VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

INKREMENTÁLNÍ STATICKÁ ANALÝZA PRO JAZYK YARA

INCREMENTAL PARSING FOR YARA LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VOJTĚCH DVOŘÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. DOMINIKA REGÉCIOVÁ**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

145037

Ústav: Ústav informačních systémů (UIFS)
Student: **Dvořák Vojtěch**
Program: Informační technologie
Specializace: Informační technologie
Název: **Inkrementální statická analýza pro jazyk YARA**
Kategorie: Bezpečnost
Akademický rok: 2022/23

Zadání:

1. Nastudujte si teorii inkrementálních parserů a jejich použití pro analýzu zdrojových kódů v textových editorech a integrovaných vývojových prostředích (IDE).
2. Seznamte se s jazykem YARA (https://yara.readthedocs.io/en/stable/) a nástroji pro jeho statickou analýzu. Zaměřte se na projekt YLS (https://github.com/avast/yls), který poskytuje server protokolu LSP (Language Server Protocol).
3. Navrhněte knihovnu, která umožní statickou analýzu jazyka YARA pro účely použití ve vývojových prostředích za využití algoritmu pro inkrementální parsování.
4. Implementujte vámi navrženou knihovnu z předešlého bodu. Proveďte integraci s projektem YLS, kde dojde k nahrazení řešení statických analýz bez inkrementálního parsování za řešení s použitím vaši knihovny.
5. Vaše řešení otestujte vůči extenzivní sadě pravidel v jazyku YARA a porovnejte s řešeními bez inkrementálního parsování.

Literatura:

- Szor: The Art of Computer Virus Research and Defense, Addison-Wesley Professional (2005), ISBN 978-0321304544
- Recorded Future: The Threat Intelligence Handbook, CyberEdge Group (2018), ISBN 978-0999035467
- Tim A. Wagner and Susan L. Graham.: Efficient and flexible incremental parsing. (1998)
- Interní dokumentace společnosti Avast
- Dále dle doporučení vedoucího či konzultanta

Při obhajobě semestrální části projektu je požadováno:
První 3 body a začátek 4.

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

Vedoucí práce: **Regéciová Dominika, Ing.**
Konzultant: Ing. Marek Milkovič
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 26.10.2022

Abstrakt

Hlavním cílem této bakalářské práce je navrhnout a implementovat programovou knihovnu, jež umožní inkrementální statickou analýzu jazyka YARA. Jedním z hlavních účelů této nové knihovny je integrace s open-source projektem Yara Language Server, který vyvíjí firma Avast. Oproti dosavadnímu řešení, jež využívá neinkrementální přístup k analýze, by mělo dojít ke snížení nároků na strojový čas. Kromě informací o programovém řešení je součástí této práce rovněž souhrn teorie zaměřující se na statickou analýzu a její inkrementální variantu, zásadní informace o nástroji YARA a také seznámení s dosavadním řešením, s knihovnou Yaramod-v3. Dále je v práci zahrnuto také srovnání nové knihovny s tímto dosavadním řešením, v němž jsou prezentovány dosažené výsledky. Provedené experimenty ukázaly, že nová knihovna je schopná provést inkrementální analýzu modifikované sady pravidel přibližně $20\times - 2000\times$ rychleji v závislosti na konkrétní sadě.

Abstract

The main goal of this bachelor thesis is to design and implement a program library that enables incremental static analysis of the YARA language. One of the main purposes of this new library is to integrate with the open-source Yara Language Server project developed by Avast. Compared to the existing solution, which uses a non-incremental approach to analysis, the machine time requirements should be reduced. In addition to information about the software solution, this thesis also includes a summary of the theory focusing on static analysis and its incremental variant, essential information about the YARA tool, and an introduction to the existing solution, the Yaramod-v3 library. The thesis also contains a comparison of the new library with the current solution, in which the achieved results are presented. The experiments performed showed that the new library is able to perform incremental analysis of a modified rule set approximately $20\times - 2000\times$ faster depending on the particular set.

Klíčová slova

inkrementální statická analýza, jazyk YARA, integrované vývojové prostředí, IDE, YLS, LSP, Yaramod, LR syntaktická analýza, sémantická analýza, lexikální analýza, textový editor, tree-sitter, visitor, linting, Avast

Keywords

incremental static analysis, YARA, integrated development environment, IDE, YLS, LSP, Yaramod, LR parsing, semantic analysis, lexical analysis, text editor, tree-sitter, visitor, linting, Avast

Citace

DVOŘÁK, Vojtěch. *Inkrementální statická analýza pro jazyk YARA*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Dominika Regéciová

Inkrementální statická analýza pro jazyk YARA

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod odborným vedením Ing. Dominiky Regéciové. Další informace mi poskytli Ing. Marek Milkovič a Ing. Matej Kašťák. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Vojtěch Dvořák
8. května 2023

Poděkování

Tímto bych chtěl poděkovat Ing. Dominice Regéciové za skvělé vedení této bakalářské práce. Dále velice děkuji také konzultantům, Ing. Markovi Milkovičovi a Ing. Matejovi Kašťákovi, za cenné rady a připomínky, které mi poskytli.

Obsah

Úvod **3**

1 Inkrementální statická analýza kódu **4**
1.1 Statická analýza kódu 4
1.2 Inkrementální provedení statické analýzy 13

2 Jazyk YARA **24**
2.1 Obecný formát zdrojových souborů 25
2.2 Modularita jazyka YARA 28
2.3 YARA ve firmě Avast 28

3 Dosavadní řešení **30**
3.1 Syntaktická a sémantická analýza pravidel 30
3.2 Tvorba nových pravidel 31
3.3 Zpřístupnění výsledků analýzy 31
3.4 Modifikace existujících pravidel 32
3.5 Aplikace dosavadního řešení 32

4 Tvorba knihovny pro inkrementální statickou analýzu **33**
4.1 Specifikace požadavků 33
4.2 Návrh knihovny a jejího rozhraní 34
4.3 Implementace 41
4.4 Integrace s projektem Yara Language Server 53
4.5 Testování 54
4.6 Příklady použití 55
4.7 Metriky kódu 56

5 Porovnání dosažených výsledků s dosavadním řešením **57**
5.1 Yaramod-v4 57
5.2 Yara Language Server 58

Závěr **61**

Literatura **62**

A Snímky obrazovky **66**

B Obsah přiloženého paměťového média **69**

Seznam obrázků

1.1 Komunikace textového editoru s language serverem 9
1.2 Obecné schéma statického analyzátoru 10
1.3 Průběh inkrementální statické analýzy založené na knihovně tree-sitter . . . 15
1.4 Změny v AST během inkrementální lexikální a syntaktické analýzy 17
1.5 Průběh inkrementální lexikální analýzy 18

4.1 Návrh architektury knihovny Yaramod-v4 35
4.2 Třídní diagram modelující řídící rozhraní knihovny a s ním související třídy 37
4.3 Třídní diagram sémantického rozhraní knihovny Yaramod-v4 39
4.4 Derivační strom vytvořený knihovnou tree-sitter na základě gramatiky pro jazyk YARA 47
4.5 Proces aktualizace AST během inkrementální statické analýzy v knihovně Yaramod-v4 51
4.6 Proces aktualizace výstupní reprezentace kódu během inkrementální statické analýzy v knihovně Yaramod-v4 52

A.1 Ukázka funkcionality Linting a Document Symbol 66
A.2 Ukázka syntaktických kontrol 67
A.3 Ukázka typových kontrol 67
A.4 Kontrola duplicitních identifikátorů 67
A.5 Ukázka funkcionality Completion 68

Úvod

Cílem této práce je vytvořit programovou knihovnu, která by umožnila inkrementální statickou analýzu jazyka YARA, jenž je dnes hojně používán pro popis a klasifikaci malwaru (blíže popsán v kapitole 2). Podstatou inkrementálního přístupu k analýze je využívání datových struktur vytvořených v rámci předchozího běhu analýzy. Díky tomu je možné analyzovat pouze ty části kódu, u kterých je to zapotřebí.

Existující statické analyzátory jazyka YARA tento přístup nepodporují, což je činí méně vhodnými pro použití v integrovaných vývojových prostředích (IDE) a textových editorech. Kvůli konvenčnímu přístupu analyzují kód kompletně znovu po každé editaci, a to i v případě, že byla modifikována pouze jeho malá část. Právě v textových editorech a IDE má však statická analýza široké uplatnění, neboť pomáhá programátorům tvořit bezpečnější kód a lépe ho pochopit. Pro uživatele to v důsledku znamená vyšší kvalitu a redukci ceny softwaru. Bližší informace o statické analýze, o jejích aplikací a její inkrementální formě jsou uvedeny v kapitole 1.

Tato práce vznikala ve spolupráci s firmou Avast, která pro tvorbu pravidel v jazyce YARA používá vlastní vývojové prostředí YDE (Yara Development Environment), jehož součástí je server YLS (Yara Language Server). Ten prostřednictvím protokolu LSP (Language Server Protocol) komunikuje s textovými editory či IDE a dodává jim tak funkcionalitu statických analyzátorů pro snazší návrh a pochopení pravidel. Tento server je implementován s využitím knihovny Yaramod-v3 (viz kapitola 3), jež je rovněž vyvíjena společností Avast. Tato knihovna provádí syntaktickou a sémantickou analýzu pravidel v jazyce YARA, čímž kontroluje jejich validitu a skrze aplikační programové rozhraní je k nim možné přistupovat. Toho lze využít pro pokročilejší statickou analýzu zdrojových souborů.

Knihovna, jež je hlavním praktickým výstupem této práce, bude umožňovat provádět statickou analýzu podobným způsobem jako Yaramod-v3. Od tohoto dosavadního řešení ji však bude odlišovat podpora inkrementálního přístupu. Jelikož tato knihovna v budoucnu nahradí stávající řešení, bude poskytovat s ním podobné rozhraní, aby bylo možné snadno provést její integraci s již existujícími projekty. Dále bude odolná vůči chybám ve zdrojovém kódu. Informace o návrhu a implementaci knihovny jsou dostupné v kapitole 4. Porovnání nového řešení s knihovnou Yaramod-v3 je dostupné v kapitole 5.

Stejně jako u projektů YLS a Yaramod-v3, tak i v případě této nové knihovny se v budoucnu bude jednat o open-source software. Díky tomu má tato práce potenciál poskytnout vylepšený způsob statické analýzy pravidel v jazyce YARA nejen společnosti Avast, ale také všem uživatelům tohoto jazyka na celém světě.

Kapitola 1

Inkrementální statická analýza kódu

První statické analyzátory zdrojových kódů se začaly objevovat již v šedesátých letech minulého století jakožto způsob, jak optimalizovat překladače a kód, který generují. Později nalezly uplatnění také v rámci integrovaných vývojových prostředí pro hledání chyb a poskytování další podpory pro tvorbu zdrojových kódů [35, str. 1 – 3].

Pro automatizovanou statickou analýzu zdrojového kódu však musí být kód nejprve převeden do podoby, která je pro jeho další zpracování vhodná. Typicky se jedná o abstraktní syntaktický strom, jenž je vytvořen syntaktickým analyzátorem na základě vstupního řetězce a pravidel gramatiky daného jazyka. Zejména tento proces, ale i jiné principy statické analýzy vycházejí z principů využívaných právě v kompilátorech [9, str. 72].

Pro účely použití v textových editorech a integrovaných vývojových prostředích je vhodné, aby statická analýza byla prováděna v době editace kódu. To ale přináší další výzvy v podobě zvýšených nároků na rychlost provádění analýzy, a navíc je zapotřebí kód reanalyzovat při každé jeho modifikaci. Jedno z možných řešení je provádět syntaktickou analýzu inkrementálně, tedy za použití již vytvořených datových struktur. Tento přístup výrazně šetří strojový čas, neboť není nutné znovu analyzovat části zdrojového kódu, jež zůstaly nezměněny [41].

V této kapitole jsou shrnuty obecné informace o inkrementální lexikální, syntaktické a sémantické analýze, jelikož tyto metody jsou předpokladem pro provádění inkrementální statické analýzy. Obecně o tomto způsobu zkoumání vlastností kódu pojednává začátek této kapitoly. V jejím závěru je shrnut význam inkrementální statické analýzy formálních jazyků v integrovaných vývojových prostředích a textových editorech.

Použití těchto metod pro podporu tvorby zdrojových souborů v jazyce YARA je ostatně jednou z hlavních motivací pro tvorbu nové programové knihovny, jež je v rámci této práce prezentována.

1.1 Statická analýza kódu

Pojmem statická analýza kódu rozumíme podle studie od P. Emanuelssona *„automatizované metody, jak zkoumat běhové vlastnosti kódu bez jeho provádění“* [14, str. 7].

Jinými slovy, se jedná o určování stavů programu, které mohou nastat v době jeho běhu, pouze na základě znalosti zdrojového kódu [4, str. 9]. Tento proces je obvykle prováděn prostřednictvím jeho vhodné *interní reprezentace* neboli *modelu programu*. Díky němu můžeme

vytvářet abstrakce stavů, jež sice nenesou všechny informace o jejich reálných ekvivalentech, ale díky tomu je snazší s nimi manipulovat a korektně je popsat [15, str. 1].

Většinou se tato metoda analýzy kódu zaměřuje na takové vlastnosti, které mohou v případě chyb vyústit v násilné ukončení programu či navrácení neplatných výsledků. Související publikace do třídy chyb odhalovaných statickou analýzou často neřadí syntaktické chyby a v některých případech ani jednoduché sémantické chyby (např. typové chyby) [14, str. 7].

V rámci této práce však budeme považovat hledání syntaktických a sémantických chyb rovněž za zodpovědnost statického analyzátoru. Důvodem je, že nezbytnou součástí většiny statických analyzátorů je právě také převod zdrojového kódu do vhodné interní reprezentace (např. prostřednictvím lexikální a syntaktické analýzy). Při této konverzi často probíhají i syntaktické a sémantické kontroly. Z pohledu uživatele tedy dochází k hledání těchto chyb a statický analyzátor by měl zajistit, aby o nich byl uživatel informován.

Komplementární technikou ke statické analýze je tzv. dynamická analýza. Ta zjišťuje vlastnosti kódu jeho prováděním. Jedná se například o testování a profilování. U tohoto typu analýzy není zapotřebí vytvářet žádné abstrakce programu, protože zkoumáme skutečné stavy, které během jeho provádění nastávají [15, str. 1].

Ačkoliv dynamická analýza poskytuje přesnější údaje, není její provádění nikterak zobecněno tak, aby byly brány v potaz i běhy s jinými vstupními proměnnými. Navzdory tomu mohou být tyto proměnné použity v budoucnu, čímž se program dostává do neotestovaného, potenciálně nebezpečného stavu. Kvalita testování se tedy významně odvíjí od kvality testovací sady. Statická analýza naproti tomu produkuje kvůli abstrakci méně přesný, více zobecněný popis stavu, jenž je však platný pro více konkrétních stavů programu. Kvalita statické analýzy kódu právě závisí na zvoleném způsobu abstrakce stavů [15, str. 1 – 2].

Je důležité mít na paměti, že ani na základě jednoho výše uvedeného typu analýzy nelze vyloučit přítomnost chyb v kódu. Díky statické analýze kódu můžeme ale zredukovat potřebu jeho testování a odhalit některé artefakty, jež jsou testováním a dynamickou analýzou obecně těžko odhalitelné [14, str. 7].

Podle studie od P. Emanuelssona [14] se může jednat například o identifikaci a eliminaci tzv. *mrtvého kódu*[1] nebo o identifikaci některých nedovolených operací (dělení nulou, přístup k prvku za hranicemi pole...).

1.1.1 Funkce statické analýzy

Kromě hledání artefaktů ve zdrojovém kódu uvedených v předchozí kapitole nám metody statické analýzy mohou poskytnout další užitečné funkce, které zlepšují kvalitu kódu a šetří programátorům čas. Například kniha Secure Programming with Static Analysis uvádí následující funkce [9, str. 24]:

- **Kontroly datových typů** – Kontrola, zda datový typ hodnoty nebo proměnné odpovídá očekávanému datovému typu (např. při přiřazování, ve výrazech...). Do jaké míry je schopen tuto kontrolu provést statický analyzátor, záleží na konkrétním jazyce, pro který je určen. Statické analyzátory s touto funkcí jsou hojně využívány v překladačích při statických typových kontrolách [1, str. 387].
- **Kontrola stylu** – Statický analyzátor se snaží minimalizovat přítomnost některých pochybných programátorských konstrukcí, které by mohly mít negativní vliv na čitelnost a udržovatelnost kódu.

[1] Tj. část programu, jež nebude za žádných okolností prováděna nebo je prováděna zbytečně.

- **Podpora pro snazší pochopení programu** – Tato funkcionalita pomáhá uživatelům mít přehled ve zdrojových kódech, což je také důvod, proč je alespoň částečně implementována ve většině moderních integrovaných vývojových prostředích. Jejím typickým příkladem je navigace na deklaraci či definici funkce.
- **Verifikace programu** – Při formální verifikaci programu je statickým analyzátorem prováděna kontrola programu vůči jeho specifikaci.
- **Hledání chybových idiomů** – Jedná se o identifikaci částí kódů, které často vedou k chybě a které mohou vykazovat chování, jež programátor pravděpodobně neočekává.
- **Kontrola bezpečnosti kódu** – Může se jednat například o identifikaci potenciálního přístupu mimo alokovanou paměť a jiných podobných zranitelností programu.

Funkcionalita poskytovaná statickými analyzátory se liší dle konkrétní aplikace. Na základě jejich konkrétního použití můžeme statické analyzátory a jejich metody rozdělit do tří kategorií [35, str. 2]:

1. **Statická analýza pro optimalizaci programu** – Detekce mrtvého kódu, detekce hodnot, jež mohou být vyhodnoceny v době překladu…
2. **Statická analýza pro korektnost programu** – Detekce přetečení, dereference ukazatele na adresu `NULL` v jazyce C…
3. **Statická analýza pro vývoj programu** – Jedná se hlavně o podporu pro snazší pochopení programu a jeho případnou refaktorizaci. Tento druh statické analýzy nalézá uplatnění v moderních integrovaných vývojových prostředích a textových editorech.

1.1.2 Nástroje pro statickou analýzu

V současné době existuje celá řada nástrojů pro statickou analýzu kódu. Některé z nich jsou přímo určeny pro integrovaná vývojová prostředí, další jsou zase distribuovány ve formě samostatných nástrojů (např. s CLI či vlastním GUI) a některé kombinují oba přístupy.

Poskytovaná funkcionalita se velmi liší stejně jako jejich jiné parametry (rychlost provádění analýzy, způsob prezentace výsledků analýzy…) v závislosti na jejich konkrétním účelu. Například analyzátory určené pro vývojová prostředí poskytují zpravidla méně komplexní funkcionalitu za cenu rychlejší analýzy a okamžité zpětné vazby uživateli.

Příkladem statických analyzátorů jsou *FindBugs* pro jazyk Java [22], *Pylint* pro jazyk Python [29] či nástroj *Splint* [16] určený pro C. Statické analyzátory najdeme také v kompilátorech, které je využívají pro optimalizaci kódu a hledání některých chyb (GCC[2], Clang[3]…).

Za zmínku stojí také jeden z prvních statických analyzátorů, program *Lint* [23] pro jazyk C. Funkcionalita, kterou tento nástroj poskytuje (kontrola stylu, hledání chybových idiomů, typové kontroly…), se na základě jeho názvu označuje jako tzv. *linting*. Tento pojem je dodnes hojně používán jako souhrnný název pro tyto základní funkce statických analyzátorů.

[2] Manuálová stránka s možnostmi nastavení analyzátoru v překladači GCC: https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc/Static-Analyzer-Options.html

[3] Manuálová stránka věnovaná statickému analyzátoru překladače Clang: https://clang.llvm.org/docs/ClangStaticAnalyzer.html

Zajímavou oblastí, kde se statická analýza kódu rovněž uplatňuje, jsou doménově specifické jazyky. Setkáváme se zde jednak s běžnou funkcionalitou jiných statických analyzátorů, ale také se specifickými funkcemi, jež pomáhají řešit problémy dané domény. Například článek [10] prezentuje rámcové řešení pro analýzu zdrojových kódů databázových aplikací, které umožňuje detekovat zranitelnost typu *SQL injection*. Do této oblasti také ostatně spadá i knihovna vytvořená v rámci této práce, jelikož provádí analýzu doménově specifického jazyka YARA.

Další exemplář statického analyzátoru používá technologie *eBPF*. Ta umožňuje rozšiřovat funkce jádra operačního systému Linux pomocí uživatelsky definovaných rozšíření, čehož se využívá např. při zpracování paketů nebo pro monitorování systému. Rozšíření jsou spouštěna v privilegovaném režimu přímo na CPU, a proto musí před jejich nahráním do jádra OS proběhnout jejich precizní statická analýza [18].

Kromě poskytované funkcionality jsou dalšími zásadními parametry statických analyzátorů *rychlost* provádění analýzy, její *přesnost* (precision), *spolehlivost* (soundness), *hloubka* (depth) a *uživatelská přívětivost*.

Zatímco přesnost udává, do jaké míry dokáže analyzátor přesně odhadnout chování programu, spolehlivost popisuje, zda je analyzátor schopen postihnout všechny jeho potenciální stavy a možná chování. Velmi spolehlivé analyzátory často do množiny těchto stavů zahrnují i ty, které během provádění programu nenastanou. Tím eliminují chyby druhého typu (false negatives) za cenu většího výskytu chyb prvního typu (false positives), což je z hlediska bezpečnosti analyzovaného programu výhodné [32, str. 22].

Hloubka, do které je analýza prováděna, popisuje, jak velké celky zdrojového programu jsou v rámci analýzy v jednom okamžiku zkoumány. Analýza může být prováděna v rámci celých projektů nebo v rámci jednotlivých souborů či pouze řádků. Omezení hloubky má sice za následek zvýšení rychlosti, ale také ztrátu kontextu, a proto může dojít k omezení funkcionality. Nemusí být pak možné např. zkoumat platnost proměnných definovaných mimo aktuální modul a podobně [9, str. 38].

Podobně protichůdné vlastnosti jsou přesnost a rychlost, neboť přesnější analýza vyžaduje více času. Z tohoto důvodu se u statických analyzátorů setkáváme s různými kompromisy, kdy na jedné straně existují velmi pomalé, ale precizní nástroje, schopné provádět analýzu nad celým projektem. Na druhé straně jsou však jednoduché, velmi rychlé analyzátory poskytující pouze základní funkcionalitu [9, str. 39]. To je také případ analyzátorů v integrovaných vývojových prostředích a textových editorech (viz 1.1.3). Ať už je ale kompromis mezi rychlostí a jinými parametry jakýkoliv, vždy je snaha zachovat co největší spolehlivost [32, str. 23].

Neméně důležitá je zmíněná uživatelská přívětivost. Statický analyzátor poskytuje pouze hlášení o výsledcích analýzy. Za změny v kódu jako takové je zodpovědný uživatel, který tento nástroj používá. Kvůli tomu je zapotřebí, aby byl prostřednictvím něj dobře informován a byl schopen kód vylepšit na základě těchto hlášení [9, str. 41].

Na to, jak jsou nástroje pro statickou analýzu v dnešní době reálně používané, odpovídá uživatelsky orientovaný průzkum z března 2022 [11]. Naprostá většina (78,3 %) dotázaných vývojářů v něm udává, že je používají, jelikož jim umožňují tvořit lepší kód. Menší část z nich (30,4 %) uvedla jako jeden z důvodu jejich používání firemní standardy, které to přímo vyžadují. I sami vývojáři tedy zjevně oceňují podporu, kterou jim tento typ analýzy kódu poskytuje [11, str. 6].

Z průzkumu také vyplývá, že pro statickou analýzu jsou používány jak nástroje zabudované v integrovaných vývojových prostředích, tak samostatně distribuované analyzátory. Dotázaní dále vypověděli, že ideální je, aby výsledky analýzy byly prezentovány přímo v pro-

středí, v němž kód editují. Nejčastější artefakty, jež jsou statickou analýzou detekovány, se týkají programovacího stylu, bezpečnosti kódu a chyb, které by mohly mít negativní vliv na funkčnost programu [11, str. 4 – 5].

Zajímavá fakta prezentuje také článek [5], jenž se zaměřuje na další, neméně podstatný aspekt této problematiky, a sice na její ekonomickou stránku. Tento průzkum ukázal, že i přes další režii a náklady spojené s používání statických analyzátorů, lze celkově díky nim dosáhnout redukce ceny softwaru v průměru o 17 %.

Není tedy pochyb o tom, že téma statické analýzy a její integrace do textových editorů a integrovaných vývojových prostředí je v dnešní době aktuální téma. To potvrzuje také fakt, že provádění statické analýzy bezpečnosti kódu (Static Analysis Security Testing neboli SAST) je součástí životního cyklu bezpečného vývoje software navrženého společností Microsoft (Microsoft Security Development Lifecycle zkráceně MSDL) [28].

1.1.3 Integrovaná vývojová prostředí a textové editory

Přestože někdy bývají pojmy integrované vývojové prostředí a textový editor zaměňovány, existují mezi nimi rozdíly.

Integrované vývojové prostředí (IDE) není pouze program pro editaci kódu, ale obsahuje také další vestavěnou funkcionalitu jako např. kompilátor (resp. interpret) kódu nebo ladící nástroje. Nástroje pro statickou analýzu kódu mohou být právě součástí této vestavěné funkcionality. Může se jednat o nástroj pro zvýrazňování syntaxe či syntaktických chyb během editace, statický analyzátor v kompilátoru nebo samostatný nástroj pro statickou analýzu. Typickými příklady IDE jsou *NetBeans*, *Eclipse* a *Visual Studio*.

Textové editory (editory kódu) jsou oproti tomu méně komplexní nástroje, zaměřené pouze na editaci kódu. Neposkytují sice v základu uživatelům tak široké spektrum služeb, díky tomu ale také nekladou tak vysoké nároky na výkon hostitelského systému jako IDE. Příklady textových editorů jsou *VSCode*, *Sublime Text* a *Vim* [34].

Pokud nějakou dodatečnou funkcionalitu tyto nástroje obsahují, tak jde většinou o základní podporu pro editaci kódu. Tou je zvýrazňování syntaxe, syntaktických chyb, automatické doplňování znaků, navigace v kódu apod. Pro tyto funkce se někdy používá obecné označení *intellisense*[4] (i v případě některých IDE).

Moderní editory kódu a IDE umožňují svoji funkcionalitu obohacovat prostřednictvím rozšíření. Editory kódu tak díky nim mohou obsahovat i nástroje, které jsou nabízeny v integrovaných vývojových prostředích. Může se jednat právě o statické analyzátory.

I když nemusí jít o statickou analýzu v pravém slova smyslu a může se jednat např. pouze o zvýrazňování syntaxe, tak i v tomto případě se jedná o proces, který zahrnuje mimo jiné také tvorbu modelu programu (viz 1.1.4). Ačkoliv by bylo možné implementovat tvorbu modelu programu v rámci každého rozšíření pro jednotlivé jazyky a jednotlivá IDE (respektive textové editory), bylo by to velmi náročné.

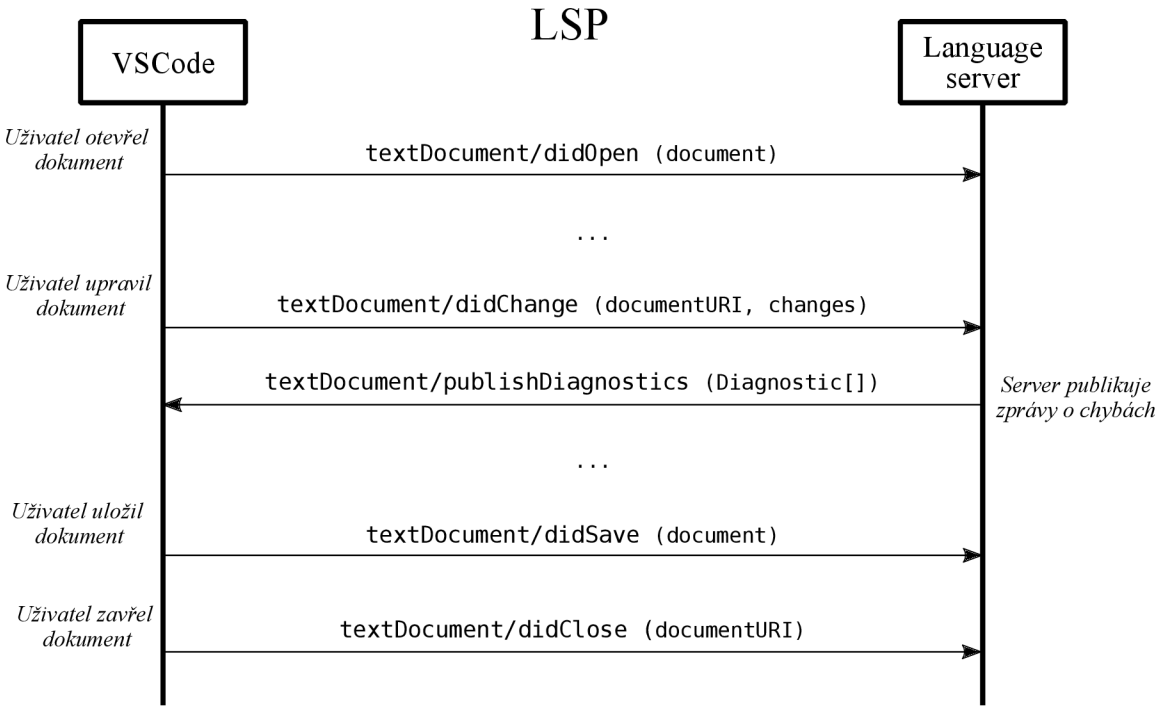
Řešením jsou tzv. *language servery*. Jedná se o samostatné aplikace, běžící v rámci vlastního procesu v OS. Díky tomu je možné, aby byly implementovány v libovolném jazyce a s využitím libovolných knihoven, které poskytují rozhraní právě pro tvorbu modelu programu a práci s ním nebo přímo implementují některou ze statických analýz kódu. S textovým editorem nebo s IDE pak komunikují pomocí protokolu *Language Server Protocol* (LSP) [26] založeném na JSON-RPC. Díky němu lze language servery propojit se všemi

[4] Dokumentace funkcí intellisense v textovém editoru VSCode: https://code.visualstudio.com/docs/editor/intellisense#_intellisense-features

IDE, resp. textovými editory, které tento protokol podporují a není tak nutné implementovat rozšíření pro každé IDE zvlášť.

