Zadání diplomové práce

Název:	Rozšíření nástroje Woke
Student:	Bc. Michal Převrátil
Vedoucí:	Ing. Josef Gattermayer, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Woke je vznikající open source nástroj pro statickou analýzu kódu smart kontraktů v jazyce Solidity. Je implementován v jazyce Python a je snadno rozšiřitelný o budoucí bezpečnostní moduly. Inspirován je nástrojem Slither, který si bere za cíl nahradit. Cílem této práce je rozšířit nástroj Woke o další funkcionalitu.

Pokyny:
- Nastudujte nástroje Slither a Woke.
- Po dohodě s vedoucím práce navrhněte (pod)množinu funkcionalit o které rozšíříte nástroj Woke.
- Navrhněte vhodný postup implementace funkcionalit do nástroje Woke.
- Proveďte implementaci a otestujte její správnost.

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 21. října 2021 v Praze.

Diplomová práce

ROZŠÍŘENÍ NÁSTROJE WOKE

Bc. Michal Převrátil

Fakulta informačních technologií
Katedra počítačové bezpečnosti
Vedoucí: Ing. Josef Gattermayer, Ph.D.
3. května 2022

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Bc. Michal Převrátil. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Převrátil Michal. *Rozšíření nástroje Woke.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Obsah

Úvod **1**

1 Ethereum **3**
1.1 Keccak-256 a SHA-3 . . . 3
1.2 Ether a Gas . . . 4
1.3 Ethereum Improvement Proposal . . . 4
1.4 Yellowpaper . . . 5
1.4.1 Recursive Length Prefix . . . 5
1.4.2 Hex-prefix kódování . . . 8
1.4.3 Modifikovaný Merkle Patricia strom (trie) . . . 8
1.4.4 Merkle proof . . . 10
1.4.5 Stav EVM . . . 11
1.4.6 Předkompilované (precompiled) smart kontrakty . . . 12
1.4.7 Transakce . . . 12
1.4.8 Bloky . . . 14
1.4.9 Ethereum Virtual Machine . . . 15
1.5 Těžba bloků . . . 21

2 Solidity **23**
2.1 Popis jazyka . . . 23
2.1.1 Doporučení pro formátování kódu . . . 23
2.1.2 Datové typy . . . 24
2.1.3 Struktura souborů . . . 26
2.2 Kompilátor *solc* . . . 30
2.2.1 Vyhodnocování importních cest . . . 31

3 Nástroje používané Ethereum komunitou **35**
3.1 Nástroj Slither . . . 35
3.1.1 Detektory zranitelností . . . 35
3.1.2 Příkazy pro vizualizaci dat . . . 37
3.1.3 Další příkazy Slitheru . . . 38
3.2 Remix IDE . . . 39
3.3 Nástroj solc-select . . . 39

4 Nástroj Woke **41**
4.1 Analýza současného stavu . . . 41
4.2 Implementace nástroje . . . 42
4.2.1 Knihovna pydantic . . . 42
4.2.2 Příprava vývojového prostředí . . . 43
4.2.3 Práce s konfiguračními soubory . . . 44
4.2.4 Správce instalací kompilátoru *solc* . . . 48
4.2.5 Parsování Solidity souborů . . . 49
4.2.6 Kompilace projektů . . . 54

4.2.7 Parsování AST dat . . . 57
4.2.8 Interakce s příkazovou řádkou . . . 57
4.3 Testování implementace . . . 59
4.3.1 Vyhodnocení pokrytí implementace testy . . . 59
4.3.2 Vyhodnocení rychlosti kompilace . . . 60

Závěr **63**

A Použití nástroje Woke **65**
A.1 Instalace . . . 65
A.2 Použití správce verzí kompilátoru *solc* . . . 66
A.3 Kompilace projektu . . . 67

B Navrhované budoucí funkcionality **69**
B.1 Automatické formátování kódu . . . 69
B.2 Interpret a symbolické vykonávání jazyka Solidity . . . 69

Obsah přiloženého média **81**

Seznam obrázků

1.1 Příklad vstupu algoritmu RLP 7
1.2 Zjednodušené znázornění modifikovaného Merkle Patricia stromu (trie) 10
1.3 Příklad aplikace algoritmu *merkle proof* 11

3.1 Ukázka *control flow grafu* vygenerovaného Slitherem 38

4.1 Příklad analýzy kompilační jednotky nástrojem Woke 57

B.1 Terminálové rozhraní symbolického debuggeru jazyka Solidity 72

Seznam tabulek

1.1 Ether a jeho dílčí jednotky 4
1.2 Předkompilované (precompiled) smart kontrakty 12

4.1 Informace o Solidity projektech zvolených pro testování kompilace 61
4.2 Porovnání rychlosti kompilace nástroje Woke s jinými nástroji 62

Seznam výpisů kódu

2.1 Ukázka komentářů v Solidity 27
2.2 Ukázka funkce jazyka Solidity 29
2.3 Ukázka modifikátoru funkce jazyka Solidity 29

3.1 Ukázka implementace vlastního detektoru ve Slitheru 36
3.2 Ukázka výstupu Slither příkazu `contract-summary` 37
3.3 Implementace bezpečného sečtení čísel knihovny OpenZeppelin 37

4.1 Příklad použití knihovny *pydantic* v nástroji Woke 42
4.2 Ukázka konfiguračního souboru nástroje Woke v YAML formátu 45
4.3 Ukázka konfiguračního souboru nástroje Woke v TOML formátu 45
4.4 Ukázka všech možností konfiguračního souboru nástroje Woke 47

4.5 Ukázka výstupu příkazu `woke config` 47
4.6 Úryvek zdrojového kódu kompilátoru *solc* zpracovávající verzovací výraz 51
4.7 Použití regulárních výrazů pro parsování Solidity verzovacích výrazů 53
4.8 Použití regulárních výrazů pro parsování Solidity `import` výrazů 54
4.9 Použití knihovny *click* v příkazu `woke compile` 58
4.10 Pokrytí kódu nástroje Woke testy 59

B.1 Funkce symbolického vykonávání příkazu `if` jazyka Solidity 71

Rád bych touto cestou poděkoval vedoucímu této práce Ing. Josefu Gattermayerovi, Ph.D. za ochotu a cenné rady, které vedly ke zkvalitnění této práce. Také bych rád za velmi přínosné technické konzultace poděkoval Dominiku Teimlovi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

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

V Praze dne 3. května 2022 ....

Abstrakt

Tato práce se zabývá implementací modulů do vznikajícího nástroje Woke určeného pro analýzu smart kontraktů na platformě Ethereum. V práci je detailně popsán Ethereum blockchain a programovací jazyk Solidity určený pro vývoj aplikací na tuto platformu. Dále je v práci analyzován nástroj Slither, který je, stejně jako nástroj Woke, napsaný v jazyce Python. Praktická část práce popisuje postup implementace jednotlivých modulů nástroje Woke a jejich testování.

Klíčová slova blockchain, Ethereum, Woke, Solidity, statická analýza

Abstract

This thesis deals with the implementation of modules in the emerging Woke tool for analyzing smart contracts on the Ethereum platform. The work describes in detail the Ethereum blockchain and the Solidity programming language for developing applications on this platform. Furthermore, the thesis analyses the Slither tool, which, like Woke, is written in Python. The practical part of the thesis describes the process of implementation of individual modules of the Woke tool and their testing.

Keywords blockchain, Ethereum, Woke, Solidity, static analysis

Seznam zkratek

ABI	Application Binary Interface
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BIP	Bitcoin Improvement Proposal
EIP	Ethereum Improvement Proposal
EVM	Ethereum Virtual Machine
IDE	Integrated Development Environment
LSP	Language Server Protocol
NIST	Národní institut standardů a technologie (National Institute of Standards and Technology)
NSA	Národní bezpečnostní agentura (The National Security Agency)
PEP	Python Enhancement Proposal
PRNG	Pseudorandom Number Generator
RLP	Recursive Length Prefix
SHA	Secure Hash Algorithm
SVM	Solc Version Manager

Úvod

Na základě dokumentu „Bitcoin: A Peer-to-Peer Electronic Cash System“ vznikla technologie blockchainu – distribuované decentralizované databáze [1]. Bitcoin blockchain byl již od začátku navržen jako virtuální platební systém. Ethereum blockchain na tento koncept navázal a rozšířil jej o myšlenku vybudovat distribuovaný počítač, který by sloužil jako platforma pro decentralizované aplikace [2]. Toto bylo umožněno díky specifikaci způsobu ukládání libovolných programových dat a turingovsky kompletní instrukční sadě.

Díky použití relativně nové technologie blockchainu a možnosti vytvářet libovolně komplexní aplikace (nazývané *smart kontrakty*) vznikl prostor pro řadu více či méně sofistikovaných útoků. Již z podstaty blockchainu značná část decentralizovaných aplikací pracuje s virtuálními finančními prostředky. I proto je bezpečnost nasazených smart kontraktů i samotného blockchainu velmi důležitým tématem. Přesto byly blockchainové sítě v minulosti úspěšným cílem velkého množství útoků, které přetrvávají dodnes [3][4]. Útokům na Ethereum blockchainu situaci ulehčuje skutečnost, že je instrukční kód všech aplikací veřejně dostupný. Často je také dostupný odpovídající kód ve vyšším programovacím jazyce, jelikož jeho zveřejnění zvyšuje důvěryhodnost projektu. Společně s kódem jsou veřejně dostupné všechny hodnoty stavových proměnných programu. V případě nalezení chyby v již nasazeném smart kontraktu je, v závislosti na implementaci, jeho oprava velmi komplikovaná nebo dokonce nemožná. V případě odcizení virtuálních finančních prostředků je po velmi krátké době transakce, ve které k odcizení došlo, nevratná[1]. Bezpečnostní analýza smart kontraktů před nasazením na blockchain je proto nutností.

V případě aplikací pro Ethereum blockchain je možné riziko úspěšného útoku minimalizovat správným návrhem aplikace, dodržováním doporučených postupů a především auditováním kódu. Problémem je však nepochybně rychlost vývoje Ethereum blockchainu včetně programovacího jazyka Solidity určeného pro vývoj na tuto platformu [6]. Díky kombinaci rychlého vývoje Ethereum blockchainu a faktu, že tato platforma je relativně nová, neexistuje velké množství kvalitních nástrojů, které by zefektivnily práci vývojářů a auditorů.

Tato skutečnost je motivací pro vytvoření nástroje Woke, který je nosným tématem této práce. Cílem nástroje je nabídnout funkcionality, které by minimalizovaly zranitelnosti smart kontraktů vytvořených v programovacím jazyku Solidity. Woke by měl svými funkcionalitami nahradit nástroj Slither, který se zaměřuje na statickou analýzu Solidity kódu. Náplní této práce je implementovat a otestovat moduly nástroje Woke, pomocí kterých může být tohoto cíle dosaženo.

[1] K výjimečné situaci došlo v červnu 2016, kdy byly na základě většinové shody vráceny zpět transakce, v rámci kterých bylo odcizeno značné množství finančních prostředků [5]. Tento útok se obvykle označuje jako *DAO attack* nebo *DAO hack*.

Kapitola 1

Ethereum

V roce 2013 přišel Vitalik Butterin s nápadem rozšířit Mastercoin (rozšiřující protokol nad Bitcoinem) o skriptovací jazyk, který by umožnil vytvářet smart kontrakty – programy fungující na blockchainu [7]. Návrh Butterina byl pro vývojářský tým Mastercoinu příliš radikální, a proto nebyl implementován. V reakci na to Vitalik Butterin v prosinci 2013 zveřejnil dokument popisující blockchain se zabudovaným turingovsky úplným (Turing complete) strojem. Tento dokument se stal základním kamenem pro vytvoření kompletní specifikace Etherea – projektu, který si dal za cíl vytvořit globální programovatelný počítač založený na technologii blockchainu.

Bitcoin byl navržený pro uchovávání informace o uložených mincích na jednotlivých adresách [8]. Přestože jsou částí specifikace Bitcoinu i jednoduché instrukce, tyto instrukce jsou primárně určené pro vytváření a ověřování digitálních podpisů při převodu mincí mezi účty. Ethereum rozvíjí tuto myšlenku o možnost ukládání libovolné stavové informace ve formátu klíč-hodnota [9]. Kromě specifikace ukládání dat Ethereum také definuje turingovsky kompletní instrukční sadu, která s těmito daty může pracovat a umožňuje vytváření libovolně komplexních aplikací. Tyto aplikace ve finále modifikují globální stav distribuovaného počítače, kterým Ethereum je. Celý koncept formálně popisuje dokument nazvaný Yellowpaper [2]. Zdrojový kód dokumentu je dostupný v repozitáři na adrese `https://github.com/ethereum/yellowpaper`.

1.1 Keccak-256 a SHA-3

2. listopadu 2007 Národní institut standardů a technologie (NIST) vyhlásil soutěž o novou standardizovanou hashovací funkci Secure Hash Algorithm 3 (SHA-3) [10]. Do posledního kola postoupila mimo jiné i hashovací funkce Keccak. V době soutěže mohli autoři funkcí provádět některé úpravy v případě, že našli potenciální zranitelnosti či jiné nedokonalosti. Některé návrhy na úpravy připravoval také přímo NIST.

Toto se odehrávalo v době, kdy Edward Snowden odhalil některé dokumenty, ze kterých se mohlo usuzovat, že NIST může být ovlivňován Národní bezpečnostní agenturou (NSA) [7]. Ethereum foundation, organizace zastřešující projekt Ethereum, proto ve svých specifikacích použila původní návrh hashovací funkce Keccak tak, jak byla autory navržena. Finální specifikace schválená institutem NIST se od původního návrhu lišila jen v některých detailech, ale z principu fungování hashovacích funkcí obě tyto specifikace pro stejná data poskytují zcela jiné výstupy.

V mnoha dokumentech i kódu týkajících se Etherea dříve často byla původní specifikace hashovací funkce Keccak-256[1] nesprávně označována jako SHA-3 [11][12]. Dnes by měla být

[1] V Ethereu se velmi často používá 256-bitová varianta hashovací funkce Keccak.

Tabulka 1.1 Ether a jeho dílčí jednotky

Jednotka	Relativní hodnota
Wei	10^0
KWei (Babbage)	10^3
MWei (Lovelace)	10^6
GWei (Shannon)	10^9
microEther (Szabo)	10^{12}
milliEther (Finney)	10^{15}
Ether	10^{18}

velká část těchto chybných pojmenování opravena. Zmatení ale v komunitě přetrvává dodnes.

1.2 Ether a Gas

Narozdíl od Bitcoinu, který byl již od začátku navrhován jako platidlo, Ethereum spíše než platforma pro platby je platformou pro vytváření decentralizovaných aplikací [9]. Od toho se odvíjí i fakt, že se v Ethereu používá více platidel.

Pro převod hodnotných mincí mezi účty se používá Ether [13]. Hodnotu Etheru (vůči konvenčním platidlům) určuje „neviditelná ruka trhu“. Ether se dělí na menší jednotky, které jsou vhodnější z pohledu popisného a implementačního hlediska. Tabulka 1.1 popisuje vztahy pro převod od nejmenší jednotky Wei, po největší Ether.

Druhou hlavní jednotkou je Gas [14]. Gas slouží jako jednotka pro vyjádření ceny za zpracování transakce. Jedná se o pomocnou jednotku – za transakce se ve finále platí Etherem, kdy převodní poměr mezi Gasem a Etherem[2] si volí odesílatel transakce. Těžaři ale mohou transakci ignorovat, pokud pro ně cena za transakci není dostatečně lákavá. Naopak v případě velkého vytížení sítě (nadprůměrného vytváření nových transakcí) může uživatel sítě zařídit prioritizaci své transakce tím, že nabídne nadprůměrný převodní poměr mezi Gasem a GWei.

1.3 Ethereum Improvement Proposal

Ethereum Improvement Proposal (EIP) je dokument určený Ethereum komunitě popisující návrh na změnu či novou funkcionalitu týkající se Etherea [15]. Autor dokumentu by měl dostatečně technicky zdokumentovat změny navrhované v dokumentu a uvést motivaci pro tyto změny. Také je zodpovědností autora, aby byl návrh přijat komunitou a aby zdokumentoval případné nesouhlasné názory.

Formát EIP je inspirovaný Bitcoin Improvement Proposal (BIP) [16], který je inspirovaný Python Enhancement Proposal (PEP) [17].

Podle EIP-1 [15] existuje několik typů EIP:

- **Standards Track EIP:** EIP popisující změny přímo v implementaci Etherea. Může se jednat o změny komunikačního protokolu, změnu pravidel pro přijetí či zamítnutí bloků nebo transakcí. EIP může dále navrhovat nové standardy nebo konvence ve vývoji aplikací souvisejících s projektem Ethereum. Tento typ EIP obvykle obsahuje dvě nebo tři části: samotný

[2] Nejčastěji se převodní poměr Gasu udává proti GWei.

dokument popisující změny, implementaci změn, a v odůvodnělých případech úpravu formální specifikace (především tedy Yellowpaperu).

Standards Track EIP se dále dělí do několika kategorií:

  - **Core:** EIP zasahující do kritických částí Ethereum protokolu, pro jejichž začlenění je potřeba zajistit shodu v komunitě a protokol aktualizovat. Patří sem například EIP přidávající nové instrukce nebo upravující formát transakcí.
  - **Networking:** EIP upravující síťové komunikační protokoly využívané v Ethereum síti.
  - **Interface:** EIP popisující rozhraní a podporované funkcionality klientských aplikací v Ethereum síti. Interface EIP také mohou popisovat změny názvů instrukcí EVM či rozhraní pro komunikaci s kontrakty na Ethereum síti.
  - **ERC:** EIP obecně zavádějící standardy a doporučení. Mimo jiné sem patří EIP-20 [18], které v jazyce Solidity popisuje (ve formě hlaviček funkcí) rozhraní kontraktů, které spravují softwarově implementované virtuální prostředky, *tokeny*, které jsou často označované jako ERC-20 tokeny. Díky tomuto EIP existuje jednotný standard pro vytváření vlastních platidel projektů, které na Ethereum síť nahrávají své aplikace.

- **Meta EIP:** Tyto EIP popisují proces či událost ve vývoji Etherea. Patří sem především EIP popisující aktualizace Ethereum protokolu. Obsahem těchto EIP je typicky událost pro aktivaci aktualizace a seznam Core EIP, které detailně ukazují jednotlivé technické změny.

- **Informational EIP:** Poslední kategorií jsou informační EIP, které kromě předání informací zavádí doporučení. Příkladem je EIP-2228 [19], které doporučuje pro síť (resp. chain) s ID 1 používat název Ethereum Mainnet (nebo jen Mainnet).

1.4 Yellowpaper

Yellowpaper je formální specifikace Etherea [2]. Základním pojmem Yellowpaperu je state machine – abstraktní výpočetní model udržující si v každém časovém okamžiku stav. Ve stavu mohou být uloženy různé informace, například vlastnictví virtuálních mincí danými účty, kód (program) uložený na některých účtech nebo libovolná data ve formátu klíč-hodnota. V kontextu Etherea se většinou mluví o Ethereum Virtual Machine (EVM).

Přechod z jednoho validního stavu do druhého je v případě Etherea prováděn prostřednitvím transakcí. Transakce jsou sbírány do bloků, které jsou za sebe řazeny pomocí hashů – nový oblok obsahuje hash předcházejícího bloku. Odtud vznikl název blockchain (řetěz bloků).

Bloky uchovávající transakce tedy slouží jako historie záznamů změn stavu EVM. Postupnou aplikací změn od prvního bloku až po aktuální blok by měl být každý schopný získat stav aktuální, tedy stav, který sdílí všichni na Ethereum blockchainu.

Tato kapitola popisuje stav Ethereum sítě po aplikaci aktualizace s názvem Berlin, konkrétně commit 4b05e0d v GitHub repozitáři Yellowpaperu [20].

1.4.1 Recursive Length Prefix

Recursive Length Prefix (RLP) je algoritmus pro kódování strukturovaných binárních dat [2, příloha B]. Výstupem je posloupnost (pole) bytů. Vstupem algoritmu může být:

- pole bytů,

- seřazený seznam, jehož položky mohou být:

  - pole bytů,
  - další vnořený seřazený seznam.

Pro účely formálního popisu jsou pole bytů ohraničená hranatými závorkami $[a_0,\ a_1,\ \ldots]$, seznamy jsou ohraničené kulatými závorkami $(l_0,\ l_1,\ \ldots)$. Pole a seznamy nejsou zaměnitelné, speciálně [ ] (prázdné pole) není rovno ( ) (prázdnému seznamu). Dále výraz $\|\mathbf{x}\|$ značí délku pole, resp. seznamu $\mathbf{x}$.

▶ **Definice 1.1** (Množina seznamů, množina polí bytů [2, příloha B] (*upraveno*))**.**

$$\begin{aligned}
&\mathbb{T} := \mathbb{L} \cup \mathbb{B}\\
&\mathbb{L} := \{\mathbf{t} \colon \mathbf{t} = (t_0,\ t_1,\ \ldots) \land \forall n < \|\mathbf{t}\| \colon t_n \in \mathbb{T}\}\\
&\mathbb{B} := \{\mathbf{b} \colon \mathbf{b} = [b_0,\ b_1,\ \ldots] \land \forall n < \|\mathbf{b}\| \colon b_n \in \mathbb{N}, 0 \leq b_n \leq 255\}
\end{aligned}$$

Vstupem algoritmu může být libovolný prvek z množiny $\mathbb{T}$. Výstupem algoritmu je prvek množiny $\mathbb{B}$.

▶ **Definice 1.2** (Zřetězení polí [2, příloha B] (*upraveno*))**.**

$$\begin{aligned}
&\forall \mathbf{a}, \mathbf{b} \in \mathbb{B}, \textit{kde } \mathbf{a} = [a_0,\ a_1,\ \ldots,\ a_m], \mathbf{b} = [b_0,\ b_1,\ \ldots,\ b_n], m, n \in \mathbb{N} \colon\\
&\mathbf{a} \cdot \mathbf{b} := [a_0,\ a_1,\ \ldots,\ a_m,\ b_0,\ b_1,\ \ldots,\ b_n]
\end{aligned}$$

▶ **Definice 1.3** (Kódování big-endian [2, příloha B] (*upraveno*))**.**

$$\forall x \in \mathbb{N} \setminus \{0\} \colon \mathtt{BE}(x) := \mathbf{b} = [b_0,\ b_1,\ \ldots] \colon b_0 \neq 0 \land x = \sum_{n=0}^{\|\mathbf{b}\|-1} b_n \cdot 256^{\|\mathbf{b}\|-1-n}$$

▶ **Definice 1.4** (Funkce RLP [2, příloha B] (*upraveno*))**.** *Výstup algoritmu RLP závisí na tom, zda je vstupem pole či seznam:*

$$\forall \mathbf{t} \in \mathbb{T} \colon \mathtt{RLP}(\mathbf{t}) := \begin{cases} \mathtt{RLP_b}(\mathbf{t}) & \textit{pro } \mathbf{t} \in \mathbb{B} \\ \mathtt{RLP_l}(\mathbf{t}) & \textit{pro } \mathbf{t} \in \mathbb{L} \end{cases}$$

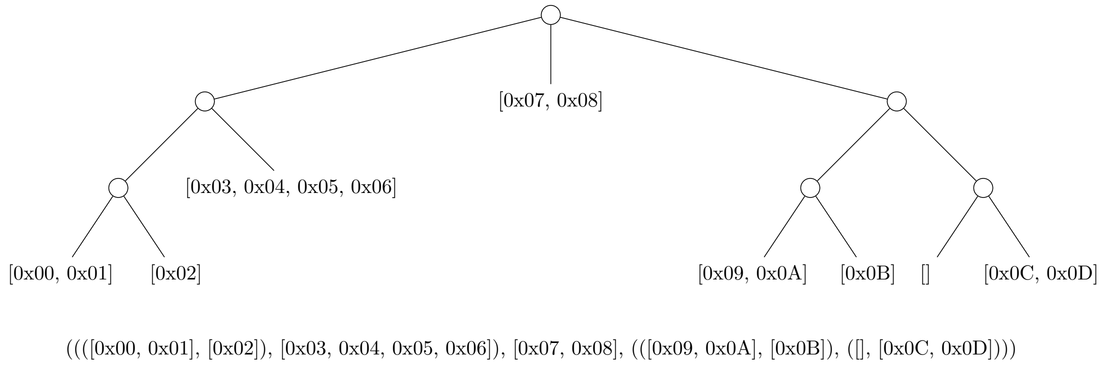
Jestliže je vstupem algoritmu pole bytů:

- Jestliže pole obsahuje jediný byte s hodnotou menší než 128, výstupem je přímo toto pole s jedinou hodnotou.
- Jestliže pole obsahuje méně než 56 bytů, výstupem je obsah pole s zřetězené s polem s jedním bytem, jehož hodnota je délka vstupního pole plus 128.
- V případech, kdy je délka vstupního pole méně než 2^{64}, je výstupem vstupní pole bytů, jemuž je předřazeno pole bytů - bytová reprezentace čísla ve formátu big-endian popisujícího délku vstupního pole. Sloučení obou těchto polí je pak ještě předřazen jediný byte s hodnotou délky předřazeného pole v předchozím kroku plus 183.
- Případy, kdy délka vstupního pole je větší nebo rovna 2^{64}, nejsou podporovány.

