, Avinash, DONG, Hunter a MANCORIDIS, Spiros. Lightweight behavioral malware detection for windows platforms. In: *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2017, s. 75–81.

34. BAZRAFSHAN, Zahra, HASHEMI, Hashem, FARD, Seyed Mehdi Hazrati a HAMZEH, Ali. A survey on heuristic malware detection techniques. In: *The 5th Conference on Information and Knowledge Technology*. IEEE, 2013, s. 113–120.

35. RAFF, Edward, ZAK, Richard, COX, Russell, SYLVESTER, Jared, YACCI, Paul, WARD, Rebecca, TRACY, Anna, MCLEAN, Mark a NICHOLAS, Charles. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*. 2018, roč. 14, č. 1, s. 1–20.

36. ANDERSON, Blake, QUIST, Daniel, NEIL, Joshua, STORLIE, Curtis a LANE, Terran. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*. 2011, roč. 7, č. 4, s. 247–258.

37. ISLAM, Rafiqul, TIAN, Ronghua, BATTEN, Lynn M a VERSTEEG, Steve. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*. 2013, roč. 36, č. 2, s. 646–656.

38. NAVAL, Smita, LAXMI, Vijay, RAJARAJAN, Muttukrishnan, GAUR, Manoj Singh a CONTI, Mauro. Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*. 2015, roč. 10, č. 12, s. 2591–2604.

39. YE, Yanfang, WANG, Dingding, LI, Tao, YE, Dongyi a JIANG, Qingshan. An intelligent PE-malware detection system based on association mining. *Journal in computer virology*. 2008, roč. 4, č. 4, s. 323–334.

40. ZAKERI, Mohaddeseh, FARAJI DANESHGAR, Fatemeh a ABBASPOUR, Maghsoud. A static heuristic approach to detecting malware targets. *Security and Communication Networks*. 2015, roč. 8, č. 17, s. 3015–3027.

41. KHODAMORADI, Peyman, FAZLALI, Mahmood, MARDUKHI, Farhad a NOSRATI, Masoud. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In: *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*. IEEE, 2015, s. 1–6.

42. MEHTAB, Anam, SHAHID, Waleed Bin, YAQOOB, Tahreem, AMJAD, Muhammad Faisal, ABBAS, Haider, AFZAL, Hammad a SAQIB, Malik Najmus. AdDroid: rule-based machine learning framework for android malware analysis. *Mobile Networks and Applications*. 2020, roč. 25, č. 1, s. 180–192.

43. ÍNCER ROMEO, Íñigo, THEODORIDES, Michael, AFROZ, Sadia a WAGNER, David. Adversarially robust malware detection using monotonic classification. In: *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. 2018, s. 54–63.

44. SAXE, Joshua a BERLIN, Konstantin. Deep neural network based malware detection using two dimensional binary program features. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015, s. 11–20.

45. HUANG, Wenyi a STOKES, Jack W. MtNet: a multi-task neural network for dynamic malware classification. In: *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, s. 399–418.

46. YE, Yanfang, CHEN, Lingwei, HOU, Shifu, HARDY, William a LI, Xin. DeepAM: a heterogeneous deep learning framework for intelligent malware detection. *Knowledge and Information Systems*. 2018, roč. 54, č. 2, s. 265–285.

47. ZHU, Dali, JIN, Hao, YANG, Ying, WU, Di a CHEN, Weiyi. DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data. In: *2017 IEEE symposium on computers and communications (ISCC)*. IEEE, 2017, s. 438–443.

48. NATARAJ, Lakshmanan, KARTHIKEYAN, Sreejith, JACOB, Gregoire a MANJUNATH, Bangalore S. Malware images: visualization and automatic classification. In: *Proceedings of the 8th international symposium on visualization for cyber security*. 2011, s. 1–7.

49. CUI, Zhihua, XUE, Fei, CAI, Xingjuan, CAO, Yang, WANG, Gai-ge a CHEN, Jinjun. Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*. 2018, roč. 14, č. 7, s. 3187–3196.

50. KALASH, Mahmoud, ROCHAN, Mrigank, MOHAMMED, Noman, BRUCE, Neil DB, WANG, Yang a IQBAL, Farkhund. Malware classification with deep convolutional neural networks. In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, s. 1–5.