Příklad komunikace textového editoru a language serveru znázorňuje sekvenční diagram na obrázku 1.1. Jak je v diagramu znázorněno, komunikace mezi textovým editorem a language serverem probíhá oběma směry. Textový editor zasílá language serveru např. notifikace o otevření dokumentu, uložení, zavření nebo o jeho editaci. Oznámení o editaci mohou být zasílána s různou granularitou, tzn. po přidání nebo odebrání jednoho znaku, ale také po několika úpravách na různých místech v dokumentu. Language server zasílá editoru notifikace, v rámci kterých např. publikuje varování a zprávy o chybách, na což může textový editor zareagovat jejich vizualizací ve svém uživatelském rozhraní.

Ačkoliv jsou pro tuto práci zásadní zejména výše uvedené druhy notifikací, LSP definuje i další typy zpráv, pomocí kterých může být realizována také další funkcionalita (např. navigace na definice symbolů).

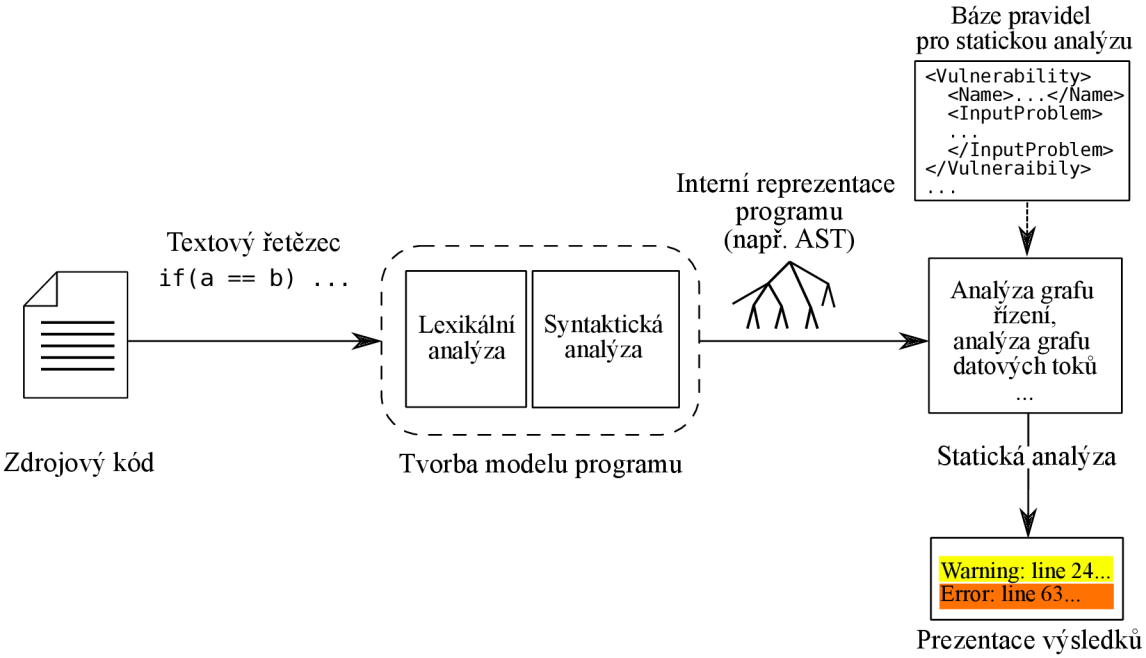


Obrázek 1.1: Komunikace textového editoru (zde VSCode) s language serverem (zjednodušená verze diagramu dostupného v [26]) – Zprávy mají svůj typ neboli metodu (např. `didSave`) a parametry (uvedeny v závorce). Textový editor nebo IDE může tímto způsobem komunikovat s více language servery, z nichž je každý specializovaný pro konkrétní jazyk.

Příkladem language serveru je *Yara Language Server* (YLS), který využívá knihovnu Yaramod-v3 (viz 3) a díky podpoře LSP je možné jej propojit s vícero textovými editory (VSCode, Vim/NeoVim, Emacs...) [25]. Oba dva projekty jsou v rámci této práce důkladněji popsány v kapitolách 2.3 a 3.

1.1.4 Průběh statické analýzy

Ačkoliv se průběh provádění statických analyzátorů může lišit v závislosti na konkrétní implementaci, obecně jej můžeme znázornit následujícím schématem:



Obrázek 1.2: Obecné schéma statického analyzátoru (sestaveno na základě [9, 37])

Fáze zpracování zdrojového kódu uvedené ve schématu mohou být různě sofistikované a v některých případech může být některá z fází vynechána. Vždy záleží na konkrétních potřebách statického analyzátoru a na jazyce, pro který je určen. Běžně je zdrojový kód analyzován v těchto třech základních krocích:

Tvorba modelu programu

První fází je převod zdrojového kódu do interní reprezentace neboli tzv. tvorba modelu programu. Tato fáze může být různě komplexní v závislosti na potřebách daného statického analyzátoru.

Jelikož pro další zpracování není zdrojový kód ve své surové formě (textový řetězec) vhodný, je převeden do vhodnější, interní reprezentace. Touto interní reprezentací je zpravidla *abstraktní syntaktický strom* (AST) vycházející ze simulace konstrukce *derivačního stromu* (parse tree). Termínem derivační strom neformálně rozumíme „*strom, který znázorňuje derivaci řetězce na základě formální gramatiky daného jazyka*“ [1, str. 45]. Úplnou formální definici tohoto pojmu najdeme například v knize Compilers: Principles, Techniques, and Tools [1, str. 45].

Abstraktní syntaktický strom na rozdíl od derivačního stromu nemusí odrážet přesnou syntaktickou strukturu programu, kterou udává příslušná gramatika, a některé detaily tak v něm mohou být vynechány (znaménka operátorů, závorky apod.) [1, str. 69].

Simulace konstrukce derivačního stromu je vytvářena *syntaktickým analyzátorem* implementovaným na základě gramatiky příslušného formálního jazyka. Syntaktické analýze však musí předcházet *lexikální analýza*, která vstupní řetězec rozdělí na proud tzv. *tokenů*. Jedná se o struktury obsahující název nebo jiné symbolické označení dané lexikální jednotky. Volitelně mohou mít také jiné atributy (např. pozici v dokumentu) [1, str. 111]. Zpracování vstupního řetězce pomocí syntaktické a lexikální analýzy může být prováděno v rámci statické analýzy kódu stejným způsobem v kompilátorech [1, str. 39].

Některé jednodušší statické analyzátory nevyžadují tak komplexní interní reprezentaci, jakou je strom. Pokud je cílem statické analýzy např. identifikovat volání některých potenciálně nebezpečných funkcí, je proud tokenů jako interní reprezentace dostačující. Tato volání v něm mohou být totiž snadno nalezena a je tedy zapotřebí pouze lexikálního analyzátoru [9, str. 103].

Velmi přímočarý způsob provádění statické analýzy, který nevyžaduje žádnou konverzi zdrojového kódu do interní reprezentace, je uveden v článku [8]. V principu se jedná o vyhledávání potenciálně nebezpečných konstrukcí daného programovacího jazyka pomocí unixového nástroje *grep*. Ten ve zdrojových souborech vyhledává dané problematické vzory, popsané regulárními výrazy.

Pro jiné, pokročilejší typy statických analyzátorů je však zapotřebí vytvořit před dalším zpracováním zdrojového kódu AST. Přestože je možné použít pro tento účel různé druhy syntaktických analyzátorů, výhodné je vybrat *LR syntaktický analyzátor*, konkrétně pak *LR(1) syntaktický analyzátor* (canonical parser). Ten totiž dokáže zpracovávat všechny programovací jazyky, pro které je možné definovat bezkontextovou gramatiku. Množina jazyků zpracovávaných LL metodami je navíc méně početná než množina jazyků, již je možné zpracovat LR syntaktickými analyzátory [1, str. 241].

Kromě toho můžeme tyto syntaktické analyzátory vygenerovat pomocí řady dostupných generátorů. Příkladem je desetiletími ověřený *GNU Bison* [12], který si navíc dokáže poradit i s nejednoznačnostmi v gramatice jazyka.

V některých případech totiž bezkontextová gramatika nemusí být jednoznačná. Není tedy možné na základě jednoho následujícího znaku rozhodnout, jaké gramatické pravidlo bude pro zpracování tokenu použito (uvažujeme LR(1) syntaktický analyzátor). Řešením je např. specifikace precedence a asociativity operátorů [1, str. 279].

Univerzálnější řešení je použít tzv. *zobecněný LR syntaktický analyzátor* (GLR syntaktický analyzátor). Ten umožňuje řešit situace, v nichž je možné zpracovat token na vstupu více než jedním způsobem. Tyto situace se označují jako *konflikty*, přičemž existují dva druhy těchto konfliktů. *Reduce/reduce* konflikty, jsou ty konflikty, u nichž není možné rozhodnout mezi provedením vícero redukcí na zásobníku. U druhého typu, *shift/reduce*, zase není možné rozhodnout, zda provést shift tokenu na zásobník nebo provést redukci [1, str. 238].

Pokud dojde ke konfliktu, GLR syntaktický analyzátor je rozdělen na dva či více samostatných syntaktický analyzátorů. Každý z těchto analyzátorů zpracuje token dle jiného pravidla a následně pokračují v analýze. Další tokeny na vstupu jsou zpracovávány všemi takto vytvořenými instancemi analyzátorů.

Dochází tak k větvení syntaktického analyzátoru (resp. jeho datových struktur). Analogicky mohou být větve dále rozdělovány, pokud nastanou další konfliktní situace. Větev syntaktické analýzy je ukončena, pokud je nalezena neplatná kombinace tokenů, přičemž zbylé větve jsou rozvíjeny dále [12, str. 17 – 23]. Další, rozšiřující metodě pro LR syntaktickou analýzu, která optimalizuje její původní koncept, se věnuje kapitola 1.2.3.

V této fázi statické analýzy může být dále provedena základní *sémantická analýza*. Jejím účelem může být obohacení interní reprezentace kódu o sémantické informace. To je kromě předcházející syntaktické analýzy umožněno také *tabulkou symbolů*. Díky sémantické analýze může statický analyzátor např. správně interpretovat přetížené operátory [9, str. 76].

Samotná statická analýza programu

Nad interní reprezentací, která může být navíc obohacena o sémantické informace, je již možné provádět statickou analýzu jako takovou. V závislosti na tom, jaké vlastnosti programu je zapotřebí zkoumat, jsou aplikovány konkrétní algoritmy.

Vyjma typových kontrol je provádění statické analýzy jako takové spíše zodpovědností nástrojů, jež budou využívat programovou knihovnu vytvořenou v rámci této práce. Z tohoto důvodu jsou uvedeny pouze příklady metod, které jsou využívány.

V rámci této fáze může být analyzováno např. předávání řízení pomocí *grafu toku řízení* (control flow graph). Tento orientovaný acyklický graf je sestrojen na základě interní reprezentace a jeho uzly obsahují seznamy instrukcí (tzv. basic blocks), jež budou provedeny vždy právě v takovém pořadí, v jakém se nachází v kódu. Zároveň platí, že jejich provádění není proloženo prováděním instrukcí z jiného seznamu. Analýzou tohoto grafu můžeme zkoumat cesty ve zdrojovém kódu nebo se může jednat o výchozí model pro další algoritmy [9, str. 77].

Dále je možné provádět *analýzu datových toků*. Při ní sledujeme, kde vznikají hodnoty uložené v proměnných a kde jsou tyto hodnoty použity. Tato analýza může být využita v kompilátorech pro eliminaci výrazů, jejichž výsledky nejsou dále potřeba [1, str. 597]. Kromě toho mohou být jejím prostřednictvím identifikována napevno vložená hesla v kódu, což je vzhledem k bezpečnosti velmi problematické [9, str. 80].

Na pomezí této fáze a sémantických kontrol prováděných v rámci předchozího kroku se nachází také *typové kontroly*. Jejich provedení a složitost se odvíjí od konkrétního jazyka a od jeho typového systému. Publikace od P. Emanuelssona [14] například považuje typové kontroly za druh statické analýzy pouze v případě netypovaných jazyků, slabě typovaných jazyků a jazyků bez statického typování.

Jiné zdroje v tomto směru však tak striktní nejsou a považují typové kontroly doslova za *„nejčastější formu statické analýzy“* [9, str. 24]. Setkáváme se s nimi totiž jak u kompilátorů, tak u dedikovaných statických analyzátorů. Uplatňují se zde dva základní principy. Prvním z nich je *typová syntéza*, při níž se datový typ výrazu určuje na základě jeho podvýrazů. Příkladem je datový typ výsledku sčítání v jazyce C, jenž je definován jeho operandy. Druhým principem je *typové odvozování* (inference), během kterého se datový typ jazykové konstrukce určuje na základě jejího použití. Pokud například funkce `len` vrací délku řetězce z argumentu, můžeme na základě volání `len(x)` konstatovat, že proměnná `x` bude datového typu řetězec [1, str. 386].

Existuje celá řada dalších algoritmů pro statickou analýzu a mnoho jejích forem. Jejich společným rysem je, že musí být velmi často doplněny o bázi pravidel, které určují, jak budou zkoumané vlastnosti kódu vyhodnoceny. Tato pravidla z pohledu uživatele udávají, jaké artefakty budou ohlášeny jako problematické, případně umožňují jejich klasifikaci. Pro snadnou tvorbu a správu těchto pravidel definují některé nástroje pro statickou analýzu vlastní formát, v němž jsou specifikována [9, 37].

Prezentace výsledků

Poněkud opomíjenou fází statické analýzy je prezentace výsledků uživatelům. Jedná se však o velmi podstatný aspekt, který určuje, zda bude analyzátor vůbec použitelný v praxi.

Je totiž důležité, aby uživatel při pohledu na výsledek statické analýzy mohl rozhodnout o jeho korektnosti a důležitosti. Dále by pak měl být výsledek prezentován v takové podobě, aby na něj mohl uživatel snadno reagovat (např. opravou chyby v kódu nebo seřízením nástroje pro statickou analýzu) [9, str. 105].

Nástroje pro statickou analýzu by měli uživatelům umožnit filtrovat výsledky a eliminovat tak např. varování, kterým se uživatel v dané chvíli věnovat nechce. Možnost filtrování je také důležitá kvůli náchylnosti některých nástrojů na chyby prvního typu (false positives). Přítomnost této funkcionality dovoluje uživateli eliminovat šum, jejž způsobují falešná chybová hlášení, a zaměřit se na skutečné problémy ve zdrojovém kódu [31]. Pokročilejší statické analyzátory umožňují seskupování a třídění výsledků podle různých kritérií [9, str. 106 – 110].

Příkladem moderního způsobu prezentace výsledků je analýza kódu v jazyce C# v rámci integrovaného vývojového prostředí *Visual Studio*. Statická analýza v tomto IDE [13] je součástí sady nástrojů *.NET Compiler Platform SDK*. Chyby a upozornění jsou jednak označené přímo v kódu samotném (barevným podtržením problematických úseků), ale jsou také přehledně uvedeny v seznamu. V tomto seznamu je možné chyby filtrovat podle různých kritérií (např. závažnosti, pozice v kódu apod.). Analyzátor kódu poskytuje kromě popisného hlášení také možnost automatické opravy nebo možnost potlačení upozornění. Hlášení jsou navíc opatřena unikátním kódem a odkazem na webovou dokumentaci s dalšími informacemi.

1.2 Inkrementální provedení statické analýzy

Jednou ze zásadních vlastností statických analyzátorů je rychlost, jakou jsou schopné analýzu provádět. Tento parametr nabývá ještě více na důležitosti při jejich použití v rámci IDE a textových editorů. V těchto případech je totiž často analýza prováděna interaktivním způsobem, během editace zdrojového kódu.

Bylo by pochopitelně možné při každé editaci provádět analýzu kompletně znovu neinkrementálním způsobem. To by ale způsobilo latenci, jež by se s rostoucí velikostí zdrojového souboru zvyšovala. U rozsáhlých zdrojových kódů by pak tím pádem statická analýza prováděná v reálném čase nebyla ani možná.

Pokud bychom například vložili na náhodný řádek ve zdrojovém souboru jeden znak, muselo by dojít k aktualizaci zobrazených výsledků analýzy. Kromě toho by ale musela být opětovně provedena i samotná statická analýza, a navíc také tvorba modelu programu, neboť modifikací vstupního řetězce došlo k jeho zneplatnění.

Jedno z možných řešení nám v tomto případě nabízí *inkrementální* přístup ke statické analýze, či k některé z jejích fází. Principem tohoto přístupu je průběžné zaznamenávání změň zdrojového kódu a následná analýza pouze těch jeho částí, které byly touto editací ovlivněny [6, str. 1].

Jelikož velkou částí knihovny, která byla v rámci této práce vytvořena, je inkrementální tvorba modelu programu a sémantické kontroly nad tímto modelem, je tato kapitola věnována zejména těmto částem statické analýzy. Ačkoliv existuje více publikací na téma inkrementální lexikální a syntaktické analýzy [6, 19], zaměříme se zejména na disertační práci Practical Algorithms for Incremental Software Development Environments [41].

Důvodem je, že je touto prací inspirována knihovna *tree-sitter* [7], která je použita pro implementaci programového řešení této bakalářské práce. V závěru této kapitoly jsou stručně shrnuty principy inkrementální sémantické analýzy, která je rovněž součástí řešení.

1.2.1 Model programu a jeho modifikací

Odkazované inkrementální statické analyzátory používají jako interní reprezentaci programu stromové struktury. Konkrétně se jedná o abstraktní syntaktické stromy (AST), které slouží buďto jako výchozí struktury pro algoritmy statické analýzy (viz výše) nebo pro tvorbu jiné reprezentace, jež může být ještě na vyšší úrovni abstrakce. Kromě toho v nich ale také mohou být uchovávány záznamy o modifikacích zdrojového souboru. To je také případ knihovny tree-sitter, která provádí inkrementální syntaktickou analýzu zdrojových souborů a v kontextu statické analýzy se tedy stará o tvorbu modelu programu. Typicky se využívá právě ve spojení s textovými editory a IDE.

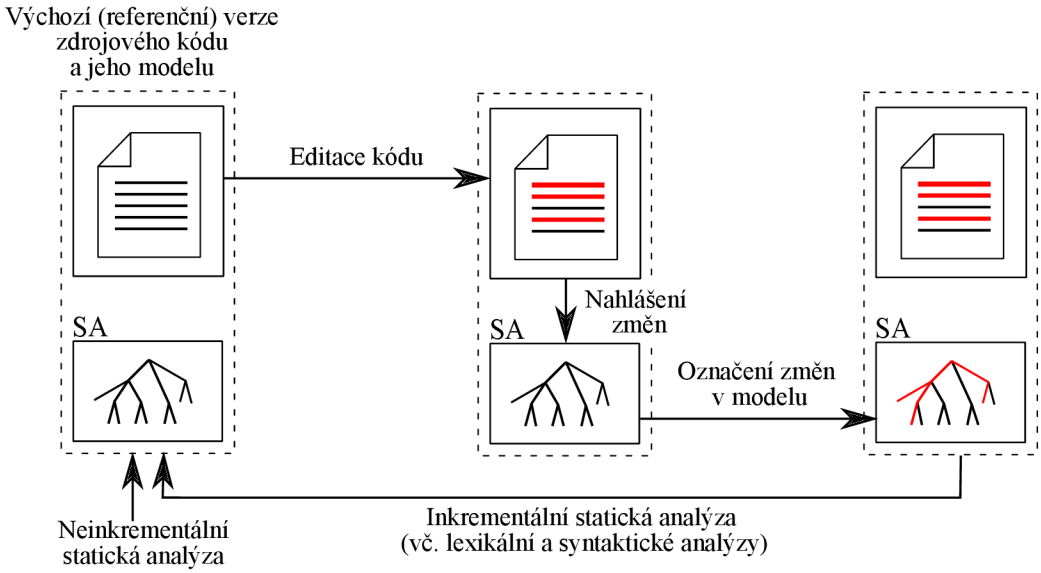
Činnost inkrementálního statického analyzátoru, jenž může být založen například na této knihovně, je schematicky znázorněna na obrázku 1.3. Nejprve je provedena počáteční analýza zdrojových souborů. Při počáteční syntaktické analýze je použit neinkrementální přístup (batch parsing). Neexistuje totiž zatím struktura, z níž by bylo možné vycházet. Tím vznikne prvotní referenční verze interní reprezentace neboli modelu programu.

Editace zdrojových souborů (např. uživatelem v IDE) způsobí nekonzistenci modelu programu se vstupním řetězcem. Dochází proto k nahlášení změn statickému analyzátoru (SA), který je propaguje dále, až k jeho části zodpovědné za správu modelu (např. knihovna tree-sitter). Ta si záznamy o změnách uchovává buďto přímo v AST nebo jiným způsobem. Například inkrementální syntaktický analyzátor prezentovaný J. Beetemem používá pro tento účel tzv. *kmn seznam* (kmn list) [6, str. 3].

Po zaznamenání změn je provedena inkrementální lexikální a syntaktická analýza, která obnoví konzistenci zdrojového souboru a modelu programu. Poté následuje sémantická analýza a případně jsou aplikovány algoritmy pro statickou analýzu (analýza datových toků, analýza grafu toku řízení atd.). Tyto typy analýz mohou být provedeny rovněž inkrementálně, ale je také možné je provést kompletně znovu nad aktualizovaným modelem.

S ukládáním záznamů o editacích zdrojového souboru souvisí také formát, v němž jsou editace nahlašovány, a způsob, jakým jsou určeny modifikované části modelu programu. V případě uchování změn v podobě kmn seznamu je zdrojový řetězec (zdrojový kód) reprezentován jako uspořádaná n-tice jeho podřetězců $(s_1, s_2, \ldots, s_N)$. Pro každý tento podřetězec je definována uspořádaná trojice (k_i, m_i, n_i), kde k je počet nemodifikovaných znaků na začátku daného podřetězce, m je počet znaků, které byly z původního řetězce odstraněny, a n je počet nově vložených znaků namísto původních m znaků. Jinými slovy, tato trojice udává počet znaků m, jež byly v původním řetězci počínaje indexem k nahrazeny novým počtem znaků n. Například tedy trojice $(10, 0, 1)$ udává, že v daném řetězci byl na index 10 vložen jeden nový znak [6, str. 3].

Knihovna tree-sitter řeší tento problém jiným způsobem. Uchovává pozice začátků a konců uzlů AST v rámci zdrojového dokumentu přímo ve strukturách reprezentující uzly daného stromu. Tyto pozice jsou uloženy v podobě indexu bytu ve zdrojovém souboru (bytový offset), ale také v podobě souřadnic tvořených indexem řádku a indexem znaku na daném řádku (struktura `TSPoint`). Editace jsou oznamovány prostřednictvím struktury `TSInputEdit`. Ta obsahuje index prvního modifikovaného bytu (`start_byte`), index prvního bytu za modifikovanou částí v původním zdrojovém souboru (`old_end_byte`) a index bytu za modifikovanou částí v nové verzi zdrojového souboru (`new_end_byte`). Tytéž údaje



Obrázek 1.3: Průběh inkrementální statické analýzy založené na knihovně tree-sitter [7] (vytvořeno na základě [41]). Zkratka SA označuje statický analyzátor, který si udržuje model programu. Ten se po nahlášení změn stává nekonzistentním se zdrojovým kódem. Jeho konzistence je obnovena inkrementální statickou analýzou, zejména pak činností inkrementálního lexikálního a syntaktického analyzátoru.

obsahuje také ve formě souřadnic ve zdrojovém kódu. Struktura, která reprezentuje editaci, je v jazyce C definována jako [7]:

```
struct TSInputEdit {
    uint32_t start_byte;
    uint32_t old_end_byte;
    uint32_t new_end_byte;
    TSPoint start_point;
    TSPoint old_end_point;
    TSPoint new_end_point;
}
```

Díky údajům ve struktuře `TSInputEdit` a pozicím uzlů uloženým v AST jsou během nahlášení změn snadno nalezeny modifikované podstromy. Mohou tak být upraveny údaje o jejich pozici a také nastaven jejich příznak modifikace. Při vkládání znaků je tedy průchodem stromu nalezen podstrom, který v dokumentu reprezentuje právě tu část, do níž byl znak vložen. Následně jsou upraveny údaje o pozici tohoto podstromu a nakonec je nastaven příznak modifikace pro daný podstrom. Tento proces probíhá rekurzivně čili pro všechny podstromy, jež jsou danou modifikací dotčeny.

Ani v jednom z výše uvedených případů není dodán v době ohlašování editace syntaktickému ani lexikálnímu analyzátoru konkrétní řetězec, který byl editací vložen nebo odstraněn. Změny jsou v obou případech popsány pouze jejich pozicemi a počty znaků, které byly odstraněny, respektive vloženy. Tyto informace nejsou zapotřebí, protože díky zaznamenávání pozic změn ve zdrojovém dokumentu mohou v něm tyto analyzátory snadno najít dané modifikované oblasti během inkrementální lexikální a syntaktické analýzy.

Uvedená disertační práce [41], na které je knihovna tree-sitter založena, definuje vlastní sofistikovaný způsob zaznamenávání změn. Díky němu je možné přistupovat k několika

verzím modelu programu a zároveň zajišťuje možnost, dotázat se uzlů v AST, zda byly změněny.

Níže popsané algoritmy pro lexikální a syntaktickou analýzu totiž pracují se třemi verzemi AST [41, str. 28]:

- **Referenční** – Jedná se o verzi, která je konzistentní se zdrojovým kódem. Vzniká po dokončení inkrementální nebo počáteční neinkrementální syntaktické analýzy.
- **Původní** – Je to verze AST před začátkem syntaktické analýzy. Vzniká editacemi referenční verze zdrojového kódu a jejich zaznamenáním. Je nekonzistentní se zdrojovým kódem. Rozdíly mezi touto a referenční verzí určují, jaké části AST bude možné opětovně využít.
- **Aktuální** – Jedná se o verzi AST, která je aktuálně vytvářena syntaktickým analyzátorem.

Uzly v AST si zároveň udržují dva booleovské atributy. První z nich určuje, zda byl modifikován uzel samotný (local change). Druhý udržuje souhrnnou informaci o jeho modifikaci, modifikaci jeho potomků a strukturálních změnách v podstromě, pro který je daný uzel kořenem (nested change). Díky druhém atributu můžeme hledat ve stromě cestu od kořene až k modifikovaným listovým uzlům [41, str. 45].

Současně však uvedená práce dovoluje konkrétním aplikacím, aby používaly jiný model pro záznam a identifikaci změn v AST, což je právě případ knihovny tree-sitter. To je možné díky tzv. *change reporting rozhraní*, které definuje sadu metod, již musí uzly derivačního uzlu poskytovat, aby bylo nad nimi možné algoritmy provádět. Pro následující kapitoly bude z množiny těchto metod důležitá hlavně `has_changes` se signaturou `bool has_changes(version_id, [local|nested])`. Tato metoda umožňuje přistupovat k atributům uchovávajícím informace o modifikaci. Pomocí prvního parametru lze přistupovat i k uzlům v nereferenční verzi AST. Pokud není specifikován druhý parametr, identifikovány jsou jakékoliv změny, které se uzlu týkají [41, str. 18].

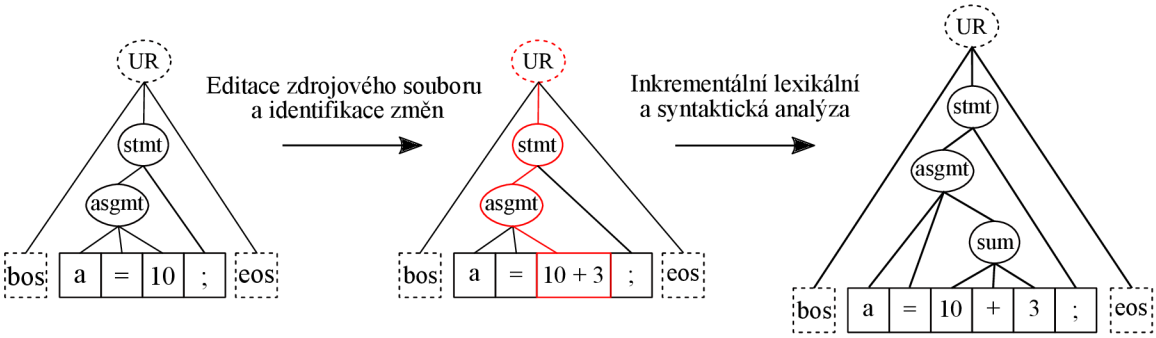
Dále se předpokládá, že v AST je přítomna trojice pomocných uzlů (sentinel nodes). První dva se v něm vykytují jako jeho nejpravější a nejlevější listový uzel, čímž označují začátek a konec proudu tokenů (uzly `bos` a `eos`). Třetím pomocným uzlem je samotný kořen AST (označovaný jako `UltraRoot`). Díky těmto uzlům také není nutné ošetřovat zpracování krajních tokenů v řetězci tokenů, čímž dochází ke zjednodušení níže popsaných algoritmů [41, str. 17].