▶ **Definice 1.5** (Funkce RLP pro vstup z množiny polí bytů [2, příloha B] (*upraveno*))**.** *Tuto skutečnost lze formálně definovat:*

$$\forall \mathbf{b} \in \mathbb{B} \colon$$

$$\mathtt{RLP_b}(\mathbf{b}) := \begin{cases} \mathbf{b} & \textit{pro } \|\mathbf{b}\| = 1 \land 0 \leq b_0 < 128 \\ [128 + \|\mathbf{b}\|] \cdot \mathbf{b} & \textit{pro } \|\mathbf{b}\| < 56 \land (\|\mathbf{b}\| = 1 \Rightarrow 128 \leq b_0 \leq 255) \\ [183 + \|\mathtt{BE}(\|\mathbf{b}\|)\|] \cdot \mathtt{BE}(\|\mathbf{b}\|) \cdot \mathbf{b} & \textit{pro } 56 \leq \|\mathbf{b}\| < 2^{64} \\ \emptyset & \textit{pro } 2^{64} \leq \|\mathbf{b}\| \end{cases}$$



Obrázek 1.1 Příklad vstupu algoritmu RLP

V případě seznamu může struktura připomínat strom, kde listy jsou pole bytů a vnitřní uzly reprezentují seznamy. V příkladu na obrázku 1.1 jsou seznamy značené kulatými závorkami, pole bytů hranatými závorkami.

V případě vstupu ve formě seznamu je výstup definován rekurentně:

- Jestliže po aplikaci RLP na každou položku seznamu a zřetězení výstupních polí vznikne pole o délce méně než 56 bytů, výstupem je právě toto pole, jemuž je předřazen byte s délkou tohoto pole plus 192.

- Jestliže po aplikaci RLP na každou položku seznamu a zřetězení výstupních polí vznikne pole o délce méně než 2^{64} bytů, výstupem je právě toto pole, jemuž je předřazeno pole bytů – bytová reprezentace čísla ve formátu big-endian popisujícího délku zřetězeného pole vzniklého aplikací RLP na jednotlivé položky seznamu. Zřetězení obou těchto polí je pak ještě předřazen jediný byte s hodnotou délky předřazeného pole v předchozím kroku plus 247.

- Případ, kdy po aplikaci RLP na každou položku seznamu a zřetězení výstupních polí vznikne pole délky 2^{64} nebo více, není podporován.

▶ **Definice 1.6** (Serializace seznamu [2, příloha B] (*upraveno*))**.** *Serializaci seznamu lze definovat jako zřetězení polí vzniklých postupnou aplikací algoritmu RLP na každou položku seznamu (za předpokladu, že jednotlivé výstupy RLP jsou definovány):*

$$\forall \mathbf{l} \in \mathbb{L}\colon s(\mathbf{l}) \coloneqq \begin{cases} \texttt{RLP}(l_0) \cdot \texttt{RLP}(l_1) \cdot \ldots & \textit{jestliže } \forall n < \|\mathbf{l}\|\colon \texttt{RLP}(l_n) \neq \emptyset \\ \emptyset & \textit{jestliže } \exists n < \|\mathbf{l}\|\colon \texttt{RLP}(l_n) = \emptyset \end{cases}$$

▶ **Definice 1.7** (Funkce RLP pro vstup z množiny seznamů [2, příloha B] (*upraveno*))**.**

$$\forall \mathbf{l} \in \mathbb{L}\colon$$

$$\texttt{RLP}_1(\mathbf{l}) \coloneqq \begin{cases} [192 + \|s(\mathbf{l})\|] \cdot s(\mathbf{l}) & \textit{pro } \|s(\mathbf{l})\| < 56 \wedge s(\mathbf{l}) \neq \emptyset \\ [247 + \|\texttt{BE}(\|s(\mathbf{l})\|)\|] \cdot \texttt{BE}(\|s(\mathbf{l})\|) \cdot s(\mathbf{l}) & \textit{pro } 56 \leq \|s(\mathbf{l})\| < 2^{64} \wedge s(\mathbf{l}) \neq \emptyset \\ \emptyset & \textit{pro } \|s(\mathbf{l})\| \geq 2^{64} \vee s(\mathbf{l}) = \emptyset \end{cases}$$

▶ **Příklad 1.8.** Z dříve uvedených definic by mělo být patrné, že prázdné pole je kódované jinou hodnotou než prázdný seznam.

$$\texttt{RLP}([\,]) = [128]$$
$$\texttt{RLP}((\,)) = [192]$$

1.4.2 Hex-prefix kódování

Hex-prefix kódování je způsob převodu libovolného množství nibblů (čtveřic bitů) na bytové pole [2, příloha C]. Tento způsob kódování navíc umožňuje uložení dodatečné informace do bytového pole. Toto kódování se využívá v konstrukci modifikovaného Merkle Patricia stromu. Z toho vycházejí i následující definice, které předpokládají použití v modifikovaném Merkle Patricia stromu a jsou tomu uzpůsobené. V tomto případě se předpokládá, že je žádoucí do výsledné hodnoty zahrnout paritu kódovaného pole nibblů a binární informaci (typ uzlu v modifikovaném Merkle Patricia stromu).

▶ **Definice 1.9** (Množina polí nibblů).

$$\mathbb{Y} = \{\mathbf{y} \colon \mathbf{y} = [y_0,\ y_1,\ \ldots] \colon \forall n < \|\mathbf{y}\|, 0 \leq y_n \leq 15\}$$

▶ **Definice 1.10** (Hex-prefix kódování [2, příloha C] (*upraveno*)). *Funkce HP kóduje posloupnost nibblů a dodatečnou binární informaci t:*

$$\forall \mathbf{y} \in \mathbb{Y}, \forall t \in \{0,1\} \colon \mathtt{HP}(\mathbf{y}, t) \coloneqq \begin{cases} [16f(t),\ 16y_0 + y_1,\ 16y_2 + y_3,\ \ldots] & \textit{pro } \|\mathbf{y}\| \textit{ sudé} \\ [16(f(t)+1) + y_0,\ 16y_1 + y_2,\ 16y_3 + y_4,\ \ldots] & \textit{pro } \|\mathbf{y}\| \textit{ liché} \end{cases}$$

$$\forall t \in \{0,1\} \colon f(t) \coloneqq \begin{cases} 2 & t \neq 0 \\ 0 & t = 0 \end{cases}$$

1.4.3 Modifikovaný Merkle Patricia strom (trie)

Modifikovaný Merkle Patricia strom (trie) je datová struktura sloužící k ukládání dvojic klíč-hodnota, kde klíč i hodnota mohou být libovolná pole bytů [2, příloha D]. Pro účely formálního popisu je však vhodnější předpokládat, že klíče jsou pole nibblů.

Slovo *Merkle* v názvu poukazuje na fakt, že na uzly ve stromu je odkazováno pomocí hashů těchto uzlů. Dále se velmi často využívá hashe kořenového uzlu stromu, tzv. *merkle root*, který umožňuje efektivně ověřit, zda je daná dvojice klíč-hodnota uložena ve stromu či nikoliv. Proces ověření se nazývá *merkle proof*.

▶ **Definice 1.11** (Množina dvojic klíč-hodnota [2, příloha D] (*upraveno*)).

$$\mathbb{J} \coloneqq \{(\mathbf{k_0} \in \mathbb{Y},\ \mathbf{v_0} \in \mathbb{B}),\ (\mathbf{k_1} \in \mathbb{Y},\ \mathbf{v_1} \in \mathbb{B}),\ \ldots\}$$

Modifikovaný Merkle Patricia strom předpokládá existenci tří druhů uzlů:

- **List (leaf):** List je optimalizační uzel, ve kterém je uložen seznam obsahující dva prvky. První prvek je hex-prefix kódované pole zbývajících nibblů, které nebyly popsány průchodem stromu až k tomuto uzlu. Tento uzel tedy popisuje právě jednu dvojici klíč-hodnota. Typ uzlu je označován číslem 1, což je zároveň druhý argument funkce `HP` (hex-prefix). Druhým prvkem seznamu je samotná hodnota dvojice klíč-hodnota ve formě pole bytů.
- **Rozšíření (extension):** Rozšíření je optimalizační uzel, ve kterém je uložen seznam obsahující dva prvky. První prvek je hex-prefix kódované pole nibblů, pro které společné pro všechny klíče v podstromu. Typ uzlu je označován číslem 0, což je zároveň druhý argument funkce `HP` (hex-prefix). Druhým prvkem dvojice je odkaz na další uzel.
- **Větev (branch):** Větev je uzel, ve kterém je uložen seznam o 17 prvcích. V prvních 16 prvcích jsou uloženy odkazy na následující uzly, kde pořadí odkazu na uzel reprezentuje jeden nibble klíče (16 hodnot kóduje právě jeden nibble). V posledním prvku seznamu může být uložena hodnota páru klíč-hodnota, kde klíč odpovídá průchodu od kořene po tento uzel. Pokud není v posledním prvku seznamu uložena hodnota, je zde uloženo prázdné pole[3].

Průchodem ve stromu vzniká zřetězení nibblů. Uzly typu list (leaf) a rozšíření (extension) mohou reprezentovat libovolně dlouhé pole nibblů. Uzel typu větev (branch) při průchodu reprezentuje právě jeden nibble. Uzly větev (branch) a list (leaf) navíc mohou ukládat až jednu hodnotu z dvojice klíč-hodnota, kde klíč odpovídá zřetězení polí nibblů při průchodu od kořene stromu k danému uzlu.

Hodnota odkazu na následující uzel závisí na několika faktorech:

- Pokud pro podstrom, který by následující uzel reprezentoval, je množina dvojic klíč-hodnota k uložení prázdná, hodnotou odkazu je prázdné pole[3].
- Pokud délka pole, vzniklého serializací hodnoty následujícího uzlu pomocí RLP, je menší než 32, hodnotou odkazu na následující uzel je přímo hodnota daného následujícího uzlu (namísto odkazu na něj). Jedná se o optimalizační techniku.
- V ostatních případech je hodnotou odkazu na následující uzel Keccak-256 hash pole bytů, které vznikne kódování hodnoty daného následujícího uzlu funkcí RLP.

▶ **Definice 1.12** (Hodnota odkazu na následující uzel v modifikovaném Merkle Patricia stromu [2, příloha D] (*upraveno*))**.** *Jestliže $\mathbb{J}$ reprezentuje množinu párů klíč-hodnota, které je potřeba uložit v následujícím uzlu, i reprezentuje délku zřetězeného pole nibblů vzniklého průchodem od kořene stromu k následujícímu uzlu, $c(\mathbb{J}, i)$ reprezentuje hodnotu následujícího uzlu a* `KEC` *představuje hashovací funkci Keccak-256, pak hodnotu odkazu na následující odkaz lze definovat funkcí n:*

$$n(\mathbb{J}, i) \coloneqq \begin{cases} [\,] \in \mathbb{B} & \textit{jestliže } \mathbb{J} = \emptyset \\ c(\mathbb{J}, i) & \textit{jestliže } \|\texttt{RLP}(c(\mathbb{J}, i))\| < 32 \\ \texttt{KEC}(\texttt{RLP}(c(\mathbb{J}, i))) & \textit{v ostatních případech} \end{cases}$$

V následující definici se využívá zápisu $\mathbf{k}[start..stop]$, který reprezentuje podpole (podposloupnost ve formě pole) od indexu $start$ (včetně) po index $stop$ (včetně).

▶ **Definice 1.13** (Hodnota uzlu v modifikovaném Merkle Patricia stromu [2, příloha D] (*upraveno*))**.** *Jestliže $\mathbb{J}$ reprezentuje množinu párů klíč-hodnota, které je potřeba uložit v podstromu, kde daný uzel je kořenem podstromu, i reprezentuje délku zřetězeného pole nibblů vzniklého průchodem od kořene stromu k danému uzlu, pak hodnotu tohoto uzlu lze definovat funkcí c:*

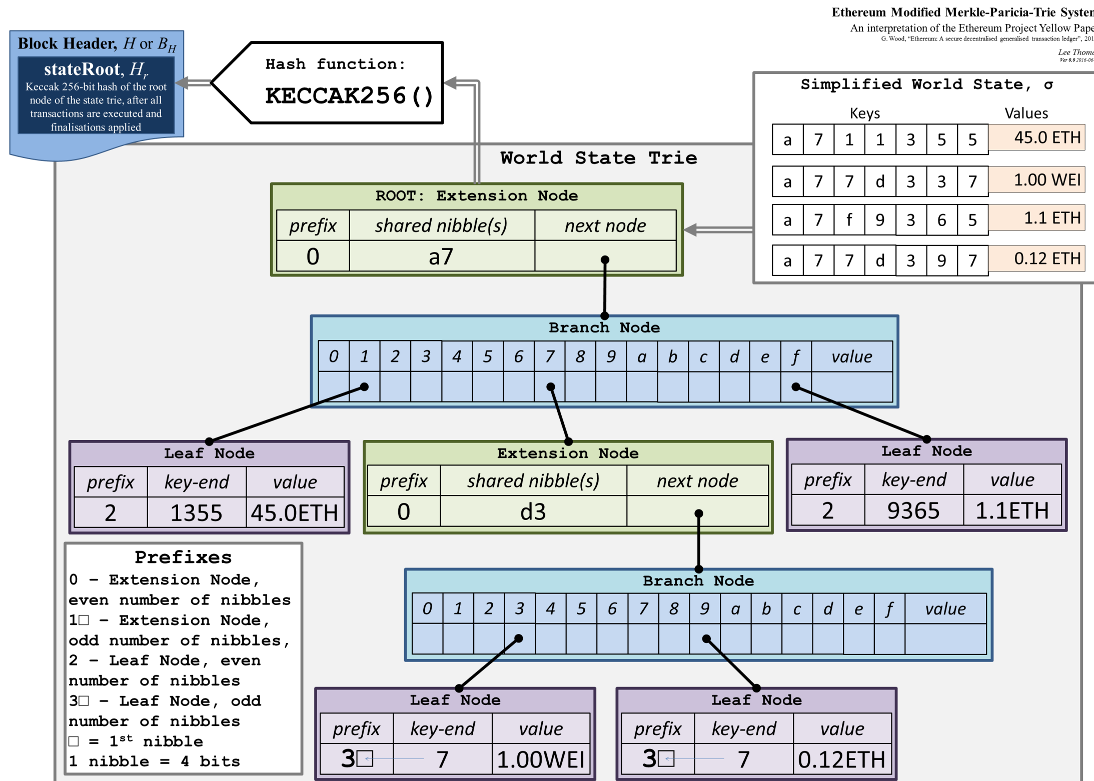
$$c(\mathbb{J}, i) \coloneqq \begin{cases} (\texttt{HP}(\mathbf{k}[i..(\|\mathbf{k}\|-1]), 1), \mathbf{v}) & \textit{jestliže } \mathbb{J} = \{(\mathbf{k},\ \mathbf{v})\}\ (\textit{tj. } \|\mathbb{J}\| = 1) \\ (\texttt{HP}(\mathbf{k}[i..(j-1)], 0), n(\mathbb{J}, j)) & \textit{jestliže } i \neq j, \\ & j = max\{x \colon \exists \mathbf{y} \colon \|\mathbf{y}\| = x \wedge \forall (\mathbf{k},\ \mathbf{v}) \in \mathbb{J} \colon \mathbf{k}[0..(x-1)] = \mathbf{y}\} \\ (u(0), u(1), \ldots, u(15), \mathbf{v}) & \textit{v ostatních případech}, \\ & u(j) \coloneqq n(\{(\mathbf{k},\ \mathbf{v}) \colon (\mathbf{k},\ \mathbf{v}) \in \mathbb{J} \wedge k_i = j\}, i+1), \\ & \mathbf{v} = \begin{cases} \mathbf{v} & \textit{jestliže } \exists (\mathbf{k},\ \mathbf{v}) \in \mathbb{J} \colon \|\mathbf{k}\| = i \\ [\,] \in \mathbb{B} & \textit{v ostatních případech} \end{cases} \end{cases}$$

▶ **Definice 1.14** (Merkle root [2, příloha D])**.** *Merkle root (v Yellowpaperu označovaný jako* $\texttt{TRIE}(\mathbb{J})$*) modifikovaného Merkle Patricia stromu, ve kterém jsou uložené dvojice klíč-hodnota z množiny $\mathbb{J}$ a kde* `KEC` *reprezentuje hashovací funkci Keccak-256, lze definovat následovně:*

$$\texttt{TRIE}(\mathbb{J}) \coloneqq \texttt{KEC}(\texttt{RLP}(c(\mathbb{J}, 0)))$$

[3] Yellowpaper bohužel v době psaní této práce nebyl zcela důsledný a z popisu nebylo patrné, zda se jedná o prázdné pole nebo prázdný seznam. Autor tohoto textu navrhl úpravu Yellowpaperu, která byla následně schválena a začleněna [21]. Tvrzení, že se jedná o prázdné pole, bylo ověřeno na základě dvou různých implementací modifikovaného Merkle Patricia stromu [22][23].

Obrázek 1.2 znázorňuje modifikovaný Merkle Patricia strom. Oproti popisu v předcházejících odstavcích je zde několik zjednodušení. V hodnotách uzlů je zde oddělený prefix (jako první nibble) od ostatních nibblů. Hodnoty dvojic klíč-hodnota jsou zde jako desetinná čísla reprezentující množství Etheru. Formálně by tyto hodnoty byly kódovány do bytových polí. Obrázek nepopisuje způsob uložení odkazů mezi uzly.



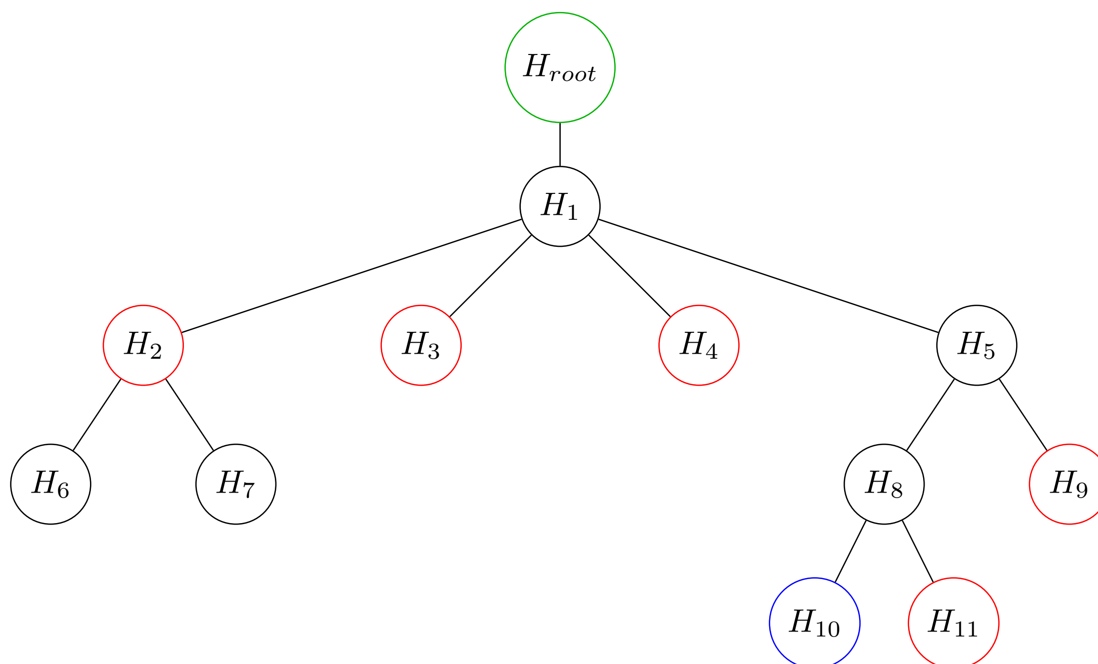
Obrázek 1.2 Zjednodušené znázornění modifikovaného Merkle Patricia stromu (trie) [24]

1.4.4 Merkle proof

V praxi může být pro každý uzel Ethereum sítě nepraktické ukládat si kompletní stav EVM. Toto je případ tzv. *light node* uzlů sítě, které ukládají pouze hlavičky bloků, což má za důsledek úsporu řádově stovek gigabytů paměti [25]. Tyto uzly nemají přístup k transakcím provedeným na blockchainu a neudržují si tedy ani globální stav EVM. Pro ověření, že daná dvojice klíč-hodnota se skutečně nachází ve stavu EVM, lze využít algoritmu *merkle proof* [26].

Součástí hlaviček bloků je mimo jiné *merkle root* stavu EVM. Jeho hodnotu lze pokládat za správnou na základě proof-of-work [27] (případně proof-of-stake [28]) konceptu. Pro ověření, že daná dvojice klíč-hodnota je uložena ve stavu EVM v daném bloku, stačí společně s hlavičkou bloku dodat informace o uzlech vedoucích od kořene stromu k uzlu, kde je pár klíč-hodnota uložen, a *merkle root* všech podstromů, které jsou pro konstrukci *merkle root* celého stromu potřeba.

Na obrázku 1.3 H_{root} představuje *merkle root* stromu – je to hodnota, které lze důvěřovat. Předpokládá se, že pár klíč-hodnota, jehož přítomnost prokazatel dokazuje, je uložen v uzlu s hashem H_{10}. Společně s tímto párem a informacemi z uzlů H_1, H_5, H_8 prokazatel dodá červeně označené hashe uzlů H_2, H_3, H_4, H_9 a H_{11}. Ze znalosti těchto hodnot může ověřovatel skutečně ověřit, že daný pár klíč-hodnota je uložen v modifikovaném Merkle Patricia stromu.



Obrázek 1.3 Příklad aplikace algoritmu *merkle proof*

1.4.5 Stav EVM

Stav EVM, nebo také globální stav (world state), je mapování mezi adresou Ethereum sítě (což je 160-bitová hodnota) a stavem účtu, který odpovídá dané adrese [2, kap. 4.1]. V blockchainu nejsou tyto stavy přímo uloženy (vytváří se postupnou aplikací jednotlivých transakcí, které na blockchainu uložené jsou), ale předpokládá se, že jsou na straně klienta Ethereum sítě ukládány pomocí modifikovaného Merkle Patricia stromu ve formě klíč-hodnota, kde klíč i hodnota jsou libovolná pole bytů. Toto úložiště Yellowpaper nazývá *stavová databáze* (state database). Na blockchainu je pak v každém bloku uložen vždy pouze *merkle root* globálního stavu EVM. Stav účtu, který odpovídá 160-bitové adrese, obsahuje tyto čtyři položky:

- **nonce:** Přirozené číslo popisující počet transakcí odeslaných z daného účtu. V případě účtu, který obsahuje spustitelný EVM kód, toto číslo reprezentuje počet vytvořených smart kontraktů pomocí tohoto účtu.
- **balance:** Přirozené číslo reprezentující počet Wei vlastněných tímto účtem.
- **storageRoot:** *Merkle root* modifikovaného Merkle Patricia stromu, ve kterém jsou uloženy dvojice ve formě klíč-hodnota, kde klíč i hodnota jsou 256-bitová čísla.
- **codeHash:** Keccak-256 hash spustitelného EVM kódu (bytekódu) patřícího danému účtu. Samotný kód (ve formě pole bytů) je uložen ve *stavové databázi*, kde klíčem je právě tento hash. EVM kód účtu lze vytvořit na základě transakcí uložených na blockchainu.