51. RAFF, Edward, BARKER, Jon, SYLVESTER, Jared, BRANDON, Robert, CATANZARO, Bryan a NICHOLAS, Charles. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*. 2017.

52. KRČÁL, Marek, ŠVEC, O., BÁLEK, M. a JAŠEK, Otakar. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. In: *ICLR*. 2018.

53. MILLS, Alan, SPYRIDOPOULOS, Theodoros a LEGG, Phil. Efficient and interpretable real-time malware detection using random-forest. In: *2019 International conference on cyber situational awareness, data analytics and assessment (Cyber SA)*. IEEE, 2019, s. 1–8.

54. IADAROLA, Giacomo, MARTINELLI, Fabio, MERCALDO, Francesco a SANTONE, Antonella. Towards an interpretable deep learning model for mobile malware detection and family identification. *Computers & Security*. 2021, roč. 105, s. 102198. ISSN 0167-4048. Dostupné z DOI: 10.1016/j.cose.2021.102198.

55. MARAIS, Benjamin, QUERTIER, Tony a CHESNEAU, Christophe. Malware Analysis with Artificial Intelligence and a Particular Attention on Results Interpretability. In: MATSUI, Kenji, OMATU, Sigeru, YIGITCANLAR, Tan a RODRIGUEZ-GONZÁLEZ, Sara (ed.). *Distributed Computing and Artificial Intelligence, Volume 1: 18th International Conference, DCAI 2021, Salamanca, Spain, 6-8 October 2021*. Springer, 2021, zv. 327, s. 43–55. LNNS. Dostupné z DOI: 10.1007/978-3-030-86261-9_5.

56. DOLEJŠ, Jan a JUREČEK, Martin. Interpretability of Machine Learning-Based Results of Malware Detection Using a Set of Rules. In: *Artificial Intelligence for Cybersecurity*. Ed. STAMP, Mark, AARON VISAGGIO, Corrado, MERCALDO, Francesco a DI TROIA, Fabio. Cham: Springer International Publishing, 2022, s. 107–136. Dostupné z DOI: 10.1007/978-3-030-97087-1_5.

57. FÜRNKRANZ, Johannes a WIDMER, Gerhard. Incremental reduced error pruning. In: *Machine learning proceedings 1994*. Elsevier, 1994, s. 70–77.

58. BRIDGE, Karl, HESHAM, Ahmed, KELDORPH, Russ, ZHU, Yong-Kang, ABRAM, Nico, KENNEDY, John, BATCHELOR, Drew, COULTER, David, KRELL, Jay, ROBERTSON, Colin, KAYSER, Maurice, WARRINGTON, Christopher, SATRAN, Michael a LEBLANC, Mark. *PE Format*. Microsoft, [b.r.]. Dostupné tiež z: `https://learn.microsoft.com/en-gb/windows/win32/debug/pe-format`. [Online; cit. 2023-02-01].

59. ANDERSON, Hyrum S a ROTH, Phil. EMBER: An open dataset for training static PE malware machine learning models. *arXiv preprint arXiv:1804.04637*. 2018.

60. HARANG, Richard a RUDD, Ethan M. SOREL-20M: A large scale benchmark dataset for malicious PE detection. *arXiv preprint arXiv:2012.07634*. 2020.

61. RONEN, Royi, RADU, Marian, FEUERSTEIN, Corina, YOM-TOV, Elad a AHMADI, Mansour. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*. 2018.

62. SEVERI, Giorgio, LEEK, Tim a DOLAN-GAVITT, Brendan. MALREC: Compact full-trace malware recording for retrospective deep analysis. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, s. 3–23.

63. CATAK, Ferhat Ozgur a YAZI, Ahmet Faruk. A benchmark API call dataset for Windows PE malware classification. *arXiv preprint arXiv:1905.01999*. 2019.

64. BOSANSKY, Branislav, KOUBA, Dominik, MANHAL, Ondrej, SICK, Thorsten, LISY, Viliam, KROUSTEK, Jakub a SOMOL, Petr. Avast-CTU Public CAPE Dataset. *arXiv preprint arXiv:2209.03188*. 2022.

65. *Classification of Malware PE Headers*. [B.r.]. Dostupné tiež z: `https://github.com/urwithajit9/ClaMP`. [Online; cit. 2023-02-03].