Poloha pomocných uzlů v AST, označení změn či jeho stav před a po inkrementální lexikální a syntaktické analýze je znázorněn na obrázku 1.4.

1.2.2 Lexikální analýza

Činnost inkrementálního lexikálního analyzátoru je z pohledu uživatele stejná jako jeho neinkrementální varianty (batch lexer), používané při kompilaci programovacích jazyků. Načítá vstupní řetězec znaků, které tvoří zdrojový kód, a identifikuje v něm tzv. *lexémy*. Na základě těchto lexémů vytváří výstupní řetězec tokenů, který předává k dalšímu zpracování [1, str. 109]. Termín lexém označuje *„sekvenci znaků ve vstupním řetězci, která je identifikována lexikálním analyzátorem jako instance příslušného tokenu“* [1, str. 111].

Z uživatelského pohledu se neinkrementální a inkrementální přístupy začnou lišit ve chvíli, kdy je nutné znovu analyzovat zdrojový kód, v němž došlo ke drobným změnám (např. v důsledku editace uživatelem).



Obrázek 1.4: Změny v AST během inkrementální lexikální a syntaktické analýzy (sestaveno na základě fungování [7] a práce [41]) – Analyzovaným řetězcem je přiřazení hodnoty v jazyce C. Ve čtvercích jsou tokeny, které zároveň plní úlohu listových uzlů v AST. Přerušovanou čarou jsou označeny pomocné uzly (zkratka UR znamená `UltraRoot`). Vidíme, že proud tokenů a některé části AST nejsou konzistentní se zdrojovým kódem po provedení změn. Změny jsou však ve stromě označeny a během opětovné analýzy je obnovena konzistence AST a zdrojového kódu.

Neinkrementální, dávkový přístup vyžaduje kompletní reanalýzu celého vstupního řetězce. *Inkrementální lexikální analýza* naproti tomu přihlíží k již existujícímu výstupnímu řetězci tokenů a reanalyzuje pouze ty části vstupního řetězce, které to vyžadují. Díky tomu dochází k redukci času, jenž je k analýze potřeba [41, str. 37]. Součástí tohoto procesu může být také označení nových a reanalyzovaných tokenů v rámci výstupního řetězce [6, str. 6].

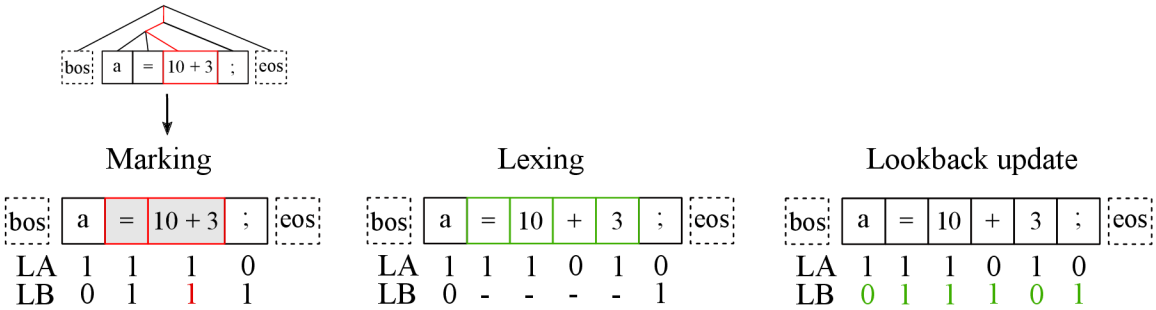
Jeden ze způsobů realizace *inkrementálního lexikálního analyzátoru* (incremental lexer) je prezentovaný ve výše odkazované disertační práci [41, str. 37 – 55]. Velkou výhodou této implementace je možnost využití běžných generátorů lexikálních analyzátorů (např. *flex*). Zároveň není zapotřebí žádný speciální formalismus pro popis lexémů daného jazyka. I zde je možné modelovat lexikální analyzátor jako konečný automat. Podpora inkrementální analýzy je umožněna pouze prostřednictvím rozšíření konvečního lexikálního analyzátoru a tokenů, které produkuje.

V tokenech, které jsou produkovány neinkrementálními lexikálními analyzátory, je běžně uchován řetězec s daným lexémem, typ tokenu a případně také odkaz na rodičovský uzel v AST (pokud jsou použity jako listové uzly AST). V případě inkrementálního přístupu je struktura reprezentující token navíc obohacena o položky:

- `state` – Jedná se o konečný stav lexikálního analyzátoru v době vytvoření tokenu.
- `lookahead` – Je to počet znaků přečtených za hranicí lexému.
- `lookback` – Udává relativní polohu prvního tokenu, který je na aktuálním tokenu závislý.

Kompletní struktura tokenu, který je produkován inkrementálním lexikálním analyzátorem, tedy vypadá následujícím způsobem (převzato z [41, str. 40]):

```
1 struct TOKEN {
2     int type, state, lookback, lookahead;
3     STRING lexeme;
4     NODE *parent;
5 }
```



Obrázek 1.5: Průběh inkrementální lexikální analýzy – Schéma zobrazuje proud tokenů na konci každé z jejích fází. Vstupem první fáze (marking) je nekonzistentní AST z obrázku 1.4. Druhé dvě fáze mohou být provedeny bezprostředně nebo až na vyžádání během syntaktické analýzy. Zkratky LA a LB označují položky `lookahead` a `lookback` jednotlivých tokenů.

Uložení stavu v tokenu umožňuje obnovit interní konfiguraci lexikálního analyzátoru, v níž se nacházel v době navrácení daného tokenu. Poslední dvě položky struktury jsou zapotřebí pro vyhodnocení závislostí mezi tokeny. V některých případech stačí, aby pro konstrukci tokenů byly přečteny pouze znaky tvořící příslušné lexémy.

Někdy je ale zapotřebí, aby analyzátor nahlédl i na znaky, které již lexému nenáleží, což vytváří právě tyto závislosti. V kontextu jazyka YARA se jedná například o identifikátory pravidel (viz 2.1.1). Ty jsou definovány jako libovolné sekvence alfanumerických znaků a podtržítek, přičemž jejich prvním znakem není číslo. U takto definovaných identifikátorů je až na základě následujícího znaku (např. bílého znaku) určen konec identifikátoru. Analogicky však mohou nastat situace, kdy tímto způsobem musí být načteno více znaků, aniž by byly součástí lexému daného tokenu [1, str. 113].

Právě počet znaků, který bylo tímto způsobem potřeba přečíst během konstrukce tokenu, uchovává položka `lookahead`. U konvenční lexikální analýzy tento jev v rámci struktury tokenu nijak reflektován nebývá [41, str. 42].

Výše popsaný problém lexikální analýzy způsobuje, že editací některého lexému není ovlivněn jen příslušný token, ale také tokeny předcházející. Kolik tokenů je takto ovlivněno udává právě položka `lookback`. Konkrétně říká, o kolik míst je zapotřebí se v proudu tokenů vrátit, abychom nalezli první token, který mohl být editací zneplatněn. Tato hodnota je vypočítána na základě položky `lookahead`.

Toto dynamické zaznamenávání hodnot `lookahead` a `lookback` je velmi obecný přístup, jenž umožňuje aplikaci tohoto algoritmu i v oblastech zpracování přirozeného jazyka. Pro většinu běžných programovacích jazyků však platí, že maximální hodnota položky `lookahead` a tedy i `lookback` je jedna nebo nula (viz příklad s identifikátory). Toho se často využívá, protože je možné nastavit tuto hodnotu fixně pro každý token. Následkem toho dochází k reanalýze některých tokenů, u nichž by nebyla zapotřebí. Celkově se tím ale velmi zredukuje režie potřebná pro výpočet hodnoty `lookback` a její udržování v konzistentním stavu, což může vést k lepším výsledkům [41, str. 43].

Podobně jako tokeny, tak i samotný inkrementální lexikální analyzátor získáme rozšířením jeho neinkrementální verze. Konkrétně je zapotřebí přidat metodu `get_state` umožňující získat aktuální stav a metodu `set_state`, jež umožňuje nastavit stav lexikálního analyzátoru.

Samotná analýza probíhá ve třech fázích: *marking*, *lexing* a *lookback update*. V první fázi dojde k označení tokenů (marking), které je zapotřebí reanalyzovat. Tokeny jsou umístěny

v AST jako listové uzly (v případě [41]). Předpokládá se, že v tomto stromě jsou již označeny uzly, jež byly ovlivněny editací zdrojového kódu. Vhodným průchodem a na základě tohoto označení je možné identifikovat část proudu tokenů, v níž byly tokeny explicitně modifikovány. Tato oblast proudu tokenů musí být navíc také rozšířena o tokeny, jež jsou závislé na explicitně modifikovaných tokenech. Ty lze určit pomocí hodnot `lookback` explicitně modifikovaných tokenů. Během označování je nastaven příznak změny, a tudíž lze celou označenou oblast během následujících fází nebo během syntaktické analýzy snadno lokalizovat.

Takto označené sekvence tokenů je poté zapotřebí reanalyzovat. Děje se tak ve druhé fázi lexikální analýzy (lexing). Nejprve je lexikální analyzátor zresetován do stavu, v němž se nacházel před analýzou označené sekvence tokenů. To je možné díky položce `state` tokenu, jenž označené oblasti přechází, a metodě `set_state`. Jelikož je použit pomocný token `bos` na začátku proudu tokenů, nemůže nastat situace, kdy by nebylo možné nalézt předcházející token pro některou z označených oblastí. Pomocný počáteční token `bos` ve své položce `state` uchovává počáteční stav lexikálního analyzátoru.

Stav analyzátoru je tedy nastaven na požadovanou hodnotu a daná označená oblast je znovu analyzována. Díky tomu je obnovena konzistence proudu tokenů a modifikovaného zdrojového kódu [41, str. 47].

Poslední fází je tzv. *lookback update*. Jak už její název napovídá, dojde k aktualizaci položek `lookback`. Ta probíhá od prvního tokenu, jehož `lookahead` zasahuje do reanalyzované sekvence tokenů. Zásadní datovou strukturou je zde seznam, jenž obsahuje položky ve tvaru `<token, lookahead, age>` (character lookahead list)[5]. Postupně pro každý zpracovaný token je vložena do seznamu položka, která obsahuje jeho název, hodnotu `lookahead` a hodnotu vyjadřující počet iterací, který položka strávila v seznamu (výchozí hodnota je 0). Po každém dalším zpracováním tokenu a přidání nového záznamu do seznamu je hodnota `lookahead` v každé položce dekrementována o délku lexému právě zpracovaného tokenu. Hodnota `age` je naproti tomu inkrementována o jedničku. Pokud `lookahead` položky v seznamu je menší či rovna nule, je položka ze seznamu odstraněna [41, str. 49].

Při zpracování tokenu můžeme na základě obsahu seznamu určit jeho novou hodnotu `lookback` jako nejvyšší hodnotu `age`, jež je v seznamu přítomná. Tato aktualizace se provádí pochopitelně pro všechny tokeny v rámci oblasti reanalyzované během předchozí fáze. Je ji ale také nutné provést u některých tokenů nacházejících se před nově vytvořenými a za nimi. První, který je aktualizován je nalezen tak, že je nejprve identifikován token nacházející se v referenční, původní i aktuální verzi (předchází oblasti označené během fáze marking). Token, jenž po něm následuje v referenční verzi, obsahuje ve své položce `lookback` počet tokenů, které se nachází před označenou (reanalyzovanou) oblastí a je u nich zapotřebí rovněž provést lookback update.

Aktualizovány jsou tak postupně tyto tokeny a všechny následující (včetně tokenů v reanalyzované oblasti). Tento proces je ukončen ve chvíli, kdy je nalezen pomocný token `eos`, nebo když se začnou uložené hodnoty `lookback` shodovat s nově vypočítanými hodnotami. Seznam využívaný pro aktualizaci (character lookahead list) přitom nesmí obsahovat už žádné tokeny z reanalyzované oblasti řetězce tokenů.

Výše popsaná inkrementální lexikální analýza může probíhat buďto na žádost inkrementálního syntaktického analyzátoru nebo samostatně, před začátkem syntaktické analýzy. V prvním případě však musí být fáze markingu oddělena a musí proběhnout před samotnou syntaktickou analýzou. Díky tomu je syntaktický analyzátor schopen snadno

[5] V odkazované disertační práci nejsou části položek v seznamu pojmenovány. Toto pojmenování bylo zavedeno pro snazší vysvětlení fáze lookback update.

identifikovat to, že zpracovává část AST, jež může obsahovat potenciálně nekonzistentní tokeny vyžadující reanalýzu.

1.2.3 Syntaktická analýza

V této kapitole se zaměříme na způsob inkrementální syntaktické analýzy, jenž je prezentovaný ve výše uvedené disertační práci [41]. Jeho podstatou je opětovné využití podstromů AST, které nebyly modifikovány. Přirozenou reprezentací podstromů jsou v procesu syntaktické analýzy neterminály gramatiky.

Inkrementální syntaktický analyzátor tedy zpracovává vstup (input stream) složený jak z *terminálů*, které jsou reprezentovány tokeny, tak z *neterminálů*, jež jsou reprezentovány vnitřními (neterminálními) uzly již vytvořeného AST. Tokeny jsou produkovány inkrementálním lexikálním analyzátorem (viz předchozí kapitola) nebo jsou získány rozkladem neterminálního uzlu AST. Vnitřní uzly AST, jež mohou být opětovně použity, jsou získávány průchodem původní verze AST [41, str. 60].

Algoritmus využívá také tyto pomocné funkce (kódy jsou převzaty z [41]):

- `right_breakdown` – Zprava rozkládá podstrom (neterminál) na vrcholu zásobníku na terminály a neterminály, které umisťuje opět na zásobník. Pokud je zpracován terminál, rozklad se zastaví, přičemž tento terminál je vložen zpět na zásobník. Tato funkce se využívá pro rozklad nejpravější větve podstromu, jelikož může být ovlivněna bezprostředně následujícím terminálem na vstupu. Proto je zapotřebí provést analýzu této pravé větve opětovně [41, str. 59].

```
right_breakdown() {
    NODE *node;
    do {
        node = parse_stack->pop();
        foreach child of node do shift(child);
    } while (is_nonterminal(node));
    shift(node);
}
```

- `left_breakdown` – Funkce je použita k rozložení následníka (tzv. lookahead), který je neterminálem a k získání bezprostředně následujícího terminálu. V případě, že ukazatel `la` ukazuje na neterminál, je navrácen jeho nejlevější potomek, který je terminálem. Děje se tak pomocí rekurzivního volání funkce `left_breakdown`. Pokud je ukazatelem odkazován terminál nebo neterminál bez potomků, je volána funkce `pop_lookahead` [41, str. 62].

```
NODE *left_breakdown (NODE *la) {
    if (la->arity > 0) {
        NODE *result = la->child(0, previous_version);
        if (is_fragile(result)) return left_breakdown(result);
        return result;
    } else return pop_lookahead(la);
}
```

- `pop_lookahead` – Vrací následníka uzlu (vždy se jedná o terminál) odkazovaného ukazatelem `la`. Nejdříve je nalezen nejpravější následník uzlu `la` a poté v závislosti na tom,

zda se jedná o neterminál či terminál, je proveden další rozklad tohoto následníka nebo je tento uzel rovnou navrácen [41, str. 62].

```
NODE *pop_lookahead (NODE *la) {
    while (la->right_sibling(previous_version) == NULL)
        la = la->parent(previous_version);
    NODE *result = la->right_sibling(previous_version);
    if (is_fragile(result)) return left_breakdown(result);
    return result;
}
```

Provádění samotného algoritmu inkrementální syntaktické analýzy lze popsat pomocí funkce `inc_parse` (převzato z [41, str. 61]):

```
1  inc_parse () {
2      parse_stack->clear(); parse_state = 0; parse_stack->push(bos);
3      NODE *la = pop_lookahead(bos);
4      while (true)
5          if (is_terminal(la))
6              if (la->has_changes(reference_version)) relex(la);
7              else
8                  switch (parse_table->action(parse_state, la->symbol)) {
9                      case ACCEPT: if (la == eos) {
10                         parse_stack->push(eos);
11                         return;
12                     } else { recover(); break; }
13                     case REDUCE r: reduce(r); break;
14                     case SHIFT s: shift(s); la = pop_lookahead(la); break;
15                     case ERROR: recover(); break;
16                 }
17         else
18             if (la->has_changes(reference_version)
19                 a = left_breakdown(la);
20             else {
21                 perform_all_reductions_possible(next_terminal());
22                 if (shiftable(la)) {
23                     shift(la); right_breakdown();
24                     la = pop_lookahead(la);
25                 }
26                 else la = left_breakdown(la);
27             }
28 }
```

Z kódu výše je patrné, že v situaci, kdy jsou na vstupu pouze terminály (např. při počáteční syntaktické analýze), je algoritmus prakticky shodný se standardní LR syntaktickou analýzou [1, str. 248]. Jedinou odlišností je přítomnost pomocného uzlu `bos`. Pokud dochází k reanalýze a na vstupu je nalezen terminál, který byl změněn (zjištěno pomocí metody `has_changes`), je spuštěna inkrementální lexikální analýza funkcí `relex` (řádek 6).

Pokud je na vstupu nalezen neterminál, je zkontrolováno, jestli došlo k jeho změně oproti referenční verzi AST (ř. 18). Pokud ano, dojde k rozkladu tohoto uzlu pomocí metody `left_breakdown`. Dochází tak k postupnému rozdělení neterminálu (podstromu) na další podstromy, které zůstaly nezměněny a ty, které danou změnu obsahují.

Jestliže je na vstupu nalezen neterminál, jenž neobsahuje žádné změny, jsou provedeny všechny redukce, které je možné na zásobníku provést, neboť není nutné znovu reanalyzovat

nezměněné podstromy AST. Poté je tento neterminál vložen na vrchol zásobníku, přičemž jeho nejpravější platná větev je rozložena pomocí `right_breakdown`, protože její platnost vzhledem k současné verzi může být ovlivněna následujícím terminálem na vstupu (ř. 23). Při LR(1) syntaktické analýze musíme totiž přihlížet také vždy k jednomu následujícímu terminálu na vstupu a tento terminál nemusel být v původní verzi stromu přítomen.

Časová složitost tohoto algoritmu je $\mathcal{O}(t + s(\log N)^2)$, kde t je počet nových tokenů, s je počet míst, kde byla provedena modifikace a N je celkový počet uzlů v daném stromě. Z pohledu uživatele tedy závisí na množství a rozsahu editací, které provedl [41, str. 61 – 62].

Článek, z nějž algoritmus pochází, prezentuje také jeho modifikace, které redukují časovou složitost (na $\mathcal{O}(t + s \log N)$). Dále do něj zavádějí zotavení z chyb nebo jej umožňují použít se zobecněnými metodami LR syntaktické analýzy [41, str. 62 – 88].

Spojení tohoto algoritmu pro inkrementální syntaktickou analýzu a GLR syntaktického analyzátoru (viz kapitola 1.1.4) je označováno jako IGLR syntaktická analýza [41, str. 81].

1.2.4 Sémantická analýza

Vhodným formalismem, který je pro popis statické sémantiky programu vhodný, jsou syntaxí řízené definice (syntax-directed definitions). Jedná se o spojení bezkontextových gramatik s tzv. *atributy* a *sémantickými akcemi*. Atributy jsou přiřazeny symbolům gramatiky (neterminály i terminály) a obohacují je tak o dodatečné informace, které mohou mít různý charakter. Může se jednat například o celočíselné hodnoty, řetězce či reference. To, jakým způsobem atributy nabývají svých hodnot, definují sémantické akce, jež jsou přidruženy pravidlům gramatiky [1, str. 304].

Podle způsobu výpočtu hodnot atributů je dělíme na *syntetizované* (synthesized) a *dědičné* (inherited). Syntetizovaný atribut neterminálu získá svoji hodnotu pouze na základě ostatních atributů toho neterminálu nebo skrze hodnoty atributů potomků neterminálu v AST. Dědičné atributy jsou určeny hodnotami ostatních atributů daného neterminálu, atributů rodičovských nebo sourozeneckých neterminálů v AST. Z tohoto popisu vyplývá, že jelikož jsou terminály listovými uzly AST, mají v jejich případě smysl pouze dědičné atributy. Jsou-li hodnoty atributů definované pouze na základě jiných atributů nebo konstant, nazýváme tyto syntaxí řízené definice *atributovými gramatikami* [1, str. 304 – 306].

Jelikož hodnoty atributů mohou být určovány na základě hodnot jiných atributů, vzniká mezi nimi vztah závislosti. O atributech figurujících v sémantické funkci definující atribut A, tak hovoříme jako o *argumentech* A. Vztah závislostí mezi atributy může být zobrazen pomocí *grafu závislostí* (dependency graphs) [38, str. 1].

V rámci sémantických akcí jsou sice běžně vyhodnocovány hodnoty atributů, ale mohou být právě skrze ně prováděny také sémantické kontroly (např. typové kontroly). U kompilátorů jsou kromě toho využívány také ke generování kódu. Atributy symbolů gramatiky mohou být vyhodnocovány buďto přímo během syntaktické analýzy, nebo dodatečně, při průchodu výsledného AST [1, str. 303].

Problémy, které přináší inkrementální přístup k sémantické analýze, popisuje T. Reps ve svém článku Optimal-time Incremental Semantic Analysis for Syntax-directed Editor [38]. Jedním z oněch problémů jsou neúplné AST, jež vznikají v důsledku rozpracovaných zdrojových souborů. Kvůli chybějícím podstromům nemusíme tak být schopni určit hodnoty atributů. Přesto je žádoucí, aby byla sémantická analýza vůči tomuto stavu odolná a dokázala provést kontroly alespoň nad některými částmi AST. Řešením je zavedení speciálních pravidel ve tvaru $X \longrightarrow \perp$ (completing production), kde symbol $\perp$ označuje chybějící pod-

strom. Existencí těchto pravidel pro každý neterminál gramatiky docílíme, že z pohledu uživatele se budou AST s chybějícími podstromy chovat jako úplné [38, str. 3].

K další problematické situaci dochází při nahrazení podstromu v AST jiným podstromem, čímž se hodnoty některých atributů mohou stát nekonzistentními. Neinkrementálním řešením by bylo znovu vyhodnotit hodnoty všech atributů v AST. Cílem inkrementálního přístupu je omezit množinu atributů, jejichž hodnotu je zapotřebí aktualizovat. Základním předpokladem pro řešení tohoto problému je, aby bylo možné vyhodnotit atributy uzlů v *samostatně stojícím stromě* (free-standing tree). To jsou stromy, jejichž kořenem není počáteční symbol dané gramatiky. Problémem při vyhodnocování atributů v těchto stromech mohou být argumenty dědičných atributů kořene, jež nejsou v daném samostatně stojícím stromu přítomné. Řešením je omezit používání dědičných atributů nebo hodnotu chybějících atributů dočasně stanovit jako vybranou výchozí hodnotu [38, str. 3].

Při nahrazování podstromu v AST se již tedy předpokládá, že díky uvedené schopnosti, vyhodnocovat atributy pro samostatně stojící stromy, byly hodnoty všech atributů v novém podstromě již určeny. Atributy kořenového uzlu nového podstromu (resp. vnitřního uzlu AST, ke kterému byl uzel připojen) jsou po připojení do AST nekonzistentní a musí být znovu vyhodnoceny. Jejich opětovné vyhodnocení sice zajistí jejich validitu, nicméně může způsobit nekonzistenci jiných atributů v AST, jež jsou na nich závislé. Z tohoto důvodu musí dojít k tzv. *propagaci změn* (change propagation), která zajistí, že po nahrazení podstromu budou hodnoty všech atributů v AST korektní. Toho je dosaženo pomocí grafu závislostí mezi atributy v nově připojeném podstromě a atributy v celém AST, který je označován jako *model*. Zjednodušeně řečeno, celý proces probíhá tak, že hodnoty atributů jsou vyhodnocovány v topologickém pořadí. Pokud se jejich původní a nová hodnota liší, je model rozšířen o závislé atributy. Předpokládáme přitom, že mezi atributy neexistuje cyklická závislost, a tudíž nemůže dojít k zacyklení během vyhodnocování [38, str. 5].

Kapitola 2

Jazyk YARA

YARA je multiplatformní nástroj, který je primárně určený k identifikaci a klasifikaci malwaru. Můžeme ho ale využít obecně k identifikaci textových a binárních souborů s určitými specifickými znaky. Těmito znaky může být např. přítomnost sekvence bytů v souboru, počet výskytů sekvence či charakteristické chování programu [2].

Původní motivací tvůrce tohoto nástroje, Victora Alvareze, byla právě automatizace rozpoznávání a rozdělování vzorků malwaru do tříd podle charakteristických znaků. Ruční metody totiž vyžadovaly memorování velkého počtu názvů těchto tříd a navíc nebylo snadné sdílet jejich typické znaky s dalšími odborníky [3].

Tento nástroj tedy nejen, že naplnil tyto potřeby tvůrce (a jistě i mnoha jiných uživatelů), ale poskytl také cestu, jak popis malwaru formalizovat prostřednictvím stejnojmenného jazyka, který je jím využíván.

Pomocí tohoto aplikačně specifického jazyka YARA totiž formulujeme ony charakteristické znaky souborů a vytváříme tzv. *pravidla*. Program *yara* je nejprve zkompiluje do podoby speciálního bytekódu, což zahrnuje také kontrolu syntaktické a sémantické správnosti pravidel. Následně zkompilovaná pravidla, která jsou již pro člověka prakticky nečitelná, interpretuje[1]. To znamená, že provede vyhledávání nad zadaným adresářem, jehož cílem je identifikovat soubory odpovídající pravidlům a určit, jakým z nich odpovídají.

Kromě programu *yara* pro příkazový řádek, poskytují vývojáři tohoto nástroje také knihovny pro jazyky Python (yara-python) a C (libyara). Díky tomu můžeme využívat tento nástroj skrze aplikační programové rozhraní těchto knihoven a snadno jej integrovat do větších projektů vytvářených v různých programovacích jazycích [2].

Gramatika jazyka YARA je dostupná pouze ve formátu pro generátor syntaktických analyzátorů GNU bison, který využívá oficiální distribuce nástroje *yara*. Doposud nebyla zveřejněna žádná jiná formální specifikace gramatiky (neuvažujeme-li dokumentaci, která má spíše neformální charakter).

Při vytváření nástroje pro inkrementální statickou analýzu tohoto jazyka je nezbytné jej dobře znát, aby bylo možné formulovat jeho gramatiku a na základě ní analyzátor vytvořit. Cílem této kapitoly je proto uvést čtenáře do tohoto jazyka, seznámit jej s některými jeho zásadními konstrukcemi a ozřejmit mu jejich sémantiku. Z tohoto důvodu obsahuje kapitola také několik jednoduchých příkladů obsahujících dané klíčové konstrukce.

[1] Chceme-li pravidla pouze zkompilovat můžeme použít nástroj *yarac* pro příkazový řádek (viz https://yara.readthedocs.io/en/stable/commandline.html)

2.1 Obecný formát zdrojových souborů

Zdrojové soubory v jazyce YARA obsahují sadu pravidel. Ta mohou být na sobě úplně nezávislá, ale je také možné se v nich odkazovat na jiná existující pravidla. V době odkazování musí být dané pravidlo již definované.

Kromě pravidel mohou zdrojové soubory obsahovat ještě jednořádkové či víceřádkové komentáře nebo klíčová slova `import` a `include`, kterými buďto importujeme moduly nebo vkládáme jiné zdrojové soubory v jazyce YARA. Za těmito klíčovými slovy následuje název modulu, respektive cesta ke zdrojovému souboru.

2.1.1 Pravidla v jazyce YARA

Pravidla označujeme klíčovým slovem `rule`, za kterým následuje název pravidla, volitelně značky (tagy) a nakonec tělo pravidla ve složených závorkách. Značky nám umožňují filtrovat výstup programu *yara* [2].

Tělo je rozděleno do několika sekcí, jejichž pořadí nelze zaměnit. Jsou jimi metadata (označována klíčovým slovem `meta`), řetězce (`strings`) a podmínka (`condition`). Sekce musí být uvedeny právě v tomto pořadí, případně lze metadata či řetězce vynechat.

Příkladem jednoduchého pravidla je:

```
1 rule sample1 : sample {
2     meta:
3         author = "Vojtech Dvorak"
4         desc = "Ukazkove pravidlo"
5     strings:
6         $s00 = "Welcome"
7     condition:
8         $s00 // Podminka je splnena, pokud soubor obsahuje retezec $s00
9 }
```

Pomocí tohoto pravidla můžeme identifikovat všechny soubory v daném adresáři, které obsahují řetězec `Welcome` na libovolné pozici. Pravidlo také obsahuje metadata tvořená páry `<identifikator> = <hodnota>`, která v tomto případě obsahují jméno autora a popis pravidla.

Za názvem proměnné je zde uvedena také značka `sample`, jež nám umožňuje odfiltrovat z výstupu programu *yara* nálezy na základě pravidel, které touto značkou označeny nejsou (při spuštění programu uvedeme parametr `-t sample`).