Jestliže **codeHash** odpovídá Keccak-256 hashi prázdného pole bytů, jedná se o tzv. *simple account* (podle Yellowpaperu). Často se tomuto typu účtu také říká *externally owned account* nebo jednoduše – účet, který není smart kontrakt. V opačném případě, kdy **codeHash** neodpovídá Keccak-256 hashi prázdného pole bytů, se totiž jedná o smart kontrakt. Výjimku z tohoto pravidla tvoří předkompilované (precompiled) smart kontrakty.

Tabulka 1.2 Předkompilované (precompiled) smart kontrakty

Adresa	Implementovaná funkcionalita	EIP
1	Obnovení (získání) veřejného klíče z ECDSA podpisu	–
2	SHA-256 hashovací funkce	–
3	RIPEMD-160 hashovací funkce	–
4	Výstupem jsou data na vstupu (funkce identity)	–
5	Modulární umocňování, kde $0^0 := 1$ a $x \ mod \ 0 := 0$ pro všechna x	EIP-198 [29]
6	Sčítání bodů na eliptické křivce *alt_bn128*	EIP-196 [30]
7	Násobení bodu eliptické křivky *alt_bn128* konstantou	EIP-196 [30]
8	Operace *optimal ate pairing check* [31] na eliptické křivce *alt_bn128*	EIP-197 [32]
9	BLAKE2b hashovací funkce	EIP-152 [33]

1.4.6 Předkompilované (precompiled) smart kontrakty

Předkompilované smart kontrakty jsou uloženy na speciálním typu účtů, který obvykle nemá přiřazený spustitelný EVM kód, jako je tomu u běžných smart kontraktů [2, příloha E]. Přesto je lze volat (spouštět) jako jakýkoliv jiný smart kontrakt. Tyto kontrakty obvykle implementují funkcionality, které by bylo pomocí EVM kódu nemožné nebo velmi neefektivní implementovat. Zároveň by ale nebylo vhodné tyto funkcionality zavádět jako samostatné EVM instrukce. Tabulka 1.2 popisuje všechny předkompilované kontrakty v době psaní této práce. První čtyři kontrakty v tabulce byly na blockchainu přítomné již od první verze Etherea.

1.4.7 Transakce

Transakce je kryptograficky podepsaný požadavek z účtu, který není smart kontrakt [2, kap. 4.2]. Předpokladem je, že tvůrce transakce je člověk, který využívá softwarových nástrojů pro vytvoření transakce.

EIP-2718 [34] zavádí speciální typ transakce, která může sloužit jako obálka pro nové typy transakcí vytvořené v budoucnu. Podle tohoto EIP existují dva druhy transakcí – původní (typ 0) a transakce podle EIP-2930 [35] (typ 1). Nezávisle na EIP-2718 se mohou transakce ještě dělit do dvou skupin. První skupinu představují transakce, které vytvářejí nové smart kontrakty na chainu. Druhá skupina je tvořena transakcemi, které volají (*message-call* podle Yellowpaperu) již nasazené kontrakty na síti. Všechny transakce se skládají z těchto dat:

- **type:** Hodnota v rozsahu 0 až 0x7f popisující typ transakce podle EIP-2718.
- **nonce:** Přirozené číslo, které značí počet transakcí odeslaných z účtu, který tuto transakci vytvořil.
- **gasPrice:** Přirozené číslo udávající počet Wei, které je odesílatel transakce ochotný zaplatit za jednotku Gasu.
- **gasLimit:** Přirozené číslo značící maximální množství Gasu, který může být použit při zpracovávání celé transakce. Tento Gas je rezervován již při čtení hlavičky transakce, před začátkem zpracovávání transakce. V průběhu transakce tento limit již nemůže být navýšen.

- **to:** Adresa (ve formě 160-bitového čísla) uvádějící příjemce transakce. V případě transakce vytvářející nový kontrakt je tato položka prázdná (prázdné pole bytů).
- **value:** Přirozené číslo udávající množství Wei, které mají být převedeny od odesílatele transakce k příjemci transakce.
- **r, s:** Čísla související s podpisem transakce, díky kterým je možné získat adresu odesílatele transakce.

Transakce popsané v EIP-2930 (typ 1 podle EIP-2718) navíc obsahují tyto dodatečné položky:

- **accessList:** Seznam dvojic, kde první prvek dvojice je adresa Ethereum účtu. Druhý prvek dvojice je seznam klíčů, pomocí kterých bude kód v transakci přistupovat ke *storage* úložišti daného Ethereum účtu. Jedná se o optimalizaci, díky které zpracovávatelé transakce předem znají některé přístupy do paměti, což vede k nižším cenám (v Gasu) těchto přístupů.
- **chainId:** ID chainu, do kterého byla tato transakce odeslána. Jedná se o ochranu proti replay útoku, kdy by útočník mohl transakci z jiného chainu (například testovacího) odeslat v hlavním chainu s validním kryptografickým podpisem.
- **yParity:** Bit popisující paritu y-ové souřadnice bodu na eliptické křivce *secp256k1n*. Pomocí této křivky se kryptograficky podepisují transakce.

Původní typ transakcí (typ 0 podle EIP-2718) položku **accessList** nepodporuje. Paritní bit **yParity** a volitelně také **chainId** jsou kódovány do jediné hodnoty pomocí přepočetních vzorců.

Všechny transakce vytvářející nové kontrakty navíc obsahují libovolně velkou položku **init**, ve které je EVM bytekód, jehož provedením vzniká EVM bytekód, který je přiřazen nově vzniklému kontraktu a který v budoucnu obsluhuje všechna volání tohoto kontraktu. Kód **init** se provádí pouze jednou při vytváření kontraktu.

Naopak transakce, které nové kontrakty nevytváří, ale volají již existující kontrakty, obsahují libovolně dlouhé pole bytů značené **data**. Tato data slouží jako argument při volání kontraktu (*message-call* podle Yellowpaperu).

1.4.7.1 Potvrzení transakcí

Potvrzení (receipts) transakcí jsou data uložená v blocích blockchainu, která zaznamenávají některé informace o vykonaných transakcích [2, kap. 4.3.1]. Jedná se o pětici s následujícími hodnotami:

- typ transakce podle EIP-2718,
- stavový kód transakce, kde hodnota 1 značí úspěch a hodnota 0 neúspěch [36],
- celkové množství použitého Gasu,
- upravený *Bloom filter* [37] vytvořený z logovacích záznamů uskutečněných v dané transakci,
- logovací záznamy uskutečněné v dané transakci.

Logovací záznam je trojice obsahující:

- adresu, která logovací záznam vytvořila,
- seznam štítků (topics), 32-bytových hodnot, rozlišujících typ logovacího záznamu,
- samotnou zprávu logovacího záznamu ve formě posloupnosti bytů.

1.4.8 Bloky

Bloky jsou základní stavební jednotkou blockchainu [2, kap. 4.3]. Kromě hlavičky obsahují seznam transakcí, které jsou v bloku zpracované, a seznam hlaviček ommer[4] bloků. Ommer bloky jsou bloky, jejichž předchozí (otcovský) blok je shodný s otcovským blokem otcovského bloku aktuálního bloku a zároveň se nejedná o otcovský blok aktuálního bloku – tedy otcovský blok aktuálního bloku není ommer. Jedná se především o případy, kdy vedle otcovského bloku vznikl ještě jeden blok v důsledku vytvoření bloků v téměř stejný čas. Díky tomuto seznamu jsou drobně odměňováni i těžaři, jejichž větev blockchainu není dále rozvíjena. Hlavička bloku obsahuje následující položky:

- **parentHash:** Keccak-256 hash hlavičky předchozího bloku.
- **ommersHash:** Keccak-256 hash seznamu ommer bloků připojeného k tomuto bloku.
- **beneficiary:** Adresa, na kterou jsou převedeny všechny poplatky (ve Wei) za zpracování transakcí v tomto bloku.
- **stateRoot:** *Merkle root* stavové databáze (state database).
- **transactionsRoot:** *Merkle root* modifikovaného Merkle Patricia stromu, ve kterém jsou uloženy všechny transakce tohoto bloku.
- **receiptsRoot:** *Merkle root* modifikovaného Merkle Patricia stromu, ve kterém jsou uloženy potvrzení všech transakcí tohoto bloku.
- **logsBloom**: Upravený *Bloom filter* [37], v němž jsou zachyceny některé informace o logovacích zprávách, které jsou zapsány v potvrzeních transakcí tohoto bloku.
- **difficulty:** Přirozené číslo popisující složitost vytvoření tohoto bloku. Jeho hodnotu je možné určit ze složitosti předchozího bloku a z unixového času – položky **timestamp** v hlavičce aktuálního bloku.
- **number:** Přirozené číslo reprezentující pořadí bloku v blockchainu. První blok, *genesis* blok, byl očíslován číslem nula.
- **gasLimit:** Maximální množství Gasu, který může být utracen za transakce v tomto bloku. Tato hodnota nepřímo limituje množství transakcí v bloku.
- **gasUsed:** Skutečné množství použitého Gasu pro zpracování všech transakcí v tomto bloku.
- **timestamp:** Unixový čas popisující počet uplynutých sekund od 00:00:00 1. ledna 1970 po okamžik, kdy byl tento blok vytvořen.
- **extraData:** Libovolná data ve formě pole bytů, jehož velikost však nesmí překročit 32 bytů.
- **mixHash:** 256-bitový hash, který společně s položkou **nonce** prokazuje, že tvůrce bloku vynaložil dostatek výpočetní síly pro vytvoření tohoto bloku.
- **nonce:** 64-bitové číslo, které společně s položkou **mixHash** prokazuje, že tvůrce bloku vynaložil dostatek výpočetní síly pro vytvoření tohoto bloku.

[4]Ommer je netradiční anglické slovo vyjadřující právě jednoho člena z dvojice (teta, strýc).

1.4.9 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) je virtuální výpočetní stroj, který je turingovsky kompletní[5] [2, kap. 9]. EVM disponuje instrukční sadou, kde vykonání každé instrukce stojí určité množství Gasu [38]. Tato cena v Gasu může být fixní, ale může se také odvíjet od argumentů instrukce nebo obecně kontextu, v jakém je instrukce vykonávána. Mimo to je Gas také účtován za vytvoření nového kontraktu, volání jiného kontraktu (*message-call*) a používání datového úložiště s názvem *memory*. Nativní jednotkou EVM pro výpočty je 256-bitové slovo. Tato velikost slova vychází například z toho, že se v Yellowpaperu často pracuje s Keccak-256 hashovací funkcí. 256-bitová slova jsou také vhodná pro výpočty nad eliptickými křivkami.

1.4.9.1 Zásobník

Při exekuci EVM pracuje s několika různými úložišti dat. Jedním z nich je zásobník [2, kap. 9.4.1][38]. Zásobník obsahuje 256-bitová slova, kterých může na zásobníku být maximálně 1024. Se zásobníkem pracuje většina instrukcí. Na vrcholu zásobníku jsou očekávané vstupní argumenty instrukcí. Tyto hodnoty jsou při zpracování instrukce ze zásobníku odebrány. Na vrchol zásobníku pak instrukce zapisují případné výstupní hodnoty. Pro explicitní manipulaci se zásobníkem slouží následující instrukce [2, příloha H][38]:

- `POP`: Instrukce ze zásobníku odebírá jedno slovo. Toto slovo není výpočetně využité.
- Instrukce typu `PUSH`: Jedná se celkem o 32 instrukcí se značením `PUSHX` (například `PUSH10`). Tyto instrukce (jako jediné) nejsou v bytekódu reprezentované jediným bytem, jelikož součástí jejich bytové reprezentace je (kromě bytu identifikujícího instrukci) neprázdná posloupnost bytů odpovídající délky `X`. Z této posloupnosti je vytvořeno 256-bitové slovo (byty jsou umisťované od nejméně významné pozice), které je vloženo na vrchol zásobníku.
- Instrukce typu `DUP`: 16 instrukcí značených `DUPX`, které naleznou slovo na pozici `X` od vrcholu zásobníku a toto slovo vloží (zduplikují) na vrchol zásobníku. Instrukce `DUP1` duplikuje nejvrchnější slovo na zásobníku, instrukce `DUP2` duplikuje druhé nejvrchnější slovo a tak dále.
- Instrukce typu `SWAP`: 16 instrukcí se značením `SWAPX`, které na zásobníku prohazují slovo na pozici 1 (nejvrchnější slovo) se slovem na pozici `X + 1`.

1.4.9.2 Memory segment

Memory segment je využíván jako dočasné úložiště při vykonávání EVM bytekódu [2, kap. 9.4.1][38]. Zápis do tohoto segmentu neovlivňuje stav blockchainu. Segment je realizován jako posloupnost bytů adresovatelná pomocí 256-bitového slova. Ve výchozím stavu je obsah *memory* segmentu vyplněn nulami. Během vykonávání EVM instrukcí je udržován nejvyšší offset, pomocí kterého bylo do *memory* segmentu přistupováno. Pokud instrukce přistupuje za tento offset, je segment expandován v násobcích 32 bytů. Expanze paměti probíhá lineárně od offsetu nula a je zpoplatněna Gasem, což motivuje k používání co nejmenšího množství paměti. Za předpokladu, že do *memory* segmentu ještě nebylo přistupováno, je cena v jednotkách Gasu za expanzi o n 32-bytových úseků určena jako $\left\lfloor \frac{n^2}{512} \right\rfloor + 3n$. S *memory* segmentem pracují tyto instrukce [2, příloha H][38]:

- `MLOAD`: Instrukce na vrcholu zásobníku očekává offset do *memory* segmentu. Z tohoto offsetu čte 256-bitové slovo, které vkládá na vrchol zásobníku.
- `MSTORE`: Instrukce na vrcholu zásobníku očekává offset a 256-bitovou hodnotu. Tato hodnota je zapsána do *memory* segmentu na daný offset.

[5] Yellowpaper používá pojem *quasi*-Turing-complete, čímž je zdůrazněno, že množství výpočetního výkonu je limitováno parametrem, kterým je Gas.

- `MSTORE8`: Instrukce na vrcholu zásobníku očekává offset a 256-bitovou hodnotu. Z této hodnoty je použit pouze nejméně významný byte, které je zapsán do *memory* segmentu na daný offset.
- `MSIZE`: Instrukce na vrchol zásobníku vkládá číslo popisující množství expandované paměti *memory* segmentu. Toto číslo musí být násobkem čísla 32.

1.4.9.3 Code segment

Dalším paměťovým segmentem je *code* segment, který vzniká při vytváření smart kontraktu, kdy je do něj uložena posloupnost bytů reprezentující EVM kód (instrukce) [2, kap. 9.3][38]. Při vykonávání EVM kódu je *code* segment určen pouze pro čtení. V tomto segmentu je uložen spustitelný EVM bytekód kontraktu. Pro práci s *code* segmentem existuje několik instrukcí [2, příloha H][38]:

- `CODESIZE`: Instrukce na vrchol zásobníku vkládá velikost *code* segmentu účtu (adresy kontraktu), jehož kód je právě prováděn.
- `CODECOPY`: Instrukce na vrcholu zásobníku očekává tři argumenty: offset od začátku *code* segmentu, offset od začátku *memory* segmentu a délku. Instrukce z aktuálního *code* segmentu od daného offsetu kopíruje bytekód o dané délce do *memory* segmentu na daný offset. Kopírováním může dojít k expanzi *memory* segmentu.
- `EXTCODESIZE`: Tato instrukce funguje obdobně jako `CODESIZE`, ale na zásobníku očekává adresu kontraktu, jehož délku bytekódu vkládá na vrchol zásobníku.
- `EXTCODECOPY`: Tato instrukce funguje obdobně jako `CODECOPY`, ale na zásobníku navíc očekává adresu kontraktu, jehož bytekód kopíruje do *memory* segmentu. Kopírováním může dojít k expanzi *memory* segmentu.
- `EXTCODEHASH`: Instrukce na vrcholu zásobníku očekává adresu kontraktu a vkládá na něj Keccak-256 hash bytekódu daného kontraktu.

1.4.9.4 Storage segment

Storage segment je permanentní datové úložiště kontraktu, do kterého se ukládají dvojice klíč-hodnota, kde klíč i hodnota jsou 256-bitová slova [2, kap. 9.1][38]. Předpokládá se, že tyto dvojice se ukládají do modifikovaného Merkle Patricia stromu, jelikož je pro každý účet na blockchainu uložena položka **storageRoot**, která reprezentuje *merkle root* tohoto stromu. Jestliže do *storage* segmentu zatím nebyla uložena data s daným klíčem, výchozí hodnota čtená pomocí tohoto klíče je nula. Se *storage* segmentem operují tyto instrukce [2, příloha H][38]:

- `SLOAD`: Instrukce na vrcholu zásobníku očekává klíč a vkládá na něj hodnotu ze *storage* segmentu uloženou pod tímto klíčem.
- `SSTORE`: Instrukce na vrcholu zásobníku očekává dvojici klíč-hodnota kterou ukládá do *storage* segmentu.

1.4.9.5 Call data segment

Call data je segment podobný *memory* segmentu, jelikož je adresovatelný pomocí 256-bitového slova a čtení za hranice segmentu vrací nulové byty [2, kap. 8][38]. Rozdílem je, že z tohoto segmentu je možné pouze číst. Také nedochází k expanzi paměti. Tento segment vzniká při odeslání transakce (položka **data** transakce) nebo při volání jiného kontraktu. Ve druhém případě tento segment vzniká pomocí některé z instrukcí, které implementují volání kontraktu, a je vytvořen

jako kopie *memory* segmentu od určitého offsetu a s danou délkou. S *call data* segmentem interagují tyto tři instrukce [2, příloha H][38]:

- `CALLDATALOAD`: Instrukce na vrcholu zásobníku očekává 256-bitové slovo jako offset do *call data* segmentu. Z tohoto offsetu čte 256-bitové slovo, které vkládá na vrchol zásobníku.
- `CALLDATASIZE`: Instrukce na vrcholu zásobníku vkládá číslo popisující velikost *call data* segmentu.
- `CALLDATACOPY`: Instrukce na vrcholu zásobníku očekává tři argumenty: offset od začátku *memory* segmentu, offset od začátku *call data* segmentu a délku. Instrukce z aktuálního *call data* segmentu od daného offsetu kopíruje posloupnost bytů o dané délce do *memory* segmentu na daný offset. Kopírováním může dojít k expanzi *memory* segmentu.

1.4.9.6 Return data segment

Return data segment vzniká při návratu z volaného kontraktu [2, kap. 9.4][38]. Tento návrat může být standardní nebo vyvolán chybou. Pomocí tohoto segmentu může volající kontrakt přistupovat k návratovým datům volaného kontraktu. Segment je adresovatelný 256-bitovým slovem, je určen pouze pro čtení a při přístupu za hranice segmentu je vyvolána chyba. Pro čtení z tohoto segmentu slouží tyto dvě instrukce [2, příloha H][38]:

- `RETURNDATASIZE`: Instrukce na vrcholu zásobníku vkládá číslo popisující velikost *return data* segmentu posledního volání kontraktu.
- `RETURNDATACOPY`: Instrukce na vrcholu zásobníku očekává tři argumenty: offset od začátku *memory* segmentu, offset od začátku *return data* segmentu a délku. Instrukce z *return data* segmentu odpovídajícího poslednímu volání kontraktu od daného offsetu kopíruje posloupnost bytů o dané délce do *memory* segmentu na daný offset. Kopírováním může dojít k expanzi *memory* segmentu.

1.4.9.7 Aritmetické a logické operace

EVM implementuje běžné aritmetické operace [2, příloha H][38]. Patří mezi ně operace pro součet, rozdíl, podíl, modulo (zbytek po celočíselném podílu), umocňování. Výsledky operací jsou na zásobník vkládány modulo 2^{256}. Pro operace sčítání a násobení navíc existuje druhá varianta instrukce, která jako třetí argument umožňuje specifikovat vlastní hodnotu pro operaci modulo. Operace dělení a modulo existují v bezznaménkové variantě a ve variantě, kdy jsou argumenty interpretovány pomocí dvojkového doplňku. Pokud je dělitel v operacích dělení a modulo nula, výsledek je nula. 0^0 je definováno jako 1. Mezi aritmetické operace lze také zařadit aritmetický posun doprava a znaménkové rozšíření čísla interpretovaného pomocí dvojkového doplňku.

Mezi logické operace patří logický součin, logický součet, operace xor, bitová negace, logický posun doprava a doleva. Také sem lze zařadit instrukci `BYTE`, která na vrcholu zásobníku očekává číslo v rozsahu 0 až 31 popisující pozici bytu (od nejvíce význameného bytu). Druhým argumentem na zásobníku je 256-bitové slovo, jehož byte na dané pozici je vložen na vrchol zásobníku.

1.4.9.8 Operace porovnání čísel

EVM implementuje běžné operace pro porovnání čísel: `<`, `>`, `==` [2, příloha H][38]. Operace *větší než* a *menší než* jsou v instrukční sadě přítomné ve dvou variantách. První varianta porovnává slova bezznaménkově, druhá varianta slova interpretuje pomocí dvojkového doplňku a následně porovnává. Dále je v instrukční sadě přítomna instrukce `ISZERO`, která vyhodnocuje, zda je na vrcholu zásobníku nulová hodnota, či nikoliv. V případě kladného výsledku porovnání je na vrchol zásobníku vložena hodnota `1`, v opačném případě je vložena hodnota `0`.

1.4.9.9 Kryptografické operace

Instrukční sada EVM podporuje jedinou kryptografickou operaci, kterou je instrukce `KECCAK256` implementující výpočet hashe pomocí Keccak-256 hashovací funkce [2, příloha H][38]. Tato instrukce byla dříve nesprávně pojmenovaná `SHA3` [12]. Instrukce na vrcholu zásobníku očekává offset od začátku *memory* segmentu a délku. Na vrchol zásobníku instrukce vkládá Keccak-256 hash posloupnosti bytů vymezené offsetem a délkou v *memory* segmentu. Instrukce může vést k expanzi paměti *memory* segmentu.

Jakékoliv jiné kryptografické operace nejsou instrukční sadou podporované. Je však možné využít předkompilovaných kontraktů, které implementují další hashovací funkce, obnovení veřejného klíče z ECDSA podpisu či operace na eliptické křivce *alt_bn128* [2, příloha E].

1.4.9.10 Skokové instrukce

Pro podporu `if` nebo `for` výrazů ve vyšším jazyce je vhodné, aby specifikace EVM podporovala skokové operace. Při vykonávání EVM bytekódu je udržována hodnota čítače, *program counteru*, která reprezentuje offset od začátku aktuálního *code* segmentu [2, kap. 9.4][38]. Hodnotu *program counteru* vkládá na vrchol zásobníku instrukce `PC`. Ve většině případů je *program counter* po zpracování instrukce inkrementovaný o jedničku. Výjimku tvoří `PUSH` instrukce, volání kontraktů a skokové instrukce. Mezi skokové instrukce patří [2, příloha H][38]:

- `JUMP`: Instrukce na vrcholu zásobníku očekává novou hodnotu *program counteru*.
- `JUMPI`: Instrukce na vrcholu zásobníku očekává novou hodnotu *program counteru* a 256-bitové slovo. Nová hodnota *program counteru* se aplikuje pouze v případě, že je druhý argument různý od nuly. Jestliže je druhý argument nulový, ke skoku nedochází a *program counter* se inkrementuje o jedničku.
- `JUMPDEST`: Tato instrukce samotná nemá význam a pouze inkrementuje *program counter* o jedničku. Pokud je však v případě předchozích dvou instrukcí skok proveden, musí nová hodnota *program counteru* odkazovat na `JUMPDEST` instrukci. V opačném případě dochází k vyvolání chyby.

1.4.9.11 Instrukce pro získání dat z blockchainu

Velkou skupinu EVM instrukcí tvoří instrukce, které na vrchol zásobníku zapisují externí data z blockchainu [2, příloha H][38]. Patří sem instrukce zjišťující informace o bloku, do kterého je právě zpracovávaná transakce začleněna. Instrukce na zásobník vkládají například unixový čas vytvoření bloku, číslo bloku, chain ID nebo hash bloku s daným číslem.