66. ARP, Daniel, SPREITZENBARTH, Michael, HUBNER, Malte, GASCON, Hugo, RIECK, Konrad a SIEMENS, CERT. DREBIN: Effective and explainable detection of Android malware in your pocket. In: *NDSS*. 2014, zv. 14, s. 23–26.

67. JOYCE, Robert J, AMLANI, Dev, NICHOLAS, Charles a RAFF, Edward. MOTIF: A Malware Reference Dataset with Ground Truth Family Labels. *Computers & Security*. 2023, roč. 124, s. 102921.

68. PLOHMANN, Daniel, CLAUSS, Martin, ENDERS, Steffen a PADILLA, Elmar. Malpedia: a collaborative effort to inventorize the malware landscape. *Proceedings of the Botconf*. 2017.

69. KARBAB, ElMouatez Billah, DEBBABI, Mourad, DERHAB, Abdelouahid a MOUHEB, Djedjiga. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*. 2018, roč. 24, S48–S59.

70. *VirusShare*. [B.r.]. Dostupné tiež z: `https://virusshare.com`. [Online; cit. 2023-02-03].

71. *VirusTotal*. [B.r.]. Dostupné tiež z: `https://www.virustotal.com/gui/`. [Online]; cit. 2023-02-04.

72. *theZoo - A Live Malware Repository*. [B.r.]. Dostupné tiež z: `https://thezoo.morirt.com/`. [Online]; cit. 2023-02-04.

73. *MalShare*. [B.r.]. Dostupné tiež z: `https://malshare.com/`. [Online; cit. 2022-10-15].

74. SZEGEDY, Christian, ZAREMBA, Wojciech, SUTSKEVER, Ilya, BRUNA, Joan, ERHAN, Dumitru, GOODFELLOW, Ian a FERGUS, Rob. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*. 2013.

75. EYKHOLT, Kevin, EVTIMOV, Ivan, FERNANDES, Earlence, LI, Bo, RAHMATI, Amir, XIAO, Chaowei, PRAKASH, Atul, KOHNO, Tadayoshi a SONG, Dawn. Robust physical-world attacks on deep learning visual classification. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, s. 1625–1634.

76. HAO-CHEN, Han Xu Yao Ma, DEB, Liu Debayan, ANIL, Hui Liu Ji-Liang Tang a JAIN, K. Adversarial attacks and defenses in images, graphs and text: A review. *International Journal of Automation and Computing*. 2020, roč. 17, č. 2, s. 151–178.

77. PAPERNOT, Nicolas, MCDANIEL, Patrick, GOODFELLOW, Ian, JHA, Somesh, CELIK, Z Berkay a SWAMI, Ananthram. Practical black-box attacks against machine learning. In: *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 2017, s. 506–519.

78. KOLOSNJAJI, Bojan, DEMONTIS, Ambra, BIGGIO, Battista, MAIORCA, Davide, GIACINTO, Giorgio, ECKERT, Claudia a ROLI, Fabio. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE, 2018, s. 533–537.

79. SUCIU, Octavian, COULL, Scott E a JOHNS, Jeffrey. Exploring adversarial examples in malware detection. In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019, s. 8–14.

80. HU, Weiwei a TAN, Ying. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*. 2017.

81. PARK, Daniel, KHAN, Haidar a YENER, Bülent. Generation & Evaluation of Adversarial Examples for Malware Obfuscation. In: *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 2019, s. 1283–1290.

82. YOU, Ilsun a YIM, Kangbin. Malware obfuscation techniques: A brief survey. In: *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, s. 297–300.

83. ROSENBERG, Ishai, SHABTAI, Asaf, ROKACH, Lior a ELOVICI, Yuval. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, s. 490–510.

84. DEMONTIS, Ambra, MELIS, Marco, PINTOR, Maura, JAGIELSKI, Matthew, BIGGIO, Battista, OPREA, Alina, NITA-ROTARU, Cristina a ROLI, Fabio. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In: *28th USENIX Security Symposium USENIX Security 19)*. 2019, s. 321–338.

85. PAPERNOT, Nicolas, MCDANIEL, Patrick a GOODFELLOW, Ian. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*. 2016.

86. PODSCHWADT, Robert a TAKABI, Hassan. On Effectiveness of Adversarial Examples and Defenses for Malware Classification. In: *International Conference on Security and Privacy in Communication Systems*. Springer, 2019, s. 380–393.