Konvence pro pojmenování pravidel, značek a položek metadat odpovídají konvencím pro názvy proměnných v jazyce C. Jedná se o řetězec složený z jakýchkoliv alfanumerických znaků nebo podtržítek (_) o maximální délce 128 znaků, přičemž prvním znakem nesmí být číslice. Velikosti písmen jsou přitom rozlišovány a daný řetězec nesmí kolidovat s některým z klíčových slov jazyka YARA [2].

2.1.2 Sekce metadat

V rámci pravidel je možné specifikovat metadata ve formě `<identifikator> = <hodnota>`. Hodnotou může být textový řetězec, celočíselná nebo pravdivostní hodnota [2].

V sekci s metadaty (v pravidlech jako *meta*) je obvykle uchováno jméno autora, stručný popis pravidla nebo specifikace třídy souborů (třídy malwaru), která je pravidlem popsána. Například se může jednat o úroveň nebezpečnosti dané třídy malwaru.

2.1.3 Sekce s řetězci

V rámci sekce s řetězci (strings) lze definovat sekvence bytů (řetězce), na které je možné se následně odkazovat v rámci podmínky pravidla. Definice se skládá z identifikátoru řetězce a hodnoty (sekvence bytů). Identifikátor řetězce vždy začíná znakem dolaru a je libovolnou posloupností alfanumerických znaků a znaku podtržítka.

Jazyk YARA podporuje tři způsoby jak řetězce specifikovat:

1. **Textové řetězce** – Sekvenci bytů uvedeme pomocí textu ohraničeného znakem `"`. Textové řetězce mohou navíc obsahovat escape sekvence `\n`, `\r`, `\t` apod.

2. **Hexadecimální řetězce** – Hodnotu řetězce zapíšeme pomocí dvojic hexadecimálních číslic, které označují jednotlivé byty řetězce. Od zbytku pravidla oddělujeme tento typ řetězců pomocí složených závorek.

   Součástí může být také operátor |, pomocí kterého můžeme uvést více alternativ pro určitou část řetězce ohraničenou jednoduchými závorkami. Hexadecimální číslice tvořící byte můžou být nahrazeny znakem otazníku, který označuje, že daný byte, resp. nibble, může být jakýkoliv.

   Podobnou funkci mají také tzv. *jumps*. Jedná se o počet nebo rozsah počtu bytů, jejichž hodnota v rámci řetězce může být libovolná. Zapisujeme je pomocí hranatých závorek. Pokud chceme například specifikovat řetězec tvořený byty `AA` a `BB`, přičemž mezi nimi může být dva až pět libovolných bytů, použijeme zápis `{ AA [2-5] BB }`.

3. **Regulární výrazy** – V dřívějších verzích byly regulární výrazy v rámci nástroje YARA zpracovávány knihovnami PCRE[2] a RE2[3]. Od verze 2.0 používá YARA svoje vlastní řešení pro práci s regulárními výrazy. Dle dokumentace však toto řešení poskytuje až na drobné výjimky stejnou funkcionalitu jako knihovna PCRE. Kompletní přehled funkcionality podporované regulárními výrazy v jazyce YARA lze najít v dokumentaci [2]. Regulární výrazy ohraničujeme lomítky (`/`). Bezprostředně za ukončující lomítko můžeme přidat také příznaky `i` a `s`. První označuje, že nezáleží na velikosti písmen a druhý udává, že metaznak `.` označuje také znak nového řádku.

Za definicí řetězce může následovat také kombinace modifikátorů. Může se jednat například o modifikátor `base64`, který způsobí, že řetězec bude hledán v kódování *base64* (budou vyhledávány všechny varianty řetězce v tomto kódování, viz [21]).

V následujícím pravidle jsou uvedeny všechny tři způsoby, jakými lze řetězce specifikovat:

```
1 rule string_rule {
2     strings:
3         $s00 = "Welcome" nocase     // Textovy retezec
4         $r00 = /y+ar{1,10}a/i      // Regularni vyraz
5         $h00 = { AA [1-3] BB CC ( DD | 1? ) }   // Hexadecimalni retezec
6     condition:
7         $s00 and $r00 and $h00
8 }
```

[2] Oficiální webové stránky knihovny PCRE: https://www.pcre.org
[3] Oficiální repozitář knihovny RE2: https://github.com/google/re2

2.1.4 Podmínka pravidla

Povinnou sekcí pravidla je podmínka (condition). Jedná se o pravdivostní výraz, jehož evaluací pro daný soubor je získána pravdivostní hodnota, která udává, zda pravidlo soubor popisuje či nikoliv. Kromě odkazování na řetězce a jiná pravidla nám jazyk YARA poskytuje celou řadu operátorů, které můžeme pro tvorbu podmínek použít. K dispozici jsou logické, relační, aritmetické a další operátory. Kompletní jejich seznam se stručným popisem je dostupný v dokumentaci [2].

Pravdivostní hodnota, kterou představuje identifikátor řetězce (např. `$s00`), je rovna hodnotě `true`, pokud se řetězec v souboru alespoň jednou vyskytuje. V opačném případě je hodnota tohoto symbolu `false`. Kromě řetězců jako takových, můžeme v podmínkách použít také odkaz na počet jejich výskytů (zaměníme-li symbol dolaru za znak `#`) nebo index bytu, na kterém n-tý výskyt řetězce začíná (zaměníme-li `$` za `@` a použijeme hranaté závorky pro odkaz na konkrétní výskyt).

Pravidlo s podmínkou *„Najdi všechny soubory se třemi výskyty řetězce 'Welcome', přičemž první výskyt tohoto řetězce začíná na bytu s indexem 2“* bychom tedy zapsali tímto způsobem:

```
1 rule num_and_offset_rule {
2     strings:
3         $s00 = "Welcome"
4     condition:
5         #s00 == 3 and @s00[1] == 2
6 }
```

Zatímco byty v souborech jsou indexovány od nuly, výskyty řetězců jsou indexovány od jedničky.

Při větším množství řetězců by bylo nepraktické používat jejich identifikátory ve spojení s logickými operátory (jako je tomu např. v pravidle `string_rule` v kapitole 2.1.3). Situace se ještě zkomplikuje, pokud chceme vytvořit pravidlo kontrolující výskyt např. libovolných dvou z desíti možných řetězců.

Z tohoto důvodu zavádí YARA operátor `of`, který jako první operand přijímá kvantifikátor, jenž udává, kolik řetězců z množiny se musí v souboru vyskytovat. Druhým operandem `of` je množina řetězců. Analogicky lze tento operátor použít i s množinou identifikátorů jiných pravidel.

YARA rovněž umožňuje iterovat přes výskyty řetězce pomocí operátoru `for...of`. Lze jej využít také pro iterování přes danou množinu čísel. Díky iterátorům lze snadno formulovat např. podmínku *„Najdi všechny soubory, kde všechny výskyty řetězce 'start' začínají nejvýše na bytu s indexem 1000“*:

```
1 rule iterator_rule {
2     strings:
3         $s00 = "start"
4     condition:
5         for all j in (1..#s00) : (@s00[i] <= 1000)
6 }
```

Ve výše uvedeném případě je využit kvantifikátor `all` označující všechny hodnoty, kterých nabývá proměnná `j` během iterace. Množina čísel je v tomto případě daná celočíselným intervalem začínajícím číslem jedna a končícím počtem výskytů řetězce `$s00`. Krajní hodnoty jsou v intervalu zahrnuty.

V rámci podmínky se lze odkazovat také na symboly z modulů nebo na externí proměnné, které dodáme programu *yara* při spuštění, čímž je možné pravidla parametrizovat.

2.2 Modularita jazyka YARA

Moduly nám umožňují rozšířit funkcionalitu nástroje YARA, přičemž sami jeho tvůrci poskytují uživatelům sadu základních (oficiálních) modulů (PE, ELF, Cuckoo...). Uživatelé ale také mohou vytvářet vlastní moduly pomocí rozhraní pro jazyk C [2, str. 73].

Moduly typicky rozšiřují nástroj YARA o konstanty, vestavěné proměnné, struktury či funkce. Například moduly *ELF* a *PE*, které jsou specializované pro stejnojmenné formáty spustitelných souborů, obsahují symbol `entry_point` s hodnotou, jež je rovna virtuální adrese vstupního bodu.

Dalším zajímavým modulem je *Cuckoo*. V případě malwaru může být problematické spoléhat při jeho identifikaci pouze na přítomnost řetězců. Řetězce mohou být v rámci binárního souboru totiž zakódovány nebo jinak skryty (tzv. obfuskace), aby nebylo snadné malware odhalit. Řešení nám poskytuje tento modul, jenž umožňuje popsat binární soubor rovněž dle jeho chování (behaviorálně). Můžeme také oba přístupy kombinovat [2, str. 60].

Chování souboru je v tomto případě analyzováno v izolovaném prostředí vyvíjeného v rámci open-source projektu *Cuckoo Sandbox* a pro každou analýzu je vytvořeno prostředí nové. Z tohoto důvodu není případným malwarem ohrožen hostitelský systém a jednotlivé běhy behaviorální analýzy se nijak neovlivňují [20, str. 16].

Při použití tohoto modulu musíme však dodat nástroji YARA behaviorální informace získané z *Cuckoo Sandbox* reportem ve formátu JSON, který nám toto prostředí vygeneruje. Užitečným modulem může být také *Console*, díky kterému můžeme realizovat ladící a kontrolní výpisy.

2.3 YARA ve firmě Avast

Nástroj YARA je ve společnosti Avast používán ke svému původnímu účelu, k identifikaci a klasifikaci malwaru. Firma používá interní konvence pro pojmenování pravidel, řetězců a pro informace, jež musí být uvedeny v metadatech pravidla. Kromě oficiálních modulů a sandboxu Cuckoo využívá také vlastní interní moduly a izolovaná prostředí.

Navíc je ve společnosti Avast zavedeno několik rozšíření pro jazyk YARA. Jedná se o sekci s proměnnými v pravidlech (variables) a přetížení operátoru `of`. Tato rozšíření, která byla představena v rámci práce Tomáše Kendera [27], nijak nekolidují s konstrukcemi originálního jazyka YARA. Tudíž všechna pravidla, jež jsou validní v oficiálním jazyce YARA, by měla být validní i v této jeho nadstavbě. Při tvorbě nové verze knihovny pro analýzu tohoto jazyka však musíme s těmito rozšířeními počítat, aby byla nová verze zpětně kompatibilní se současným řešením.

Pro nedostatek nástrojů pro podporu tvorby pravidel v integrovaných vývojových prostředích a textových editorech, byl v rámci Avastu vytvořen open-source projekt *YARA Language Server* (zkráceně YLS). Jedná se o aplikaci, která komunikuje s textovým editorem (s klientem) prostřednictvím protokolu LSP. Díky tomu poskytuje jednotnou podporu pro více různých textových editorů (VSCode, Vim...) [25]. Protokol LSP byl blíže popsán v kapitole 1.1.3.

Jelikož se jedná o nástroj určený pro analýzu pravidel v době jejich editace, poskytuje z hlediska statické analýzy spíše základní funkcionalitu. Ta zahrnuje podporu pro snazší

pochopení pravidel (strukturování zdrojového souboru, kontextové popisy symbolů...), linting, podporu pro udržování jednotného stylu pravidel a další užitečné funkce. Díky integraci s textovými editory a díky rychlosti analýzy poskytuje uživatelům zpětnou vazbu přímo v prostředí, kde je kód vytvářen [25]. Díky těmto vlastnostem YLS umožňuje uživatelům tvořit pravidla efektivněji a s menším množstvím chyb.

Projekt YLS je vyvíjen v jazyce Python s využitím knihovny *pygls*, která implementuje LSP a poskytuje tak rámcové řešení pro tvorbu konkrétních language serverů [36]. Dále je v rámci YLS využívána knihovna Yaramod-v3, jejíž zodpovědností je samotná statická analýza zdrojových souborů v jazyce YARA. Tato knihovna je blíže popsána v kapitole 3.

Kapitola 3

Dosavadní řešení

Yaramod-v3 je open-source knihovna vyvíjená firmou Avast, která poskytuje aplikační programové rozhraní pro syntaktickou a sémantickou analýzu pravidel v jazyce YARA, tvorbu nových pravidel a modifikaci již existujících [33]. Ačkoliv je implementována v jazyce C++, můžeme k ní přistupovat i prostřednictvím jazyka Python. To je možné díky využití knihovny *pybind11* [42].

Knihovna se sestává z následujících čtyř hlavních částí [33, str. 39]:

- Syntaktický a sémantický analyzátor pravidel v jazyce YARA (zodpovědnost třídy `ParserDriver`)
- Generování nových pravidel (`YaraRuleBuilder`)
- Formátování pravidel (`TokenStream`)
- Modifikace pravidel (`ModifyingVisitor`)

Rozhraní této knihovny využívá objektově orientované programování a návrhové vzory *builder* a *visitor*. Yaramod-v3 podporuje jak konstrukci *include*, tak import modulů. Symboly obsažené v modulech jsou popsány prostřednictvím souborů ve formátu JSON. To je důležité zejména pro sémantickou analýzu zdrojových souborů. Uživatelé mohou sami vytvářet soubory s popisem uživatelsky definovaných modulů, takže se používání nestandardních modulů nijak nevylučuje s používáním knihovny Yaramod-v3.

3.1 Syntaktická a sémantická analýza pravidel

Základní funkcionalitou, kterou knihovna poskytuje, je analýza pravidel ve zdrojovém souboru nebo v řetězci. Jak již bylo uvedeno výše, o tento úkol se stará třída `ParserDriver`, která v jedné ze svých metod obsahuje definici lexémů pro jazyk YARA a v jiné zase definici pravidel gramatiky tohoto jazyka. Navíc jsou součástí definice pravidel také sémantické kontroly a jiné akce, které jsou prováděny po redukci na základě daného pravidla.

Samotná syntaktická analýza je prováděna pomocí LALR(1) (lookahead-LR) syntaktického analyzátoru. Jejím výsledkem je objekt třídy `YaraFile`, který agreguje všechny elementy v analyzovaném zdrojovém souboru. V případě, že během analýzy dojde k chybě, je vyhozena výjimka s popisným chybovým hlášením. Analýza je však při prvním nálezu syntaktické nebo sémantické chyby ukončena.

3.2 Tvorba nových pravidel

Rozhraní pro tvorbu nových pravidel je implementováno pomocí návrhového vzoru *builder*, který je řazen mezi tzv. *tvořivé návrhové vzory* (creational design patterns).

Jeho hlavní myšlenkou je oddělit tvorbu komplexního objektu od jeho chování. V situaci, kdy instance třídy má velké množství atributů, které navíc mohou být dalšími objekty, by tvorba nových instancí vyžadovala volání složitého a nepřehledného konstruktoru. Situaci bychom se mohli pokusit vyřešit pomocí dědičnosti, nicméně, pokud bychom potřebovali využívat mnoho různých variant tohoto komplexního objektu, potřebovali bychom i stejně velké množství těchto tříd [39].

Řešením je právě návrhový vzor *builder*, který instanciaci objektu a inicializaci jeho atributů deleguje na jiné objekty – buildery. Pomocí metod těchto objektů tak můžeme snadno inicializovat části komplexního objektu. Jejich voláním určujeme, jak bude onen konkrétní objekt vypadat. Pokud některou z variant objektu vytváříme opakovaně, můžeme volání metod daného builderu agregovat skrze objekt označovaný jako *director* [17, str. 129].

Komplexními objekty, jež je zapotřebí instanciovat a inicializovat, jsou v případě Yaramodu pravidla v jazyce YARA. Instance třídy `YaraRuleBuilder` plní funkci objektu typu builder. Voláním metod tohoto objektu inicializujeme jednotlivé části pravidla (tagy, řetězce, podmínku...) a nakonec voláním `get` vytvoříme objekt třídy `Rule` [33, str. 17].

Ačkoliv je způsob tvorby nových pravidel implementačně zajímavý, nejedná se o statickou analýzu jazyka, a tudíž se v této práci toto téma dále nerozvíjí.

3.3 Zpřístupnění výsledků analýzy

Výsledky statické analýzy jsou uživateli zpřístupněny prostřednictvím struktury objektů. Jedná se o model zdrojového kódu, jehož struktura odráží sémantiku zdrojového souboru. Z tohoto důvodu jsou třídy, které se na tvorbě této reprezentace podílejí, dále označovány také jako *sémantické rozhraní*.

Uživatel tedy primárně manipuluje s objekty, které reprezentují celé konstrukce, jež mají v kontextu jazyka YARA svoji sémantiku. Na druhou stranu, je-li to zapotřebí, uživatel má možnost přistoupit i k jednotlivým tokenům tvořícím tyto konstrukce a určit tak např. jejich pozici v rámci zdrojového kódu. Jak již bylo uvedeno výše, výsledkem analýzy je objekt třídy `YaraFile`, jenž jako svoje atributy obsahuje seznam pravidel a konstrukcí `import`. Pokud je použita konstrukce `include`, tyto seznamy obsahují i elementy, které jsou přítomny ve vložených souborech.

Pravidla obsažená ve zdrojových souborech jsou reprezentována jako objekty třídy `Rule` a obsahují další seznamy, které odkazují na metadata daného pravidla, značky a řetězce. Důležitou součástí rozhraní pravidla je také odkaz na stromovou strukturu reprezentující podmínku.

Přistupovat k jednotlivým objektům, které tvoří podmínku lze pomocí návrhového vzoru *visitor*. Jedná se o *návrhový vzor chování* (behavioral design pattern). Ten je využíván například v situacích, kdy je zapotřebí provádět nad každým uzlem v grafu rekurzivně určitou operaci. Uzly přitom mohou být instancemi různých tříd, které mají odlišná rozhraní. Operace pro různé typy uzlů navíc může mít různý průběh.

Naivním řešením by bylo tyto operace definovat jako metody ve třídách těchto uzlů. To však vyžaduje zásah do každé třídy, což neprospívá udržovatelnosti kódu, a navíc je velká šance, že do něj takto zaneseme chyby. Přidání nového druhu operace by vyžadovalo tentýž proces zopakovat [40].

Podobně jako u návrhového vzoru builder, tak i zde je řešením delegovat provádění operací na speciální objekt – *visitor*. Tento objekt obsahuje sadu metod, z nichž je každá určena pro jinou třídu. Tyto metody jsou tak seskupeny pouze v rámci jedné třídy a ve třídách uzlů definujeme většinou pouze triviální metodu (např. `accept`), která volá metodu visitoru pro danou třídu [17, str. 455].

V knihovně Yaramod je tento návrhový vzor používán pro průchod podmínkami ve zdrojovém souboru. Rozhraní definuje bázovou třídu `ObservingVisitor`, ze které pomocí dědičnosti můžeme vytvořit vlastní visitory, jež budou během průchodu podmínkou provádět libovolné operace [33, str. 11].

3.4 Modifikace existujících pravidel

Rozhraní pro modifikace podmínek existujících pravidel využívá opět návrhový vzor visitor a zvláštní třídu `ModifyingVisitor` [33, str. 31]. Celý proces modifikace podmínek funguje velmi podobně jako při prostém průchodu. Kromě podmínek lze modifikovat i další komponenty pravidel, jako jsou metadata, řetězce a tagy. V těchto případech je však proces modifikace jednodušší, neboť je prováděn pouze skrze rozhraní příslušných objektů bez využití třídy `ModifyingVisitor`.

Stejně jako v případě rozhraní pro tvorbu nových pravidel, tak ani zde se nejedná o součást statické analýzy, a tudíž toto téma nebude dále rozvíjeno.

3.5 Aplikace dosavadního řešení

Tato knihovna je užitečná zejména v projektech, kde je zapotřebí manipulovat s pravidly automaticky, prostřednictvím programu nebo je staticky analyzovat. Jednou z aplikací, která tuto knihovnu používá právě pro statickou analýzu a podporu tvorby pravidel je zmíněný projekt YLS (viz 2.3).

Dalším zajímavým projektem je *Arya*[1], který, na základě pravidel v jazyce YARA, umožňuje vygenerovat pseudomalware, tj. binární soubor, jenž sice odpovídá pravidlům, ale není nijak nebezpečný. Ten může být použit například pro testování v případech, kdy není možné použít reálný malware.

[1] Oficiální repozitář projektu Arya: https://github.com/claroty/arya

Kapitola 4

Tvorba knihovny pro inkrementální statickou analýzu

Hlavním cílem této bakalářské práce je vytvořit novou knihovnu pro statickou analýzu jazyka YARA. Na rozdíl od dosavadního řešení, které je popsáno v předchozí kapitole, by měla umožňovat provádět analýzu inkrementálně. Díky tomu bude optimalizována pro použití v rámci textových editorů a IDE, které jsou jednou z hlavních motivací pro její tvorbu. Vzhledem k tomu, že se bude jednat o novou verzi knihovny Yaramod-v3 a bude v některých aplikacích její náhradou, je toto nové řešení dále označováno jako *Yaramod-v4* či pouze jako knihovna.

V této kapitole jsou popsány všechny fáze vývoje programového řešení od specifikace požadavků přes návrh a implementaci až po testování. Nakonec je také popsána implementace tohoto nového řešení s projektem YLS a příkladů použití knihovny.

4.1 Specifikace požadavků

Nejzásadnějším požadavkem, který je ostatně také zmíněn v zadání, je podpora inkrementálního přístupu ke statické analýze. Na základě používání současné verze knihovny (Yaramod-v3), byly formulovány také další nároky na novou verzi. Zásadní funkcionalita knihovny Yaramod-v4 a vlastnosti, kterými by měla disponovat, jsou:

- Podpora inkrementální statické analýzy
- Tolerance vůči chybám
- Podpora uživatelsky definovaných modulů
- Možnost specifikovat vstupní zdrojový kód jak pomocí bezprostředního řetězce, tak skrze cestu k souboru se zdrojovým kódem
- Oddělení elementů jednotlivých zdrojových souborů (při použití `include`)
- Možnost lokalizovat některé sémantické celky ve zdrojovém souboru (pravidla, řetězce...)
- Serializace výsledků analýzy

Ačkoliv to není v seznamu výše uvedeno, bude rovněž vhodné, aby rozhraní knihovny bylo podobné rozhraní její dosavadní verze. Díky tomu se bude snáz integrovat s projekty, které využívají Yaramod-v3.

Podporou inkrementální statické analýzy je myšlena možnost provedení syntaktických a základních sémantických kontrol (kontrola datových typů, kolize identifikátorů apod.) nad zdrojovými soubory při vytváření výstupní reprezentace zdrojového kódu. Díky této reprezentaci bude knihovna také umožňovat pokročilejší statickou analýzu, jež bude řízena klientskou aplikací (např. YLS). Může se jednat například o optimalizace podmínek pravidel zjednodušením pravdivostních výrazů.

Od nové verze knihovny se také očekává, že přítomnost chyby ve zdrojovém kódu neukončí rozpracovanou analýzu a bude tak moci být identifikováno více chyb než pouze jedna. Toto chování by měla zaručit právě tolerance vůči chybám.

Jelikož moduly jsou v jazyce YARA velmi používané, je zapotřebí, aby nové řešení podporovalo jejich import. Protože uživatelé mohou využívat své vlastní, mělo by být možné dodat analyzátoru jejich specifikace. Z důvodu zpětné kompatibility bude ideální podporovat stejný formát specifikace jako u dosavadního řešení (JSON, viz 3).

Schopnost provádět statickou analýzu jak ze souboru, tak z řetězce bude výhodná zejména pro co největší flexibilitu použití knihovny. Například pro použití v textových editorech bude vhodné podporovat analýzu bezprostředního řetězce, která je v dosavadním řešení omezená a z toho důvodu je upřednostňováno provádění analýzy nad souborem. To dále limituje jeho aplikace (např. YLS), protože není možné provádět analýzu v reálném čase[1].

V některých případech je nutné přistoupit pouze k elementům specifického zdrojového souboru a odfiltrovat tedy všechny elementy z jiných souborů, jež byly vloženy pomocí `include`. Tento problém by měl řešit požadavek na oddělení elementů jednotlivých souborů. Pokud chceme pomocí dosavadního řešení přistoupit pouze k pravidlům obsaženým ve specifickém souboru, musí být objekty reprezentující pravidla filtrovány klientskou aplikací. Nové řešení by mělo poskytnout takové rozhraní, aby toto filtrování nebylo zapotřebí.

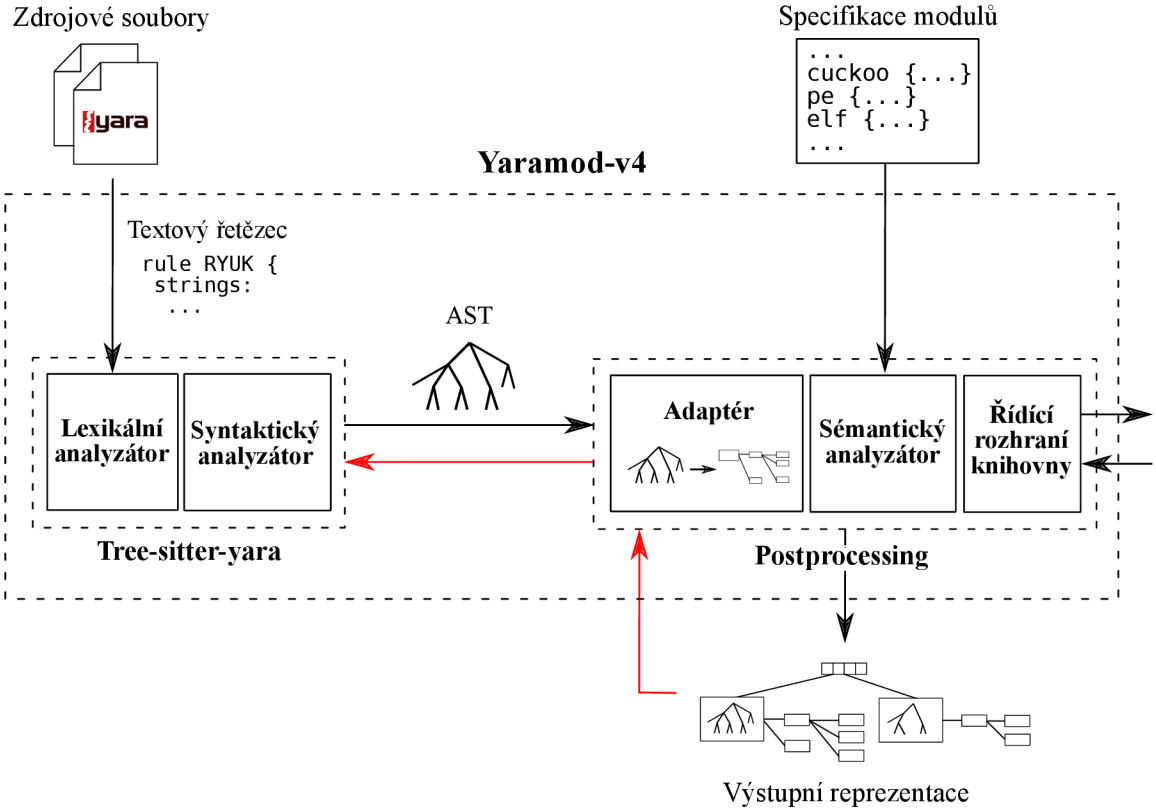
Požadavek na lokalizaci sémantických celků je důležitý zejména pro analýzu v rámci textových editorů a v IDE. Pokud například klientská aplikace poskytuje navigaci na definice symbolů, je potřeba znát jejich polohu v rámci zdrojového souboru. Tuto informaci by měla poskytnout právě knihovna.

Serializací je zde míněn převod struktur reprezentující výsledek analýzy zpět, do podoby formátovaného zdrojového kódu, jenž bude sémanticky shodný se vstupním zdrojovým kódem. Funkcionalita umožňující serializaci je sice méně důležitým požadavkem, nicméně bude vhodná pro testování a ladění knihovny. Pro uživatele může být ovšem také benefitem, neboť mu poskytne možnost automaticky formátovat zdrojové soubory v jazyce YARA.

4.2 Návrh knihovny a jejího rozhraní

Na základě obecných znalostí o jazyce YARA, fungování dosavadního řešení, inkrementální statické analýze kódu a také na základě specifikace požadavků byl vyhotoven návrh knihovny Yaramod-v4. Návrh probíhal ve dvou fázích metodou shora dolů. Nejprve byla navržena architektura knihovny a poté byla provedena objektově orientovaná analýza problému, z níž vzešly třídní diagramy, které novou knihovnu modelují. Díky tomuto přístupu

[1] Vždy je analyzován pouze uložený obsah souboru, a tudíž nedojde k reanalýze, pokud uživatel text v souboru upravil, aniž by jej uložil.



Obrázek 4.1: Návrh architektury knihovny Yaramod v4 – Schéma zobrazuje knihovnu, její komponenty a transformace zdrojových kódů, které provádí. Červené hrany znázorňují předávání datových struktur při spuštění inkrementální statické analýzy. Nejprve je výstupní reprezentace s označením změn předána zpět modulu postprocessing. Ten požádá modul tree-sitter-yara o aktualizaci AST a poté sám provede aktualizaci výstupní reprezentace. Adaptér se stará o konverzi AST na výstupní reprezentaci. Knihovna dále obsahuje řídící rozhraní, pomocí kterého je možné ji ovládat.

došlo k dekompozici celé knihovny na moduly a rozdistribuování zodpovědností modulů mezi jednotlivé třídy.

Návrh architektury vychází hlavně z obecných znalostí o fungování statické analýzy. Jak již bylo uvedeno v kapitole 1.1.4, první fází statické analýzy kódu je převést jej do interní reprezentace. Pro dosažení funkcionality srovnatelné s dosavadním řešením nebude dostačující provést pouze lexikální analýzu, ale bude muset být provedena i analýza syntaktická. Interní reprezentací zdrojového kódu je z tohoto důvodu v případě Yaramod-v4 abstraktní syntaktický strom (AST) vytvořený inkrementálním syntaktickým analyzátorem. Ten se stará i o aktualizaci AST při úpravách vstupního řetězce.