EVM během vykonávání instrukcí rozlišuje adresu odesílatele transakce, tu je možné získat instrukcí `ORIGIN`, a adresu, ze které byl aktuální kontrakt volán, tu na zásobník vkládá instrukce `CALLER`. Adresa získaná instrukcí `ORIGIN` je vždy účet bez EVM bytekódu (jedná se o *externally owned account*). Na začátku vykonávání EVM bytekódu transakce jsou obě adresy shodné. Adresa získaná instrukcí `CALLER` se může změnit v momentě, kdy je vykonaná jedna z instrukcí pro volání kontraktu.

1.4.9.12 Volání kontraktů (*message-call*)

Yellowpaper volání kontraktů nazývá *message-call*[2, kap. 8]. Obecně může být volání kontraktu iniciované transakcí nebo EVM bytekódem kontraktu. Kontrakt typicky volá jiný kontrakt (na jiné adrese), ale může volat i sám sebe.

Yellowpaper a jeho definice EVM nezná pojem *funkce kontraktu*. Ve vyšších programovacích jazycích jsou však kontrakty složené z volatelných funkcí, které mohou přijímat různé argumenty. Programovací jazyk Solidity toto řeší tak, že v *call data* segmentu očekává jako první

čtyři byty začátek hashe Keccak-256 signatury funkce, která se má volat [39]. Další byty v *call data* segmentu jsou pak jednotlivé argumenty funkce. Na začátku bytekódu každého kontraktu vytvořeného v Solidity je pak skoková tabulka, která na základě prvních čtyř bytů v *call data* segmentu skáče na místo v bytekódu, kde je požadovaná funkce implementovaná.

Pro volání kontraktů existuje více instrukcí [2, příloha H][38]:

- `CALL`: Instrukce na zásobníku očekává celkem sedm argumentů:
  - **gas:** Množství Gasu alokované pro vykonávání EVM bytekódu ve volaném kontraktu. Nevyužitý Gas je vrácen aktuálnímu kontextu.
  - **address:** Adresa smart kontraktu, jehož kód má být prováděn.
  - **value:** Množství Wei, které má být připsáno volanému kontraktu.
  - **argsOffset:** Offset do *memory* segmentu, odkud jsou data kopírována do *call data* segmentu volaného kontraktu.
  - **argsSize:** Délka posloupnosti bytů, která je kopírována z *memory* segmentu do *call data* segmentu volaného kontraktu.
  - **retOffset:** Offset do *memory* segmentu, kam mají být zkopírována data z *return data* segmentu volaného kontraktu po dokončení volání.
  - **retSize:** Délka posloupnosti bytů, která je po dokončení volání kopírována z *return data* segmentu volaného kontraktu do *memory* segmentu aktuálního kontaktu.

  Vykonáním instrukce vzniká nový kontext – nový zásobník, nový *memory* segment, *code* a *storage* segmenty odpovídají volanému kontraktu. *Call data* segment volaného segmentu vzniká kopií dat z daného offsetu o dané délce z *memory* segmentu volajícího kontraktu. *Program counter* je v novém kontextu nastaven na nulovou hodnotu – na první instrukci bytekódu. Adresu volajícího kontraktu je možné získat instrukcí `CALLER`. Instrukce `ORIGIN` stále vrací adresu uživatele, který vytvořil transakci.

  Volání kontraktu může být ukončeno standardní způsobem, nebo může být vyvoláno chybou. Jestliže volání skončí chybou, je na vrchol zásobníku volajícího kontextu zapsána hodnota `0`. V opačném případě je na vrchol zásobníku volajícího kontextu zapsána hodnota `1`.

- `CALLCODE`: Instrukce má téměř identické chování jako instrukce `CALL`. Očekává stejný počet argumentů se stejným významem. Rozdíl je v tom, že nově vytvořený kontext pro vykonávání kódu volaného kontraktu je vytvořen tak, jako by byl volán volající kontrakt (až na *code* segment). Toto především znamená, že *code* segment odpovídá volanému kontraktu a *storage* segment odpovídá volajícímu kontraktu. Volající kontrakt musí důvěřovat volanému kontraktu, jelikož umožňuje kódu volaného kontraktu upravovat svůj *storage* segment.

  Tato instrukce však není příliš využívaná, a dokonce existuje EIP-2488 [40], které navrhuje instrukci vyřadit. Instrukce totiž stejně jako `CALL` instrukce upravuje adresu získanou `CALLER` instrukcí, což je v praxi nepraktické. Z pohledu vyšších jazyků měla tato instrukce za cíl implementovat volání knihovních funkcí.

- `DELEGATECALL`: Tato instrukce vznikla jako náhrada za `CALLCODE` s opravenou sémantikou. Oproti `CALL` instrukci není přijímán argument **value**. Instrukce tedy očekává pouze šest argumentů, které mají stejný význam jako předcházející instrukce. Stejně jako `CALLCODE` tato instrukce využívá *code* segmentu volaného kontraktu, ale zachovává *storage* segment volajícího kontraktu. Rozdílem je, že adresa získaná `CALLER` instrukcí se nemění. Také není možné změnit hodnotu udávající množství Wei odeslaného odesílatelem transakce (argument **value** předchozích instrukcí).

- `STATICCALL`: Instrukce je podobná instrukci `CALL`, ale neumožňuje volanému bytekódu měnit stav blockchainu. Konkrétně není možné používat:

- instrukce pro vytváření nových kontraktů,
- instrukce pro zrušení existujících kontraktů,
- instrukce pro vytváření logovacích zpráv,
- `SSTORE` instrukci,
- `CALL` instrukci, jestliže je volaná s nenulovou hodnotou argumentu **value**[6].

Pro ukončení kontextu vykonávání (návrat z volaného kontraktu) existuje více instrukcí:

- `STOP`: Instrukce ukončuje vykonávání aktuálního kontextu standardním způsobem (nedochází k chybovému ukončení). Jestliže je instrukcemi `CALL`, `CALLCODE`, `DELEGATECALL` a `STATICCALL` volaná adresa *externally owned* účtu (který neobsahuje bytekód), chování odpovídá vykonání této instrukce. Ihned dochází k ukončení vykonávání volaného kontextu nechybovým způsobem.

- `RETURN`: Instrukce na vrcholu zásobníku očekává offset od začátku *memory* segmentu a délku. Instrukce ukončuje aktuální kontext a ve volaném kontextu vytváří *return data* segment s daty zkopírovanými z *memory* segmentu aktuálního segmentu od daného offsetu a s danou délkou.

- `REVERT`: Instrukce na vrcholu zásobníku očekává offset od začátku *memory* segmentu a délku. Instrukce ukončuje vykonávaný kontext chybovým způsobem a ve volaném kontextu vytváří *return data* segment s daty zkopírovanými z *memory* segmentu aktuálního segmentu od daného offsetu a s danou délkou. Všechny změny upravující stav blockchainu jsou vráceny zpět. Zbývající nevyužitý Gas je vrácený volajícímu kontextu.

  Chování popsané touto instrukcí je uplatněno i ve všech případech, kde během vykonávání EVM bytekódu dochází k chybově. Jedná se například o přístup mimo rozsah *return data* segmentu, situaci, kdy na zásobníku není dostatek hodnot pro vykonání instrukce, nebo když není dostatek alokovaného Gasu pro provedení aktuální instrukce.

- `INVALID`: Jedná se o instrukci přiřazenou všem hodnotám bytu, které nepopisují nějakou jinou instrukci. Instrukce se chová obdobně jako `REVERT` instrukce, ale zpracovává všechen alokovaný Gas. *Return data* segment volajícího kontextu je vytvořen prázdný.

1.4.9.13 Logovací zprávy

Logovací zprávy mohou být vytvořené EVM instrukcemi. Zprávy jsou pak v blockchainu viditelné v potvrzeních (receipts) instrukcí [kap. 4.3.1][2]. Zprávy může následně zpracovávat také software mimo blockchain [42]. Pro vytváření logovacích zpráv existuje pět instrukcí [2, příloha H][38]:

- Instrukce typu `LOG`: Jedná se celkem o 5 instrukcí se značením `LOGX`, kde se `X` pohybuje od 0 do 4 a značí počet štítků (topics), které jsou zprávě přiřazeny. Instrukce na zásobníku očekává offset od začátku *memory* segmentu, délku a variabilní počet (závisející na konkrétní instrukci) štítků reprezentovaných 256-bitovými slovy. Instrukce vytváří logovací zprávu vymezenou offsetem a délkou v *memory* segmentu s daným počtem štítků.

1.4.9.14 Vytváření a odstraňování kontraktů

Kontrakty mohou vznikat transakcemi nebo vykonáváním bytekódu již existujících kontraktů [2, kap. 7]. Ve druhém případě jsou za vytváření nových kontraktů zodpovědné tyto instrukce [2, příloha H][38]:

[6] Instrukci `CALLCODE` je možné ve `STATICCALL` kontextu používat i s nenulovou hodnotou argumentu **value** [41].

- `CREATE`: Instrukce na vrcholu zásobníku očekává tři argumenty: množství Wei, které má být odesláno na adresu nově vzniklého kontraktu, offset od začátku *memory* segmentu a délku. Nový kontrakt je vytvořen na adrese, která odpovídá nejnižším 160 bitům Keccak-256 hashe seznamu (adresa kontraktu, jehož bytekód je vykonáván; **nonce** účtu kontraktu, jehož bytekód je vykonáván)[7] zakódovaného pomocí RLP. *Code* segment nově vytvořeného kontraktu vzniká jako návratová hodnota vykonaného bytekódu z *memory* segmentu aktuálního kontextu od daného offsetu s danou délkou. Na vrchol zásobníku je vložena adresa vytvořeného kontraktu.

- `CREATE2`: Instrukce se chová obdobně jako `CREATE` instrukce, ale očekává dodatečný čtvrtý argument, *salt*, který se využívá při výpočtu adresy nově vytvořeného kontraktu. Způsob výpočtu adresy nového kontraktu je upravený tak, aby bylo možné adresu deterministicky určit ještě před vytvořením kontraktu. V případě instrukce `CREATE` je problém v tom, že adresa nového kontraktu záleží na **nonce** účtu, který nový kontrakt vytváří.

- `SELFDESTRUCT`: Instrukce zajišťuje, že je aktuální kontrakt označený pro odstranění po dokončení aktuální transakce. Množství Wei, které aktuální kontrakt vlastní, je odesláno na adresu, kterou instrukce očekává na vrcholu zásobníku. Segmenty *storage* a *code* odstraňovaného kontraktu jsou vymazány.

1.5 Těžba bloků

Velkou skupinou uzlů v Ethereum síti jsou tzv. těžaři (miners) [43]. Jedná se o uzly, které sbírají transakce a umisťují do svých mempoolů – seznamů transakcí, které ještě nebyly zpracovány a začleněny do bloků.

Jakmile daný uzel nashromáždí dostatek transakcí, začne vytvářet blok. Obvykle jsou transakce přidávané do bloku zvoleny tak, aby to bylo pro těžaře nejvýhodnější, tj. aby z poplatků ve formě Gasu jednotlivých transakcí plynul pro těžaře co největší výdělek.

Při vytváření bloku těžař postupně prochází jednotlivé transakce, které mají být do bloku umístěny. Každá transakce musí být ověřena, že je validní. Toto především zahrnuje kontrolu kryptografického podpisu transakce, kontrolu formátu transakce a ověření, že aplikací transakce (tedy případným voláním kontraktu) vznikne přechod do validního stavu blockchainu a že uživatel vlastní dostatek Etheru pro zaplacení transakce. Aplikací transakce těžař přepíše svůj lokální stav virtuálního stroje (EVM).

Před tím, než je blok broadcastován do sítě, musí těžař vynaložit jistý druh úsilí, který jej opravňuje k vytvoření bloku a rozeslání po síti. Bez tohoto úsilí (resp. důkazu o vynaložení úsilí) by blok nebyl přijat ostatními uzly sítě. Tímto způsobem se síť chrání proti některým útokům.

Je potřeba, aby se všichni uživatelé sítě jednomyslně shodovali na pořadí bloků a transakcí v nich. Také je potřeba, aby v řetězci byly přítomné pouze takové bloky, které splňují všemi akceptovaná pravidla popsaná v Yellowpaperu [2]. Jedním z těchto pravidel například je, že daný uživatel nemůže své prostředky (Ether) utratit dvakrát. Často je tento problém nazýván jako *double spending problem* [44]. Uzly sítě tato pravidla kontrolují a přijímají pouze bloky, které jsou v souladu s těmito pravidly. V momentě, kdy by se měl řetězec bloků rozvětvit, je všemi uznáván jako „správný“ řetězec ten s největším počtem bloků. Předpoklad je takový, že většina uzlů v síti je legitimní a dodržuje stanovená pravidla. Pokud tedy někdo vytvoří blok, který tato pravidla nesplňuje, většina tento blok nepřijme a řetězec bloků se začne větvit před místem, kde byl přidán neplatný blok. Protože je legitimních uzlů více než uzlů útočníka (většina uzlů je legitimní), validní větev blockchainu brzy získá mnohem více bloků než větev útočníka a větev útočníka se tedy efektivně stane slepou větví.

[7] Naopak v případě vytváření kontraktu pomocí transakce adresa i **nonce** odpovídají *externally owned* účtu, který transakci vytvořil.

Mechanizmus popsaný v předchozím odstavci vyžaduje několik předpokladů. Jedním z nich je, že vytváření nových bloků je netriviální. Pokud by vytváření bloků bylo jednoduché, mohl by útočník i s malými prostředky vytvořit řetězec bloků delší než zbytek legitimních uživatelů. Z tohoto důvodu je potřeba, aby při vytváření bloků těžař vynaložil určitý typ úsilí. Konceptů pro vynaložení tohoto úsilí je více. Obecně se těmto konceptům říká *consensus mechanisms* [45] – na jejich základě dochází ke shodě v síti o tom, jaký blok bude následovat.

Nejznámějším konceptem je proof-of-work [2, příloha J][27]. Ten spočítá v tom, že těžař může kontrolovat jednu hodnotu uvnitř bloku nazývanou **nonce**[8]. Na základě této položky a hlaviček bloků blockchainu je vypočítána hodnota **mixHash**, která je následně v hlavičce bloku také uložena. Díky této hodnotě je obtížné pro účely těžby bloků využít specializovaný hardware – Application Specific Integrated Circuit (ASIC). Cílem je nalezení takové hodnoty **nonce**, aby hash celého bloku měl menší hodnotu než je nějaká globálně uznávaná mez. Hodnota **nonce** nepřevídatelně mění výsledný hash bloku.

Motivací pro legitimní chování těžařů je financí odměna. Těžař vytvořením validního bloku má nárok na určité množství Etheru. Tímto způsobem vznikají nové Ethery. Narozdíl od Bitcoinu není počet Etherů v budoucnu nějak fixně omezen či limitován [46]. Kromě toho těžař získává odměnu ve formě poplatků od uživatelů – tvůrců transakcí. Pokud by těžař začlenil transakci, která není legitimní, nebo vytvořil jinak nevalidní blok, ostatní těžaři by jej nepřijali. Efektivně by těžař nedostal odměnu ve formě Etherů a poplatků v Gasu. Obdobně těžař nemá důvod vytvářet nový blok v řetězci s jiným blokem, který nepovažuje za validní. Lze předpokládat, že ostatní těžaři tento blok také nebudou pokládat za validní a těžař za vynaložené úsilí nezíská odměnu.

Logicky se nabízí možnost útoku na síť prostřednictvím tzv. 51% útoku [47]. Jedná se o stav, kdy útočník vlastní většinu výpočetní síly v síti a je tedy schopný sám vytvářet nejdelší řetězec bloků bez ohledu na ostatní těžaře v síti.

Další *consensus mechanism* je proof-of-stake [28]. V tomto případě se těžařům spíše říká validátoři. Jejich úkolem je také vytváření nových bloků a kontrola ostatních validátorů. Princip vytváření nových bloků je zde ale jiný. Nejde totiž o výpočetně náročnou operaci, jako je tomu v případě proof-of-work, ale o vlastnost vlastnictví platidla daného projektu (v tomto případě Etheru). Uživatel musí pro začlenění mezi validátory dočasně poskytnout (na dobu určitou či neurčitou) dané množství Etheru – tomuto procesu se říká *staking*. Tento Ether slouží jako důkaz o validátorově dobrých úmyslech. Pokud by validátor schvaloval nelegitimní bloky, mohl by mu být Ether, který poskytl, odebrán. Navíc by to mohlo poškodit jméno projektu, čímž by snižoval hodnotu svých vlastních mincí. V tomto konceptu je snažší stát se aktivním těžařem/validátorem. Validátoři pro vytváření nového bloku jsou voleni náhodně. Nedochází tedy k soutěži mezi validátory pomocí výpočetních prostředků. Pokud validátor není k dispozici pro vytvoření bloku (například je offline), může opět přijít o část svých Etherů. Toto částečně vylučuje možnost odstavení sítě v případě, že by validátoři byli nezodpovědní či účelově blokovali vytváření nových bloků.

Ethereum aktuálně využívá mechanizmu proof-of-work (stejně jako Bitcoin), ale je plánovaný přechod na proof-of-stake. Tento přechod se často označuje jako Ethereum 2.0 nebo *The Merge* [48].

[8]Tato hodnota **nonce** nesouvisí s položkou **nonce** účtu Ethereum blockchainu.

Kapitola 2
Solidity

Solidity je objektově orientovaný, vysokoúrovňový programovací jazyk určený pro implementaci smart kontraktů na Ethereum blockchainu [6]. Jazyk se stále velmi vyvíjí, což mimo jiné komplikuje vývoj nástrojů pracujících s tímto jazykem (nástroje pro statickou a dynamickou analýzu). Také kompilátor pro převod ze Solidity kódu do EVM bytekódu není tolik optimalizovaný, jako u jiných konvenčních jazyků, a velké množství jeho funkcionalit je stále v experimentálním režimu. Tato kapitola popisuje programovací jazyk Solidity ve verzi 0.8.13[1].

2.1 Popis jazyka

Syntaxe jazyka byla velmi ovlivněna jazykem C++ [49]. Dále při vzniku Solidity některé funkcionality vycházely z jazyka JavaScript. Část těchto funkcionalit byla odstraněna (například klíčové slovo `var`). Zůstala především syntaxe importů a způsob verzování, u kterých lze pozorovat podobnost s jazykem JavaScript. Z podpory pro vícenásobnou dědičnost či použití klíčového slova `super` lze usuzovat, že jazyk Solidity byl také ovlivněn jazykem Python. Kompletní gramatika Solidity je velmi dobře znázorněna v dokumentaci [50].

2.1.1 Doporučení pro formátování kódu

Stejně jako Python v PEP-8 [51] popisuje doporučované konvence pro formátování kódu, Solidity ve své dokumentaci také uvádí některá doporučení [52]. Jedná se například o tato pravidla:

- Text by měl být odsazován ve formě čtyř mezer za každou úroveň odsazení. Používání tabulátorů není doporučeno.
- Deklarační příkazy na nejvyšší úrovni v Solidity souboru by měly být vzájemně odděleny dvěma prázdnými řádky.
- Deklarace funkcí uvnitř kontraktu by měly být odděleny jedním prázdným řádkem s některými výjimkami, kdy je vhodnější prázdný řádek vypustit (například v situaci, kdy součástí deklarace funkcí není tělo s definicí funkce).
- Maximální délka řádku by se měla pohybovat v rozmezí 79 až 99 znaků.
- V případě rozdělení jednoho příkazu do více řádků by první prvek (argument) měl být na samostatném řádku (neměl by být na řádku společně s otevírací závorkou), každý prvek

[1] Jedná se o nejnovější verzi jazyka Solidity v době psaní této práce.

(argument) příkazu by měl být na samostatném řádku oddělen právě jednou úrovní odsazení od okolních řádků a uzavírací závorka příkazu by měla být také umístěna na samostatném řádku.

- Zdrojové soubory by měly být kódovány pomocí UTF-8[2] kódování.
- Importní příkazy by měly být umístěné na začátku souboru po `pragma` příkazech.

2.1.2 Datové typy

Jazyk Solidity je staticky typovaný a pro každou proměnnou tedy musí být známý její datový typ [53]. Narozdíl od některých jiných jazyků, každý datový typ v Solidity má svoji výchozí hodnotu, která odpovídá bytové reprezentaci ve formě posloupnosti nulových bytů. Toto odpovídá faktu, že zkompilovaný kód běží v EVM, kde přítupy do paměti mimo aktuální rozsah často vedou na implicitní rozšíření paměti o nulové byty. Podobně jako Python, Solidity rozlišuje předávání argumentů hodnotou a referencí.

2.1.2.1 Hodnotové typy

Hodnotové typy v paměti ukládají přímou hodnotu proměnné [53]. V případě předání argumentu hodnotového typu funkci se vždy vytváří kopie této proměnné.

Mezi hodnotové typy patří:

- Typ `bool` se svými logickými operátory. Výchozí hodnota tohoto typu je `false`.
- Znaménkové a bezznaménkové celočíselné typy. Celočíselné typy s největším rozsahem hodnot jsou typy `int` a `uint`, které odpovídají typům `int256`, respektive `uint256`. Číslo v datovém typu reprezentuje počet bitů určených k uložení čísla. Typy `int` a `uint` tedy velikostí odpovídají slovu EVM. Nejmenší celočíselné typy jsou typy `int8` a `uint8`. Solidity podporuje všechny celočíselné typy reprezentovatelné celočíselným počtem bytů v rozmezí 1 až 32. Existují tedy typy `int8`, `uint8`, `int16`, `uint16`, `int24`, `uint24` a tak dále. Celočíselné typy podporují celou řadu aritmeticko-logických operací, které přímo vycházejí z instrukční sady EVM. Od verze 0.8.0 jazyka Solidity je kontrolováno přetečení a podtečení při aritmetických operacích. Jestliže je detekováno přetečení nebo podtečení, dochází k *revert* sémantice. Kontrola přetečení a podtečení může být vypnuta vložením příkazů do `unchecked { ... }` bloku. Ve verzích nižších než 0.8.0 nebyly tyto kontroly prováděny a mohlo docházet k přetečení a podtečení při aritmetických výpočtech. Z tohoto důvodu vznikly v knihovně OpenZeppelin funkce pro bezpečné provádění aritmetických operací [54].
- Dokumentace jazyka Solidity zmiňuje datové typy `fixed` a `ufixed` pro reprezentaci čísel s pevnou desetinnou čárkou. Tyto typy však nejsou podporovány – je možné je deklarovat, ale není možné jim přiřazovat ani z nich získávat hodnotu.
- Speciální typ `address` reprezentuje adresu Ethereum sítě ve formě 20-bytového (160-bitového) čísla. Adresa může být explicitně převedena na typ kontraktu, který je uložen na dané adrese. Dále existuje datový typ `address payable`, který navíc poskytuje členské funkce pro převod Etheru na danou adresu. Tímto je možné v kódu rozlišovat adresy, kterým by neměl být zasílán Ether, od adres, kterým je možné v programu Ether zaslat.
- Pro každý kontrakt deklarovaný v jazyce Solidity existuje jeho odpovídající datový typ. Tento typ je v paměti reprezentován adresou kontraktu.

[2] V implementační části této práce bylo autorem textu zjištěno, že kompilátor Solidity nepřijímá zdrojové soubory kódované pomocí některých jiných běžně používaných kódování jako je například CP-1250.

- Hodnotového typu jsou také fixní posloupnosti bytů `bytes1`, `bytes2`, ..., `bytes32` o délce 1 až 32 bytů. Typy jsou podobné celočiselným datovým typům, ale nepodporují některé artimetické operace. Naopak je možnost indexace jednotlivých bytů posloupnosti.

  Dále jsou jako hodnotové typy vnímány literály:

- Literál reprezentující adresu Ethereum sítě je hexadecimální číslo složené z 39 až 41 číslic. Hexadecimální číslice `A` až `F` musí být v literálu ve správné posloupnosti velkých a malých písmen, aby adresa odpovídala kontrolnímu součtu, který je popsán v EIP-55 [55].
- Číselné literály mohou být zapsány v desítkové nebo šestnáctkové soustavě (s prefixem `0x`). Literály v osmičkové soustavě nejsou podporovány a použití úvodní nuly v číslech není povoleno. Také je možné použití vědeckého zápisu ve formě `<mantisa>e<exponent>` (například `5e2`). Stejně jako v jazyce Python je umožněno v číselných literálech používat znak podtržítka pro přehlednost.
- Textové literály jsou ohraničeny dvojicí jednoduchých nebo dvojitých uvozovek. Součástí textových literálů mohou být escape sekvence reprezentující bílé znaky nebo znak pomocí ASCII nebo UTF-8 kódování.