87. XIA, Xiao-Ling, DING, Y., JIANG, J. a ZENG, R. Malware detection based on ontology. *2017 International Conference on Machine Learning and Cybernetics (ICMLC)*. 2017, roč. 1, s. 21–26.

88. DING, Yuxin, WU, Rui a ZHANG, Xiao. Ontology-based knowledge representation for malware individuals and families. *Computers & Security*. 2019, roč. 87, s. 101574.

89. JASIUL, Bartosz, ŚLIWA, Joanna, GLEBA, Kamil a SZPYRKA, Marcin. Identification of malware activities with rules. In: *2014 Federated Conference on Computer Science and Information Systems*. IEEE, 2014, s. 101–110.

90. FASANO, Fausto, MARTINELLI, Fabio, MERCALDO, Francesco, NARDONE, Vittoria a SANTONE, Antonella. Spyware Detection using Temporal Logic. In: *ICISSP*. 2019, s. 690–699.

91. RAZZAQ, Abdul, LATIF, K., AHMAD, H., HUR, Ali, ANWAR, Z. a BLOODSWORTH, Peter Charles. Semantic security against web application attacks. *Inf. Sci.* 2014, roč. 254, s. 19–38.

92. CARVALHO, R., GOLDSMITH, M. a CREESE, S. Investigating Malware Campaigns With Semantic Technologies. *IEEE Security Privacy*. 2019, roč. 17, č. 1, s. 43–54.

93. CHU, Ge a LISITSA, Alexei. Ontology-based Automation of Penetration Testing. In: *ICISSP*. 2020, s. 713–720.

94. SIKOS, Leslie F. OWL ontologies in cybersecurity: conceptual modeling of cyber-knowledge. *AI in Cybersecurity*. 2019, s. 1–17.

95. BROMANDER, Siri, JØSANG, Audun a EIAN, Martin. Semantic Cyberthreat Modelling. In: *STIDS*. 2016, s. 74–78.

96. MAVROEIDIS, Vasileios a BROMANDER, Siri. Cyber threat intelligence model: An evaluation of taxonomies, sharing standards, and ontologies within cyber threat intelligence. In: *2017 European Intelligence and Security Informatics Conference (EISIC)*. IEEE, 2017, s. 91–98.

97. SYED, Zareen, PADIA, Ankur, FININ, Tim, MATHEWS, Lisa a JOSHI, Anupam. UCO: A unified cybersecurity ontology. *UMBC Student Collection*. 2016.

98. GRÉGIO, André, BONACIN, Rodrigo, MARCHI, Antonio Carlos de, NABUCO, Olga Fernanda a GEUS, Paulo Lıcio de. An ontology of suspicious software behavior. *Applied Ontology*. 2016, roč. 11, č. 1, s. 29–49.

99. RASTOGI, Nidhi, DUTTA, Sharmishtha, ZAKI, Mohammed J, GITTENS, Alex a AGGARWAL, Charu. Malont: An ontology for malware threat intelligence. In: *International workshop on deployable machine learning for security defense*. Springer, 2020, s. 28–44.

100. ULICNY, Brian E, MOSKAL, Jakub J, KOKAR, Mieczyslaw M, ABE, Keith a SMITH, John Kei. Inference and ontologies. In: *Cyber Defense and Situational Awareness*. Springer, 2014, s. 167–199.

101. GRUBER, Thomas R. A translation approach to portable ontology specifications. *Knowledge acquisition*. 1993, roč. 5, č. 2, s. 199–220.

102. ROSSE, Cornelius a MEJINO JR, José LV. The foundational model of anatomy ontology. In: *Anatomy ontologies for bioinformatics: principles and practice*. Springer, 2008, s. 59–117.

103. ALEKSANDER, Suzi A, BALHOFF, James, CARBON, Seth, CHERRY, J Michael, DRABKIN, Harold J, EBERT, Dustin, FEUERMANN, Marc, GAUDET, Pascale, HARRIS, Nomi L et al. The gene ontology knowledgebase in 2023. *Genetics*. 2023, roč. 224, č. 1, iyad031.