Jako inkrementální lexikální a syntaktický analyzátor je využit open-source projekt *tree-sitter*. Zejména z tohoto důvodu se architektura knihovny sestává ze dvou základních modulů, které jsou do velké míry dvěma samostatnými projekty.

Prvním z nich je *tree-sitter-yara*, inkrementální lexikální a syntaktický analyzátor jazyka YARA založený na projektu tree-sitter (více v 4.3.2). Výstupem této části je AST a poskytuje tzv. *syntaktické rozhraní* pro druhý modul knihovny, jehož úkolem je konverze AST

na výstupní reprezentaci a provádění sémantických kontrol. Tento modul je dále označován jako *postprocessing*.

Uživatel knihovny Yaramod-v4 je díky tomuto přístupu odstíněn od AST a manipuluje pouze s reprezentací kódu tvořenou objekty. Ta bude výsledky analýzy prezentovat na vyšší úrovni abstrakce. Třídy a metody, které se podílejí na vytváření této výstupní reprezentace, jež je produkována modulem postprocessing, jsou dále označovány také jako *sémantické rozhraní*.

Kromě toho, že zodpovědností modulů je vytváření příslušné reprezentace zdrojového kódu, mají také za úkol aktualizovat danou reprezentaci v případě uplatnění inkrementální analýzy po editacích zdrojového souboru. Návrh architektury ve formě jednoduchého schématu je možné vidět na obrázku 4.1.

Dva základní moduly je dále možné dělit na jednodušší komponenty. Tree-sitter-yara je možné rozdělit na lexikální a syntaktický analyzátor. Část postprocessing lze dále dekomponovat na řídící rozhraní, sémantické rozhraní, sémantický analyzátor a adaptér, který zajišťuje převod syntaktického rozhraní na sémantické rozhraní knihovny.

Jak je možné vidět, součástí výstupní reprezentace je i AST, který jinak knihovna využívá pouze jako svoji interní strukturu. Přestože se nepředpokládá, že by k němu uživatel přistupoval, princip fungování inkrementální syntaktické analýzy vyžaduje pro svoje fungování existující AST (viz 1.2.3) a bude tedy potřebný pro její provedení po editaci zdrojového kódu.

V rámci modulu tree-sitter-yara je hlavním úkolem vytvořit gramatiku jazyka YARA a popis tokenů ve specifickém formátu, vyžadovaném knihovnou tree-sitter. O samotnou lexikální a syntaktickou analýzu se stará tato knihovna implementovaná v jazyce C, jež pro definice poskytuje sadu funkcí. V případě tohoto modulu tedy nemá příliš smysl znázorňovat jej pomocí třídních diagramů.

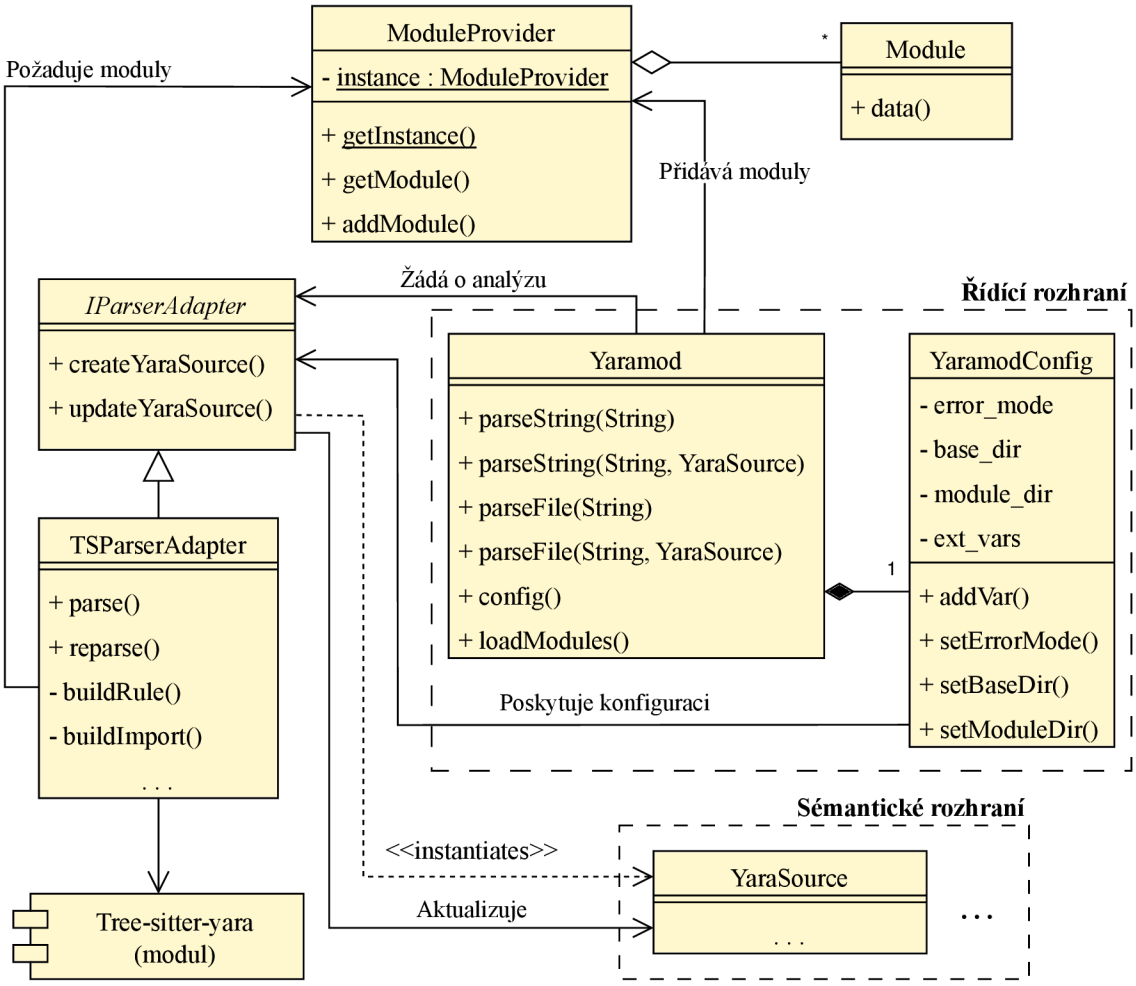
Druhý modul bude implementován v jazyce C++ (důvod výběru tohoto jazyka je uveden v 4.3.1) s využitím objektově orientovaného paradigmatu. Z tohoto důvodu jej lze modelovat právě pomocí diagramů tříd. Diagramy tříd obsažené v této práci pro větší přehlednost neobsahují všechny detaily (chybí např. kompletní signatury metod, řešení modifikátorů pravidel a řetězců, některé metody, tabulky symbolů…).

Třídní diagram řídícího rozhraní knihovny je možné vidět na obrázku 4.2. Řídící rozhraní se skládá ze dvou tříd, `YaramodConfig` a `Yaramod`. První z těchto tříd, `YaramodConfig`, udržuje stav statického analyzátoru mezi jednotlivými běhy analýzy. Konkrétně uchovává uživatelská nastavení a spravuje externí proměnné. V rámci uživatelských nastavení je možné deaktivovat toleranci chyb, vypnout analýzu vložených zdrojových souborů, nastavit cestu k bázovému adresáři se zdrojovými soubory nebo k adresáři se specifikací modulů.

Objekt třídy `Yaramod` reprezentuje statický analyzátor jako takový. Uživatel může pomocí něj staticky analyzovat kód a také přistupovat k příslušné instanci `YaramodConfig`. U metod pro provádění analýzy, `parseFile` a `parseString`, je využito přetěžování funkcí v jazyce C++. Pro každou z těchto funkcí existují dvě implementace.

První je použita při prvotní neinkrementální analýze, kdy uživatel skrze argumenty poskytne pouze zdrojový kód. V tomto případě tyto metody navrací ukazatel na objekt třídy `YaraSource`, která je základem sémantického rozhraní knihovny.

Pokud uživatel kromě zdrojového kódu poskytne metodám už existující instanci třídy `YaraSource`, je použita jejich druhá implementace, jež provádí inkrementální statickou analýzu. Dojde tak k aktualizaci objektu typu `YaraSource` a jeho komponent (byla-li zapotřebí).



Obrázek 4.2: Třídní diagram modelující řídící rozhraní knihovny a s ním související třídy – Uživatel v rámci řídícího rozhraní manipuluje se třídami `Yaramod` a `YaramodConfig`. Třída `YaramodConfig` udržuje uživatelská nastavení a poskytuje je adaptéru provádějícímu konverzi AST na výstupní reprezentaci tvořenou objekty. Adaptér se poté stará také o aktualizaci.

Ačkoliv je jedním z požadavků na knihovnu tolerance vůči chybám ve zdrojovém kódu, mohou existovat projekty, kde tato vlastnost není zapotřebí a kde se předpokládá vyhození výjimky při nalezení chyby. Právě kvůli tomu umožňuje `YaramodConfig` toleranci chyb deaktivovat.

Cesta k bázovému adresáři bude uplatněna ve chvíli, kdy uživatel analyzuje kód přímo z řetězce a vyskytuje se v něm konstrukce `include` s relativní cestou. Díky specifikaci cesty k bázovému adresáři je možné vložené zdrojové soubory jednoduše vyhledat. Dále je možné skrze konfiguraci také plně vypnout analýzu takto vložených souborů. To může být užitečné ve chvíli, kdy klientská aplikace ošetřuje vkládání souborů svým vlastním způsobem nebo pokud chceme provést sémantické kontroly, které vložené soubory nevyžadují. Nejdůležitějším uživatelským nastavením je nejspíše cesta k adresáři se specifikací modulů, která je prohledána, nalezené soubory jsou deserializovány a v podobě objektu `Module` jsou uchovány v jediné instanci třídy `ModuleProvider`. Pro udržování právě jedné instance tohoto typu je zde aplikován návrhový vzor *jedináček* (singleton) [17, str. 172]. Díky tomuto návr-

hovému vzoru je možné k modulům přistoupit odkudkoliv. Po přidání nového modulu do adresáře nebo při změně cesty k tomuto adresáři, může uživatel moduly znovu nahrát skrze volání metody `loadModules`.

Objekt typu `Yaramod` si dále udržuje objekt, jenž implementuje rozhraní abstraktní třídy `IParserAdapter`. Jedná se o komponentu provádějící konverzi AST, který poskytuje modul tree-sitter-yara, na výstupní reprezentaci, tvořenou instancemi tříd sémantického rozhraní. Tato část je označována také jako adaptér rozhraní. Ačkoliv existuje stejnojmenný návrhový vzor, nejedná se v tomto případě přímo o jeho aplikaci, protože rozhraní modulu tree-sitter-yara není objektově orientované, a tudíž zde nelze tento vzor v originální podobě uplatnit.

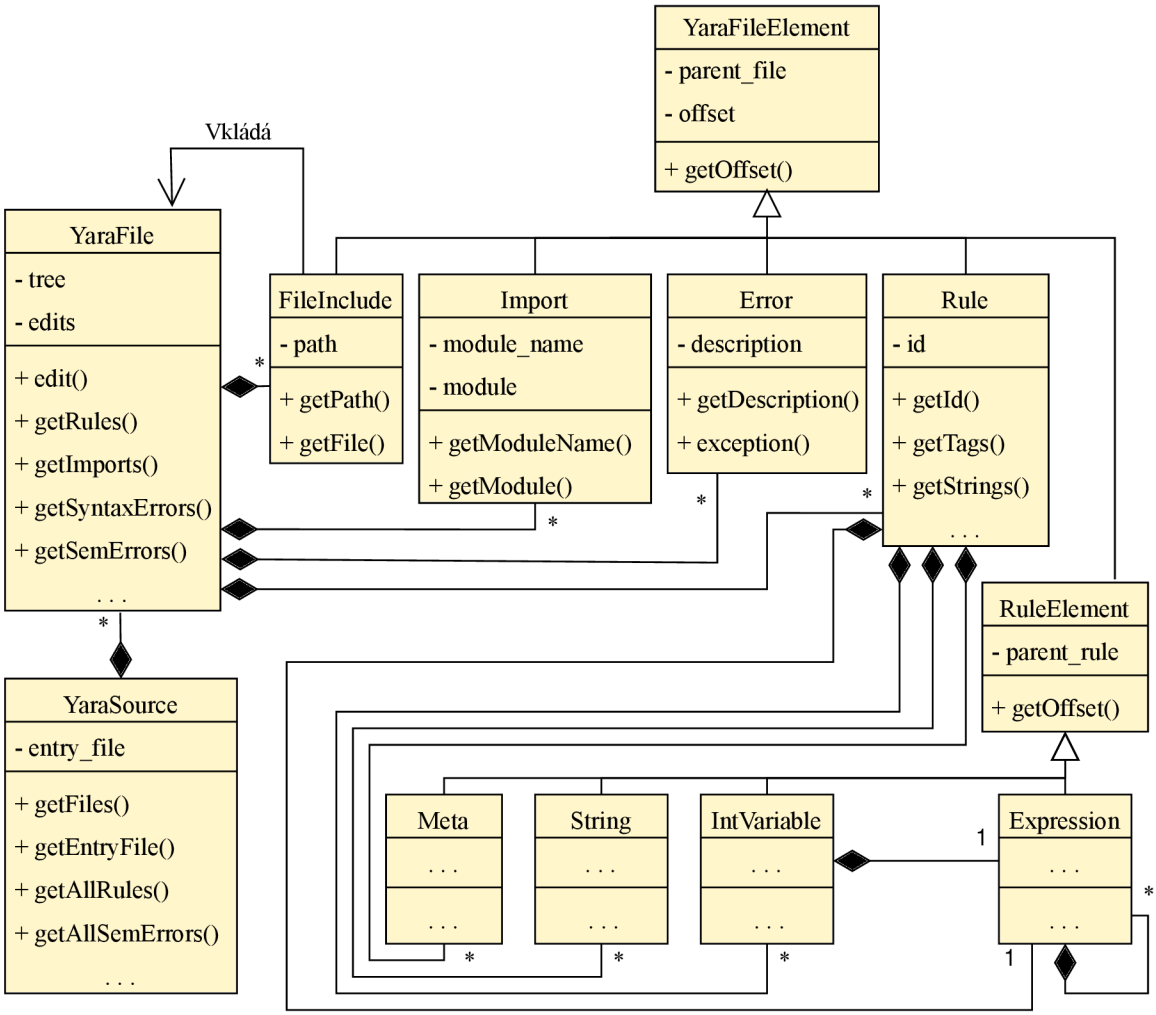
Abstraktní třída `IParserAdapter` je zde použita z důvodu větší flexibility. Díky ní je odstíněno řídící rozhraní od konkrétní implementace adaptéru a kdyby tedy v budoucnu došlo např. ke změně rozhraní modulu tree-sitter-yara, může být řídící rozhraní zachováno. Rozhraní této třídy implementuje však zatím pouze třída `TSParserAdapter`. Instance této třídy komunikuje pomocí sady funkcí poskytovaných knihovnou tree-sitter s modulem tree-sitter-yara a vytváří výstupní reprezentaci zdrojového kódu pomocí svých privátních metod `buildRule`, `buildImport` atd. V případě inkrementální analýzy se stará o aktualizaci tohoto modelu zdrojového kódu.

Dále se tento objekt stará také o identifikaci některých syntaktických chyb, které nebylo možné kvůli použití syntaktického analyzátoru tree-sitter odhalit, nebo jejich odhalení při syntaktické analýze nebylo žádoucí (více v kapitole 4.3). Částečně také provádí kontroly sémantické. Kontroly, které jím realizovány nejsou, jsou uskutečňovány samotným sémantickým rozhraním. To, která třída má danou kontrolu na starosti, závisí na její povaze. Například kontrola toho, zda je symbol definován, je prováděna objektem třídy `TSParserAdapter`. Naproti tomu kontrola duplicitních identifikátorů řetězců je již zodpovědností objektu `Rule`, neboť ten má za úkol správu řetězců, jež mu náleží.

Další velkou částí modulu postprocessing je sémantické rozhraní knihovny Yaramod-v4. Tato část je popsána diagramem tříd na obrázku 4.3. Struktura tohoto rozhraní je na úrovni tříd a jejich interakce do velké míry stejná jako u knihovny Yaramod-v3 [33] kvůli zpětné kompatibilitě. Přesto jsou zde koncepty, které v rozhraní předchozí verze knihovny nenajdeme. Konkrétně jde o třídy `YaraSource`, `FileInclude` `YaraFileElement`, `Error` a `RuleElement`.

Zásadním vztahem mezi třídami je zde kompozice. Instance typu `YaraFile` představuje právě jednu *překladovou jednotku*. Pro každý objekt tohoto typu byl právě jednou spuštěn lexikální a syntaktický analyzátor. Modul tree-sitter-yara tedy pro každou tuto jednotku vytváří simuluje tvorbu derivačního stromu, vytvoří AST a ten je poté konvertován na výstupní reprezentaci adaptérem. Objekt třídy `YaraFile`, reprezentující zdrojový soubor v jazyce YARA, je tvořen libovolným počtem objektů typu `FileInclude`, `Import` a `Rule`. Tyto typy v daném pořadí reprezentují konstrukce `include`, `import` a pravidlo v jazyce YARA. Jelikož objekty těchto typů nedávají samostatně smysl, jsou s `YaraFile` ve vztahu kompozice.

Instance `FileInclude` si udržuje referenci na další objekt typu `YaraFile`, čímž vytváří stromovou strukturu překladových jednotek. Tato stromová struktura sice dobře odráží to, jakým způsobem jsou zdrojové soubory vkládány jeden do druhého, nicméně neposkytuje vhodný pohled na možné závislosti mezi nimi. Pokud například vložíme do zdrojového souboru další soubory pomocí konstrukce `include`, v posledním takto vloženém souboru je možné odkazovat také symboly obsažené i ve dříve vložených souborech. Ty jsou však z pohledu stromové reprezentace sourozeneckými uzly nebo jejich potomky. To je důležité reflektovat při implementaci sémantických kontrol.



Obrázek 4.3: Třídní diagram sémantického rozhraní knihovny Yaramod-v4

Celou sadu překladových jednotek, jež jsou jedna do druhé vkládány, obaluje instance třídy `YaraSource`. Ta si udržuje buffer s ukazateli na všechny objekty třídy `YaraFile`, které jsou dostupné ze vstupní překladové jednotky (entry file). Obsah této překladové jednotky je explicitně poskytnut uživatelem skrze cestu ke zdrojovému souboru či skrze řetězec se zdrojovým kódem. Díky tomuto bufferu je snadné kontrolovat, zda není některý ze zdrojových souborů vložen dvakrát. Zároveň objekt třídy `YaraSource` poskytuje uživateli seznam všech dostupných pravidel, chyb atd. ve všech zdrojových souborech (ve vstupní překladové jednotce i ve vložených).

Dalším zajímavým prvkem je třída `Error`, což je bázová třída pro všechny třídy, jež reprezentují chyby. Knihovna Yaramod-v3 ohlašuje chyby ve zdrojovém souboru vyhozením výjimky. Zde jsou chyby primárně reprezentovány jako objekty[2], jež jsou uchovány v rámci příslušného objektu typu `YaraFile`. Ten z pohledu chyb představuje zdrojový soubor, v němž se chyba vyskytla.

[2]Deaktivování tolerance chyb má ze následek vyhození výjimky při výskytu první chyby, jako je tomu u dosavadního řešení.

Zásadní metodou `YaraFile` je z hlediska inkrementální statické analýzy `edit`. Pomocí ní může uživatel nebo klientská aplikace ohlásit editaci zdrojového kódu a následně spustit inkrementální analýzu pro aktualizaci výstupní reprezentace. Pro obecnější použití knihovny se předpokládá, že editací může být mezi prováděním analýz libovolný počet. Zároveň pro odstínění uživatele od implementačních detailů inkrementálního přístupu je vhodné, aby i těsně po provedení editací byl schopen přistoupit k výstupní reprezentaci, byť bude zastaralá. Z těchto důvodů jsou editace shromažďovány do bufferu a aplikovány až těsně před inkrementální analýzou. To také umožňuje snadnou implementaci metody `undo`, neboť pro zrušení zatím neaplikovaných modifikací zdrojového kódu stačí buffer vyčistit.

Komponenty objektů třídy `YaraFile` jsou ukládány prostřednictvím tabulek, jež jsou vždy indexovány dle bytového offsetu, na němž začínají ve zdrojovém souboru. Indexování podle pozice ve zdrojovém souboru je důležité zejména pro podporu inkrementálního přístupu. Analyzátor musí být totiž schopen určit všechny objekty, které mohly být editací ovlivněny. Bytové offsety editovaných částí dokumentu a uchovaná pozice objektu budou zásadní pro určování objektů, které je zapotřebí aktualizovat. Jelikož uchování bytového posunu jejich začátku a další vlastnosti (např. délka syntaktické konstrukce, kterou reprezentují) jsou společným rysem všech komponent, jsou vytknuty do bázové třídy `YaraFileElement`.

Objekt typu `Rule` lze dále dekomponovat na objekty reprezentující metadata (třída `Meta`), řetězce (`String`), interní proměnné a podmínku pravidla (`Expression`). Dále si udržuje seznam značek (tagů). Komponenty, které tvoří pravidlo, mají společnou bázovou třídu `RuleElement`. Tato třída je navíc odvozená od již zmíněné třídy `YaraFileElement`, neboť prvek pravidla je zároveň prvkem zdrojového souboru. Třída `RuleElement` přepisuje některé z metod pro uchovávání bytového offsetu a přístup k němu (v diagramu znázorněno pouze metodou `getOffset`), protože v případě komponent pravidel je uchováván pouze relativní bytový offset vůči začátku pravidla, ke kterému náleží. Díky tomu není nutné udržovat konzistenci pozic všech komponentů pravidla, ale pouze pravidla jako takového. To souvisí s granularitou aktualizací výstupní reprezentace během inkrementální analýzy, jež je popsána blíže v kapitole 4.3. Stejně jako v případě překladové jednotky, tak i pravidla mají svoje tabulky, v nichž uchovávají své řetězce, metadata a interní proměnné.

Třída `Expression` představuje výraz v podmínce pravidla a hodnoty interních proměnných. V diagramu tříd 4.3 je možné vidět, že s třídami reprezentující pravidlo a interní proměnné je opět ve vztahu kompozice, neboť výraz sám o sobě pozbývá smysl. Jak pravidlo, tak proměnné mají pouze jednu komponentu tohoto typu. `Expression` je opět bázovou třídou celé hierarchie tříd (na diagramu není zobrazena), z nichž každá reprezentuje nějaký operátor jazyka YARA, literál, či symbol. Operátory mohou mít v závislosti na jejich aritě různý počet operandů, a proto i objekt typu `Expression` může být složen z N dalších objektů tohoto typu. Vzniká tak stromová struktura. To je v diagramu tříd 4.3 modelováno opět vztahem kompozice, která má třídu `Expression` jak za komponentní, tak za kompozitní prvek.

Pro efektivní průchod tímto stromem je využit návrhový vzor visitor, což je stejný přístup jako v dosavadním řešení [33]. Tento návrhový vzor je blíže popsán právě v kapitole 3, která se mu věnuje.

Serializace objektů je zajišťována pomocí abstraktní třídy `Printable`, jejíž rozhraní disponuje metodou `getTextFormatted`. Všechny serializovatelné objekty jsou z této třídy odvozeny. V diagramu tříd tato část rozhraní pro větší přehlednost není znázorněna, neboť kromě tříd `YaraFileElement` a `RuleElement` jsou z ní odvozeny všechny uvedené třídy.

4.3 Implementace

Po vytvoření návrhu byla knihovna Yaramod-v4 realizována. Díky tomu, že byla v rámci návrhu rozdělena na dva moduly, jež na sebe navazují, bylo možné i fázi implementace rozdělit na dvě po sobě jdoucí části. Nejdříve byl vytvořen modul tree-sitter-yara, respektive gramatika pro knihovnu tree-sitter. Po dokončení této komponenty byl implementován modul postprocessing.

4.3.1 Použité technologie

Co se implementačního jazyku týče, pro knihovnu jako takovou byl vybrán jazyk C++. Vzhledem k tomu, že se jedná o kompilovaný jazyk, je pro tento účel vhodný zejména díky rychlosti, kterou je kód v tomto jazyce prováděn. Jak již bylo uvedeno v teoretické části, rychlost provádění statické analýzy je jedním z jejích zásadních parametrů.

Oproti alternativě, kterou zadání umožňuje pro tento účel použít, jazyku Rust, je C++ výhodnější zejména z důvodu snazšího zajištění kompatibility mezi aplikacemi využívajícími dosavadní verzi knihovny a Yaramod-v4. Na druhou stranu je zapotřebí výsledek důkladněji otestovat, protože C++ na rozdíl od Rustu negarantuje paměťovou bezpečnost. Konkrétně je používán standard C++20.

Dále je zapotřebí použít jazyka Python, protože klientské aplikace využívají Yaramod-v3 také skrze rozhraní v tomto jazyce. Prostřednictvím tohoto rozhraní je realizována integrace s projektem Yara Language Server, která je také součástí zadání práce.

Kromě výše uvedých jazyků byl také použit JavaScript pro specifikaci gramatiky jazyka YARA. Pro překlad knihovny do binární podoby je využíván překladových systém *CMake* a pro generování dokumentace nástroj *Doxygen*. Pro testovaní rozhraní v jazyce Python je použita knihovna *pytest*. Další moduly a rámcová řešení pro jazyk Python využívá projekt YLS [25]. Součástí programového řešení této práce jsou nicméně pouze ty části projektu, kde došlo k nahrazení knihovny Yaramod-v3 za novou verzi. Z tohoto důvodu zde nejsou závislosti YLS uvedeny.

Mimo standardních knihoven jazyka C++ a vestavěných modulů jazyka Python bylo při vývoji Yaramod-v4 využito několik knihoven a rámcových řešení, které jsou dílem jiných autorů. Tyto projekty, jež jsou rovněž uvedeny se všemi náležitostmi na přiloženém paměťovém médiu, jsou popsány v následujících kapitolách.

Tree-sitter

Knihovna *tree-sitter* [7] plní naprosto zásadní úlohu v rámci modulu tree-sitter-yara, jehož zodpovědností je inkrementální lexikální a syntaktická analýza kódu v jazyce YARA. Princip fungování tohoto nástroje byl popsán v kapitolách 1.1.4 a 1.2.3. Skládá se ze dvou základních komponent. První její částí je nástroj *tree-sitter-cli*. Ten je v rámci Yaramod-v4 využíván pro generování tabulek pro LR syntaktickou analýzu ve formě hlavičkových souborů před samotným překladem zdrojového kódu knihovny. Tyto tabulky jsou využívány druhou částí knihovny, IGLR syntaktickým analyzátorem *tree-sitter*. Ten je již nedílnou součástí modulu tree-sitter-yara. Tento syntaktický analyzátor je implementován jazyce C, a tudíž se jedná o rychlé a přenositelné řešení, které navíc nemá žádné další závislosti.

Zmíněný nástroj tree-sitter-cli není pouze generátorem tabulek. Disponuje také jednoduchým textovým uživatelským rozhraním či rámcovým řešením pro testování a ladění uživatelsky definovaných gramatik. Podkladem pro generování tabulek, jsou totiž bezkon-

textové gramatiky, které uživatel může specifikovat pomocí jazyka JavaScript a sady k tomu určených funkcí. To umožňuje uživatelům analyzovat různé formální jazyky.

Funkce syntaktického analyzátoru je možné využívat prostřednictvím sady funkcí v jazyce C, ale také skrze rozhraní pro Rust, Python, Javu a spoustu jiných jazyků. Kromě toho je také vestavěn do tree-sitter-cli, což usnadňuje vytváření prototypů gramatik. Více o praktickém používání této knihovny a o její úloze v rámci této práce je uvedeno v kapitole 4.3.2.

JSON for Modern C++

Jedním z požadavků na výslednou knihovnu byla také podpora modulů jazyka YARA a uživatelsky definovaných modulů. Tento požadavek komplikuje provádění sémantické analýzy, neboť musí být použitým symbolům přiřazen jejich kontext. Definice symbolů, které jsou obsaženy v modulech, je zapotřebí knihovně dodat prostřednictvím souborů ve formátu JSON. Jde tak o stejný přístup jako u dosavadního řešení.

Z tohoto důvodu bude zapotřebí, aby knihovna dokázala analyzovat kromě cílového jazyka YARA také soubory ve formátu JSON. K tomuto účelu je v Yaramod-v4 využívána právě knihovna *JSON for Modern C++* [30]. Ta umožňuje deserializaci souborů a řetězců ve formátu JSON na struktury a objekty jazyka C++. Tato reprezentace specifikace modulů je podstatně vhodnější pro určování kontextu symbolů než surový text.

Pybind11

Již bylo dříve uvedeno, že Yaramod-v3 je velmi často využíván skrze rozhraní v jazyce Python. Jelikož se v budoucnu předpokládá nahrazení tohoto dosavadního řešení knihovnou Yaramod-v4, bude rovněž vhodné, aby byla přístupná i z prostředí tohoto jazyka.