Výčtové typy (`enum`) také patří do hodnotových typů, neboť jsou jednotlivé položky výčtu reprezentovány čísly od 0 do 255. Hodnotovým typem jsou také funkce. Funkce je možné ukládat do proměnných či je předávat jako argumenty jiným funkcím.

2.1.2.2 Referenční typy

Referenční datové typy na uložená data ukazují pomocí reference [53]. Na jednu datovou položku tedy může odkazovat více proměnných. Při deklaraci proměnných referenčního typu je nutné znát segment, do kterého jsou data uložena. Těmito segmenty může být *memory* segment, *storage* segment nebo *call data* segment.

Mezi referenční typy patří:

- Pole libovolného datového typu, která jsou značena jako `T[]` v případě pole dynamické velikosti, `T[k]` v případě pole fixní velikosti a `T[][3]` v případě tří polí obsahujících pole dynamické velikosti. Prvky pole jsou indexovány od nuly. K přidávání či odebírání prvků na konci pole je možné využít členských funkcí `.push()`, `.push(value)` a `.pop()`.
- Speciální datové typy `bytes` a `string` jsou založené na polích. Datový typ `bytes` odpovídá typu `bytes1[]`, ale je v paměti efektivněji reprezentován. Datový typ `string` je téměř shodný s typem `bytes`, ale neumožňuje indexaci znaků ani zjišťování délky řetězce.

Literály polí jsou zapsány v hranatých závorkách, kde jsou jednotlivé prvky pole odděleny čárkami. Mezi referenční typy patří také struktury (`struct`), jejichž členské proměnné mohou být libovolného typu. Speciálním referenčním typem je typ `mapping`. Jedná se o datový typ, který reprezentuje slovník či hashovací tabulku. Součástí deklarace typu `mapping` je uvedení typu klíče a typu hodnoty. Proměnné typu `mapping` jsou deklarovány pomocí syntaxe:

```
mapping(KeyType => ValueType) variableName;
```

Pro každý možný klíč existuje výchozí hodnota, která odpovídá výchozí hodnotě daného typu. Typ `mapping` může být také vícenásobně vnořen, například:

```
mapping(address => mapping(address => uint)) map;
```

2.1.3 Struktura souborů

2.1.3.1 SPDX identifikátor licence

Solidity má přímou podporu pro uvádění licence ve zdrojových souborech [56]. Předpokládá se, že každý zdrojový soubor v Solidity začíná SPDX identifikátorem [57] licence (například v případě MIT licence `// SPDX-License-Identifier: MIT`). V případě closed source licence je možné využít speciální hodnoty `UNLICENSED`. Kompilátor konkrétní typ licence zadaný v identifikátoru dále nezkoumá (může být zadán i neplatný typ licence). SPDX identifikátor s licencí nemusí být nutně umístěn na první řádce (je to ale doporučeno). Pokud není SPDX identifikátor v souboru přítomný vůbec, kompilátor vypíše varování.

2.1.3.2 Pragma

Klíčové slovo `pragma` ovlivňuje chování kompilátoru [56]. Je podporováno několik příkazů typu pragma.

Příkazem `pragma solidity identifikator_verze;` je možné specifikovat, pro jaké verze kompilátoru Solidity je zdrojový kód určen. Během kompilace kompilátor porovnává svoji verzi s tímto identifikátorem a pokud verze kompilátoru tomuto identifikátoru neodpovídá, je kompilace ukončena s chybovou hláškou. Jako `identifikator_verze` může být uvedena jedna konkrétní verze (například `0.8.13`) nebo seznam rozsahů verzí ve formátu shodném s verzováním nástroje *npm*[3] [58].

Příkazy `pragma abicoder v1;` a `pragma abicoder v2;` aktivují danou verzi Application Binary Interface (ABI) kóderu [39]. ABI kóder má za úkol převod (kódování) datových typů Solidity na datové typy, které je možné snadno reprezentovat v paměti EVM pomocí posloupností bytů. Využívá se při popisu aplikačního rozhraní[4] smart kontraktů. Pro každou veřejně volatelnou funkci smart kontraktu je poskytnut popis vstupů a výstupů funkce, což umožňuje její volání v Ethereum síti standardizovaným způsobem.

ABI kóder ve verzi 2 oproti verzi 1 umožňuje kódování vnořených polí a struktur. Od verze Solidity 0.6.0 již není nový ABI kóder považován za experimentální a od verze 0.8.0 je používán při kompilaci implicitně.

Velmi málo využívanou i vyvíjenou funkcionalitou je SMT checker [59], který umožňuje během kompilace formální ověření kódu SMT (Satisfiability Modulo Theories) solverem podle `assert` a `require` příkazů. Pro podporu této funkcionality musí být navíc kompilátor sestaven ve speciálním režimu. SMT checker se aktivuje příkazem `pragma experimental SMTChecker;`.

2.1.3.3 Import

Solidity má formát `import` příkazů, pro načítání symbolů z dalších zdrojových kódů, velmi podobný jazyku JavaScript [56]. Příkaz je podporován v několika variantách:

```
import "filename";
```

Příkaz načte globální symboly z daného souboru do globálního jmenného prostoru aktuálního souboru.

```
import * as symbolName from "filename";
```

Vytvoří nový globální symbol `symbolName` a všechny globální symboly z daného souboru zpřístupní ve jmenném prostoru symbolu `symbolName`.

[3] Přesnému formátu verzování se věnuje implementační část této práce.
[4] Odtud pochází název Aplication Binary Interface (ABI).

```
import "filename" as symbolName;
```

Má stejný význam jako předchozí případ, pouze využívá jinou syntaxi.

```
import {symbolName1 as aliasName1, symbolName2} from "filename";
```

Příkaz umožňuje zvolit konkrétní symboly, které jsou načtené z daného souboru. Každý symbol je navíc možné přejmenovat. V uvedeném příkladu je načtený symbol `symbolName2` ze souboru `filename`. Dále je v aktuálním souboru vytvořený nový globální symbol `aliasName1`, který odkazuje na symbol `symbolName1` ze souboru `filename`.

2.1.3.4 Komentáře

Solidity nabízí dva druhy komentářů: jednořádkové a víceřádkové [56]. Jednořádkové komentáře začínají dvojicí znaků `//` a končí koncem řádku. Víceřádkové komentáře začínají dvojicí znaků `/*` a končí dvojicí znaků `*/`.

V obou případech je možné do komentáře vepsat speciální metadata v *NatSpec* formátu [60]. V takovém případě jednořádkové komentáře začínají trojicí znaků `///` a víceřádkové komentáře trojicí `/**`. Jak ukazuje výpis 2.1, *NatSpec* formát umožňuje například přidat titulek, autora či popis argumentů a výstupu funkce. Kromě zavedených typů metadat je možné přidat metadata s vlastním štítkem, což může být využíváno aplikacemi třetích stran.

Výpis kódu 2.1 Ukázka komentářů v Solidity

```
1  // A single-line comment example
2
3  /*
4  A multi-line
5  comment
6  example.
7  */
8
9  /// @title Add two numbers
10 /// @author Michal Převrátil
11 /// @return Sum of the arguments
12 function sum(uint a, uint b) public pure returns (uint) {
13    return a + b;
14 }
```

2.1.3.5 Struktury

Struktury jsou uživatelské datové typy seskupující více proměnných do jednoho záznamu [61]. Jsou deklarované klíčovým slovem `struct`:

```
struct Person {
   string name;
   bool verified;
}
```

2.1.3.6 Výčtové typy

Výčtové typy (enumerace) popisují výčet z fixních uživatelsky pojmenovaných možností [53][61]. Interně jsou jednotlivé možnosti reprezentovány pomocí hodnot jednoho bytu (od 0 do 255). Výčtové typy jsou deklarovány pomocí klíčového slova `enum`:

```
enum State { Visible, Hidden, Deactivated }
```

2.1.3.7 Chybové typy

Chybové typy mohou být používané v `revert` příkazech jazyka Solidity způsobujících vykonání *revert* sémantiky EVM [61][62]. Chybové typy popisují data vrácená volajícímu kontextu pomocí instrukce `REVERT`. Chybové typy jsou deklarované klíčovým slovem `error`:

```
error InsufficientBalance(uint currentBalance, uint transferAmount);
```

2.1.3.8 Události

Události přímo souvisí s logovacími zprávami EVM [61][62]. Prostřednictvím těchto logovacích zpráv mohou smart kontrakty komunikovat s vnějším softwarem mimo blockchain. Součástí deklarace událostí může být limitovaný počet proměnných s klíčovým slovem `indexed`. Tyto proměnné pak popisují štítky (topics) logovacích zpráv. Události je možné deklarovat následujícím způsobem:

```
event Transfer(
   address indexed from,
   address indexed to,
   uint value
);
```

2.1.3.9 Funkce

Funkce jsou typicky deklarované uvnitř kontraktů [61][62]. Funkce aktuálního kontraktu mohou být volány pomocí interního volání. To je realizováno pomocí skokových instrukcí EVM. Funkce jiných kontraktů musí být volány pomocí externího volání, které je realizováno na úrovni EVM pomocí volání kontraktu (*message-call*). Interní volání funkce je ukutečněno uvedením názvu funkce s kulatými závorkami a případnými argumenty – `add(10, 20)`. Externí volání funkce je provedeno spojením názvu instance kontraktu a její funkce tečkou – `contract1.sub(20, 10)`. Funkce aktuálního kontraktu mohou být také s použitím klíčového slova `this` volány externím způsobem – `this.mul(2, 3)`. Se způsobem volání souvisí klíčová slova, která mohou být uvedena jako součást deklarace funkce:

- `external`: Funkce může být volaná pouze externím způsobem volání.
- `public`: Funkce může být volaná externím způsobem volání z jiného kontraktu a interním nebo externím způsobem z aktuálního kontraktu.
- `internal`: Funkce může být volaná pouze interním způsobem volání z aktuálního kontraktu nebo z kontraktu, který dědí od kontraktu, ve kterém je funkce deklarovaná.
- `private`: Funkce může být volaná pouze interním způsobem volání z aktuálního kontraktu.

Funkce dále mohou být volitelně označeny jedním z těchto klíčových slov v závislosti na tom, jak pracují se stavem blockchainu:

- `view`: Funkce označená tímto klíčovým slovem nemůže měnit stav blockchainu. Může ale tento stav číst. Pokud je to možné, je volání této funkce implementováno pomocí instrukce `STATICCALL`.
- `pure`: Funkce deklarovaná s tímto klíčovým slovem nemůže zapisovat ani číst stav blockchainu. Pokud je to možné, je volání této funkce implementováno pomocí instrukce `STATICCALL`.

Funkce kromě argumentů může také specifikovat seznam návratových hodnot. Ukázka 2.2 představuje funkci, která může být označena klíčovým slovem `pure`, jelikož nečte stav blockchainu ani jej nemodifikuje. Na vstupu funkce jsou dvě znaménková čísla a výstupem funkce je jejich součet a rozdíl.

Výpis kódu 2.2 Ukázka funkce jazyka Solidity

```
function addSub(int a, int b) public pure returns (int, int) {
   return (a + b, a - b);
}
```

2.1.3.10 Modifikátory funkcí

Pomocí modifikátorů funkcí je možné měnit chování funkce [61][62]. Modifikátory mohou být specifikovány mezi klíčovými slovy v deklaraci funkce. Také jim mohou být předány některé argumenty funkce. Při volání funkce s modifikátorem není prováděno tělo funkce, ale tělo modifikátoru. Jestliže modifkátor obsahuje speciální příkaz `_;`, je na tomto místě v kódu modifikátoru provedeno (vloženo) tělo prováděné funkce. Jestliže je v deklaraci jedné funkce specifikováno více modifikátorů, jsou tyto modifikátory prováděny ve stejném pořadí, v jakém jsou uvedeny v deklaraci funkce. Výpis 2.3 ukazuje modifikátor zabraňující zanořenému volání té samé funkce.

Výpis kódu 2.3 Ukázka modifikátoru funkce jazyka Solidity [63] (upraveno)

```
contract FunctionModifier {
   uint public x = 10;
   bool public locked;

   modifier noReentrancy() {
      require(!locked, "No reentrancy");

      locked = true;
      _;
      locked = false;
   }

   function decrement(uint i) public noReentrancy {
      x -= i;

      if (i > 1) {
         decrement(i - 1); // selže díky modifikátoru funkce
      }
   }
}
```

2.1.3.11 Stavové proměnné

Speciálním typem proměnných jsou proměnné, které jsou deklarované uvnitř kontraktu [61][62]. Tyto proměnné se nazývají stavové proměnné a jsou ukládány ve *storage* segmentu kontraktu. Tyto proměnné mohou být označené klíčovým slovem `constant` nebo `immutable`. V prvním případě musí být hodnota stavové proměnné známa již při kompilaci a poté nemůže být upravována. Ve druhém případě může být hodnota stavové proměnné nastavena uvnitř konstruktoru kontraktu. Poté stavová proměnná také nemůže být upravena. Stavové proměnné označené těmito klíčovými slovy nejsou uložené ve *storage* segmentu. Dále je stavovým proměnným nastavovaná viditelnost podobně, jako je tomu u funkcí:

- `public`: Pro stavovou proměnnou je automaticky vygenerovaná funkce, která umožňuje získat hodnotu proměnné deklarované uvnitř kontraktu z libovolného jiného kontraktu.
- `internal`: Ke stavové hodnotě může být přistupováno z kontraktu, uvnitř kterého je proměnná deklarována, a z kontraktu, který od tohoto kontraktu dědí. Pokud není specifikováno žádné klíčové slovo modifikující viditelnost stavové proměnné, je použita tato viditelnost.
- `private`: Ke stavové proměnné může být přistupováno pouze z kontraktu, ve kterém je proměnná deklarována.

Až na případná klíčová slova se stavové proměnné syntaxí neliší od proměnných deklarovaných uvnitř funkce:

```
contract ContractExample {
   string constant constantStateVariable = "abc";
   bool stateVariable;
}
```

2.1.3.12 Kontrakty

Kontrakty jazyka Solidity odpovídají kontraktům popsaným v Yellowpaperu [62]. Mohou obsahovat konstruktor ve formě funkce, která je volaná při vytváření kontraktu. Kontrakty také podporují vícenásobnou dědičnost – v tomto ohledu jsou kontrakty blízkém třídám z jiných programovacích jazyků.

2.2 Kompilátor *solc*

Kompilátor *solc* převádí zdrojový kód v jazyce Solidity do EVM bytekódu [64]. Mimo to kompilátor poskytuje velké množství dalších informací, které mohou být užitečné například pro nástroje pracující se Solidity kódem. Ve výchozím stavu kompilátor nevypisuje žádný výstup. Jednotlivé typy výstupů je možné vyžádat pomocí přepínačů příkazové řádky. Mezi typy výstupu patří také Abstract Syntax Tree (AST). Jedná se o strukturovaný popis syntaxe Solidity souboru v JSON formátu. Jelikož jsou data strukturovaná ve formě stromu, často se uzel v tomto stromu (konkrétní datová položka AST) označuje jako AST node. AST data jsou velmi výhodná pro všechny nástroje pro analýzu Solidity souborů. Díky těmto datům nemusí nástroje provádět parsování Solidity kódu. AST výstup kompilátoru však není formálně zdokumentován.

Kompilátor *solc* také nabízí komunikační rozhraní pomocí standardního vstupu (stdin) a standardního výstupu (stdout) v JSON formátu. Toto rozhraní je výhodné především pro nástroje pracující s kompilátorem. Konkrétní formát je popsán na adrese `https://docs.soliditylang.org/en/v0.8.13/using-the-compiler.html#compiler-input-and-output-json-description`.

2.2.1 Vyhodnocování importních cest

Součástí kompilace je také převod importních řetězců jazyka Solidity na systémové cesty k souborům [65]. Sestavení kompilátoru pro některé platformy (například WebAssembly) nabízí specifikaci vlatních *import callbacků*. Díky tomu mohou nástroje pro tyto platformy namísto cest k souborům zpracovávat také URL adresy (odkazující například na GitHub). V případě staticky sestavených verzí kompilátoru pro Linux OS, Windows OS a macOS však tato možnost chybí.

2.2.1.1 Remapping

První popis použití kompilátoru *solc* se v dokumentaci jazyka Solidity nachází v dokumentaci pro verzi 0.4.10 [66]. Dokumentace popisuje volitelné argumenty ve formě `prefix=path` nazývané *remappingy*, které mohou být předány kompilátoru. Kompilátor při zpracování importního řetězce prochází jednotlivé remappingy a zkoumá, zda počátek importního řetězce přesně odpovídá `prefix` části remappingu. Z odpovídajících remappingů je zvolen ten s nejdelším prefixem. Z importního řetězce je odstraněn `prefix` zvoleného remappingu a namísto toho je vložena `path` část remappingu. Importní řetězec začínající textem `github.com/ethereum/dapp-bin/` může být přepsán na importní řetězec začínající textem `/usr/local/lib/dapp-bin/` pomocí remappingu `github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/`. Prefixová část remappingu může být také prázdný řetězec. Kompilátor ve verzi 0.4.10 odmítá číst ze souborů, pokud nejsou přímo specifikovány příkazovou řádkou nebo pokud nejsou cílem remappingu.

2.2.1.2 Přepínač `--allow-paths`

Aby bylo možné specifikovat dodatečné adresáře, ze kterých může kompilátor číst, byl ve verzi 0.4.11 přidán přepínač příkazové řádky `--allow-paths` [67]. Tento přepínač umožňuje specifikovat čárkou oddělené cesty k libovolnému počtu adresářů. V případě využití rozhraní standardního vstupu a výstupu v JSON formátu je nutné tento přepínač také uvést na příkazové řádce.

2.2.1.3 Úprava formátu remappingu

Ve verzi 0.5.0 dochází ke změně, kdy prefixová část remappingu již nemůže být prázdný řetězec [68].

2.2.1.4 Přepínač `--base-path`

Verze 0.6.9 přidává nový přepínač příkazové řádky `--base-path` [69]. Přepínačem je možné nastavit adresář, který je předřazen před všechny cesty při přístupu do souborového systému. Ve výchozím nastavení je hodnota tohoto přepínače kořen souborového systému. To je konzistentní s faktem, že importní řetězec `/var/lib/solidity_lib/a.sol` by měl být ve výchozím nastavení vyhledávat soubor `a.sol` v adresáři `/var/lib/solidity_lib`. Ve výchozím nastavení však není kořenový adresář souborového systému přidán mezi adresáře, ke kterým je umožněn přístup (přepínač `--allow-paths`). Naopak, pokud je přepínač `--base-path` specifikován uživatelem, je tento adresář zařazen také mezi povolené adresáře. Přepínač je nutné specifikovat na příkazové řádce nezávisle na tom, zda je použit standardní JSON vstup, nebo ne. Motivace za přepínačem `--base-path` je taková, že kompilátor *solc* pak může ve vnitřní reprezentaci pracovat s relativními cestami souborů. Toto umožňuje vytváření reprodukovatelných sestavení (kompilací) na různých platformách. Toto řešení však může být považováno za neintuitivní, neboť při použití `--base-path=/tmp` je importním řetězcem `/var/lib/solidity_lib/a.sol` soubor vyhledáván v systémové cestě `/tmp/var/lib/solidity_lib/a.sol`.

2.2.1.5 Source unit name

Ve verzi 0.8.5 byla v dokumentaci jazyka Solidity vytvořena nová sekce zabývající se zpracováním importních řetězců [70]. Tato část dokumentace přichází s novým pojmem, kterým je *source unit name*. Jedná se o textový řetězec, který popisuje lokaci zdrojového souboru jazyka Solidity. Pro jeden konkrétní zdrojový soubor může existovat více řetězců *source unit name*. *Source unit name* vzniká jiným postupem podle toho, jak je soubor do kompilace zařazen:

- Jestliže je soubor pro kompilaci předán jako argument na příkazové řádce, je *source unit name* tohoto souboru přímo cesta zapsaná jako argument příkazové řádky. Oddělovače adresářů v cestě jsou nahrazeny za jedno lomítko `/`.

- Jestliže je soubor pro kompilaci načten pomocí importního příkazu, vzniká *source unit name* z textového řetězce importního příkazu:

  - Jestliže importní řetězec není relativní (nezačíná znaky `./` nebo `../`), jedná se o přímý (absolutní) importní řetězec. *Source unit name* vzniká z importního řetězce po případné aplikaci remappingu na importní řetězec.

  - Jestliže je importní řetězec relativní, je postup komplexnější. Dokumentace zavádí pojem segment cesty souborového systému. Segment cesty je neprázdný textový řetězec ohraničený znaky lomítka `/`, začátkem řetězce nebo koncem řetězce. Například řetězec `.///abc/../xyz//` obsahuje popořadě segmenty `.` (tečka), `abc`, `..` (dvě tečky) a `xyz`.

    Relativní importní řetězec je normalizován. Segmenty obsahující pouze tečku jsou odstraněny. Za každý segment `..` (dvě tečky) je odebrán předchozí segment, pokud tento segment rovněž neobsahuje dvě tečky. Více lomítek `/` uvedených bezprostředně za sebou je nahrazeno za jediné lomítko.

    Jelikož se v tomto případě jedná o relativní importní řetězec, je relativně vyhodnocen vzhledem k jinému řetězci. Tímto řetězcem je *source unit name* souboru, jehož importní příkaz je právě analyzován. Jedná se tedy o *source unit name* importujícího souboru za cílem získání *source unit name* importovaného souboru. *Source unit name* importujícího souboru je nejprve upraven. Je z něj odebrán poslední segment cesty a také všechna lomítka na konci řetězce. Za každý segment `..` (dvě tečky) na začátku normalizovaného importního řetězce je z již upraveného *source unit name* importujícího souboru odstraněn jeden segment. *Source unit name* importovaného souboru vzniká jako zřetězení upraveného *source unit name* importujícího souboru a normalizovaného relativního importního řetězce. Pokud by první řetězec nekončil lomítkem a zároveň by druhý řetězec nezačínal lomítkem, je při zřetězení mezi řetězce vložen znak lomítka `/`. Po získání *source unit name* importovaného souboru je na tento řetězec ještě případně aplikován remapping.

Po získání *source unit name* souboru je tento soubor vyhledán v souborovém systému s použitím získaného *source unit name* a předřazeného adresáře `--base-path`. Případně na některých platformách je tento *source unit name* předán *import callbackům*.

Ve verzi 0.8.5 byla také syntaxe remappingu rozšířena o novou část – `context`. Celý formát remappingu je od této verze `context:prefix=target`. Části `prefix` a `target` byly již vysvětleny dříve. Část `prefix` je jedinou povinnou částí, ostatní části jsou volitelné. Také znak dvojtečky `:` je pak ve většině případů volitelný. Výjimku tvoří například remapping obsahující webovou adresu `:https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin`. Pokud by nebyla úvodní dvojtečka uvedena, jako `context` by byla vyhodnocena část `https`. Část `context` omezuje použití remappingu jen na případy, kdy *source unit name* importujícího souboru začíná na řetězec odpovídající `context` části a remapping má být aplikován na *source unit name* importovaného souboru (tento *source unit name* musí začínat na řetězec odpovídající `prefix` části).

2.2.1.6 Přepínače `--include-path`

Verze 0.8.8 přidává nový typ přepínače `--include-path` příkazové řádky [71]. Tento přepínač může být specifikován na příkazové řádce několikrát. Hodnotou tohoto přepínače by měl být adresář. Tento adresář je stejně jako adresář specifikovaný přepínačem `--base-path` předřazován před výsledný *source unit name* souboru (po případné aplikaci remappingu). Jestliže po předřazení více adresářů `--include-path` nebo adresáře `--base-path` vznikne více validních cest vedoucích k existujícímu souboru, je *source unit name* nejednoznačný vzhledem k dodaným adresářům a jedná se o chybu kompilace. V opačném případě je použit adresář `--base-path` nebo některý z adresářů `--include-path`, jehož předřazení vede ke vzniku systémové cesty vedoucí k existujícímu souboru. Adresáře specifikované pomocí přepínače `--include-path` není nutné uvádět do povolených adresářů přepínačem `--allow-paths`. Přepínač `--base-path` se na základě tohoto popisu chová stejně jako `--include-path` a mohl by být odstraněn. Z důvodů zpětné kompatibility je však předpokládáno, že přepínač prozatím zůstane kompilátorem *solc* podporován [72].