104. RAIMOND, Yves, GIASSON, Frédérick, JACOBSON, Kurt, FAZEKAS, George, GANGLER, T a REINHARDT, Simon. Music ontology specification. *Specification Document (November 28, 2010), latest version http://purl. org/ontology/mo/(RDF/XML, Turtle)*. 2010.

105. STAAB, Steffen a STUDER, Rudi. *Handbook on ontologies*. Springer Science & Business Media, 2013.

106. BAADER, Franz, CALVANESE, Diego, MCGUINNESS, Deborah, PATEL-SCHNEIDER, Peter, NARDI, Daniele et al. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

107. RUDOLPH, Sebastian. Foundations of description logics. In: *Reasoning Web International Summer School*. Springer, 2011, s. 76–136.

108. LEHMANN, Jens. *Learning OWL class expressions*. Zv. 22. IOS Press, 2010.

109. BAADER, Franz, HORROCKS, Ian, LUTZ, Carsten a SATTLER, Uli. *An Introduction to Description Logic*. Cambridge University Press, 2017.

110. SCHMIDT-SCHAUSS, Manfred a SMOLKA, Gert. Attributive concept descriptions with complements. *Artificial intelligence*. 1991, roč. 48, č. 1, s. 1–26.

111. SCHILD, Klaus. *A correspondence theory for terminological logics: Preliminary report*. Citeseer, 1991.

112. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. [B.r.]. Dostupné tiež z: `https://www.w3.org/TR/owl2-overview/`. [Online; cit. 2022-12-03].

113. *OWL 2 Web Ontology Language Profiles*. [B.r.]. Dostupné tiež z: `https://www.w3.org/TR/owl2-profiles/`. [Online; cit. 2023-01-08].

114. KEET, Maria. *An introduction to ontology engineering*. Zv. 1. [B.r.].

115. PAPADIMITRIOU, Christos H. Computational complexity. In: *Encyclopedia of computer science*. 2003, s. 260–265.

116. LEHMANN, Jens a HITZLER, Pascal. Concept learning in description logics using refinement operators. *Machine Learning*. 2010, roč. 78, č. 1, s. 203–250.

117. FUNK, Maurice, JUNG, Jean Christoph, LUTZ, Carsten, PULCINI, Hadrien a WOLTER, Frank. Learning Description Logic Concepts: When can Positive and Negative Examples be Separated? In: KRAUS, Sarit (ed.). *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. 2019, s. 1682–1688. Dostupné z DOI: 10.24963/ijcai.2019/233.

118. D'AMATO, Claudia, FANIZZI, Nicola a ESPOSITO, Floriana. Inductive learning for the semantic web: what does it buy? *Semantic Web*. 2010, roč. 1, č. 1-2, s. 53–59.

119. LAVRAC, Nada a DZEROSKI, Saso. Inductive Logic Programming. In: *WLP*. Springer, 1994, s. 146–160.

120. BÜHMANN, Lorenz, LEHMANN, Jens a WESTPHAL, Patrick. DL-Learner—A framework for inductive learning on the Semantic Web. *Journal of Web Semantics*. 2016, roč. 39, s. 15–24.

121. BÜHMANN, Lorenz, LEHMANN, Jens, WESTPHAL, Patrick a BIN, Simon. Dl-learner structured machine learning on semantic web data. In: *Companion Proceedings of the The Web Conference 2018*. 2018, s. 467–471.

122. TRAN, An C, DIETRICH, Jens, GUESGEN, Hans W a MARSLAND, Stephen. An approach to parallel class expression learning. In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2012, s. 302–316.

123. TRAN, An C, DIETRICH, Jens, GUESGEN, Hans W a MARSLAND, Stephen. Parallel symmetric class expression learning. *The Journal of Machine Learning Research*. 2017, roč. 18, č. 1, s. 2145–2178.

124. KARP, Richard M. *Reducibility among combinatorial problems*. Springer, 2010.

125. QUINLAN, J. Ross. Learning logical definitions from relations. *Machine learning*. 1990, roč. 5, s. 239–266.

126. FANIZZI, Nicola, D'AMATO, Claudia a ESPOSITO, Floriana. DL-FOIL concept learning in description logics. In: *International Conference on Inductive Logic Programming*. Springer, 2008, s. 107–121.