Jak je ale uvedeno výše, jako implementační jazyk byl zvolen C++, protože díky tomu bude mít knihovna větší potenciál z hlediska výkonu. Právě knihovna *pybind11* [42] zajišťuje v Yaramod-v4 interoperabilitu mezi C++ a Pythonem. Toto rámcové řešení disponuje dobrou dokumentací a podporou pro všechny běžné kompilátory jazyka C++ (GCC 4.8+, Clang/LLVM 3.3+...). Kromě toho také podporuje veškerou běžně používanou funkcionalitu jazyka C++ (standardní datové struktury, chytré ukazatele apod.).

GoogleTest

Pro průběžné jednotkové testování modulu postprocessing bylo použito rámcové řešení *GoogleTest* [24]. To umožnilo snadné testování nově přidané funkcionality v rámci tohoto modulu replikovatelným způsobem. Kromě samotného provádění automatizovaných jednotkových testů nabízí také možnost jejich filtrování. Toho bylo velmi často využíváno, zejména v situaci, kdy se testovací sady začaly rozšiřovat.

4.3.2 Tree-sitter-yara

První velkou částí programového řešení je gramatika jazyka YARA, která ve spojení s knihovnou pro inkrementální lexikální a syntaktickou analýzu, tree-sitter, tvoří modul tree-sitter-yara. Ten je možné využívat také samostatně pro jednoduchou inkrementální syntaktickou analýzu zdrojových souborů. Díky tomu mohou být realizovány např. syntaktické kontroly a zvýrazňování syntaxe. V rámci procesu statické analýzy kódu má tato část za

úkol konverzi zdrojového kódu v podobě textového řetězce do interní reprezentace vhodné k dalšímu zpracování, kterou je AST.

Inkrementální GLR syntaktický analyzátor, který je součástí knihovny tree-sitter, již disponuje rozhraním potřebným pro napojení na postprocessing. Důležité je tomuto syntaktickému analyzátoru dodat korektní tabulku pro LR syntaktickou analýzu cílového jazyka. Během implementace modulu tree-sitter-yara byla tedy vytvářena bezkontextová gramatika v jazyce JavaScript, na základě které nástroj tree-sitter-cli tuto tabulku vygeneruje. Ta se skládá jednak z popisu pravidel pro neterminální symboly, ale také obsahuje definice terminálních symbolů pro lexikální analyzátor, jenž je rovněž součástí tree-sitteru. Inkrementální lexikální a syntaktický analyzátor tedy nejsou součástí programového řešení této bakalářské práce, ale je jí tato gramatika. Hlavním zdrojem informací pro její tvorbu byla dokumentace jazyka YARA [2]. Před popisem vybraných částí implementace gramatiky je však zapotřebí nejdříve uvést obecné principy, pomocí kterých se gramatiky pro tree-sitter definují.

Definice gramatiky pro knihovnu tree-sitter je zapouzdřena ve speciálním objektu. Tree-sitter pro tento účel definuje funkci `grammar`. Té jsou pravidla gramatiky a popisy tokenů předány prostřednictvím argumentu. Jako argument je očekáván objekt, jehož vlastnosti jsou předepsány knihovnou. Jednou z těchto vlastností je také `rules`. V její hodnotě se očekávají již samotné definice terminálů a pravidla gramatiky, resp. definice neterminálních symbolů gramatiky. Z pohledu uživatele nejsou tyto dva druhy definic nijak odlišeny a zda se jedná o terminální nebo neterminální symbol závisí pouze na dané definici. Tyto definice jsou z hlediska jazyka JavaScript opět vlastnostmi objektu, kde jejich název odpovídá názvu definovaného symbolu. Hodnotami vlastností jsou řetězce, regulárními výrazy či objekty, které jsou navraceny funkcemi předdefinovanými knihovnou tree-sitter. Konvence pro zápis definic využívá tzv. *arrow functions*, což je zkrácený zápis funkce pomocí symbolu `=>`.

Definice terminálních symbolů

V rámci vytvořené gramatiky pro jazyk YARA jsou využívány všechny tři možné způsoby definice terminálních symbolů, a sice řetězcem, regulárním výrazem nebo předdefinovanou funkcí `token`. Předdefinovaná funkce `token` umožňuje použít další funkce poskytované knihovnou tree-sitter, které jsou jinak určeny pro vyjádření syntaktické struktury neterminálního symbolu. Gramatika jazyka YARA tohoto využívá např. pro popis modifikátorů pravidla, který vypadá následujícím způsobem:

```
1 rule_modifier: _ => token(
2     choice(
3         "private",
4         "global"
5     )
6 )
```

Výše uvedený popis udává, že symbol `rule_modifier` je v gramatice terminálem (tokenem ve výsledném AST) a představuje jeden z řetězců `private` nebo `global`. V rozvinuté Backus-Naurově formě (EBNF) bychom ekvivalentní definici zapsali:

```
rule_modifier = "private" | "global"
```

Alternativně by bylo možné pro tento účel použít také regulární výrazy. V tomto případě se však jedná o méně rozšiřitelnou variantu, protože přidáváním dalších variant modifikátorů pravidla by se regulární výraz dále prodlužoval. To by mělo za následek zhoršenou

čitelnost. Regulární výrazy jsou nicméně využívány pro popis literálů a identifikátorů. Dále jsou pomocí nich popsány části hexadecimálních řetězců a regulárních výrazů[3]. Například definice terminálního symbolu, který reprezentuje jeden byte v hexadecimálním řetězci je:

```
_hex_str_byte: _ => /[A-Fa-f\d]{2}|\?[A-Fa-f\d]|[A-Fa-f\d]\?/
```

Pomocí řetězců jsou definovány nepojmenované symboly, které figurují v pravidlech gramatiky. Díky tomu, že nejsou pojmenované, jsou ve výsledném AST tokeny, které je reprezentují, skryté. To usnadňuje jeho zpracování v modulu postprocessing. Toho se využívá např. u čistě syntaktických prvků jazyka YARA (složené závorky ohraničující definici pravidla, dvojtečka oddělující značky pravidla...). Pokud chceme explicitně označit symbol (ať už terminální, nebo neterminální) jako skrytý, použijeme podtržítko jako jeho prefix. To je možné vidět u výše uvedené definice terminálního symbolu `_hex_str_byte`. Jelikož v tomto případě není zapotřebí již kontrolovat žádnou další sémantiku těchto symbolů, jsou ve výsledném AST skryty, což dělá podstromy reprezentující hexadecimální řetězce výrazně úspornější.

Dále je ve výsledné gramatice využívána lexikální precedence. Tento princip je uplatňován v situacích, v nichž není jednoznačné, který token z načteného řetězce vytvořit. Pomocí explicitní definice lexikální precedence můžeme lexikální analyzátor přinutit vytvořit požadovaný token i přes kolizi pravidel. Toho se v gramatice YARA využívá například pro definici jednoho znaku v řetězcovém literálu. Řetězcové literály jsou ve výsledné gramatice definovány jako posloupnosti libovolných znaků nebo řídících sekvencí (escape sequences) a jsou to tedy neterminální symboly. Díky tomu je možné řídící sekvence ve výsledném AST snadno lokalizovat a zkontrolovat je (viz dále). Ostatní znaky v řetězci jsou popsány pomocí:

```
token.immediate(
    prec(LEX_PREC.COMMENT + 1, /[^\"\\\r\n]/)
)
```

Metoda `immediate` objektu `token` poskytnutého knihovnou tree-sitter zakazuje výskyt bílého znaku před terminálním symbolem. Ve spojení s voláním `prec` je specifikována právě lexikální precedence, která je v tomto případě vždy vyšší než precedence komentáře. Díky tomu jsou automaticky všechny znaky, které odpovídají regulárnímu výrazu považovány za součást řetězce a nikoliv za začátek komentáře.

Jedním z problémů, který se při vytváření definic pro terminální symboly gramatiky objevil, je možný výskyt klíčového slova na místě identifikátoru. Tento problém nastává v důsledku toho, že popis identifikátorů pomocí regulárního výrazu odpovídá i klíčovým slovům samotným. Nástroj tree-sitter nenabízí žádný aparát, jak tuto situaci jednoduše ošetřit, neboť jedním z jeho cílů je umožnit definovat gramatiku pro co největší množství jazyků a některé jazyky nezakazují používat klíčová slova jako některé identifikátory[4]. Proto musí být tato situace ošetřena až v modulu postprocessing.

Pomocí výše uvedených mechanismů byly definovány také všechny zbývající terminální symboly jazyka YARA. Tree-sitter nabízí také možnost dodat implementaci vlastního lexikálního analyzátoru v jazyce C. To však pro jazyk YARA nebylo zapotřebí.

[3] Součástí gramatiky jsou také definice neterminálních a terminálních symbolů, které reprezentují regulární výrazy v jazyce YARA.

[4] Příkladem jsou objekty v jazyce JavaScript. Název vlastnosti objektu může být shodný s jakýmkoliv klíčovým slovem. Tedy například `obj = {if: "test"};` je v JS validní.

Definice neterminálních symbolů

Pro definice neterminálních symbolů (pravidel gramatiky) je již nezbytné použít pomocné funkce poskytované knihovnou tree-sitter. Počáteční symbol výsledné gramatiky jazyka YARA je pojmenován jako `yara_file` a je definován jako:

```
yara_file: $ => repeat(
    choice(
        $.include,
        $.import,
        $.rule
    )
)
```

V popisu symbolu `yara_file` je možné vidět aplikaci dalších pomocných funkcí poskytovaných knihovnou tree-sitter, `repeat` a `choice`. První z nich označuje 0 až N opakování symbolu v argumentu a druhá výběr jednoho ze všech symbolů v argumentech. Výše uvedené pravidlo tedy říká, že derivací symbolu `yara_file` můžeme získat libovolnou sekvenci neterminálních symbolů `include`, `import`, `rule`. Argument `$`, obsahuje kontext dané gramatiky, který doplní sama knihovna. Díky tomu je možné se v definici odkazovat na další symboly gramatiky. V EBNF bychom výše uvedené pravidlo zapsali jako:

```
yara_file = { include | import | rule }
```

Důležitým konceptem pro zpracování výsledného AST modulem postprocessing je možnost pojmenování následníku neterminálního symbolu v AST. To je realizováno pomocí funkce `field`. Díky pojmenování je možné dodat uzlům význam a při využívání rozhraní knihovny tree-sitter klientskými aplikacemi (kterou je také právě knihovna Yaramod-v4) lze takto pojmenované následníky v AST snadněji vyhledat. Gramatika jazyka YARA tohoto konceptu využívá v mnoha případech. Na následujícím příkladu vidíme použití pojmenování u definice neterminální symbolu `import`:

```
import: $ => seq(
    "import",
    field("module", $.string_literal),
)
```

Zajímavou částí definic neterminálních symbolů, kde je rovněž pojmenování uzlů využito, jsou výrazy. Kromě využití rekurze pro jejich definici jsou zajímavé také používáním modulu `operators`, který je součástí programového řešení této práce. Modul, na rozdíl od knihovny tree-sitter samotné, umožňuje definovat operátory prostřednictvím vytváření instancí tříd v jazyce JavaScript. Pokud bychom definovali např. operátor pro aritmetické sčítání způsobem standardním pro knihovnu tree-sitter, použili bychom zápis:

```
add: $ => prec.left(130,
    seq(
        field("lop", $._primary_expression),
        "+",
        field("rop", $._primary_expression)
    )
)
```

Tato definice by byla plně funkční a v souladu s dokumentací jazyka YARA. Metoda `prec.left` udává asociativitu operátoru, hodnota `130` udává jeho precedenci a funkce `seq` popisuje sekvenci po sobě jdoucích symbolů, jež může být přerušena bílými znaky. Jako operandy pro sčítání jsou očekávány na obou stranách symboly `_primary_expression`, které by byly pojmenovány jako `lop` a `rop`.

Ačkoliv by pravidlem ve stejném tvaru mohly být definovány i další binární operátory, gramatika by obsahovala velké množství pravidel v tomto tvaru. Udržování těchto pravidel by při změnách a přidávání dalších operátorů bylo náročné.

Z tohoto důvodu byl vytvořen modul `operators`. Nabízí sadu tříd pro vytváření pravidel pro operátory. Díky ní můžeme definice pravidel pro operátory vydělit do tabulky mimo gramatiku a tím ji výrazně zpřehlednit. Definice operátorů pomocí této tabulky je realizována následujícím způsobem:

```
const PRIMARY_EXPRESSION_OP = {
...
DIV: new BinaryOperatorSym("div", "\\\\", "$_primary_expression", new Precedence(140)),
ADD: new BinaryOperatorSym("add", "+", "$_primary_expression", new Precedence(130)),
SUB: new BinaryOperatorSym("sub", "-", "$_primary_expression", new Precedence(130)),
...
}
```

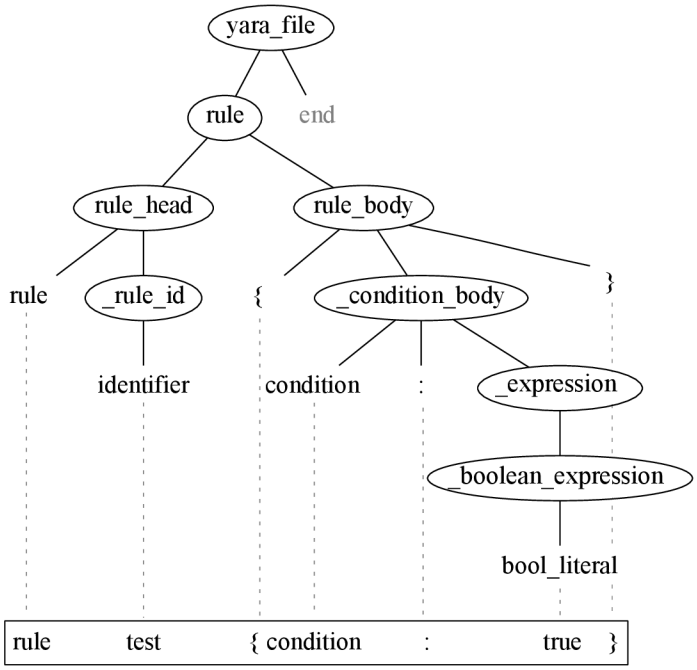
Modul `operators` využívá toho, že návratovými hodnotami pomocných funkcí knihovny tree-sitter jsou pouze objekty. Ty obsahují všechny potřebné informace pro vygenerování tabulky pro LR syntaktický analyzátor. Díky znalosti struktury těchto objektů vytvoří identický objekt, který by vznikl pomocí standardní definice operátoru. Jedná se tedy o rámcové řešení pro definice operátorů, jež uživateli poskytuje abstraktnější pohled na definice operátorů než jejich standardní definice v knihovně tree-sitter. Velkou výhodou oproti nim je udržitelnost a rozšiřitelnost těchto tabulek. Úpravy nebo přidání operátoru znamenají pouze přidání nebo modifikaci řádků v tabulce a nikoliv manipulaci s pravidly gramatiky.

Nástroj tree-sitter-cli kromě generování tabulek pro LR syntaktickou analýzu umožňuje také vizualizovat průběh syntaktické analýzy a výsledný AST. To je velmi užitečné pro ladění gramatiky. Příklad derivačního stromu pro triviální pravidlo v jazyce YARA, který byl vytvořen na základě výsledné implementace gramatiky, je možné vidět na obrázku 4.4.

Výstupní AST knihovna tree-sitter dokáže zobrazit jako řetězec ve formátu S-expression. Pro analýzu stejného řetězce jako je uveden na obrázku vypadá například AST tímto způsobem:

```
(yara_file [0,0]-[4,0]
  (rule [0,0]-[3,1]
    head: (rule_head [0,0]-[0,9]
      id: (identifier [0,5]-[0,9]))
    body: (rule_body [0,10]-[3,1]
      condition: (bool_literal [2,8]-[2,12]))))
```

Ve výpisu je dobře vidět také použití pojmenování uzlů ve výstupním stromě pomocí funkce `field`, kdy například identifikátor pravidla je označen jako `id`. Díky tomu jej bude snazší lokalizovat při zpracování AST. Dále je vidět, že oproti derivačnímu stromu na obrázku 4.4, neobsahuje výstupní AST skryté symboly. V neposlední řadě je u jednotlivých



Obrázek 4.4: Derivační strom vykreslený knihovnou tree-sitter na základě gramatiky pro jazyk YARA – V obdélníku v dolní části obrázku je manuálně doplněn vstupní řetězec.

uzlů uveden také rozsah, který ve zdrojovém kódu reprezentují (hodnoty v hranatých závorkách). Například symbol `bool_literal` reprezentující `true` nebo `false` se nachází na řádku s indexem 2 od bytu s indexem 8 (včetně) až po byte s indexem 12.

Vytvořená gramatika jazyka YARA je kompletní a podporuje také rozšíření jazyka vytvořené firmou Avast. Knihovna tree-sitter je již nativně odolná vůči chybám ve vstupním řetězci a nebylo z tohoto důvodu zapotřebí tuto vlastnost na úrovni syntaktické a lexikální analýzy implementovat.

Z důvodu ještě vyšší odolnosti je gramatika tolerantní vůči neplatným řídícím sekvencím (escape sequences) v řetězcových literálech. Kontrola, zda je řídící sekvence validní, je prováděna až modulem postprocessing. To zabraňuje zahození zbytku řetězce po nalezení první neplatné řídící sekvence. Rozdíl mezi tolerantním přístupem a kontrolou platnosti řídících sekvencí je ilustrován následujícími dvěma výpisy fragmentu AST, který odpovídá řetězcovému literálu `"a\bc\de\f"`:

```
1 value: (string_literal [2,13]-[2,24]
2  str: (string_literal_str [2,14]-[2,23]
3   (string_esc_seq [2,15]-[2,17])
4   (string_esc_seq [2,18]-[2,20])
5   (string_esc_seq [2,21]-[2,23])))
```

```
1 value: (string_literal [2,13]-[2,24]
2  str: (string_literal_str [2,14]-[2,15])
3   (ERROR [2,15]-[2,23]
4    (identifier [2,16]-[2,18])
5    (identifier [2,19]-[2,21])
6    (identifier [2,22]-[2,23])))
```

V levém výpisu vidíme fragment AST pro daný řetězcový literál, který byl vytvořen na základě gramatiky implementované v rámci modulu tree-sitter-yara. V pravém výpisu je možné vidět alternativní řešení, které není tolerantní vůči nevalidním řídícím sekvencím. Vidíme, že ačkoliv ve druhém případě došlo správně k identifikaci chyby, zbytek řetězce byl označen jako chybový uzel `ERROR`. Není tedy již možné jej snadno zkoumat, ačkoliv

v tomto případě obsahuje další nevalidní prvky. Díky aktuálnímu řešení je možné odhalit všechny nevalidní řídící sekvence ve výše uvedeném řetězcovém literálu v rámci modulu postprocessing.

4.3.3 Postprocessing

Tvorba výstupní reprezentace

Jednou z nejdůležitějších částí modulu postprocessing je třída `TSParserAdapter`. Instance této třídy, kterou si uchovává objekt typu `Yaramod`, má za úkol konvertovat AST, jenž je vytvářen modulem tree-sitter-yara, na výstupní reprezentaci (model) zdrojového kódu. Tato konverze je prováděna průchodem tohoto AST pomocí funkcí, které jsou součástí rozhraní knihovny tree-sitter.

Modul postprocessing v rámci své metody `parse` nejprve zažádá tree-sitter-yara o lexikální a syntaktickou analýzu vstupního řetězce pomocí funkce `ts_parse_string`. Následně ve třech průchodech stromem vystaví výstupní reprezentaci.

Během prvního průchodu jsou vyhledány syntaktické chyby identifikované syntaktickým analyzátorem. Tento průchod není příliš náročný, protože knihovna tree-sitter udržuje v rámci uzlů v AST atribut, jenž signalizuje výskyt chyby u daného uzlu či u jeho následníků. Pokud tedy ve zdrojovém kódu není přítomna žádná syntaktická chyba, kterou by tree-sitter-yara odhalil, je průchod ihned ukončen u kořenového uzlu `yara_file`.

Druhý průchod slouží k identifikaci tzv. globálních uzlů v syntaktickém stromě. Jsou jimi jednak komentáře, které se mohou vyskytovat takřka kdekoliv v AST, ale také uzly označující chybějící token. Knihovna tree-sitter totiž v případě, že token zcela chybí, nevytvoří uzel označující syntaktickou chybu, ale doplní do AST listový uzel s příznakem `MISSING`. Během druhého průchodu jsou tedy nalezeny jednak komentáře, které jsou následně uloženy do tabulky komentářů, a také tyto uzly, s nimiž je dále nakládáno jako se syntaktickými chybami.

Během třetího průchodu jsou nejdříve nalezena všechna pravidla a konstrukce `include` či `import`. Ve všech třech případech je nejdříve vytvořen objekt příslušného typu, který je na základě informací dostupných z AST inicializován.

V případě pravidel je inicializace nejkomplexnější, protože pravidlo může obsahovat řetězce, metadata a interní proměnné. Tyto entity jsou dále souhrnně označované jako lokální symboly. Kromě toho v pravidlech mohou být přítomny také výrazy. V principu je s těmito celky však nakládáno podobným způsobem jako s pravidly ve zdrojovém souboru. Nejprve dojde k identifikaci daného elementu v podstromě reprezentující pravidlo, poté je vytvořen objekt příslušného typu (např. `String`) a ten je dále inicializován. Na rozdíl od pravidla, které by bylo uloženo do tabulky udržované objektem typu `YaraFile`, jsou lokální symboly uchovávány v tabulkách, jež si udržují sama pravidla.

Co se týče hexadecimálních řetězců a regulárních výrazů, dochází u nich k průchodu podstromem, který je reprezentuje, za účelem sémantických kontrol (např. v `/a{n,m}/` musí platit $n \leq m$) a dalších syntaktických kontrol (např. platnost řídících sekvencí). Ve výstupní reprezentaci jsou však hodnoty těchto řetězců uchovávány pouze v serializované podobě jako datový typ `std::string`, kvůli času a paměti, jež by byla potřebná pro vytváření komplexních struktur.

V případě těch částí AST, které byly vytvořeny pomocí pravidel gramatiky využívající rekurzi, bylo zapotřebí využít rekurzi i pro jejich zpracování v modulu postprocessing. To je případ výrazů, které mohou tvořit hodnoty interních proměnných a podmínky pravidel. Vzhledem k velkému typu podtříd třídy `Expression`, je pro výběr správné funkce pro zpra-

cování uzlu ve výrazu použita hashovací tabulka (`std::unordered_map`) mapující název uzlu na ukazatel na metodu adaptéru. V ostatních případech je výběr metody realizován pomocí podmínek.

U konstrukce `include` je nejprve ověřena existence vkládaného souboru. Následně se dle cesty k souboru zkontroluje, zda buffer s překladovými jednotkami udržovaný objektem typu `YaraSource` již soubor neobsahuje. Tím se zamezí dvojnásobnému vložení stejného souboru a detekuje se cyklická závislost zdrojových souborů, která by mohla vést k zacyklení programu. Poté dojde k rekurzivnímu volání metody `parse`.

Ačkoliv jsou dva průchody AST relativně drahou operací v celém procesu tvorby výstupní reprezentace, díky prvnímu průchodu se ve druhém již nemusí kontrolovat přítomnost globálních uzlů. To výrazně zpřehledňuje kód, protože ve spoustě případech není zapotřebí provádět iteraci přes potomky uzlu, ale je plně dostačující nalézt konkrétního potomka skrze jeho pojmenování (viz 4.3.2). Nalézt uzel se specifickým označením umožňuje funkce `ts_node_child_by_field_name` z knihovny tree-sitter.

Entity ve zdrojovém kódu si udržují pozici ve formě bytové offsetu ve zdrojovém kódu a také v podobě souřadnic, tvořených indexem řádku a indexem bytu na daném řádku. Pro fungování aktualizace výstupní reprezentace je důležitá pouze první forma. Některé klientské aplikace (např. YLS) ale vyžadují druhou z těchto forem. Udržování obou dvou forem pozic je na první pohled zbytečné a komplikované. Z tohoto důvodu byly původně souřadnice získávány dynamicky z bytového offsetu, sekvenčním průchodem vstupního řetězce. To však způsobovalo značné zpomalení klientských aplikací, a tudíž je nyní implementováno uchovávání pozic v obou uvedených formátech.

Syntaktické a sémantické kontroly

Přestože je většina syntaktických kontrol zodpovědností modulu tree-sitter-yara, v některých případech jsou delegovány na modul postprocessing. Těmito případy jsou kontrola platnosti řídících sekvencí v řetězcích, kontrola přítomnosti non-ASCII znaků v regulárních výrazech a kontrola kolize identifikátorů s klíčovými slovy.

Sémantické kontroly jsou prováděny v rámci vytváření výstupní reprezentace objektem třídy `TSParserAdapter` a třídami sémantického rozhraní knihovny. Probíhají během vytváření výstupní reprezentace, protože oddělení kontrol by vyžadovalo, uskutečnit další průchod výstupní reprezentací nebo AST. Sémantické kontroly byly implementovány pro odhalování následujících artefaktů ve zdrojovém kódu:

- **Regulární výrazy** – Platnost rozsahu znaků (např. `/[a-b]/`), platnost intervalu vyjadřujícího opakování znaku (pořadí hraničních hodnot a jejich hodnota)
- **Hexadecimální řetězce** – Platnost intervalu u tzv. jumps (viz 2.1.3)
- **Modifikátory řetězců** – Duplicitní modifikátory, platnost argumentů modifikátorů
- **Přetečení** – Identifikace příliš vysokých číselných hodnot
- **Vkládání zdrojových souborů** – Existence vloženého souboru, identifikace dvojnásobného vložení jednoho zdrojového souboru
- **Identifikátory** – Kolize identifikátorů řetězců, interních proměnných a pravidel
- **Modifikátory pravidla** – Duplicitní modifikátory pravidla

- **Výrazy** – Platnost referencí ve výrazech, datové typy, platnost rozsahů...

Ačkoliv nástroj YARA označuje za chybu také řetězec, který byl definován, ale není používán, knihovna tento artefakt nekontroluje. Jelikož se předpokládá, že vstupní zdrojový kód bude často nedokončený, sémantická kontrola tohoto typu by pro uživatele představovala zbytečný šum. Jedná se však o kontrolu, kterou je díky sémantickému rozhraní možné implementovat v rámci klientské aplikace (viz kapitola 4.6).

Pro provádění některých z výše uvedených kontrol je nezbytné udržování tabulek symbolů (viz další kapitola) a také přístup ke specifikacím modulů. Moduly jsou načteny během instanciace objektu typu `Yaramod`, a jsou deserializovány knihovnou JSON for Modern C++. Díky tomu sice trvá vytvoření tohoto objektu déle, ale v rámci samotné statické analýzy již deserializace nepředstavuje žádnou dodatečnou režii. Zároveň se v rámci každého modulu vytvoří cache mapující všechny obsažené symboly na jejich kontext. To opět zrychluje analýzu, neboť není nutné rekurzivně prohledávat jmenný prostor modulu. Stačí pouze provést vyhledávání v této cache.

Typové kontroly jsou prováděny na základě kontextu získaného z těchto modulů a na základě popisu jednotlivých operátorů v jazyce YARA. Většina operátorů má pevně stanovený datový typ návratové hodnoty a není tedy zpravidla nutné používat typové odvozování ani syntézu. Výjimkou jsou aritmetické výrazy, kde se využívá právě typové syntézy. Datový typ aritmetického výrazu je určen na základě operandů daného operátoru.

Tabulky entit a symbolů

Nezbytnou součástí sémantického rozhraní jsou také tabulky, které umožňují uživateli přístup k entitám ve zdrojovém souboru. Entitami jsou zde myšleny sémantické celky, např. pravidla, importy modulů apod. Modul postprocessing využívá tři typy tabulek. Společným rysem všech tří typů je indexování dle bytového offsetu, na němž se entita ve zdrojovém kódu nachází. Toto indexování je potřeba pro aktualizaci výstupní reprezentace a také umožňuje uživateli přístup k entitám v pořadí, v jakém se ve zdrojovém kódu objevují. To může být důležité např. pro provádění statické analýzy klientskými aplikacemi.

První typ tabulky indexuje prvky pouze pomocí bytového offsetu. Tabulka je realizována skrze datovou strukturu `std::multimap`. V této tabulce jsou uchovávány objekty reprezentující chyby (podtřídy třídy `Error`), komentáře a neplatné entity (např. pravidlo s duplicitním identifikátorem).

Druhý typ tabulky kromě indexování dle bytového offsetu indexuje položky také pomocí identifikátoru, který je v rámci tabulky jednoznačný. Tento typ tabulky je implementován pomocí datových struktur `std::multimap` a `std::unordered_map`. Slouží k uchovávání všech entit s jednoznačným identifikátorem, kterými jsou pravidla, řetězce a interní proměnné.

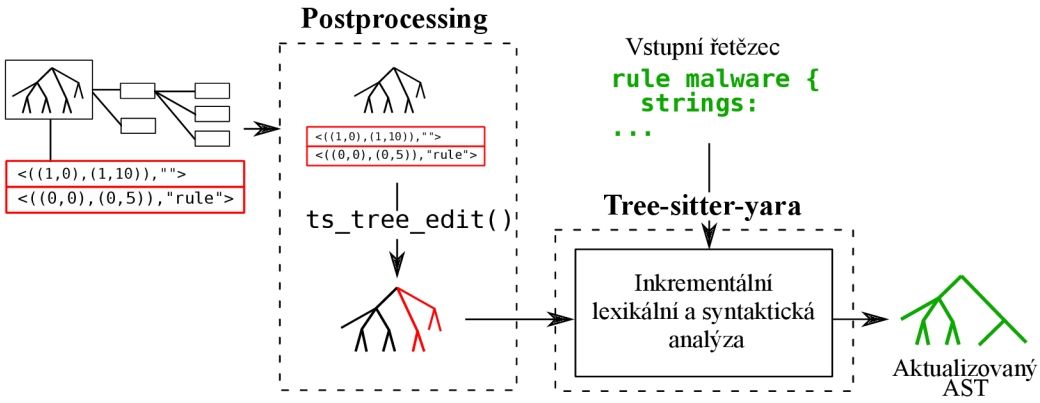
Třetí typ tabulky dovoluje duplikaci identifikátoru. To je využíváno pro metadata pravidel a objekty typu `Import`[5]. Realizovány jsou podobně jako předchozí typ tabulek s tím rozdílem, že v `std::unordered_map` je uložen celý vektor ukazatelů namísto ukazatelů samotných.