2.2.1.7 Budoucí změny

V době psaní této práce existují v GitHub repozitáři projektu Solidity [73] čtyři otevřené issues týkající se vyhodnocování importních řetězců.

V jednom z issue je uvedeno, že kompilátor *solc* by měl v případě problémů s kompilovanými soubory uvádět relevantnější chybové hlášky [74]. Jestliže vyhledávaný soubor existuje, ale není zahrnut do povolených adresářů (například přepínačem `--allow-paths`), měla by být tato skutečnost kompilátorem oznámena. V případě, že není možné importovaný soubor nalézt, měl byt kompilátor pomocí chybových hlášek lépe specifikovat, které adresáře byly prohledávány. Jestliže je kompilátorem zpracováván importní řetězec vypadající jako URL adresa, avšak pro kompilátor dané platformy nejsou *import callbacky* implementovány, mělo by toto být uživateli oznámeno.

Další issue popisuje, že by kompilátor měl být schopný detekovat importní řetězce obsahující znaky systémové cesty pro specifickou platformu [75]. Tímto je především zamýšleno, že by kompilátor měl detekovat některá specifika cest souborového systému Windows OS. Příkladem je následující importní příkaz:

```
import "C:\\project\\lib\\token.sol";
```

Source unit name importovaného souboru by byl řetězec `C:\project\lib\token.sol`[5], což není pro *nix platformy žádoucí. Uživatel kompilátoru by měl být varován, že importní řetězce by měly být nezávislé na platformě.

Další issue popisuje změny, které nejsou zpětně kompatibilní [76]. Je zde popsáno, že by všechny importní řetězce měly být stejně normalizovány tak, aby každému zdrojovému Solidity souboru odpovídal právě jeden řetězec *source unit name*. Toho má být dosaženo úpravou pravidel pro zpracování importních řetězců. Tato pravidla by se měla vztahovat pouze na importní řetězce, které neprezentují URL nějakého protokolu. Mělo by se jednat o tato pravidla:

- Měly by být odstraněny segmenty `.` (tečka) a `..` (dvě tečky) ze všech importních řetězců. Pokud by segment `..` (dvě tečky) odstraňoval také segment obsahující dvě tečky, měla by být kompilátorem vyvolána chyba. Chyba by měla být vyvolána také v případě, kdy by pro segment `..` (dvě tečky) již neexistoval předchozí segment, který by byl odstraněn (segment obsahující dvě tečky by byl prvním segmentem).
- Posloupnosti více bezprostředně následujících lomítek `/` by měly být sloučeny do jednoho lomítka.

[5] Za předpokladu, že by na tento řetězec nebyl aplikován žádný remapping

Některá tato pravidla jsou již od verze 0.8.5 aplikována na relativní importní řetězce.

Poslední issue také není zpětně kompatibilní [77]. Popisuje snahu zakázání absolutních cest v importních řetězcích. Přesněji řečeno by *source unit name* libovolného souboru neměl začínat znakem lomítko `/`. Toho by mělo být uživatelem docíleno specifikací přepínačů `--base-path` a `--include-path`, které by pokrývaly cesty ke všem kompilovaným Solidity souborům. Přepínač `--base-path` by měl nově jako výchozí hodnotu adresář, ze kterého je kompilátor spuštěn. Součástí diskuze je také popis vyhodnocování importních cest v různých nástrojích pracujících s kompilátorem *solc*. Pravděpodobně i z tohoto důvodu je součástí návrhu také možnost použití absolutních cest ve standardním JSON vstupu za určitých podmínek.

Kapitola 3

Nástroje používané Ethereum komunitou

Kromě nástrojů kritických pro správné fungování Ethereum sítě existuje velké množství dalších nástrojů, které usnadňují práci programátorů i auditorů na úrovni EVM i Solidity kódu.

3.1 Nástroj Slither

Z pohledu auditorů jsou zajímavé především nástroje pro analýzu Solidity kódu. Některé nástroje se zaměřují čistě na statickou či čistě na dynamickou analýzu, jiné kombinují oba druhy analýzy. Slither patří mezi nástroje zaměřující se čistě na statickou analýzu [78]. Jedná se o nástroj napsaný v jazyce Python.

Slither vychází z AST reprezentace Solidity kódu, ze které vytváří svoji speciální datovou reprezentaci nazývanou *SlithIR* (Slither Intermediate Representation). Tato reprezentace oproti AST lépe popisuje některé výrazy v Solidity (například členskou funkci `push` datového typu pole). *SlithIR* také převádí složitější výrazy Solidity na jednodušší podvýrazy (mezivýsledky), které označuje speciálními typy proměnných. Kromě typů proměnných přímo vycházejících z jazyka Solidity jsou navíc přidané dočasné proměnné pro mezivýsledky, které nejsou v kódu přímo uložené do žádné proměnné, a referenční proměnné značící přístup do mappingu nebo pole. *SlithIR* reprezentace je velmi využívána v detektorech zranitelností.

3.1.1 Detektory zranitelností

Slither pomocí statické analýzy nad AST reprezentací kódu implementuje velké množství detektorů zranitelností. V době psaní této práce je implementováno 76 detektorů a u každého je uveden případný dopad zranitelnosti či problému a míra jistoty, s jakou je tento problém detekován správně. Rozsah detekovatelných zranitelností a jiných problémů je široký. Jedná se například o:

- vícenásobné použití stejného jména kontraktu,
- neinicializované proměnné,
- použití slabého PRNG,
- funkce umožňující komukoliv zničení kontraktu.

Ve výchozím nastavení jsou příkazem `slither` spuštěny všechny detektory.

3.1.1.1 Implementace vlastních detektorů

Slither skrz svoje API umožňuje vytváření vlastních detektorů zranitelností. Ukázka kódu 3.1 vychází ze šablony uvedené v dokumentaci Slitheru [79]. V ukázce je implementovaný jednoduchý detektor vícenásobného umocňování. Solidity verze 0.8.0 jako jednu ze zásadních změn přineslo úpravu asociativity umocňování. To znamená, že výraz `a ** b ** c` je ve straších verzích Solidity vyhodnocovaný v jiném pořadí než ve verzi 0.8.0 a novější. Implementovaný detektor se na tuto skutečnost snaží upozornit pomocí detekce vícenásobného umocňování. Detektor pracuje s nízkou hladinou přesnosti detekce, jelikož není schopný rozpoznat, zda je asociativita vícenásobného umocňování upravena pomocí závorek, či nikoliv. Implementace této upřesňující detekce je poměrně komplikovaná vzhledem k AST reprezentaci, ze které Slither vychází, a je mimo rozsah této ukázky. Celý projekt s implementovaným ukázkovým detektorem je k dispozici na přiloženém médiu.

Výpis kódu 3.1 Ukázka implementace vlastního detektoru ve Slitheru

```
1  class ExponentiationAssociativity(AbstractDetector):
2      ARGUMENT = "exponentiation-associativity"
3      HELP = "Possible exponentiation associativity issue"
4      IMPACT = DetectorClassification.HIGH
5      CONFIDENCE = DetectorClassification.LOW
6
7      @staticmethod
8      def __search(root_node: Node) -> List:
9          visited = set()
10         results = set()
11         queue: Deque[Node] = deque([root_node])
12         while len(queue):
13             node = queue.pop()
14             if node in visited:
15                 continue
16             visited.add(node)
17             exponentiations = set()
18             for ir in node.irs:
19                 if isinstance(ir, Binary):
20                     if ir.type == BinaryType.POWER:
21                         exponentiations.add(ir.lvalue)
22                         if set(ir.read) & exponentiations:
23                             results.add(node)
24             queue.extend(node.sons)
25         return list(results)
26
27     def _detect(self):
28         results = []
29         for contract in self.contracts:
30             for function in contract.functions_declared:
31                 if not function.entry_point:
32                     continue
33                 for ir in self.__search(function.entry_point):
34                     out = ["Possible exponentiation associativity issue found
                       ↪  in ", function, ":\n", ir, "\n"]
35                     results.append(self.generate_result(out))
36         return results
```

3.1.2 Příkazy pro vizualizaci dat

Další velkou podmnožinou funkcionalit Slitheru jsou příkazy pro vizualizaci dat. Vizualizace může být provedena pro celý projekt, pro jeden konkrétní Solidity soubor nebo pro adresu s kontraktem na jedné z Ethereum sítí (včetně hlavní sítě a testovacích sítí). Příkazy se obecně dělí na dva druhy podle formátu výstupu. První skupinu tvoří příkazy pro strukturovaný výpis textových dat a patří mezi ně například:

- souhrn informací o kontraktu,
- výpis modifikátorů funkce,
- seznam `require` a `assert` příkazů uvnitř funkce,
- výpis datových závislostí mezi proměnnými kontraktů a funkcí.

Výpis 3.2 ukazuje výstup příkazu `contract-summary` pro soubor `ECDSA.sol` knihovny OpenZeppelin [54]. V souboru jsou implementované tři funkce. První dvě implementují obnovení (získání) Ethereum adresy z hashe transakce a ECDSA podpisu.

Výpis kódu 3.2 Ukázka výstupu Slither příkazu `contract-summary`

```
1 + Contract ECDSA (Most derived contract)
2   - From ECDSA
3     - recover(bytes32,bytes) (internal)
4     - recover(bytes32,uint8,bytes32,bytes32) (internal)
5     - toEthSignedMessageHash(bytes32) (internal)
```

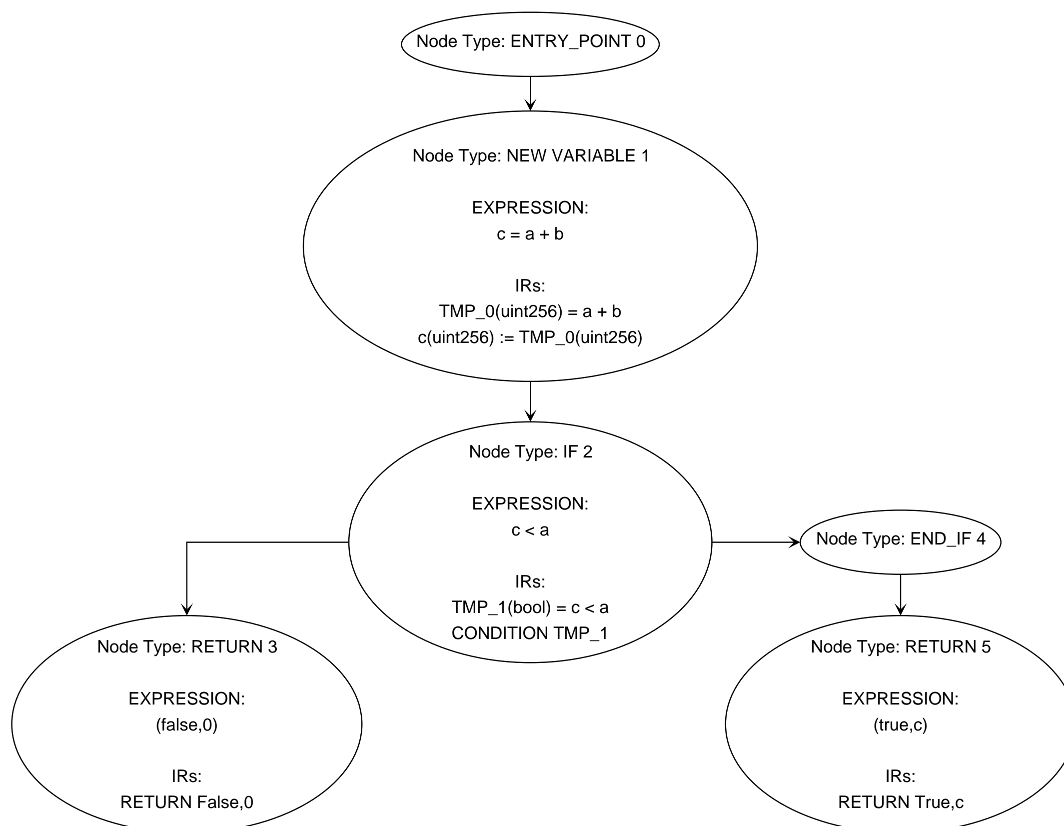
Do druhé skupiny patří příkazy zapisující do souborů v *dot* formátu [80]. Jedná se o data vykreslovaná ve formě grafů či jiných diagramů. Příkazy umí vygenerovat:

- graf volání funkcí,
- control flow graf každé funkce,
- graf dědičnosti kontraktů.

Výpis kódu 3.3 Implementace bezpečného sečtení čísel knihovny OpenZeppelin

```
1 function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
2     uint256 c = a + b;
3     if (c < a) return (false, 0);
4     return (true, c);
5 }
```

Obrázek 3.1 představuje *control flow graf* [81] funkce `tryAdd` knihovny OpenZeppelin uvedené ve výpisu 3.3. Jedná se o funkci, která byla velmi využívána ve starších verzích jazyka Solidity, kdy ještě nebyly prováděny automatické kontroly přetečení a podtečení. Funkce přijímá na vstupu dvě bezznaménková čísla (slova EVM). Funkce sčítá tato čísla a na výstupu vrací dvojici. První prvek dvojice je hodnota typu `bool`, která popisuje, zda při sečtení čísel nedošlo k přetečení (hodnota `true`), nebo došlo k přetečení (hodnota `false`). V prvním případě je druhým prvkem návratové hodnoty součet vstupních argumentů. Ve druhém případě je druhým prvkem hodnota nula.



Obrázek 3.1 Ukázka *control flow grafu* vygenerovaného Slitherem

3.1.3 Další příkazy Slitheru

Slither dále implementuje několik specializovaných funkcionalit, mezi které patří kontrola správného rozhraní kontraktů implementovaných podle některých EIP nebo kontrola správné implementace kontraktů, jejichž kód je možné pomocí speciálních technik aktualizovat.

3.1.3.1 Kontrola rozhraní kontraktu podle EIP

Mezi EIP patří také návrhy, které standardizují aplikační binární rozhraní (ABI) kontraktů, které mají být využívané specifickým způsobem. Jedná se především o specifikaci rozhraní kontraktů, které mají implementovat řídící logiku virtuálních mincí projektů na síti Ethereum. Yellowpaper jako platidlo v Ethereum síti popisuje Ether [2]. Platba za zpracování transakcí je uživateli prováděna prostřednictvím Gasu, který je na Ether převeden. Jelikož však lze vnímat síť Ethereum jako platformu pro vytváření libovolných aplikací, mohou programátoři pomocí EVM bytekódu implementovat své vlastní virtuální platidla, která se obecně nazývají *tokeny*. Aplikační rozhraní implementace *tokenů* je standardizováno. Jedná se především o EIP-20 [18], EIP-721 [82], EIP-777 [83] a EIP-1155 [84]. Podle těchto EIP se obvykle hovoří o tokenech typu ERC-20, ERC-721, ERC-777 a ERC-1155[1]. Nástroj Slither dokáže správnost implementace rozhraní těchto tokenů v Solidity kódu zkontrolovat.

[1] EIP formalizující tokeny patří do kategorie ERC.

3.1.3.2 Kontrola aktualizovatelnosti kódu kontraktu

Aktualizace kódu kontraktu je na Ethereum blockchainu obecně komplikovaný problém. V Yellowpaperu není uvedena explicitní podpora pro aktualizaci kódu nasazeného kontraktu [2]. Přesto existuje několik technik, pomocí kterých lze kód kontraktu aktualizovat. Knihovna OpenZeppelin tuto možnost nabízí se zachováním *storage* segmentu i adresy kontraktu [85]. Dále je vhodné v případě aktualizovatelných kontraktů oddělovat kontrakt s daty (uloženými ve *storage* segmentu) a kontrakt s řídící logikou. Jeden z typů implementace tohoto rozdělení kontraktů využívá instrukce `DELEGATECALL` [86].

Implementace projektu je rozdělena do datových kontraktů, které udržují data ve *storage* segmentu, a kontraktů implementujících logiku projektu. Datové kontrakty volají funkce kontraktů implementujících logiku pomocí instrukce `DELEGATECALL`. Díky tomu mohou kontrakty implementující logiku manipulovat se *storage* segmentem datových kontraktů. Při aktualizaci je vytvořen nový kontrakt s logikou. Také je aktualizován datový kontrakt, jehož volání musí směřovat na nově nasazený kontrakt implementující logiku projektu. Při aktualizaci datového kontraktu však musí být zaručeno, že značná část kontraktu zůstane zachována. Speciálně data ve *storage* segmentu původního datového kontraktu musí být kompatibilní s novým datovým kontraktem. Nástroj Slither dokáže detekovat některé změny kontraktu, které mohou zapříčinit změnu rozložení *storage* segmentu, a tedy nekompatibilitu dat starého a nového kontraktu.

3.2 Remix IDE

Z pohledu Solidity vývojářů je především důležité mít k dispozici vývojové prostředí, Integrated Development Environment (IDE), které umožňuje snadnější vývoj smart kontraktů v Solidity. Remix IDE [87], vývojové prostředí vyvíjené Ethereum Foundation, se snaží nabídnout kompletní ucelené prostředí zahrnující editor, základní analýzu, kompilátor kódu, nástroj pro nasazení kontraktů na různé Ethereum sítě a debugger transakcí. Nástroj je distribuován ve formě spustitelných souborů pro všechny hlavní platformy i ve formě webového rozhraní. Do vývojového prostředí je možné doinstalovat další komunitní rozšíření.

Editor nabízí zvýrazňování syntaxe kódu. Dále je k dispozici jednoduché bezkontextové našeptávání. Stiskem pravého tlačítka lze vyvolat kontextové menu. V nabídce jsou kromě akcí *vyjmout*, *kopírovat* a *vložit* dvě další užitečné funkcionality. První z nich je *Change All Occurrences*, což umožňuje nahradit všechny výskyty daného slova v souboru. Příkaz je bezkontextový – umožňuje nahradit libovolné slovo nezávisle na tom, zda se jedná o název proměnné či klíčové slovo jazyka Solidity. Druhou funkcionalitou je *Command Palette*, což otevírá nabídku s dalšími příkazy. Většina těchto příkazů se týká práce s editorem (přesun kurzoru, kopírování řádek, vyhledávání v textu, . . .).

3.3 Nástroj solc-select

Jelikož je kompilátor *solc* distribuován ve verzích odpovídajících verzím Solidity, může být nutné mít na systému nainstalováno více verzí kompilátoru. Nástroj solc-select funguje jako správce instalací kompilátoru *solc* [88]. Také poskytuje spustitelný soubor (skript jazyka Python) nazvaný *solc*, který funguje jako rozhraní pro uživatelem zvolenou verzi kompilátoru. Toto řešení je však problematické v případě, kdy spustitelný soubor s názvem *solc* poskytuje také jiný nástroj, nebo je kompilátor *solc* nainstalovaný pomocí balíčkovacího systému operačního systému. Nástroj jako součást stahování kompilátoru také provádí kontrolní součty ve formě SHA-256 a Keccak-256[2] hashů.

[2]Autor této práce při analýze nástroje solc-select objevil chybu, kdy byl očekávaný Keccak-256 hash dodávaný se spustitelným souborem kompilátoru nesprávně porovnáván proti vypočtenému algoritmicky odlišnému hashi SHA-3. Tato chyba se v nástroji neprojevovala, neboť při kontrole bylo dostačující, aby hash SHA-256 odpovídal očekávané hodnotě. Autoři nástroje solc-select byli na obě chyby upozorněni a chyby byly brzy opraveny [89].

Kapitola 4

Nástroj Woke

Woke je nástroj pro analýzu kódu v jazyce Solidity. Je napsaný v jazyce Python a jeho kód je veřejně dostupný v repozitáři `https://github.com/Ackee-Blockchain/woke`*. Jelikož je Ethereum ekosystém relativně mladý, na poli nástrojů pro analýzu kódu je stále velmi prostoru pro uplatnění nových projektů. Ethereum a speciálně Solidity se navíc velmi rychle vyvíjí a velké množství dříve kvalitních nástrojů dnes již nelze považovat za dostatečně účinné. Nástroj Woke se snaží chybějící funkcionality implementovat a dokonce některé nástroje nahradit.*

4.1 Analýza současného stavu

V době vytváření zadání této práce byl nástroj Woke v návrhové fázi. Modul pro parsování AST výstupu kompilátoru *solc* byl již připraven. Byly také formulovány hlavní myšlenky a motivace pro vytvoření nástroje. Mezi jednu z hlavních myšlenek patří to, že výstup nástroje Slither může být nepraktický. Výstup nástroje pro statickou analýzu je mnohem užitečnější ve vývojovém prostředí, kde programátor či auditor vidí zdrojový kód a k němu v intuitivní formě uvedené výstupy nástroje pro analýzu kódu. Analýza se navíc může dynamicky měnit s tím, jak programátor kód upravuje. I díky tomu mohou analýzu více využívat programátoři jazyka Solidity. Naopak nástroj Slither, který také poskytuje statickou analýzu, je spíše určen pro auditory. Pro vývoj v jazyce Solidity sice existuje Remix IDE, ale pokročilejší analýza a příkazy v něm nejsou podporovány. Například nastavení používání klávesových editoru *vim* je v Remix IDE velmi komplikované nebo dokonce nemožné. Nově příchozí vývojáři z jiných jazyků navíc mohou preferovat jiná vývojová prostředí, jako je například Visual Studio Code [90]. Nástroj Woke by měl analýzu těmto vývojovým prostředím poskytovat.

Pro komunikaci mezi vývojovým prostředím a nástrojem Woke byl zvolen Language Server Protocol (LSP) [91]. Protokol definuje komunikaci mezi klientem implementovaným na straně vývojového prostředí a serverem poskytujícím analýzu kódu. Cílem této práce je implementace modulů, pomocí kterých by mohl LSP server snadno zkompilovat libovolný Solidity projekt a získat z něj všechny potřebné informace. Analýza nástroje Woke spoléhá na data, která jsou výstupem kompilátoru *solc*. Aby byl schopný LSP server rychle reagovat na změny ve zdrojovém kódu, je nutné projekt správně analyzovat a navrhnout nejefektivnější způsob kompilace.

Pro vývoj nástroje Woke byl zvolen programovací jazyk Python. Přestože je tento jazyk dynamicky typovaný, jeho syntaxe umožňuje na volitelné bázi vkládat dodatečné informace o datových typech objektů [92]. Tyto typové informace jsou sice většinou za běhu ignorovány, ale umožňují kontrolu kódu pomocí speciálních nástrojů. Typové informace také mohou být velmi nápomocné pro programátory, kteří by se snažili kódu porozumět. Již od začátku vývoje nástroje Woke byla snaha tyto typové informace, pokud je to možné, využívat.

4.2 Implementace nástroje

Po dohodě s vedoucím této práce byly implementovány následující moduly:

- modul pro čtení konfiguračních souborů,
- modul pro správu instalací kompilátoru *solc*,
- modul pro parsování informací ze Solidity souborů,
- modul pro kompilaci Solidity projektů.

Kód implementace je dostupný na adrese `https://github.com/Ackee-Blockchain/woke/tree/0d27de25720142beb9619a89619b7a94c3556af1`.

4.2.1 Knihovna pydantic

V nástroji Woke je knihovna *pydantic* [93] velmi využívaná. Knihovna využívá typových informací jazyka Python pro validaci a parsování dynamicky načtených dat. Pro načítaná data je vytvořen datový model ve formě tříd jazyka Python. Jednotlivé atributy třídy jsou doplněny o typové a případně také další informace. Mezi dodatečné volitelné informace patří například výchozí hodnota nebo název, pod kterým je atribut uložen v načítaných datech (pokud se tento název neshoduje s názvem atributu). Pro každý atribut nebo dokonce celou třídu může být implementován validátor knihovny *pydantic*, který nejen slouží k validaci dat, ale také může načtená data transformovat. Pro celý datový model může být také nastaveno, zda je možné jednotlivé atributy po prvotním načtení dat měnit. Tato možnost je užitečná v případě, kdy se očekává, že načtená data budou určena pouze pro čtení. Také je možné specifikovat chování pro případ, kdy jsou v načítaných datech některé atributy navíc ve srovnání s datovým modelem.