127. FANIZZI, Nicola, RIZZO, Giuseppe, D'AMATO, Claudia a ESPOSITO, Floriana. DLFoil: Class expression learning revisited. In: *European Knowledge Acquisition Workshop*. Springer, 2018, s. 98–113.

128. RIZZO, Giuseppe, FANIZZI, Nicola a D'AMATO, Claudia. Class expression induction as concept space exploration: From DL-Foil to DL-Focl. *Future Generation Computer Systems*. 2020, roč. 108, s. 256–272.

129. RIZZO, Giuseppe, FANIZZI, Nicola, D'AMATO, Claudia a ESPOSITO, Floriana. A framework for tackling myopia in concept learning on the web of data. In: *Knowledge Engineering and Knowledge Management: 21st International Conference, EKAW 2018, Nancy, France, November 12-16, 2018, Proceedings 21*. Springer, 2018, s. 338–354.

130. GENDREAU, Michel, POTVIN, Jean-Yves et al. *Handbook of metaheuristics*. Zv. 2. Springer, 2010.

131. BOBILLO, Fernando a STRACCIA, Umberto. Fuzzy ontology representation using OWL 2. *International journal of approximate reasoning*. 2011, roč. 52, č. 7, s. 1073–1094.

132. STRACCIA, Umberto a MUCCI, Matteo. pFOIL-DL: Learning (fuzzy) EL concept descriptions from crisp OWL data using a probabilistic ensemble estimation. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, s. 345–352.

133. LISI, Francesca Alessandra a STRACCIA, Umberto. Dealing with Incompleteness and Vagueness in Inductive Logic Programming. In: *CILC*. Citeseer, 2013, s. 179–193.

134. CARDILLO, Franco Alberto a STRACCIA, Umberto. Fuzzy OWL-Boost: Learning fuzzy concept inclusions via real-valued boosting. *Fuzzy Sets Syst.* 2022, roč. 438, s. 164–186. Dostupné z DOI: 10.1016/j.fss.2021.07.002.

135. NOCK, Richard a NIELSEN, Frank. A Real generalization of discrete AdaBoost. *Artificial Intelligence*. 2007, roč. 171, č. 1, s. 25–41.

136. CARDILLO, Franco Alberto, DEBOLE, Franca a STRACCIA, Umberto. PN-OWL: A Two Stage Algorithm to Learn Fuzzy Concept Inclusions from OWL Ontologies. *arXiv preprint arXiv:2303.07192*. 2023.

137. AGARWAL, Ramesh a JOSHI, Mahesh V. PNrule: a new framework for learning classifier models in data mining (a case-study in network intrusion detection). In: *Proceedings of the 2001 SIAM International Conference on Data Mining*. SIAM, 2001, s. 1–17.

138. HEINDORF, Stefan, BLÜBAUM, Lukas, DÜSTERHUS, Nick, WERNER, Till, GOLANI, Varun Nandkumar, DEMIR, Caglar a NGOMO, Axel-Cyrille Ngonga. EvoLearner: Learning Description Logics with Evolutionary Algorithms. In: LAFOREST, Frédérique, TRONCY, Raphaël, SIMPERL, Elena, AGARWAL, Deepak, GIONIS, Aristides, HERMAN, Ivan a MÉDINI, Lionel (ed.). *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 2022, s. 818–828. Dostupné z DOI: 10.1145/3485447.3511925.

139. KOTSIANTIS, Sotiris B. Decision trees: a recent overview. *Artificial Intelligence Review*. 2013, roč. 39, s. 261–283.

140. BIAU, Gérard a SCORNET, Erwan. A random forest guided tour. *Test*. 2016, roč. 25, s. 197–227.

141. BORDES, Antoine, USUNIER, Nicolas, GARCIA-DURÁN, Alberto, WESTON, Jason a YAKHNENKO, Oksana. Translating Embeddings for Modeling Multi-relational Data. In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5–8, 2013, Lake Tahoe, Nevada, United States*. 2013, s. 2787–2795.

142. DE SOUSA RIBEIRO, Manuel a LEITE, João. Aligning Artificial Neural Networks and Ontologies towards Explainable AI. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2–9, 2021*. AAAI Press, 2021, s. 4932–4940.