Tabulky posledních dvou typů zároveň slouží jako tabulky symbolů pro sémantické kontroly. Indexace dle identifikátorů umožňuje entity rychle vyhledávat, což se využívá pro detekci duplicitních identifikátorů a kontroly, zda je symbol v době použití definován.

[5] Jazyk YARA dovoluje duplicitní identifikátory metadat a vícenásobný import jednoho modulu.

Aktualizace výstupní reprezentace

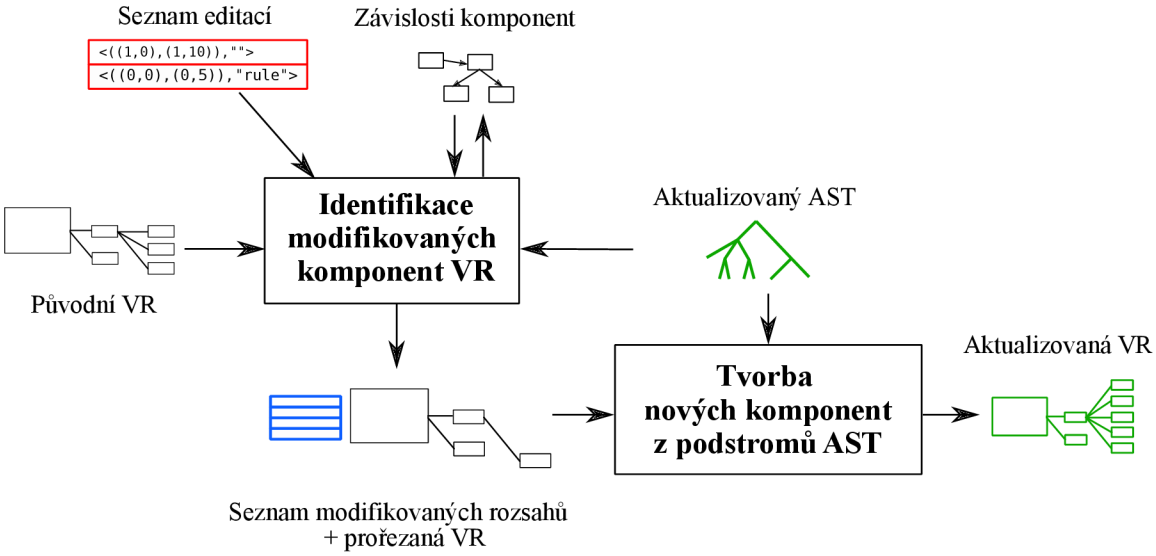
Před aktualizací výstupní reprezentace jsou uživatelem nahlášeny změny ve zdrojovém kódu metodou `edit`. Tyto změny jsou nahlášeny ve formátu `<range, text>`, kde `range` jsou souřadnice začátku a konce modifikovaného rozsahu a `text` je nový obsah této části. Pokud je například vložen řetězec `start` na úplný začátek dokumentu, jsou tyto údaje `<((0,0),(0,0)), "start">` . Pokud dojde ke smazání tohoto řetězce, jsou modifikace nahlášeny jako `<((0,0),(0,5)), "">`. Tento formát odpovídá informacím, které poskytuje LSP v rámci zprávy o změně dokumentu [26]. Jak už bylo uvedeno výše, modifikace jsou uchovány v bufferu a jsou aplikovány až ve chvíli, kdy si uživatel vyžádá samotnou aktualizaci výstupní reprezentace spuštěním inkrementální analýzy.



Obrázek 4.5: Proces aktualizace AST během inkrementální statické analýzy – Z výstupní reprezentace je nejprve extrahován původní AST a seznam editací provedených uživatelem. Poté dojde k označení změn v AST a ten je předán modulu tree-sitter-yara, který ho aktualizuje skrze analyzátory z knihovny tree-sitter. Tomuto modulu je zároveň dodán také aktualizovaný vstupní řetězec.

Během samotné aktualizace jsou nejprve modifikace předány modulu tree-sitter-yara pomocí funkce `ts_tree_edit`, poskytované knihovnou tree-sitter. Následně je provedena inkrementální lexikální a syntaktická analýza zajištěná opět touto knihovnou (viz obrázek 4.5). Poté je zapotřebí aktualizovat také výstupní model zdrojového kódu. V principu se jedná o smazání modifikací dotčeného pravidla, konstrukce `include` nebo `import` a vytvoření aktuálního objektu. Bylo by pochopitelně možné provádět aktualizace s ještě jemnější granularitou. To by však zároveň vedlo ke zvýšení režie potřebné pro aktualizaci. Proto v současné době knihovna Yaramod-v4 provádí aktualizaci pouze na úrovni výše uvedených entit, tedy na úrovni komponent objektu třídy `YaraFile`.

Aktualizace je ve zjednodušené podobě popsána na obrázku 4.6. Základní strukturou pro její provedení je seznam rozsahů bytových offsetů, v nichž mají být komponenty výstupní reprezentace aktualizovány. Položky do tohoto seznamu jsou získávány několika způsoby. V cyklu jsou do tohoto seznamu přidávány explicitně editované oblasti zdrojového kódu uživatelem. V této oblasti jsou všechny entity smazány a ty, které se nacházejí za modifikovanou oblastí jsou posunuty o počet znaků, které byly vloženy, resp. smazány. Průběžně je tento seznam doplňován o pozice závislých entit na právě smazaných entitách. To je možné díky objektu typu `YaraSource`, který si udržuje záznamy o vzájemných závislostech těchto entit.



Obrázek 4.6: Proces aktualizace výstupní reprezentace kódu (VR) během inkrementální statické analýzy v knihovně Yaramod-v4 – Nejprve jsou prostřednictvím seznamu editací, závislostí komponent a aktualizovaného AST nalezeny modifikované komponenty. Ty jsou z VR odstraněny a je vytvořen seznam modifikovaných rozsahů v dokumentu. Poté jsou nalezeny podstromy v aktualizovaném AST, jež rozsahům odpovídají, a na základě nich jsou vytvářeny nové, aktuální komponenty pro VR.

Pokud dojde např. ke smazání pravidla, toto pravidlo v rámci svého destruktoru notifikuje rodičovský objekt třídy `YaraSource`. Ten pomocí záznamů zjistí, zda některé z jiných pravidel používalo v rámci svojí definice právě smazané pravidlo, tzn. je na něm závislé. Pokud ano, přidá oblast, kde se toto závislé pravidlo nachází, do speciálního bufferu. Tyto oblasti jsou opět uchovávány ve formě rozsahů bytových offsetů.

Z tohoto bufferu jsou během provádění aktualizace přesunuty do seznamu rozsahů, u nichž má dojít k aktualizaci. S každým dalším rozsahem, který byl explicitně modifikován uživatelem jsou rovněž posunuty o příslušný počet znaků také položky v tomto seznamu, aby odpovídaly vždy aktuálním pozicím.

Po vyčtení všech explicitně modifikovaných rozsahů jsou do seznamu přidány také pozice nových uzlů v AST navráceného modulem tree-sitter-yara. To je možné díky další funkci z knihovny tree-sitter, a sice `ts_tree_get_changed_ranges`.

Po získání všech rozsahů určených k aktualizaci jsou seřazeny tak, aby jejich pořadí odpovídalo pořadí, v jakém se nachází ve zdrojovém dokumentu. Následně jsou spojeny rozsahy, které se překrývají, aby nedošlo k vícenásobné aktualizaci. Takto připravený seznam je poté v cyklu procházen a pro každý rozsah, který se v něm nachází, je nalezena sekvence uzlů v AST, jež svojí pozicí ve zdrojovém kódu odpovídá daném rozsahu. Díky zvolené granularitě aktualizací mohou být v této sekvenci pouze uzly reprezentující pravidlo, konstrukci `include` nebo konstrukci `import`.

Pro každý uzel v této sekvenci jsou nejdříve vymazány zbylé entity, které se nachází na pozici tohoto uzlu. Následně je způsobem, jenž je popsán v kapitole 4.3.3, konvertován podstrom AST, pro nějž je tento uzel kořenem, do podoby výstupní reprezentace.

Proces aktualizace výstupní reprezentace je prováděn pro každý objekt typu `YaraFile` uložený v bufferu daného objektu třídy `YaraSource`. Aktualizace je však prováděna pouze

tehdy, pokud je to zapotřebí, tzn. došlo u daného objektu typu `YaraFile` k explicitní modifikaci či byla smazána některá z entit, na níž byla komponenta tohoto objektu závislá.

Alternativa k výše uvedenému přístupu by byla provádět inkrementálním způsobem pouze lexikální a syntaktickou analýzu skrze knihovnu tree-sitter. Výstupní reprezentace by v tomto případě byla pokaždé kompletně smazána a znovu vystavěna. Docházelo by tak ve velké míře k dealokaci a alokaci objektů, jež reprezentují pravidla, importy modulů a vkládání souborů, u nichž není aktualizace zapotřebí. To by však také znamenalo podstatně jednodušší implementaci. Na to, zda by byl tento přístup výhodný, či nikoliv, odpovídá tabulka 4.1. Z tabulky vidíme, že aktuální řešení i přes svoji relativní složitost je z hlediska času výrazně efektivnější než popsaná alternativa, jak pro malou sadu pravidel, tak pro rozsáhlou.

Sledování závislostí mezi entitami však může způsobit, že i při dosavadním řešení mohou být aktualizovány všechny entity ve výstupní reprezentaci. Tato situace může nastat například v případě, že všechna pravidla ve zdrojovém kódu budou používat jedno jiné pravidlo, u kterého dojde k modifikaci. Ačkoliv tato situace běžně nenastává, může k ní dojít a výsledky by v tomto případě odpovídaly alternativnímu řešení (v tabulce jako Full update).

Sada pravidel	Partial update (s)		Full update (s)	
	Analýza	Reanalýza	Analýza	Reanalýza
`small.yar` (11,2 kB)	0,018	0,0003	0,013	0,003
`big.yar` (14,4 MB)	4,364	0,037	4,050	1,004

Tabulka 4.1: Porovnání efektivity aktuálního řešení a alternativního řešení z hlediska potřebného času – Sloupec *Partial update* označuje současné řešení. Sloupec *Full update* označuje alternativní řešení, kdy je celá výstupní reprezentace kompletně smazána a poté znovu vytvořena. Použité testovací sady pravidel jsou blíže popsány v kapitole 5.

4.4 Integrace s projektem Yara Language Server

V rámci projektu YLS byla provedena integrace Yaramod-v4 ve třech jeho komponentách, z nichž každá doposud využívala dosavadní řešení. Každá z těchto částí YLS poskytuje specifickou funkcionalitu, pro jejíž implementaci je zapotřebí využívat sémantické rozhraní knihovny. V následujícím seznamu jsou uvedeny všechny tyto části projektu YLS spolu s funkcionalitou, kterou poskytují, a stručným popisem úlohy nového řešení v dané části:

1. **Linting** – Poskytuje uživatelům syntaktické a sémantické kontroly zdrojového kódu. Je implementován pomocí čtení tabulky s chybami, která je součástí rozhraní knihovny Yaramod-v4. Pro větší uživatelskou přívětivost poskytuje také stručný popis chyby a její pozici v dokumentu, díky které je možné ji snadno vyhledat.

2. **Document Symbol** – Vytváří strukturovaný pohled na editovaný zdrojový kód. To zahrnuje jednak vyvářeení osnovy dokumentu (outline) a také navigaci na symboly uvedené v osnově. Úkolem knihovny Yaramod-v4 je zde poskytovat tabulky s těmito symboly a jejich pozice ve zdrojovém kódu. Oproti dosavadnímu řešení, jsou v osnově uvedeny také konstrukce `import` a `include`, díky kterým může uživatel snadno vidět, které moduly importoval a které zdrojové soubory do aktuálního dokumentu vložil.

3. **Completion** – Umožňuje uživateli doplnit chybějící znaky rozepsaného symbolu. To je realizováno pomocí seznamu všech symbolů, které mají prefix shodný s rozepsaným řetězcem[6]. Yaramod-v4 (stejně jako dosavadní řešení) se na této funkcionalitě podílí pouze částečně. Nejdříve je pomocí výsledků analýzy prováděné touto knihovnou určeno pravidlo, v němž se nachází kurzor uživatele. Poté je seznam symbolů obohacen o položky z tabulek sémantického rozhraní, která mohou představovat rozepsaný symbol (dříve definovaná pravidla, řetězce v aktuálním pravidle apod.).

Zcela novým konceptem v YLS je průběžné nahlašování změn v dokumentu, které je potřebné zajistit pro podporu inkrementálního přístupu k analýze zdrojového kódu. Nahlašování změn je realizováno pomocí události typu `TEXT_DOCUMENT_DID_CHANGE`, která je součástí implementace LSP knihovnou pygls [36]. Parametry této události jsou editovaný interval v dokumentu a nový text, který se v tomto intervalu po editaci vyskytuje, což přesně odpovídá argumentům jedné z implementací metody `edit` v Yaramod-v4. Díky tomu je možné v modelu programu změnu snadno uchovat a při další statické analýze z něj vycházet. Výše uvedená funkcionalita YLS, kterou Yaramod-v4 pomáhá zajišťovat, je demonstrována skrze snímky obrazovky z editoru VSCode, jež jsou dostupné v příloze A.

YLS po integraci s knihovnou Yaramod-v4 má také dvě známá omezení. Prvním z omezení je nekompatibilita těchto dvou projektů při používání non-ASCII znaků ve zdrojovém kódu, respektive znaků, jež jsou kódovány pomocí více než jednoho bytu. Yaramod-v4 nahlíží na vstupní řetězec jako na sekvenci bytů, ale YLS jej považuje za sekvenci znaků v příslušném kódování. Při smazání znaku, který je kódován pomocí více bytů, YLS nahlásí editaci, jenž odpovídá smazání pouze jednoho znaku, což Yaramod-v4 interpretuje jako jeden byte a smaže tedy pouze jeden byte daného znaku. To vede na falešné ohlášení syntaktické chyby. Zpětně se tato nekompatibilita projevuje při poskytování pozicí v dokumentu (např. při navigaci na definici symbolu pomocí funkcionality Document Symbol) a to i u dosavadního řešení. Používání non-ASCII znaků v jazyce YARA však není běžné[7].

Druhým omezením je více rozpracovaných souborů, které jsou vzájemně provázány pomocí `include`. Analýza vložených zdrojových souborů probíhá na základě uložených souborů na disku, a tudíž neuložené změny nejsou reflektovány. Z tohoto důvodu je jako vstupní soubor sady pravidel (`entry_file`) vždy považován ten, který byl otevřen jako první či je právě editovaný.

4.5 Testování

Testování knihovny Yaramod-v4 probíhalo v několika fázích. První fází bylo průběžné automatizované testování modulů tree-sitter-yara a postprocessing. Cílem této fáze testování, která probíhala souběžně s vývojem, bylo odzkoušet nově přidanou funkcionalitu těchto modulů a zároveň se přesvědčit, že její přidání nezpůsobilo chyby v jejich již implementovaných částech.

Testování modulu tree-sitter-yara bylo prováděno automatizovaně pomocí nástroje tree-sitter-cli. Tento nástroj umožňuje vytvářet testovací případy skrze specifikaci vstupního řetězce a očekávaného výstupního AST ve formátu S-expression. Tímto způsobem bylo vytvořeno 173 testovacích případů.

[6] Výjimkou jsou řetězce v pravidlech, kde je ignorován počáteční znak `$`.

[7] Existuje více možností, jak tyto znaky popsat bez jejich explicitního použití. Od verze jazyka YARA 4.1 jsou při explicitním použití navíc označovány za chybu samotným nástrojem YARA (viz https://github.com/VirusTotal/yara/wiki/Unicode-characters-in-YARA).

Kromě tohoto řešení byl také jako součást této práce vytvořen testovací skript s názvem `depth_test.py` pro hloubkové testování gramatiky. Tento skript byl implementován pomocí rozhraní knihovny tree-sitter pro jazyk Python. Jeho výhodou je, že kontroluje také přítomnost nepojmenovaných tokenů (závorek, klíčových slov atd.), které jsou ve výstupním AST skryty. Díky tomuto skriptu je možné gramatiku ještě důkladněji otestovat. Na druhou stranu je tvorba testovacích případů pro něj o něco náročnější, neboť v očekávaném výsledku, jenž je opět specifikován pomocí S-expression, musí být přítomny všechny tokeny. Z toho důvodu bylo pro tento skript vytvořeno pouze 10 testovacích případů.

Modul postprocessing byl testován jednotkovými testy, jež byly vytvořeny prostřednictvím frameworku GoogleTest. Testy jsou rozděleny do čtyř testovacích sad, aby bylo možné se při testování zaměřit vždy na určitou část tohoto modulu. Jednotkové testy byly také spouštěny pod nástrojem Valgrind za účelem eliminace úniků paměti. Počet všech jednotkových testů modulu postprocessing je 114.

Po dokončení programového řešení byly manuálně provedeny integrační testy. V rámci těchto testů byla prováděna analýza rozsáhlé sady pravidel (sada `big.yar` z kapitoly 5) v jazyce YARA používané firmou Avast, čímž došlo k ověření, zda jsou opravdu všechny elementy tohoto jazyka knihovnou podporovány. Současně byly prováděny také testy na menších zdrojových souborech opět pod nástrojem Valgrind.

Zároveň v této fázi vývoje probíhalo také profilování. To bylo realizováno skrze nástroj *gprof* a byla tímto nástrojem hledána úzká místa ve zdrojovém kódu. Profilování bylo prováděno nad sadou `big.yar`, jež je blíže popsána v 5. Díky profilování bylo zavedeno několik vylepšení jako např. cache symbolů v modulech nebo uchovávání souřadnic symbolů v dokumentu.

Po vytvoření rozhraní pro jazyk Python bylo použito rámcové řešení pytest pro otestování i této části knihovny. Pro otestování integrace s projektem YLS byly využity existující integrační testy tohoto projektu, které byly doplněny o několik nových testů. Jejich cílem bylo ověřit fungování poskytování zpětné vazby uživateli během editace zdrojového kódu. Pro jejich snazší parametrizaci byl použit modul *difflib* pro jazyk Python. Testovací případ se díky tomuto modulu sestává pouze z originální podoby zdrojového kódu, jeho upravené verze a očekávaných diagnostických zpráv. Funkce `ndiff` z tohoto modulu identifikuje rozdíly mezi dvěma verzemi zdrojového kódu a ty tak mohou být znak po znaku nahlašovány YLS jako editace zdrojového kódu.

Tímto mechanismem je simulováno psaní uživatele na klávesnici. Po nahlášení všech změn jsou zkontrolovány poslední navrácené diagnostické zprávy a ty jsou porovnány s očekávanými výsledky. První sada testů se zaměřuje na modifikaci pouze jednoho souboru. Druhá již kontroluje výsledky při modifikaci vícero zdrojových souborů, které jsou propojeny pomocí konstrukce `include`.

4.6 Příklady použití

Součástí programového řešení bakalářské práce jsou také tyto jednoduché programy, které demonstrují použití knihovny Yaramod-v4:

- `reparsing.cpp` – Demonstrace základního použití knihovny Yaramod-v4 pro statickou analýzu pravidel v jazyce YARA a pro jejich opětovnou inkrementální analýzu. Během obou běhů analýzy jsou měřeny spotřebované zdroje pomocí funkce `getrusage`. Tento program byl použit také pro porovnání knihovny Yaramod-v4 s dosavadním řešením (viz kapitola 5).

- `observing_visitor.cpp` – Ukázka využití návrhového vzoru visitor, jenž je součástí sémantického rozhraní, v C++. Tento program umožňuje vizualizovat pravdivostní výrazy pomocí stromů, které vypisuje na terminál nebo na příkazový řádek.
- `calculator.py` – Skript pro jazyk Python využívající sémantické rozhraní pro přístup k výrazům (návrhový vzor visitor). Provede vyhodnocení aritmetických výrazů v podmínkách pravidel v zadaném zdrojovém souboru.
- `unused_strings.py` – Ukázka použití knihovny Yaramod-v4 pro identifikaci definovaných, ale nepoužitých řetězců. Jedná se opět o skript v jazyce Python, jenž využívá sémantické rozhraní knihovny.

4.7 Metriky kódu

Programové řešení se skládá z vícero částí, které jsou různě rozsáhlé a využívají různé jazyky. Pro lepší přehled jsou proto v tabulkách 4.2 a 4.3 uvedeny počty řádků ve zdrojových kódech jednotlivých částí programového řešení. Uvedené počty byly určeny pomocí nástrojů *Tokei*[8] a *wc*. U projektu YLS byly modifikované řádky vypočteny pomocí příkazu `git diff`.

Část programového řešení	**Jazyk**	**Kód**	**Komentáře**
Postprocessing	C/C++	12795	2835
Tree-sitter-yara	JavaScript	1082	113
YLS (pouze upravená část)	Python	509	42
Příklady použití	C/C++	320	115
	Python	134	75

Tabulka 4.2: Počty řádků ve zdrojových kódech funkčních částí programového řešení

Testovaná část	**Jazyk**	**Kód**	**Komentáře**
Tree-sitter-yara	Plain Text	8439	–
	Python	232	20
Postprocessing	C/C++	2528	36
	Python	778	19
YLS (pouze nové testy)	Python	310	17

Tabulka 4.3: Počty řádků ve zdrojových kódech, které tvoří testy a testovací případy pro funkční části

[8] Oficiální repozitář projektu Tokei: https://github.com/XAMPPRocky/tokei

Kapitola 5

Porovnání dosažených výsledků s dosavadním řešením

Porovnání programového řešení této bakalářské práce s knihovnou Yaramod-v3 bylo prováděno na stroji s architekturou x86_64, operačním systémem Ubuntu 22.04.1, procesorem Intel Core i7-9850H 2,6 Ghz a s 32 GB RAM. Mezi jednotlivými experimenty byla promazávána cache pomocí `sync; echo 3 > /proc/sys/vm/drop_caches`, aby se tak minimalizoval její vliv. Nejdříve byly porovnány knihovny jako takové z hlediska nároků na strojový čas. Při tomto porovnání byl měřen čas analýzy samotné pomocí funkce `getrusage` z hlavičkového souboru `sys/resource.h`. Poté bylo provedeno porovnání odezvy dosavadní implementace projektu YLS a YLS po integraci s knihovnou Yaramod-v4, kde byla součástí porovnání jednak samotná analýza, ale také přístup k jejím výsledkům a kvalita integrace.

5.1 Yaramod-v4

Pro porovnání výkonu obou knihoven byl pro každou z nich vytvořen triviální program, jenž provádí analýzu zdrojového kódu a následnou reanalýzu jeho modifikované verze. U programu pro knihovnu Yaramod-v4 byl využit inkrementální přístup, přičemž pozice změn v souboru byla ohlášena pomocí argumentů programu. Dosavadní řešení, knihovna Yaramod-v3, takovou možnost nenabízí, a tudíž byl zdrojový kód a jeho modifikovaná verze analyzovány konvenčním způsobem. Pro provádění experimentů byly vybrány čtyři reálné sady pravidel, jež používá pro svoje účely firma Avast. Každá z těchto sad měla jiné charakteristiky, které jsou popsány v tabulce 5.1.

První tři sady jsou charakterem velmi podobné. Liší se hlavně svým rozsahem a počtem používaných modulů. V praxi bývají rozsáhlé sady pravidel realizovány pomocí propojo-

Sada pravidel	Celková velikost	Celkový počet pravidel	Počet souborů
`small.yar`	11,2 kB	12	1
`medium.yar`	795 kB	1818	1
`big.yar`	14,4 MB	16260	1
`includes.yar`	1,8 MB	2810	365

Tabulka 5.1: Popis sad pravidel použitých pro porovnání knihovny Yaramod-v3 a programového řešení této bakalářské práce, knihovny Yaramod-v4

vání zdrojových souborů konstrukcí `include`. Jednotlivé soubory pak zpravidla obsahují pouze několik pravidel, která spolu souvisí. Tento případ je zde zastoupen čtvrtou sadou `includes.yar`. Modifikací bylo v jejím případě přidání jednoho pravidla do náhodného zdrojového souboru v testovací sadě.

Modifikací zdrojového kódu bylo buďto přidání celého nového pravidla, nebo jeho odebrání. Zdrojové kódy byly před i po modifikacích syntakticky a sémanticky správné, neboť kvůli rozdílným chováním obou řešení při nálezu chyby by je jinak nebylo možné objektivně porovnat. Obě knihovny byly přeloženy v konfiguraci `Release`, aby byly aktivovány případné optimalizace překladačem.

Výsledky porovnání můžeme vidět v tabulce 5.2. Vidíme, že za cenu malého zpomalení při prvotní analýze (dosavadní řešení je 1,22 – 1,75× rychlejší), dokáže nové řešení reanalyzovat modifikovaný zdrojový kód výrazně rychleji. S rostoucí velikostí sady pravidel se zrychlení oproti dosavadnímu řešení ještě více zvyšuje. U velmi rozsáhlé sady je knihovna Yaramod-v4 rychlejší více než 177×. V případě sady `includes.yar` je nové řešení během reanalýzy rychlejší dokonce více jak 2000×. Tohoto zajímavého výsledku je dosaženo pomocí cache zdrojových souborů, kterou uchovává objekt třídy `YaraSource`. Díky této cache se reanalyzuje pouze modifikovaný soubor a ostatní jsou ignorovány, neboť v nich nedošlo k žádným změnám. To je také patrné z toho, že čas opětovné analýzy v tomto případě přibližně odpovídá času, jenž je potřebný pro opětovnou analýzu malého zdrojového souboru.

Dle uvedených výsledků lze usoudit, že vytvořená knihovna je skutečně vhodná pro použití právě ve spojení s textovými editory a IDE. Za cenu mírně pomalejší prvotní analýzy, která může být prováděna např. při otevření zdrojového souboru, získáváme výrazněji rychlejší reanalýzu modifikované sady pravidel. Ta je dostatečně rychlá natolik, aby mohla být prováděna po přidávání jednotlivých znaků i v případě velmi rozsáhlých sad pravidel v jazyce YARA.

Sada pravidel	Yaramod-v3 (s)		Yaramod-v4 (s)		Zrychlení	
	Analýza	Reanalýza	Analýza	Reanalýza	Analýza	Reanalýza
`small.yar`	0,008	0,006	0,014	0,0003	0,57	20
`medium.yar`	0,336	0,204	0,385	0,002	0,87	102
`big.yar`	3,449	4,083	4,224	0,023	0,82	177,52
`includes.yar`	1,075	0,973	1,152	0,0004	0,93	2432,5

Tabulka 5.2: Porovnání časů potřebných k analýze jednotlivých testovacích sad knihovnou Yaramod-v3 a novým řešením, knihovnou Yaramod-v4 – Jedná se o součty systémových časů a uživatelských časů. Zrychlení bylo vypočítáno jako podíl času potřebného pro (re)analýzu knihovnou Yaramod-v3 a knihovnou Yaramod-v4.

5.2 Yara Language Server

Oproti dosavadní implementaci YLS, využívající Yaramod-v3, došlo po integraci knihovny Yaramod-v4 ke zlepšení zejména ve třech aspektech, co se funkcionality týče. Tím prvním je odolnost vůči chybám ve zdrojovém kódu při provádění statické analýzy. Dosavadní řešení ukončilo analýzu kódu vyhozením výjimky při nalezení první chyby. Pozice chyby a její popis byly extrahovány z řetězce výjimky pomocí regulárních výrazů.

Nové řešení umožňuje identifikovat více než jednu chybu v dokumentu (ať už se jedná o syntaktické nebo sémantické chyby) a chyby jsou reprezentovány jako objekty, které disponují rozhraním pro získání jejich popisu a pozice v dokumentu. Není tedy nutné používat regulární výrazy pro určení těchto údajů.

Dále je díky použití Yaramod-v4 možné analyzovat zdrojový kód přímo z řetězce. Díky tomu lze provádět analýzu rozpracovaných dokumentů, které dosud nebyly uloženy. To souvisí také s třetím aspektem, v němž se projekt YLS díky Yaramod-v4 zlepšil, a sice s prováděním statické analýzy během editace zdrojového kódu.

Analýza je prováděna po jednotlivých editacích, čímž je uživateli poskytnuta okamžitá zpětná vazba. To je kromě možnosti provádět analýzu řetězce umožněno také nízkou latencí analýzy, za kterou vděčíme inkrementálnímu přístupu.

V tabulce 5.3 vidíme porovnání implementace YLS s využitím Yaramod-v3 (dosavadní řešení) a nové implementace, která využívá knihovnu Yaramod-v4 z hlediska času potřebného pro poskytnutí dané funkcionality.