Výpis kódu 4.1 Příklad použití knihovny *pydantic* v nástroji Woke

```
class WokeConfigModel(BaseModel):
    class Config:
        allow_mutation = False
        json_encoders = {SolidityVersion: str}
        extra = Extra.forbid

class SolcWokeConfig(WokeConfigModel):
    allow_paths: List[Path] = []
    evm_version: Optional[EvmVersionEnum] = None
    include_paths: List[Path] = []
    target_version: Optional[SolidityVersion] = None

    @validator("allow_paths", pre=True, each_item=True)
    def set_allow_path(cls, v):
        return Path(v).resolve()

    @validator("include_paths", pre=True, each_item=True)
    def set_include_path(cls, v):
        return Path(v).resolve()

class CompilerWokeConfig(WokeConfigModel):
    solc: SolcWokeConfig = Field(default_factory=SolcWokeConfig)
```

Ukázkou použití knihovny *pydantic* je výpis 4.1, který ukazuje (upravený) datový model používaný pro načítání konfiguračních souborů nástroje Woke. Po načtení datového modelu není možné jeho atributy přepisovat. Předpokládá se, že potřebné datové transformace jsou provedeny pomocí validátorů a jakýkoliv další zásah do atributů modelu je chybou v programu. Položky konfiguračního souboru, které nejsou v datovém modelu uvedeny, způsobují chybu při načítání. Díky tomu je uživatel informován v případě, že do konfiguračního souboru uvede nevalidní konfigurační možnost. Validátory implementované ve výpisu provádějí normalizaci systémové cesty. V tomto případě především jde o převod na absolutní systémovou cestu.

4.2.2 Příprava vývojového prostředí

Na počátku nového projektu je vhodné připravit nástroje, které mohou vývoj v budoucnu zefektivnit a automaticky hledat některé chyby kódu. Při přípravě prostředí pro vývoj nástroje Woke bylo samozřejmostí použití verzovacího nástroje *git*. Pro testování kódu byl zvolen *pytest* framework [94]. Pro automatické formátování kódu byl zvolen nástroj *black* [95], který dokáže formátovat kód Pythonu nad rámec doporučení PEP-8 [51]. Díky typovým informacím v kódu také bylo možné použít nástroj *pyright* pro typovou kontrolu [96].

Jelikož je projekt hostovaný na GitHubu, byly vytvořeny GitHub Actions [97], které při nahraní nových commitů na GitHub spouští všechny testy a provádí typovou kontrolu. Tyto kontroly jsou prováděny na virtuálním zařízení `ubuntu-latest` (tedy na Linux OS). Některé testy jsou dále označeny jako závislé na platformě. Tyto testy se navíc při každém nahrání commitů spouští na virtuálních zařízeních `windows-latest` a `macos-latest`. Tím je zajištěno testování kódu pro všechny podporované platformy nástroje Woke. Označování testů jako závislých na platformě je však zodpovědností programátora. GitHub Actions se provádí pro nástrojem Woke podporované verze Pythonu `3.7`, `3.8`, `3.9` a `3.10`. V případě vytvoření commitu, který neprochází testy nebo typovou kontrolou, je však vhodné o problému vědět již po vytvoření tohoto commitu. Upozornění až při nahrání commitů na GitHub může být nepraktické v tom, že od vytvoření vadného commitu mohly vzniknout další commity, které mohou z vadného commitu ve svých úpravách vycházet. Také může být náročnější samotné vyhledávání vadného commitu. Z tohoto důvodu byly v projektu vytvořeny *git hooky*.

V adresáři `woke/.githooks` se nachází skripty pro *bash*[1], které jsou spuštěny před vytvořením (v případě souboru `pre-commit`) a po vytvoření (v případě souboru `post-commit`) commitu verzovacího nástroje *git* [98]. Jejich cílem je ve značné míře usnadnit práci vývojáře. Pro jejich správné používání je však nutná pokročilejší znalost nástroje *git*. Nástroj *git* obecně rozeznává soubory, které nejsou do verzovacího systému začleněny (jedná se o *untracked* soubory) a soubory, které jsou verzovacím systémem sledovány. Ve druhé skupině souborů mohou být soubory, které se od poslední revize (od posledního commitu) změnily. Jednotlivé provedené změny buď jsou, nebo nejsou označeny pro začlenění do budoucího commitu. Jestliže změny jsou označeny pro začlenění, obvykle se hovoří, že tyto změny jsou ve *staging area*. Některé změny souboru mohou být ve *staging area*, zatímco jiné změny toho samého souboru nemusí být označeny pro začlenění. Cílem *git hooků* v projektu je spouštění *pytest* testů, nástroje *black* pro formátování kódu a nástroje *pyright* pro typovou analýzu nad kódem, který bude odpovídat poslední revizi po tom, co bude právě vytvářený commit dokončen. Jinak řečeno není žádoucí, aby nástroje prováděly svoji analýzu nad soubory, které jsou v *untracked* části, a nad změnami, které nejsou ve *staging area*. V těchto částech totiž může být kód, který je rozpracovaný a nemusí být typově korektní nebo může způsobit selhání některých testů. Také není vhodné rozpracovaný kód formátovat.

Kód `pre-commit` a `post-commit` hooků je proto poměrně komplexní. `pre-commit` hook je spouštěn před vytvořením commitu. V tomto skriptu je nejprve vytvořen dočasný commit. Není žádoucí, aby pro tento commit byl znovu spuštěn `pre-commit` hook. Toho lze dosáhnout

[1] Je předpokládáno, že většina vývojářů nástroje Woke bude při vývoji používat operační systém založený na Unixu.

přepínačem `--no-verify`[2] příkazu `git commit`. Také je explicitně zakázáno podepsání tohoto dočasného commitu nástrojem GPG[3]. Tímto commitem je zajištěno, že změny, které uživatel chtěl začlenit do verzovacího systému, jsou v této fázi začleněny ve formě dočasného commitu. Následující příkaz totiž odkládá veškeré změny *git* repozitáře (včetně *untracked* souborů) na speciální zásobník *gitu* nazývaný *stash*. Při odkládání na *stash* je vytvořen pseudonáhodný identifikátor o 16 znacích, který je použit jako zpráva při odkládání a také je zapsán do dočasného souboru. Pomocí tohoto identifikátoru je později rozhodováno, zda se má vrchní položka *stash* obnovit či nikoliv[4]. Odložením na *stash* je zaručeno, že je repozitář „vyčištěn" od kódu, který nemá být analyzován. Následující příkaz ruší dočasný commit a jeho změny vrací do *staging area*. Repozitář v této fázi obsahuje všechny uživatelem commitované změny ve *staging area*, žádné dalšího změny ani *untracked* soubory nejsou v repozitáři přítomné. Následně se provádí jednotlivé příkazy pro analýzu kódu. Jestliže je nastavena systémová proměnná `WOKE_HOOKS_RUN_ALL_TESTS`, jsou provedeny všechny testy nástroje *pytest*. V opačném případě jsou spuštěny pouze ty testy, které nejsou programátorem označené jako pomalé (slow). Pomalé testy mohou zásadním způsobem zpomalovat vytváření commitu a tedy i vývoj obecně. Příkladem pomalého testu může být test, který stahuje jedno či více sestavení kompilátoru *solc*. Po provedení testů je spuštěna typová kontrola. Jako poslední je spuštěn příkaz *black* pro formátování kódu. Tento příkaz může vytvářet nové změny, které však nejsou vloženy do *staging area*, takže jsou na logické úrovni nástroje *git* odděleny původní změny uživatele a změny vytvořené nástrojem *black*. Tímto práce `pre-commit` hooku končí.

Následně je vytvořen commit uživatelem. Poté je spuštěn `post-commit` hook. Jeho úkolem je vytvořit další commit obsahující pouze změny vygenerované nástrojem *black* a poté obnovit případné odložené změny uložené na *stash*. Commit se změnami nástroje *black* je vytvářen jako *fixup* commit posledního commitu. *Fixup* commit je snadné pomocí příkazů nástroje *git* sloučit s commitem, na který odkazuje. Očekává se, že změny nástroje *black* budou později sloučené se změnami uživatele do jednoho společného commitu. Použití *fixup* commitu však má několik výhod. Uživatel si může ve *fixup* commitu prohlédnout pouze změny vytvořené nástrojem *black*. Tyto změny je snadné upravit nebo dokonce zcela vyřadit z verzovacího systému. *Fixup* commit je vytvářen s přepínačem `--no-verify`, ale není pro něj explicitně zakázáno podepsání pomocí nástroje GPG. Po dokončení hooků totiž tento commit zůstává ve verzovacím systému, zatímco dočasný commit vytvářený v `pre-commit` hooku by v běžném případě ve verzovacím systému neměl být přítomný. Po vytvoření commitu jsou případně obnoveny změny z *git stash*. Změny na *stash* byly ukládány před spuštěním nástroje *black*, ale repozitář v této fázi již obsahuje změny tohoto nástroje. Důsledkem je, že při aplikování změn uložených na *stash* může dojít k *merging conflictu* nástroje *git*, který musí programátor manuálně řešit. Výhody při používání hooků však při běžném použití tento problém převažují. Díky hookům jsou po každém commitu spuštěny testy, provedena typová kontrola a nový kód je konzistentně zformátován.

4.2.3 Práce s konfiguračními soubory

Jedním ze základních požadavků bylo, aby nástroj Woke uměl číst konfigurační soubory v nějakém běžně používaném formátu. Dalším požadavkem bylo, aby z konfiguračního souboru bylo možné načítat další konfigurační soubory nazývané *subconfigy*. Konfigurační možnosti načtené z těchto souborů by měly mít větší prioritu. Měly by „přepisovat" možnosti v souboru, ze kterého jsou *subconfigy* načítány. Tato funkcionalita je praktická například v situaci, kdy auditor dostane k dispozici projekt s již připraveným konfiguračním souborem pro Woke. Během auditu ale může

[2] Přepínač `--no-verify` je také možné použít v případě, kdy programátor nechce, aby pro jím vytvářený commit byly spuštěny git hooky – tedy testy, typová analýza a formátování kódu.

[3] Dočasný commit je vytvářen bez explicitního vědomí uživatele. Tento commit by se nikdy neměl dostat do veřejného repozitáře, proto podpis pomocí GPG není potřeba nebo může být dokonce považován za nežádoucí. Praktickým důvodem pro zákaz podepsání dočasného commitu může být i fakt, že GPG podpis může být generován prostřednictvím hardwarového zařízení, které může vyžadovat interakci uživatele při každém podpisu.

[4] Jestliže je při odkládání na *stash* seznam změn prázdný, není položka na zásobníku vůbec vytvořena.

být pro auditora vhodné některé konfigurační možnosti upravovat, aniž by musel příliš zasahovat do původního konfiguračního souboru.

Dalším předpokladem bylo, že bude existovat jeden globální konfigurační soubor společný pro všechny projekty a konfigurační soubor umístěný v kořenovém adresáři projektu popisující konfigurační možnosti konkrétního projektu.

Bylo vybíráno z několika formátů konfiguračních souborů:

- `XML`: Specifikace tohoto formátu pochází z roku 1998 [99]. Přestože je formát stále často používaný, je pro účely konfiguračních souborů moderního nástroje nepraktický. XML formát není příliš dobře čitelný a z pohledu psaní konfiguračních souborů člověkem je uživatelsky nepřívětivý.
- `JSON`: Specifikace první verze JSON formátu pochází z roku 2013 [100]. Tento formát je značně využíván. Při použití vhodného formátování (odsazení a zalomení řádků) lze také považovat za dobře čitelný. Nevýhodou je opět uživatelská nepřívětivost při psaní souborů člověkem. Tento formát navíc nepodporuje psaní komentářů.
- `YAML`: První verze YAML formátu pochází z roku 2001 [101]. Formát je velmi dobře čitelný. K formátování strukturovaných dat slouží odsazení pomocí bílých znaků podobně jako je tomu v jazyce Python. Formát je vhodný i z pohledu vytváření souborů člověkem.
- `TOML`: Prvotní verze TOML formátu vznikla v roce 2013 [102]. Data v tomto formátu jsou strukturovaná pomocí sekcí, jejichž názvy jsou zapsané v hranatých závorkách. Formát je velmi dobře čitelný a také je vhodný z hlediska vytváření souborů člověkem.

Vhodnými kandidáty se ukázaly formáty YAML a TOML. Výpisy 4.2 a 4.3 ukazují použití těchto formátů pro jednoduchý konfigurační soubor nástroje Woke.

Výpis kódu 4.2 Ukázka konfiguračního souboru nástroje Woke v YAML formátu

```yaml
subconfigs: ["./config1.yaml", "config2.yaml"]

compiler:
  solc:
    evm_version: berlin
    include_paths: ["./node_modules"]
    remappings: ["@openzeppelin/=node_modules/@openzeppelin/"]
    target_version: 0.8.10
```

Výpis kódu 4.3 Ukázka konfiguračního souboru nástroje Woke v TOML formátu

```toml
subconfigs = ["./config1.toml", "config2.toml"]

[compiler.solc]
evm_version = "berlin"
include_paths = ["./node_modules"]
remappings = ["@openzeppelin/=node_modules/@openzeppelin/"]
target_version = "0.8.10"
```

Jako vhodnější byl zvolen TOML formát ze dvou důvodů. Prvním důvodem je, že odsazení v YAML formátu bývá problematické z pohledu správného formátování. YAML specifikace vyžaduje, aby zanořené bloky byly více odsazeny než blok rodičovský a aby položky v rámci bloku stejné úrovně byly odsazeny stejným počtem mezer [103]. Tato příliš volná specifikace může vést k nedorozumnění nebo nekonzistentnímu používání formátu. Odsazení může být také

nepraktické při větším počtu zanořených bloků. Druhým důvodem je, že by při použití YAML formátu bylo obtížné z pohledu uživatelské příručky vysvětlit, jak funguje přepisování hodnot při načítání subconfigů. Naopak v TOML formátu lze jasně specifikovat, že jsou přepisovány celé hodnoty oddělené znakem `=` (rovná se) od názvu konfigurační možnosti. Toto znamená, že konfigurační možnost, jejíž hodnotou je pole položek, je subconfigem přepsaná celá. Subconfigy pole nerozšiřují o nové položky, ale celá je přepisují novými poli.

Výběr knihovny pro čtení TOML souborů byl velmi ovlivněn PEP 680 [104], kde je navrhováno, že by do standardních knihoven Pythonu měla být začleněna knihovna pro práci s TOML soubory. Součástí návrhu je také seznam knihoven pro práci s TOML soubory implementovaných v jazyce Python. U každé knihovny je uveden důvod jejího zamítnutí. Dále je uvedena knihovna *tomli*, ze které by měla standardní knihovna vycházet. Autor této práce se s názory v PEP 680 ztotožnil, což vedlo k použití knihovny *tomli*.

Implementovaný kód využívá knihovny *pydantic* pro validaci a parsování dat z načtených konfiguračních souborů. Kromě ověření správnosti dat (včetně přebývajících konfiguračních možností, které nejsou očekávány) je díky tomu možné ke konfiguračním možnostem přistupovat pomocí objektů. Také je možné konfigurační možnosti načtené jako řetězce převést na složitější objekty.

Implementace si pro účely slučování konfiguračních možností (přepisování pomocí subconfigů) kromě zparsovaných dat ukládá i původní načtená data ve formě slovníku jazyka Python. Vzhledem ke transformacím, které se při parsování dat provádí, by nebylo možné spolehlivě implementovat slučování dvou sad zparsovaných konfiguračních možností. Při slučování dvou slovníků obsahujících konfigurační možnosti se postupně prochází jednotlivé položky slovníků. Jestliže položka není slovník, je přepsána. Jestliže položka je slovník, je zpracována rekurzivní způsobem.

Při načítání konfiguračních souborů je ukládán orientovaný graf závislostí mezi jednotlivými soubory ve formě config-subconfig. Díky tomu je možné detekovat smyčky při načítání subconfigů. V případě nenalezení konfiguračního souboru je vyvolána hláška s varováním. Woke očekává konfigurační soubor specifický pro daný projekt v kořenovém adresáři projektu pod názvem `woke.toml`. Globální konfigurační soubor je očekáván v cestě:

- ~`/.config/Woke/config.toml` pro *nix systémy,
- `%USERPROFILE%\Woke\config.toml` pro Windows OS.

V konfiguračních souborech nástroje Woke se často pracuje s cestami v souborovém systému. Z pohledu uživatelské přívětivosti je vhodné, aby byly podporované relativní i absolutní cesty nezávisle na platformě. Na práci s cestami souborového systému se v celém projektu nástroje Woke využívá standardní knihovna pathlib [105]. Jelikož subconfigy mohou být umístěné v jiných adresářích než konfigurační soubory, ze kterých jsou načítané, je vhodné, aby byly relativní cesty načítaných souborů vyhodnocovány vzhledem k systémové cestě načítajícího konfiguračního souboru. Tento požadavek koliduje s implementací slučování konfiguračních souborů. Konfigurační soubory jsou slučovány na úrovni základních datových typů Pythonu, kde není zřejmé, která položka je cesta v souborovém systému a která položka je pouze textový řetězec. Tyto informace je možné zjistit z datového modelu využívajícího knihovny *pydantic*. Aby byly relativní cesty v konfiguračních souborech správně vyhodnocovány vzhledem k cestě konfiguračního souboru, je vhodné, aby byly interně ukládány ve formě absolutní cesty. Tuto transformaci je vhodné provádět při načítání daného konfiguračního souboru.

Implementace nástroje Woke pro vyřešení popsaných problémů využívá triku. Před čtením konfiguračního souboru je aktuální pracovní adresář změněn na adresář, ve kterém je soubor uložený. Následně se konfigurační soubor načítá *tomli* knihovnou, která vrací slovník (Python dict). V tomto slovníku jsou systémové cesty uloženy jako řetězce přesně tak, jak byly zapsané v souboru. Následně se slovník parsuje pomocí *pydantic* datového modelu. Pro atributy datového modelu, které odpovídají systémové cestě, jsou definovány *pydantic* validátory, které cestu převádějí na absolutní. Zparsovaný datový model se následně zpět převádí do slovníku.

V tomto slovníku je zaručené, že všechny atributy, které mají být reprezentované jako systémové cesty, jsou cestami absolutními ve formě textového řetězce. Takto upravený slovník se slučuje se slovníkem, který reprezentuje dosud načtené konfigurační možnosti (z jiných souborů).

Výpis 4.4 ukazuje všechny konfigurací možnosti, které je možné v době psaní práce nastavit.

Výpis kódu 4.4 Ukázka všech možností konfiguračního souboru nástroje Woke

```
# cesty k subconfigům, které se mají načíst po tomto souboru
subconfigs = ["config1.toml", "/home/michal/config2.toml"]

[compiler.solc]
# seznam argumentů --allow-paths pro kompilátor solc
allow_paths = ["/usr/lib/solidity_libs"]
# požadovaná verze EVM pro kompilaci pomocí kompilátoru solc
evm_version = "berlin"
# seznam adresářů pro vyhledávání Solidity souborů při kompilaci
include_paths = ["node_modules"]
# seznam mapování názvů Solidity souborů v import příkazech
remappings = ["hardhat/=node_modules/hardhat/"]
# požadovaná verze kompilátoru solc pro kompilaci
target_version = "0.7.6"
```

Díky subconfigům nemusí být zcela jasné, jaké konfigurační možnosti nástroj Woke ve výsledku používá. Proto je implementován příkaz příkazové řádky `woke config`, který na výstupu vypisuje načtené konfigurační možnosti v JSON formátu. Pro účely výstupu je JSON formát přehlednější i proto, že některé konfigurační možnosti mohou být oproti formátu konfiguračního souboru ve výpisu podrobněji strukturované. Pro výpis je navíc využívána knihovna *rich* [106], která pomocí barev výstup ještě více zpřehledňuje. Výpis 4.5 představuje výstup příkazu `woke config` pro konfigurační možnosti uvedené ve výpisu 4.4 za předpokladu, že subconfigy nemění žádnou z uvedených konfiguračních možností. Z výpisu je patrné, že všechny cesty byly převedeny na absolutní a že konfigurační možnost `remappings` je podrobněji rozložena do jednotlivých částí.

Výpis kódu 4.5 Ukázka výstupu příkazu `woke config`

```
{
  "subconfigs":
    ["/home/michal/git/the_graph/config1.toml", "/home/michal/config2.toml"],
  "compiler": {
    "solc": {
      "allow_paths": ["/usr/lib/solidity_libs"],
      "evm_version": "berlin",
      "include_paths": ["/home/michal/git/the_graph/node_modules"],
      "remappings": [
        {
          "context": null,
          "prefix": "hardhat/",
          "target": "node_modules/hardhat/"
        }
      ],
      "target_version": "0.7.6"
    }
  }
}
```

4.2.4 Správce instalací kompilátoru *solc*

Jelikož je kompilátor *solc* distribuován v jednotlivých verzích odpovídajících verzím Solidity, je nutné, aby nástroj Woke uměl spravovat různé verze *solc* kompilátoru. K tomuto účelu slouží modul nástroje Woke s názvem Solc Version Manager (SVM), který zprostředkovává stahování, odebírání a získávání systémové cesty různých verzí *solc* kompilátoru.

Repozitář `https://binaries.soliditylang.org`[5] nabízí staticky sestavené verze kopilátoru *solc* pro všechny nástrojem Woke podporované platformy. Dále jsou k dispozici verze kompilátoru pro webové použití pomocí technologií *asm.js* [107] (pro starší verze kompilátoru) a *WebAssembly* [108] (pro nové verze kompilátoru). Pro každou platformu je k dispozici soubor `list.json`, který detailně popisuje jednotlivá sestavení kompilátoru. Součástí popisu je mimo jiné název souboru odpovídající dané verzi kompilátoru, číslo sestavení (ve formě čísla commitu) či kontrolní součty sestavení ve formě Keccak-256 a SHA-256 hashů. Pro starší verze sestavení kompilátoru využívající technologie *asm.js* jsou k dispozici také *nightly* sestavení. Tato sestavení jsou v sémantickém systému verzování [109] ostře menší než sestavení ve stejné verzi, která přídomek *nightly* nemají. V případě platforem podporovaných nástrojem Woke však *nightly* sestavení k dispozici nejsou.

Nástroj Woke ukládá sestavení kompilátoru do adresáře:

- `~/.config/Woke/compilers` pro *nix systémy,
- `%USERPROFILE%\Woke\compilers` pro Windows OS.

V adresáři je uložený soubor `list.json` z repozitáře pod názvem `solc.json`. Soubor je využíván v případě, kdy zařízení je offline a není možné stáhnout poslední verzi souboru. Informace ze souboru jsou využívány pro určení systémové cesty, kde je (nebo kam má být) sestavení konkrétní verze kompilátoru uloženo. Každá verze kompilátoru je uložena ve svém vlastním podadresáři adresáře `compilers`. Název podadresáře je shodný s názvem spustitelného souboru (až na případnou koncovku) a odpovídá celému názvu sestavení uvedenému v souboru `list.json`. Tímto názvem je například `solc-linux-amd64-v0.8.6+commit.11564f7e` v případě kompilátoru ve verzi `0.8.6` pro Linux OS.

Implementace odstranění dané verze kompilátoru a získání systémové cesty, kde má být daná verze kompilátoru uložena, je poměrně přímočará. Jestliže dříve nebyl stažen nebo načten soubor `list.json`, je provedeno jeho stažení a zparsování pomocí datového modelu knihovny *pydantic*. Jestliže soubor není možné stáhnout z internetu, je použita lokální kopie souboru s názvem `solc.json`. Systémová cesta dané verze kompilátoru je vyhodnocena pomocí obsahu souboru `list.json`.

Instalace verze kompilátoru je z pohledu implementace komplexnější. Funkcionalita je implementována s využitím asynchronního programování jazyka Python. Díky tomu je možné během stahování kompilátoru současně provádět jiný kód. Také je možné stahovat více různých verzí kompilátoru současně. Je zodpovědností programátora, aby nedošlo k vícenásobnému paralelnímu stahování stejné verze kompilátoru. V praxi však nebylo pozorováno zrychlení instalace více verzí kompilátoru pomocí paralelního zpracování oproti sekvenčnímu. Po stažení dané verze kompilátoru jsou kontrolovány oba její kontrolní součty ve formě hashů SHA-256 a Keccak-256. V případě platformy Windows OS jsou starší verze kompilátoru distribuovány společně s DLL knihovnami. Proto je nutné kontrolovat, zda soubor kompilátoru končí příponou `.zip`, a případně obsah souboru extrahovat. Z tohoto důvodu je každá verze kompilátoru umístěna v samostatném adresáři. DLL knihovny přiložené k různým verzím kompilátoru mohou být stejně pojmenovány, ale jejich obsah nemusí být nutně stejný.