143. LINARDATOS, Pantelis, PAPASTEFANOPOULOS, Vasilis a KOTSIANTIS, Sotiris. Explainable ai: A review of machine learning interpretability methods. *Entropy*. 2020, roč. 23, č. 1, s. 18.

144. RIBEIRO, Marco Túlio, SINGH, Sameer a GUESTRIN, Carlos. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In: KRISHNAPURAM, Balaji, SHAH, Mohak, SMOLA, Alexander J., AGGARWAL, Charu C., SHEN, Dou a RASTOGI, Rajeev (ed.). *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 2016, s. 1135–1144. Dostupné z DOI: 10.1145/2939672.2939778.

145. LUNDBERG, Scott M. a LEE, Su-In. A Unified Approach to Interpreting Model Predictions. In: GUYON, Isabelle, LUXBURG, Ulrike von, BENGIO, Samy, WALLACH, Hanna M., FERGUS, Rob, VISHWANATHAN, S. V. N. a GARNETT, Roman (ed.). *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 2017, s. 4765–4774.

146. VINAYAKUMAR, R., ALAZAB, Mamoun, SOMAN, K. P., POORNACHANDRAN, Prabaharan a VENKATRAMAN, Sitalakshmi. Robust Intelligent Malware Detection Using Deep Learning. *IEEE Access*. 2019, roč. 7, s. 46717–46738. Dostupné z DOI: 10.1109/ACCESS.2019.2906934.

147. LIU, Xinbo, LIN, Yaping, LI, He a ZHANG, Jiliang. A novel method for malware detection on ML-based visualization technique. *Comput. Secur.* 2020, roč. 89. Dostupné z DOI: 10.1016/j.cose.2019.101682.

148. GHOUTI, Lahouari a IMAM, Muhammad. Malware classification using compact image features and multiclass support vector machines. *IET Inf. Secur.* 2020, roč. 14, č. 4, s. 419–429. Dostupné z DOI: 10.1049/iet-ifs.2019.0189.

149. WESTPHAL, Patrick, BÜHMANN, Lorenz, BIN, Simon, JABEEN, Hajira a LEHMANN, Jens. SML-Bench–A benchmarking framework for structured machine learning. *Semantic Web*. 2019, roč. 10, č. 2, s. 231–245.

150. SARKER, Md Kamruzzaman a HITZLER, Pascal. Efficient concept induction for description logics. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2019, zv. 33, s. 3036–3043. Č. 01.

151. MITRE CORP. *Malware Attribute Enumeration and Characterization*. 2020. Dostupné tiež z: https://maecproject.github.io/. [Online; cit. 2023-03-04].

152. MOJŽIŠ, Ján a KENYERES, Martin. Interpretable Rules with a Simplified Data Representation-a Case Study with the EMBER Dataset. In: *Proceedings of the Computational Methods in Systems and Software*. Springer, 2023, s. 1–10.

153. MITRE CORP. *MAEC™ 5.0 Specification. Vocabularies*. 2017. Dostupné tiež z: https://maecproject.github.io/releases/5.0/MAEC_Vocabularies_Specification.pdf. [Online; cit. 2023-02-05].

154. SIKORSKI, Michael a HONIG, Andrew. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

155. KLEYMENOV, Alexey a THABET, Amr. *Mastering Malware Analysis: A malware analyst's practical guide to combating malicious software, APT, cybercrime, and IoT attacks*. Packt Publishing Ltd, 2022.

156. *Malware Initial Assessment*. [B.r.]. Dostupné tiež z: `https://www.winitor.com/`. [Online; cit. 2023-03-04].

157. REFAEILZADEH, Payam, TANG, Lei a LIU, Huan. Cross-Validation. In: LIU, Ling a ÖZSU, M. Tamer (ed.). *Encyclopedia of Database Systems*. Springer, 2009, s. 532–538. Dostupné z DOI: `10.1007/978-0-387-39940-9\_565`.

158. *Processor And CPU Time*. [B.r.]. Dostupné tiež z: `https://www.gnu.org/software/libc/manual/html_node/Processor-And-CPU-Time.html`. [Online; cit. 2023-17-15].