Funkcionalita YLS		Yaramod-v3 (s)	Yaramod-v4 (s)	Zrychlení
Linting	Počáteční	7,750	4,784	1,62
	Po úpravě	8,365	0,056	149,38
DocumentSymbol	Počáteční	6,514	3,673	1,77
	Po úpravě	6,303	3,723	1,69
Completion		1,147	1,177	0,97

Tabulka 5.3: Porovnání odezvy YLS při poskytování funkcionality – V prvním sloupci jsou odezvy implementace, která využívá Yaramod-v3, a ve druhém jsou odezvy implementace YLS, jež využívá knihovnu Yaramod-v4. Třetí sloupec udává zrychlení nové implementace vůči dosavadnímu řešení (vypočítáno jako podíl odezvy dosavadního řešení a nového řešení).

Pro měření časů uvedených v tabulce byla použita sada pravidel `big.yar` z předchozí kapitoly. Úpravou bylo vložení nového pravidla. V obou případech bylo vkládáno na stejné místo v dokumentu. Čas byl měřen od příchodu notifikace na server až po odeslání odpovědi zpět. U lintingu se v případě Yaramod-v3 jednalo o notifikace typu `didOpen` a `didSave`, které právě tuto funkcionalitu aktivují. U nové verze knihovny se sledovala odpověď na notifikace `didOpen` a `didChange`, neboť linting je prováděn po každé úpravě dokumentu. Díky tomu je také v tomto případě do času potřebného k opětovné inkrementální analýze započítán i čas potřebný pro ohlášení změn v dokumentu.

Z tabulky vidíme, že nové řešení exceluje zejména v provádění lintingu zdrojového kódu po jeho úpravách, kdy dokázalo obsloužit danou událost více jak 149× rychleji než dosavadní. Dále vidíme, že ačkoliv z předchozího porovnání vycházela při počáteční analýze o něco lépe knihovna Yaramod-v3, zde takřka ve všech případech vítězí YLS s využitím knihovny Yaramod-v4. Tento výsledek je způsoben tím, že během obsluhy událostí v dosavadním řešení provádí knihovna Yaramod-v3 více běhů statické analýzy. To alespoň v rámci nové implementace není pro zajištění požadované funkcionality nutné. Počet běhů analýzy byl proto redukován na minimum, což ušetřilo čas v případech, v nichž by jinak byla knihovna Yaramod-v4 pomalejší. U funkcionality Completion jsou výsledky obou implementací podobné, protože ani v jednom případě není analýza prováděna a přistupuje se pouze k výsledkům analýzy získaným během jejího posledního běhu.

Z pohledu uživatele jsou skutečné časy odezvy obou implementací o něco vyšší než ty uvedené v tabulce, protože čas potřebný pro výměnu dat těchto objemů mezi klientem YLS a samotným YLS je řádově v desetinách vteřin.

Závěr

V rámci této práce byl nejprve nastudován účel, obecný princip fungování statické analýzy kódu a její úloha v textových editorech a integrovaných vývojových prostředích. Následně bylo popsáno také fungování inkrementálních analyzátorů. Poté byl prostudován jazyk YARA a dosavadní řešení pro jeho statickou analýzu.

Získané teoretické znalosti byly využity pro tvorbu programového řešení této bakalářské práce, knihovny Yaramod-v4. Ta umožňuje provádět inkrementální statickou analýzu kódu v jazyce YARA. Knihovna je plně funkční a otestovaná jak manuálně, tak pomocí automatizovaných testů, které byly rovněž vytvořeny při tvorbě praktické části. Kromě možnosti inkrementální analýzy knihovna disponuje také dalšími výhodami oproti dosavadním řešením. Jedná se především o odolnost vůči chybám ve vstupním zdrojovém kódu. Díky ní a díky inkrementálnímu přístupu je knihovna vhodná pro použití v textových editorech a integrovaných vývojových prostředích.

To ukázalo také porovnání Yaramod-v4 s dosavadním řešením, knihovnou Yaramod-v3. Porovnání bylo prováděno na různě velkých sadách pravidel používaných v praxi. Dosavadní řešení je sice při prvotní analýze přibližně 1,22× až 1,5× rychlejší, nicméně při analýze modifikovaného vstupního řetězce je nové řešení rychlejší 20× až 177× (v závislosti na konkrétní sadě). Ještě lepších výsledků bylo dosaženo v případě sady pravidel, která se skládala z mnoha zdrojových souborů. V tomto případě Yaramod-v4 provedl reanalýzu sady pravidel po modifikacích více jak 2000× rychleji než dosavadní řešení. Tohoto výsledku bylo dosaženo také díky uchovávání výstupní reprezentace nemodifikovaných souborů v programové cache.

Dále byla součástí práce také integrace knihovny Yaramod-v4 s projektem YLS, kde došlo k nahrazení dosavadního řešení. Toto nahrazení bylo provedeno v několika částech projektu, a sice v případě provádění samotné statické analýzy kódu, vytváření strukturovaného pohledu na zdrojový kód a automatického doplňování nedokončených identifikátorů. Tato funkcionalita je po integraci nové knihovny rovněž funkční a dosahuje lepších výsledků než před ní. Analýza kódu po modifikacích se díky Yaramod-v4 zrychlila 149×. Díky zrychlení a odolnosti vůči chybám je nyní možné kód reanalyzovat po každé úpravě uživatelem.

V blízké době by měly být všechny součásti programového řešení open-source, což by mělo přinést výhody spojené s používáním této knihovny také dalším uživatelům jazyka YARA. V rámci dalšího vývoje knihovny Yaramod-v4 bude kladen důraz hlavně na rozhraní pro modifikaci a tvorbu nových pravidel. Díky tomu bude možné nahradit dosavadní řešení i v případech, kdy je zapotřebí upravovat a vytvářet pravidla pro nástroj YARA také z prostředí programovacích jazyků.

Literatura

[1] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2. vyd. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[2] Alvarez, V. M. *Yara Documentation – Release 4.2.1* [online]. 2022 [cit. 2022-12-20]. Oficiální dokumentace nástroje YARA. Dostupné z: https://yara.readthedocs.io/_/downloads/en/latest/pdf/.

[3] Alvarez, V. M. *YARA: The pattern matching swiss knife for malware researchers and everyone else* [online]. Prosinec 2014 [cit. 2022-12-20]. Záznam z konference SECURE 2014 z kanálu CERT Polska. Dostupné z: https://www.youtube.com/watch?v=TTLuy0gx5vE.

[4] Artho, C. a Biere, A. Combined Static and Dynamic Analysis. In: Cortesi, A. a Logozzo, F., ed. *Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005)*. Elsevier, 2005, sv. 131, s. 3–14. Electronic Notes in Theoretical Computer Science. DOI: 10.1016/j.entcs.2005.01.018. ISSN 1571-0661. Dostupné z: https://doi.org/10.1016/j.entcs.2005.01.018.

[5] Baca, D., Carlsson, B. a Lundberg, L. Evaluating the Cost Reduction of Static Code Analysis for Software Security. In: ACM. *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: Association for Computing Machinery, 2008, s. 79–88. PLAS '08. DOI: 10.1145/1375696.1375707. ISBN 9781595939364. Dostupné z: https://doi.org/10.1145/1375696.1375707.

[6] Beetem, J. F. a Beetem, A. F. Incremental Scanning and Parsing with Galaxy. *IEEE Trans. Softw. Eng.* IEEE Press. Červen 1991, sv. 17, č. 7, s. 641–651. DOI: 10.1109/32.83901. ISSN 0098-5589. Dostupné z: https://doi.org/10.1109/32.83901.

[7] Brunsfeld, M. *Tree-sitter* [online]. 2022 [cit. 2022-12-30]. DOI: 10.5281/zenodo.7045041. Oficiální repozitář a dokumentace projektu tree-sitter. Dostupné z: https://tree-sitter.github.io/tree-sitter/.

[8] Chess, B. a McGraw, G. Static analysis for security. *IEEE Security & Privacy*. 2004, sv. 2, č. 6, s. 76–79. DOI: 10.1109/MSP.2004.111.

[9] Chess, B. a West, J. *Secure Programming with Static Analysis*. Addison-Wesley, 2007. Addison-Wesley software security series. ISBN 9780321424778.

[10] Dasgupta, A., Narasayya, V. a Syamala, M. A Static Analysis Framework for Database Applications. In: IEEE. *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, s. 1403–1414. DOI: 10.1109/ICDE.2009.98. ISBN 9781424434220.

[11] Do, L. N. Q., Wright, J. R. a Ali, K. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering*. 2022, sv. 48, č. 3, s. 835–847. DOI: 10.1109/TSE.2020.3004525.

[12] Donnely, C. a Stallman, R. *Bison*. GNU, 2021 [cit. 2022-12-29]. Oficiální dokumentace nástroje GNU Bison. Dostupné z: https://www.gnu.org/software/bison/manual/bison.pdf.

[13] Dumont, M., Hogenson, G. a Vasani, M. e. a. Overview of source code analysis. *Microsoft Learn* [online]. 2023 [cit. 2023-4-26]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022.

[14] Emanuelsson, P. a Nilsson, U. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*. Elsevier. 2008, sv. 217, s. 5–21. DOI: 10.1016/j.entcs.2008.06.039. ISSN 1571-0661.

[15] Ernst, M. D. Static and dynamic analysis: Synergy and duality. In: IEEE. *WODA 2003: Workshop on Dynamic Analysis*. Portland, OR, USA: IEEE, Květen 2003, s. 24–27. ISSN 0270-5257.

[16] Evans, D. *Splint Manual* [online]. 2003 [cit. 2022-12-29]. Oficiální dokumentace nástroje Splint. Dostupné z: https://splint.org/manual/.

[17] Gamma, E., Helm, R., Johnson, R. a Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. Addison-Wesley Professional Computing Series. ISBN 9780321700698. Dostupné z: https://books.google.cz/books?id=6oHuKQe3TjQC.

[18] Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J. A. et al. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In: ACM. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2019, s. 1069–1084. PLDI 2019. DOI: 10.1145/3314221.3314590. ISBN 9781450367127. Dostupné z: https://doi.org/10.1145/3314221.3314590.

[19] Ghezzi, C. a Mandrioli, D. Augmenting Parsers to Support Incrementality. *J. ACM*. New York, NY, USA: Association for Computing Machinery. jul 1980, sv. 27, č. 3, s. 564–579. DOI: 10.1145/322203.322215. ISSN 0004-5411. Dostupné z: https://doi.org/10.1145/322203.322215.

[20] Guarnieri, C. *Cuckoo Sandbox Book – Release 2.0.7* [online]. 2020 [cit. 2022-12-20]. Oficiální dokumentace projektu Cuckoo Sandbox. Dostupné z: https://yara.readthedocs.io/en/stable/.

[21] Holmes, L. Searching for Content in Base-64 Strings. *Precision Computing – Software Design and Development* [online]. 2017 [cit. 2022-12-21]. Dostupné z: https://www.leeholmes.com/searching-for-content-in-base-64-strings/.

[22] Hovemeyer, D. a Pugh, W. *FindBugs Manual* [online]. 2015 [cit. 2022-12-29]. Oficiální dokumentace projektu FindBugs. Dostupné z: https://findbugs.sourceforge.net/manual/index.html.

[23] Johnson, S. C. *Lint, a C program checker*. Bell Telephone Laboratories, Murray Hill, 1977. Dokument prezentující nástroj Link. Dostupné z: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf.

[24] Joshi, A., Balázs, D. a Sigoure, B. e. a. *GoogleTest*. Google, duben 2023 [cit. 2022-04-25]. Oficiální repozitář frameworku GoogleTest. Dostupné z: https://github.com/google/googletest.

[25] Kašťák, M. YLS: First Step Towards YARA Development Environment. *Avast Engineering* [online]. 2022 [cit. 2022-12-30]. Dostupné z: https://engineering.avast.io/yls-first-step-towards-yara-development-environment/.

[26] Kelner, M., Hogenson, G. a Sherer, T. e. a. Language Server Protocol. *Microsoft Learn* [online]. 2022 [cit. 2022-12-31]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=vs-2022.

[27] Kender, T. *Rozšiřování jazyka YARA*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23242/.

[28] Lipner, S. The trustworthy computing security development lifecycle. In: IEEE. *20th Annual Computer Security Applications Conference*. IEEE, Leden 2004, s. 2– 13 [cit. 2023-4-11]. DOI: 10.1109/CSAC.2004.41. ISBN 0-7695-2252-1. Dostupné z: https://ieeexplore.ieee.org/document/1377202/keywords#keywords.

[29] Logilab, PyCQA, e. a. *Pylint Documentation – Release 2.7.1* [online]. Logilab and PyCQA, únor 2021 [cit. 2023-04-20]. Oficiální dokumentace nástroje Pylint. Dostupné z: https://pylint.readthedocs.io/_/downloads/en/2.7.1/pdf/.

[30] Lohmann, N. *JSON for Modern C++*. Březen 2023 [cit. 2022-04-25]. Oficiální dokumentace knihovny JSON for Modern C++. Dostupné z: https://json.nlohmann.me/.

[31] Louridas, P. Static Code Analysis. *IEEE Softw.* Washington, DC, USA: IEEE Computer Society Press. Červenec 2006, sv. 23, č. 4, s. 58–61. DOI: 10.1109/MS.2006.114. ISSN 0740-7459. Dostupné z: https://doi.org/10.1109/MS.2006.114.

[32] Madsen, M. *Static Analysis of Dynamic Languages*. Aarhus, DK, 2015. Disertační práce. Department of Computer Science, Aarhus University. Dostupné z: https://pure.au.dk/portal/files/85299449/Thesis.pdf.

[33] Milkovič, M., Kučera, T. a Kašťák, M. e. a. *Yaramod*. Avast, 2022 [cit. 2022-12-30]. Oficiální dokumentace knihovny Yaramod-v3. Dostupné z: https://yaramod.readthedocs.io/_/downloads/en/latest/pdf/.

[34] Mino, S. IDE vs Code Editor – Why and When to Use Them. *Jobsity* [online]. Duben 2022 [cit. 2022-12-31]. Dostupné z: https://www.jobsity.com/blog/ide-vs-code-editor-why-and-when-to-use-them.

[35] Møller, A. a Schwartzbach, M. I. *Static Program Analysis* [online]. Říjen 2018 [cit. 2023-4-11]. Department of Computer Science, Aarhus University. Dostupné z: http://cs.au.dk/~amoeller/spa/.

[36] Open Law Library. *Pygls Documentation*. Duben 2023 [cit. 2023-04-23]. Oficiální dokumentace knihovny Pygls. Dostupné z: https://pygls.readthedocs.io/_/downloads/en/v1.0.0/pdf/.

[37] Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H. a Grillenberger, F. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In: IEEE. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies and Factory Automation (ETFA 2012)*. IEEE, 2012, s. 1–8. DOI: 10.1109/ETFA.2012.6489535. ISBN 9781467347358.

[38] Reps, T. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In: ACM. *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1982, s. 169–176. POPL '82. DOI: 10.1145/582153.582172. ISBN 0897910656. Dostupné z: https://doi.org/10.1145/582153.582172.

[39] Shvets, A. Builder. *Refactoring Guru* [online]. 2021 [cit. 2022-12-30]. Dostupné z: https://refactoring.guru/design-patterns/builder.

[40] Shvets, A. Visitor. *Refactoring Guru* [online]. 2021 [cit. 2022-12-30]. Dostupné z: https://refactoring.guru/design-patterns/visitor.

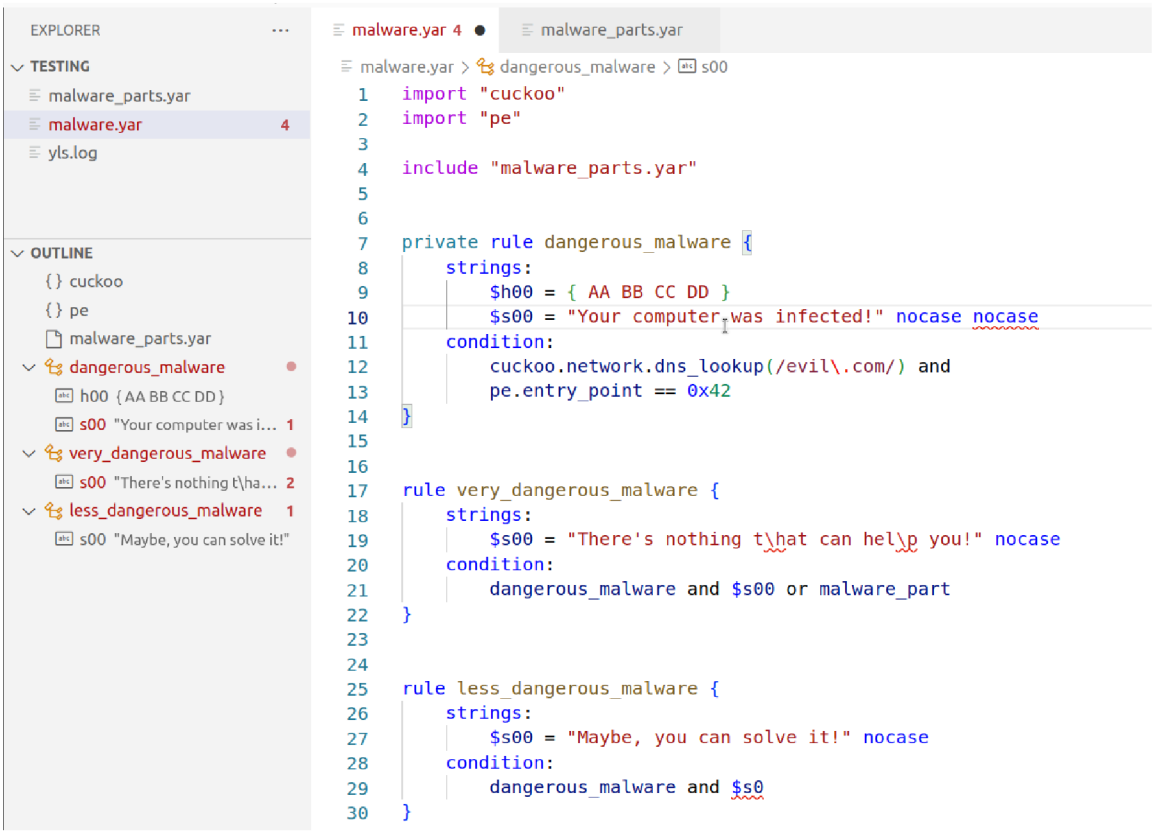
[41] Wagner, T. A. *Practical Algorithms for Incremental Software Development Environments*. Berkeley, 1998. Disertační práce. EECS Department, University of California. Dostupné z: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html.

[42] Wenzel, J. *Pybind11 Documentation*. Březen 2023 [cit. 2023-04-23]. Oficiální dokumentace knihovny Pybind11. Dostupné z: https://pybind11.readthedocs.io/_/downloads/en/stable/pdf/.

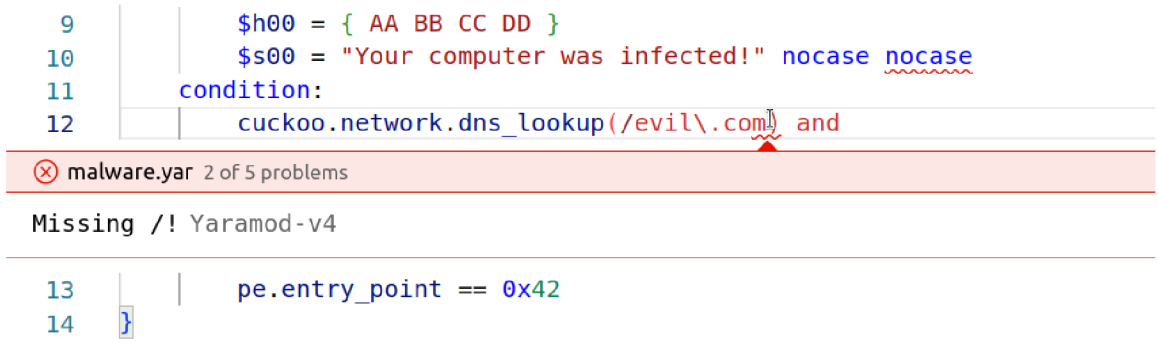
Příloha A

Snímky obrazovky

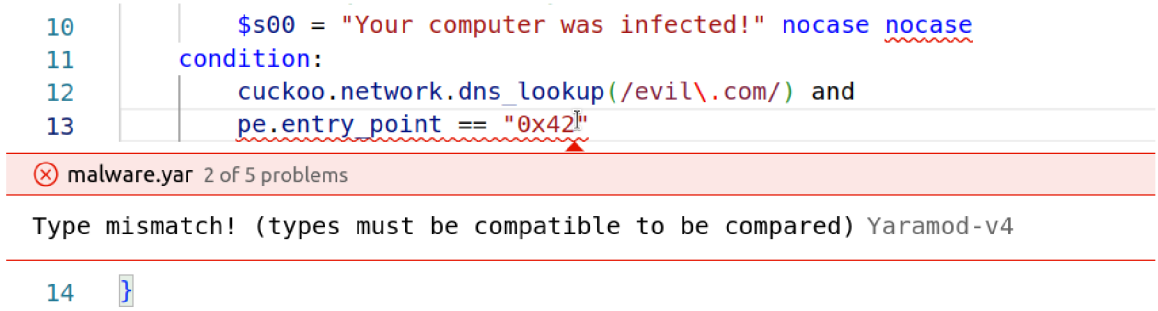
Následující snímky obrazovky byly pořízeny v textovém editoru VSCode komunikujícím s YLS, v rámci kterého došlo k integraci s programovým řešením této práce, knihovnou Yaramod-v4.



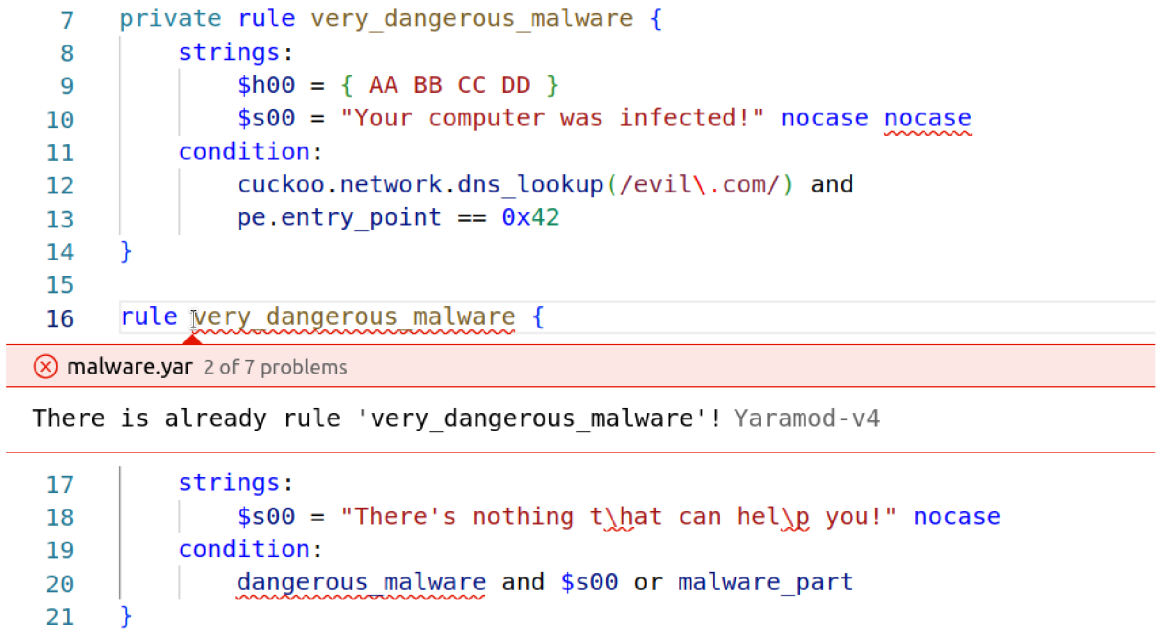
Obrázek A.1: Ukázka funkcionality Linting a Document Symbol – Yaramod-v4 poskytuje hierarchický seznam symbolů, který je možné ve VSCode vidět v levé dolní části okna. Dále je v horní liště přítomný popis pozice kurzoru v této hierarchii. Na snímku jsou také červená podtržení, což jsou chyby identifikované knihovnou tree-sitter.



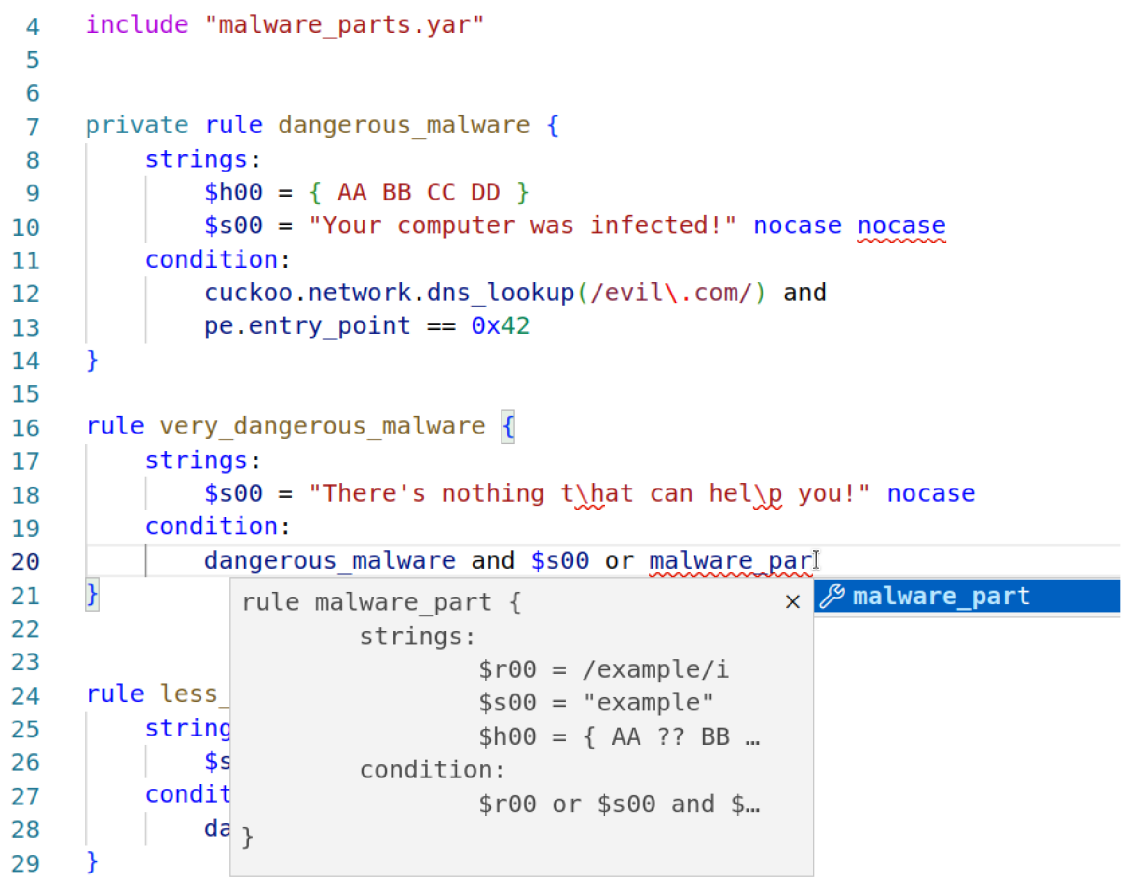
Obrázek A.2: Ukázka syntaktických kontrol prováděných knihovnou Yaramod-v4



Obrázek A.3: Ukázka typových kontrol prováděných knihovnou Yaramod-v4



Obrázek A.4: Kontrola duplicitních identifikátorů prováděná knihovnou Yaramod-v4



Obrázek A.5: Ukázka funkcionality Completion – Yaramod-v4 našel dříve definované pravidlo (ve vloženém zdrojovém souboru `malware_parts.yar`), které odpovídá rozepsanému symbolu, a nabídl ho uživateli. Knihovna zároveň poskytuje definici tohoto pravidla pomocí serializace.

Příloha B

Obsah přiloženého paměťového média

V následující tabulce je stručně popsán obsah adresáře, který je přítomen na přiloženém paměťovém médiu. Bližší popis a další informace jsou uvedeny v souboru `README.md`.

Cesta	Popis
`bp-src/`	Zdrojové soubory pro vytvoření písemné zprávy
`yaramod-v4/`	Zdrojové soubory knihovny Yaramod-v4
`yaramod-v4/bin/`	Knihovna Yaramod-v4 v binární podobě
`yaramod-v4/bindings/python/`	Zdrojové soubory rozhraní knihovny pro Python
`yaramod-v4/doc/`	Soubory potřebné pro generování dokumentace
`yaramod-v4/examples/`	Zdrojové soubory příkladů použití knihovny
`yaramod-v4/lib/`	Hlavičkové soubory knihovny
`yaramod-v4/modules/`	Popis modulů ve formátu JSON (převzato z [33])
`yaramod-v4/src/`	Zdrojové soubory knihovny Yaramod-v4
`yaramod-v4/tests/`	Testy knihovny Yaramod-v4
`yaramod-v4/tree-sitter-yara/`	Zdrojové soubory modulu tree-sitter-yara
`yaramod-v4/utils/json/`	Knihovna JSON for modern C++ (převzato z [30])
`yaramod-v4/refman.pdf`	Ref. manuál (vygenerovaný nástrojem Doxygen)
`yls/`	Soubory potřebné pro integraci s projektem YLS
`bp.pdf`	Písemná zpráva ve formátu PDF
`DEMO.mp4`	Video demonstrující fungování YLS s Yaramod-v4
`README.md`	Detailní informace o obsahu adresáře na p. médiu

Tabulka B.1: Popis obsahu adresáře na přiloženém paměťovém médiu