159. MAIMON, Oded a ROKACH, Lior. Data mining and knowledge discovery handbook. 2005.

160. FAWCETT, Tom. An introduction to ROC analysis. *Pattern recognition letters*. 2006, roč. 27, č. 8, s. 861–874.

161. OYAMA, Yoshihiro, MIYASHITA, Takumi a KOKUBO, Hirotaka. Identifying Useful Features for Malware Detection in the Ember Dataset. In: *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. 2019, s. 360–366. Dostupné z DOI: `10.1109/CANDARW.2019.00069`.

162. BIONDI, Fabrizio, ENESCU, Michael A, GIVEN-WILSON, Thomas, LEGAY, Axel, NOUREDDINE, Lamine a VERMA, Vivek. Effective, efficient, and robust packing detection and classification. *Computers & Security*. 2019, roč. 85, s. 436–451.

163. *Beware of malware Zusy! The notorious banking trojan*. [B.r.]. Dostupné tiež z: `https://reasonlabs.com/blog/beware-of-malware-zusy`. [Online; cit. 2023-09-23].

164. JUREČKOVÁ, Olha, JUREČEK, Martin a LÓRENCZ, Róbert. Classification and Online Clustering of Zero-Day Malware. *arXiv preprint arXiv:2305.00605*. 2023.

165. *Emerging Threat on FAREIT Resurgence*. [B.r.]. Dostupné tiež z: `https://success.trendmicro.com/dcx/s/solution/1118407-emerging-threat-on-fareit-resurgence`. [Online; cit. 2023-09-23].

166. *Rewterz Threat Alert – Vtflooder Trojan – Active IOCs*. [B.r.]. Dostupné tiež z: `https://www.rewterz.com/rewterz-news/rewterz-threat-alert-vtflooder-trojan-active-iocs/`. [Online; cit. 2023-09-23].

Prílohy

A Zdrojové kódy

- Zdrojové kódy softvéru DL-Learner, ktorý bol použitý v práci je dostupný v rámci repozitára:

  `https://github.com/mousetom-sk/DL-Learner`.

- Konfiguračné súbory k jednotlivým experimentom uvedeným v práci sú dostupné v rámci repozitára:

  `https://github.com/orbis-security/experiment-configurations`

- Ontológia navrhnutá v práci (vrátane datasetov), spolu so skriptami je dostupná v rámci repozitára:

  `https://github.com/orbis-security/pe-malware-ontology`

- Konceptové výrazy pre jednotlitvé rodiny malvéru (viď. kapitola 5.7) sú dostupné v rámci repozitára:

  `https://github.com/orbis-security/malware-families`

B Spúšťanie experimentov

DL-Learner je možné zostaviť pomocou nasledujúceho skriptu (nachádza sa koreňovom adresári repozitára):

```
./BuildRelease.sh
```

Po zostavení sa v priečinku `./interfaces/target` bude nachádza súbor `dllearner-1.4.1-SNAPSHOT.zip`, obsahujúci spustiteľnú verziu softvéru DL-Learner. Trénovací cyklus sa následne spustí pomocou príkazu v priečinku `/dllearner-1.4.1-SNAPSHOT/bin`:

```
./cli ./configuration.conf
```

Tento príkaz spustí trénovanie s konfiguračným súborom `configuration.conf` (alternatívne je možné spustiť súbor `cli.bat` na operačnom systéme Windows). Vo výpise B.1 môžeme vidieť ukážkový konfiguračný súbor. V samotnom konfiguračnom súbore môžeme vidieť okrem definovania použitej ontológie (`dataset_8_1000.owl`), rôzne hyperparametre a definíciu pozitívnych a negatívnych príkladov.

```
{
   prefixes = [ ("ex","https://orbis-security.com/pe-malware-ontology#") ]

   ks.type = "OWL File"
   ks.fileName = "dataset_8_1000.owl"

   alg.type = "ocel"

   alg.maxExecutionTimeInSeconds = 7200
   alg.noisePercentage = 10

   op.type="rho"
   op.useHasValueConstructor=true

   lp.positiveExamples = {
   "ex:f93aca5068966242dbeb17557f7413d8",
   "ex:dba5bfc7a80212f9e9005272551d68eb",
   ...
   }

   lp.negativeExamples = {
   "ex:37ba79f81f4a688587c2bcf5a3dd9118",
   "ex:9c078344875c6e99ff196d7b95b5ddc9",
   ...
   }
}
```

Výpis B.1: Ukážka konfiguračného súboru pre DL-Learner.