Funkcionality SVM modulu jsou běžnému uživateli zprostředkovány pomocí příkazů `woke svm` příkazové řádky. Také je implementován spustitelný soubor (skript) `woke-solc`, který předává

[5] Obsah repozitáře je také k dispozici v GitHub projektu `solc-bin` na adrese `https://github.com/ethereum/solc-bin`.

všechny své argumenty uživateli zvolené verzi kompilátoru *solc*. Konkrétně jsou implementovány tyto příkazy:

- `woke svm install` pro instalaci dané verze kompilátoru,
- `woke svm list` pro výpis nainstalovaných (a případně také všech dostupných) verzí kompilátoru,
- `woke svm remove` pro odinstalaci dané verze kompilátoru,
- `woke svm switch` pro zvolení dané verze kompilátoru příkazu `woke-solc`,
- `woke svm use` pro zvolení a v případě nutnosti instalaci dané verze kompilátoru.

Použití příkazů je dále popsáno v příloze A.2.

4.2.5 Parsování Solidity souborů

Hlavním cílem této práce je implementace modulů pro snadné napojení na LSP server. Nástroj Woke je navržen tak, že informace o Solidity kódu získává z AST formátu, který je výstupem *solc* kompilátoru. Kompilace zdrojových kódů však může být pro větší projekty pomalá. Aby bylo možné kompilaci urychlit, nástroj Woke provádí vlastní analýzu zdrojových souborů pro určení importních závislostí mezi jednotlivými soubory a pro určení verzí kompilátoru, pomocí kterých je možné zdrojové soubory zkompilovat. Z pohledu nástroje Woke je tedy nutné parsování příkazů `pragma solidity` a všech typů příkazu `import`. Všechny ostatní informace mohou být získány z AST formátu po kompilaci.

Parsování Solidity souborů je v nástroji Woke implementováno ve zjednodušené verzi, jelikož není potřeba parsovat celý soubor, ale jen několik příkazů. Po načtení konkrétního zdrojového souboru v Solidity se vytváří BLAKE2b hash, pomocí kterého je později možné určit, zda se obsah souboru změnil či nikoliv. Následně jsou z načteného obsahu souboru (v podobě textového řetězce) odstraněny jednořádkové i víceřádkové komentáře. Tím je zaručeno, že nebudou čteny příkazy, které uvnitř komentáře nemají žádný efekt. Odstraňování komentářů funguje pomocí regulárních výrazů.

Následně jsou z upraveného obsahu souboru získávány informace o podporovaných verzích kompilátoru *solc* a souborech importovaných pomocí příkazů `import`. Nejprve jsou pomocí regulárních výrazů nalezeny základní klíčová slova příkazů. Konkrétně se jedná o tyto regulární výrazy pro vyhledávání příkazů typu `pragma solidity <version>;` a všech variant příkazu `import`:

- `pragma\s+solidity\s+(?P<version>[^;]+)\s*;`,
- `import\s*(?P<import>[^;]+)\s*;`.

Následně je na všechny nálezy regulárního výrazu aplikovaná heuristika, která se snaží zjistit, jestli je nalezený výraz uvnitř textového řetězce jazyka Solidity či nikoliv. Jelikož jazyk Solidity nepodporuje víceřádkové textové řetězce [53], je implementace této heuristiky snadná. Jestliže se nalezený výraz nenachází v textovém řetězci, je zparsovaná jeho zbývající část.

4.2.5.1 Parsování verze Solidity

Dokumentace Solidity k příkazům `pragma solidity` bohužel není příliš podrobná a nelze z ní určit přesnou syntaxi ani sémantiku verzovacích výrazů [50, 56]. Dokumentace se odkazuje na verzování balíčkovacího nástroje *npm*. Brzy však bylo zjištěno autorem této práce, že syntaxe a dokonce ani sémentika verzovacích výrazů kompilátoru *solc* v některých případech neodpovídá dokumentaci nástroje *npm* [58].

Autor této práce přesto vycházel z dokumentace nástroje *npm*, v případě nejasností také čerpal z balíčku *semantic-version*[6] [110] balíčkovacího systému *PyPI* napsaného v jazyce Python. Případné odlišnosti proti implementaci verzovacích výrazů v kompilátoru *solc* byly testovány a v některých případech také ověřovány ve zdrojových souborech kompilátoru [111].

Specifikace sémantického verzování [109] popisuje sémantickou verzi jako povinnou trojici čísel s volitelnými dvěma řetězci ve formátu <major>.<minor>.<patch>-[pre-release]+[build] (například 0.1.3-nightly.2015.9.25+commit.4457170b). Číselné položky major, minor a patch jsou povinné, musí být nezáporné a bez zbytečných úvodních nul. Následovat může volitelný *pre-release tag* oddělený pomlčkou. *Pre-release tag* může být složen z více částí oddělených tečkou. Jednotlivé části mohou být složené z alfanumerických znaků a pomlčky. Nezávisle na přítomnosti či nepřítomnosti *pre-release tagu* může následovat *build tag* oddělený znakem plus. Jeho formát je shodný s formátem *pre-release tagu*. Často se také sémantické verze uvádí s prefixem v (například v0.8.7). Toto však podle specifikace sémantického verzování není správně. Specifikace sémantického verzování nástroje *npm* tuto syntaxi podporuje, implementace kompilátoru *solc* nikoliv. Jednotlivá sestavení kompilátoru *solc* jsou verzována sémantickými verzemi.

Dokumentace balíčkovacího nástroje *npm* popisuje rozsahy sémantických verzí (semantic version ranges) [58]. Definice těchto rozsahů již není uvedena ve specifikaci sémantického verzování [109]. Rozsahy sémantických verzí lze intuitivně vnímat jako nejnižší a nejvyšší podporovanou verzi s tím, že obě hraniční verze mohou i nemusí být v rozsahu zahrnuty. Nejvyšší podporovaná verze nemusí být rozsahem vůbec specifikovaná. Jestliže není uvedena nejnižší podporovaná verze, Woke jako nejnižší verzi považuje verzi 0.0.0, jelikož sestavení kompilátoru *solc* nejsou vydávána v *pre-release* verzích. Výrazy popisující rozsahy sémantických verzí mohou být vzájemně oddělené bílými znaky, v takovém případě se jedná o množinový průnik verzí reprezentovaných rozsahy, nebo dvojicí znaků ||, v takovém případě se jedná o množinové sjednocení verzí reprezentovaných rozsahy. V případě složitejších výrazů má množinový průnik větší prioritu než množinové sjednocení.

Jedním z výrazů popisujících rozsahy verzí jsou částečné verze (partial versions). Ty vznikají vynecháním některých z položek major, minor a patch. Namísto vynechání položky také může být uveden znak x, X nebo *. Může se jednat o následující případy:

- 1.2.x ≡ 1.2 ≡ >=1.2.0 <1.3.0 (chybějící položka patch),
- 1.x.x ≡ 1 ≡ >=1.0.0 <2.0.0 (chybějící položky minor a patch),
- x ≡ X ≡ * ≡ >=0.0.0 (chybějící všechny položky major, minor a patch).

Částečné verze, stejně jako další dále uvedené výrazy, jsou nástrojem Woke reprezentovány ve formě nejnižší a případně také nejvyšší verze rozsahu. V případě obou hraničních verzí implementace navíc rozlišuje, zda je hraniční verze součástí rozsahu či nikoliv. V případě částečných verzí je rozsah popsaný výrazy „větší nebo rovno než“ a „menší než“ v uvedených ekvivalencích. Kompilátor *solc* navíc podporuje uvádění zástupného symbolu x na libovolné pozici – například x.2.3. Tento výraz však nereprezentuje rozsah a není specifikací *npm* popisován. Částečné verze navíc mohou být kombinovány s dalšími operátory – například >=1.2.x. Výraz >=x.2.3 by však nedával dobrý smysl. Z tohoto důvodu nástroj Woke podporuje pouze částečné verze, které mohou být reprezentované rozsahem. Jedná se tedy o tři dříve popsané případy.

Rozsah může být intuitivně reprezentován operátory nerovnosti >, >=, < a <=. Také je možné použít operátor rovnosti =, který je zcela volitelný a sémantika výrazu jím není ovlivněna. Složitější sémantiku má částečná verze zkombinovaná s operátorem nerovnosti:

- >1.2.x ≡ >=1.3.0 (nejpravější číselná hodnota je zvětšena o jedničku a je použit operátor >=),
- >1.x.x ≡ >=2.0.0 (nejpravější číselná hodnota je zvětšena o jedničku a je použit operátor >=),

[6] Balíček *semantic-version* nebyl v nástroji Woke využit právě z toho důvodu, že syntaxe a sémantika verzovacích výrazů zcela neodpovídá implementaci v kompilátoru *solc*.

- >=1.2.x ≡ >=1.2.0 (chybějící části verze jsou doplněny nulami),
- >=1.x.x ≡ >=1.0.0 (chybějící části verze jsou doplněny nulami),
- >=x.x.x ≡ >=0.0.0 (chybějící části verze jsou doplněny nulami),
- <1.2.x ≡ <1.2.0 (chybějící části verze jsou doplněny nulami),
- <1.x.x ≡ <1.0.0 (chybějící části verze jsou doplněny nulami),
- <=1.2.x ≡ <1.3.0 (nejpravější číselná hodnota je zvětšena o jedničku a je použit operátor <),
- <=1.x.x ≡ <2.0.0 (nejpravější číselná hodnota je zvětšena o jedničku a je použit operátor <).

Výrazy typu `>x`, `<x` a `<=x` nejsou nástrojem Woke podporovány (je vyvolána chyba). Ostatní výrazy nástroj Woke interpretuje analogicky k uvedeným ekvivalencím.

Rozsah verzí může být také reprezentován výrazem s pomlčkou a dvěma sémantickými verzemi ve formě `verze1 - verze2`. Tento výraz je ekvivalentní výrazu `>=verze1 <=verze2`. Tato ekvivalence platí i v případech, kdy je alespoň jedna z verzí verzí částečnou. Například tedy platí:

- 1.2.3 - 4.5.6 ≡ >=1.2.3 <=4.5.6,
- 1.x.x - 4.5.6 ≡ >=1.x.x <=4.5.6 ≡ >=1.0.0 <=4.5.6,
- 1.2.3 - 4.5.x ≡ >=1.2.3 <=4.5.x ≡ >=1.2.3 <4.6.0.

Nástroj Woke vždy ve své interní reprezentaci ukládá výrazy, které neobsahují částečné verze. Kompilátor *solc* díky nesprávné implementaci navíc přijímá výrazy, kde `verze1` či `verze2` výrazu `verze1 - verze2` může obsahovat libovolný operátor. Kompilátor *solc* tedy například zpracovává i výraz <=1.2.3 - >=4.5.6. Tento výraz je však vyhodnocen jako rozsah >=1.2.3 <=4.5.6, jelikož implementace kompilátoru *solc* dodatečné operátory ignoruje. Toto lze tvrdit na základě zdrojového kódu kompilátoru *solc*. Relevantní část kódu je uvedena ve výpisu 4.6. Kód členské funkce `parseMatchComponent` byl zkrácen, což je naznačeno na řádce 39. Na řádce 6 se načítá výraz `verze1` pomocí členské funkce `parseMatchComponent`. Z řádků 24 až 38 v tělu funkce je patrné, že funkce také zpracovává operátory před samotným číselným výrazem. Na řádku 7 je podmínka testující, jestli se jedná o rozsah s pomlčkou. Na řádku 11 je načítán výraz `verze2`. Na řádcích 9 a 12 je pak vidět, že nezávisle na dříve načtených operátorech výrazů `verze1` a `verze2` jsou použity operátory `>=` a `<=`. Nástroj Woke toto chybné zpracování výrazu s pomlčkou neimplementuje – v případě výrazu s pomlčkou musí být obě verze uvedeny bez operátorů. Z pohledu implementace nástroje Woke je také velmi důležitá poznámka na řádce 40, díky které je možné předpokládat, že součástí verzovacích výrazů příkazu `pragma solidity` nejsou *pre-release* a *build* tagy.

Výpis kódu 4.6 Úryvek zdrojového kódu kompilátoru *solc* zpracovávající verzovací výraz [111] (zkráceno)

```
1  void SemVerMatchExpressionParser::parseMatchExpression()
2  {
3      // component - component (range)
4      // or component component* (conjunction)
5      SemVerMatchExpression::Conjunction range;
6      range.components.push_back(parseMatchComponent());
7      if (currentToken() == Token::Sub)
8      {
9          range.components[0].prefix = Token::GreaterThanOrEqual;
10         nextToken();
```

```
11            range.components.push_back(parseMatchComponent());
12            range.components[1].prefix = Token::LessThanOrEqual;
13        }
14        else
15            while (currentToken() != Token::Or && currentToken() !=
               ↪ Token::Illegal)
16                range.components.push_back(parseMatchComponent());
17        m_expression.m_disjunction.push_back(range);
18    }
19
20    SemVerMatchExpression::MatchComponent
       ↪ SemVerMatchExpressionParser::parseMatchComponent()
21    {
22        SemVerMatchExpression::MatchComponent component;
23        Token token = currentToken();
24        switch (token)
25        {
26        case Token::BitXor:
27        case Token::BitNot:
28        case Token::LessThan:
29        case Token::LessThanOrEqual:
30        case Token::GreaterThan:
31        case Token::GreaterThanOrEqual:
32        case Token::Assign:
33            component.prefix = token;
34            nextToken();
35            break;
36        default:
37            component.prefix = Token::Assign;
38        }
39        ... // zkráceno
40        // TODO we do not support pre and build version qualifiers for now in
           ↪ match expressions
41        // (but we do support them in the actual versions)
42        return component;
43    }
```

Rozsah sémantických verzí může být také specifikován operátorem ∼. Tento operátor popisuje rozsah, který odpovídá použití libovolné `patch` části verze pokud je `minor` část specifikována. V opačném případě jsou umožněny libovolné hodnoty `minor` a `patch` částí. Platí tedy:

- ∼1.2.3 ≡ >=1.2.3 <1.3.0,
- ∼1.2.x ≡ >=1.2.0 <1.3.0,
- ∼1.x.x ≡ >=1.0.0 <2.0.0.

Posledním operátorem je operátor ˆ. Tento operátor vychází z praxe, kdy k zásadním změnám („breaking“ změnám) softwaru obvykle dochází při změně první nenulové položky z trojice `major`, `minor`, `patch` (v tomto pořadí). Operátor ˆ neumožňuje použití novější verze s „breaking“ změnami:

- ˆ1.2.3 ≡ >=1.2.3 <2.0.0,

- ^0.2.3 ≡ >=0.2.3 <0.3.0,
- ^0.0.3 ≡ >=0.0.3 <0.0.4.

V případě použití částečné verze je sémantika tohoto operátoru komplikovanější. Chybějící části verze jsou nahrazeny nulami. Dále se postupuje podle uvedených pravidel. Pokud je však první nenulová část chybějící část, je pro tuto chybějící část umožněna libovolná hodnota. Platí pak následující ekvivalence:

- ^1.2.x ≡ >=1.2.0 <2.0.0 (jednoduchý případ, kdy je pouze chybějící část nahrazena nulou),
- ^0.0.x ≡ >=0.0.0 <0.1.0 (první nenulová položka je `patch`, která je však vynechána, a proto je pro ni umožněna libovolná hodnota),
- ^0.x.x ≡ >=0.0.0 <1.0.0 (první nenulová položka je `minor`, která je však vynechána, a proto je pro ni umožněna libovolná hodnota).

Jedním z důsledků implementace kompilátoru *solc* je, že je umožněno použití (nebo naopak nepoužití) bílých znaků uprostřed verzovacích výrazů. Například výraz 0.8.7 může být ve kterémkoliv místě rozdělen mezerou, tabulátorem nebo zalomením řádku. Výraz >=0.7.6<=0.9.1 je také validní. Přestože se jedná o velmi netradiční formátování verzovacích výrazů, nástroj Woke parsování těchto výrazů podporuje.

Nástroj Woke nejprve výraz rozděluje podle dvojice znaků `||` na jednotlivé podvýrazy. Podvýraz obsahující pomlčku je vyhodnocen jako rozsah s pomlčkou. V opačném případě jsou v podvýrazu očekávány verze s operátory. Tyto verze s operátory jsou vyhodnoceny podle popsaných pravidel a následně je proveden jejich množinový průnik. Množinové sjednocení rozsahů oddělených znaky `||` není prováděno. Místo toho je výsledkem parsování seznam rozsahů, které jsou odděleny znaky `||`. Toto znamená, že se v seznamu mohou vyskytovat překrývající nebo dokonce vzájemně vnořené rozsahy. Implementace nástroje Woke však kompletní normalizaci zparsovaného výrazu nevyžaduje.

Jednotlivé části verzovacího výrazu jsou parsované pomocí regulárních výrazů. Použitím regulárních výrazů v nástroji Woke však mohou vznikat chyby v podobě nepřesných chybových hlášek. Příkaz `pragma solidity` uvnitř těla kontraktu není podporován a vede k chybě při kompilaci. Nástroj Woke však díky regulárním výrazům tento příkaz rozeznává, což může například v některých případech vést k zavádějící chybové hlášce, že daný projekt není možné pomocí specifikovaných verzí zkompilovat. Použité regulární výrazy jsou velmi složité, proto jsou budovány postupně od jednodušších částí, které jsou pak do složitějších výrazů vloženy jako textové řetězce. To je znázorněno ve výpisu 4.7.

Výpis kódu 4.7 Použití regulárních výrazů pro parsování Solidity verzovacích výrazů v nástroji Woke

```
1  NUMBER = r"x|X|\*|0|[1-9][0-9]*"
2  PARTIAL = (r"(?P<major>{number})\s*(?:\.\s*(?P<minor>{number}))?\s*"
3             r"(?:\.\s*(?P<patch>{number}))?").format(number=NUMBER)
4  PARTIAL_RE = re.compile(r"^\s*{partial}\s*$".format(partial=PARTIAL))
5  PART = r"(?P<operator>\^|~|<|<=|>|>=|=)?\s*{partial}".format(partial=PARTIAL)
6  RANGE_RE = re.compile(r"\s*{part}\s*".format(part=PART))
7  RANGES_RE = re.compile(r"^(\s*{part}\s*)+$".format(part=PART))
```

4.2.5.2 Parsování importních příkazů

Zpracování importních příkazů je mnohem jednodušší. Cílem je totiž získání názvů souborů, na které nejsou kladeny žádné požadavky – může se jednat o libovolný textový řetězec jazyka

Solidity. Nástroj Woke implementuje pro každý typ importního příkazu jeden regulární výraz. Regulární výrazy jsou postupně aplikované, dokud není nalezena shoda. Součástí regulárních výrazů již není klíčové slovo `import`, které bylo zpracováno dříve v kódu. Ze zparsovaného regulárního výrazu je získán pouze název souboru, ze kterého je import proveden. Zbývající informace mohou být snadnějším a spolehlivějším způsobem získány z dat v AST formátu po kompilaci. Výpis 4.8 ukazuje všechny čtyři typy regulárních výrazů. Tyto výrazy jsou opět vytvářeny skládáním jednodušších podvýrazů. Implementace také počítá s tím, že v Solidity textovém řetězci mohou být přitomné escape sekvence `\'` a `\"`.

Výpis kódu 4.8 Použití regulárních výrazů pro parsování Solidity `import` výrazů v nástroji Woke

```
1  FILENAME = r"""(?P<filename>'.*[^\\]'|".*[^\\]")"""
2  SYMBOL = r"[_a-zA-Z][_a-zA-Z0-9]*"
3  ALIAS = r"\s*{symbol}(?:\s+as\s+{symbol})?\s*".format(symbol=SYMBOL)
4  IMPORT_FILENAME_RE = re.compile(r"\s*{filename}\s*".format(filename=FILENAME))
5  IMPORT_AS_FROM_RE = re.compile(
6      r"\s*\*\s*as\s+{symbol}\s+from\s*{filename}\s*".format(
7          filename=FILENAME, symbol=SYMBOL
8      )
9  )
10 IMPORT_AS_RE = re.compile(
11     r"\s*{filename}\s*as\s+{symbol}\s*".format(filename=FILENAME,
    ↪   symbol=SYMBOL)
12 )
13 IMPORT_ALIAS_LIST = re.compile(
14     r"\s*{{{alias}(?:,{alias})*}}\s*from\s*{filename}\s*".format(
15         alias=ALIAS, filename=FILENAME
16     )
17 )
```

4.2.6 Kompilace projektů

Modul pro kompilaci projektu je nejkomplexnějším a pro budoucí použití také nejdůležitějším modulem celého nástroje Woke. Jeho cílem je za pomoci některých uživatelem předaných konfiguračních možností co nejefektivněji zkompilovat libovolně velký projekt. Modul pro kompilaci je rozdělen do submodulu komunikujícího přímo s kompilátorem *solc* (tento submodul je v projektu nazýván *solc frontend*) a samotné logiky zajišťující rozdělení projektu do více volání kompilátoru.

4.2.6.1 Frontend kompilátoru *solc*

Tento submodul zajišťuje vytvoření procesu se správnou verzí kompilátoru *solc* a zprostředkovává komunikaci prostřednictvím dat ve formátu standardního JSON vstupu a výstupu. Data v těchto formátech jsou komunikována standardním vstupem (stdin) a standardním výstupem (stdout) procesu. Součástí tohoto submodulu jsou proto *pydantic* datové modely popisující oba formáty. Datový model výstupu kompilátoru není záměrně zcela kompletní, jelikož součástí výstupu mohou být i data ve formátu AST. Formát AST dokumentací Solidity není vůbec zdokumentovaný. Je proto vhodné AST data při čtení standardního výstupu kompilátoru neparsovat a označit je pro účely *pydantic* datového modelu jako slovník jazyka Python. AST data navíc mohou být velmi rozsáhlá, a tudíž jejich parsování může být časově náročné. Ne vždy jsou zparsovaná AST data po kompilaci potřeba.

Přestože je většina dat kompilátoru komunikována prostřednictvím standardního JSON vstupu, existuje několik argumentů, které je nutné kompilátoru předávat na příkazové řádce. Patří

mezi ně argumenty `--base-path`, `--allow-paths` a `--include-path` [71]. Hodnota argumentu `--allow-paths` je nastavena na čárkou oddělené cesty nastavené pomocí možnosti `allow_paths` v konfiguračním souboru v sekci `[compiler.solc]`. Jedná se o jakousi pojistku pro případ, že by v nástroji Woke byla chyba a volání kompilátoru by končilo neúspěchem díky tomu, že by nebyla explicitně specifikovaná některá systémová cesta pro přístup k importovanému souboru. Argumenty `--base-path` a `--include-path` jsou nastaveny pouze v případě, že je cílová verze kompilátoru alespoň 0.8.8. V nižších verzích totiž argument `--include-path` není podporován. Pro verze kompilátoru 0.8.8 a vyšší je hodnota argumentu `--base-path` vždy nastavena na kořenový adresář projektu, odkud se také spouští proces kompilátoru *solc*. V budoucích verzích kompilátoru má být aktuální adresář výchozí hodnotou argumentu `--base-path` [77]. Nástroj Woke se tedy snaží být konzistentní s budoucími verzemi kompilátoru. Hodnoty argumentů `--include-path` pak přímo odpovídají systémovým cestám předaným prostřednictvím `include_paths` možnosti konfiguračního souboru v sekci `[compiler.solc]`. Díky argumentům `--include-path` je pak možné ve standardním JSON vstupu uvádět relativní cesty k souborům vzhledem k adresářům specifikovaným těmito argumenty. Toto je chování, které má být v budoucích verzích kompilátoru vyžadováno [77]. Naopak v případě použití kompilátoru ve verzi nižší než 0.8.8 není argument `--include-path` k dispozici a cesty ve standardním JSON vstupu musí být absolutní. V tomto případě pak argument `--base-path` také není potřeba a jeho nastavení může dokonce v některých případech způsobovat chyby.

Konfigurační možnost `include_paths` takto zajišťuje konzistentní chování při specifikování knihoven používaných v importních příkazech souborů v projektu. Uživatel nástroje Woke v konfigurační možnosti jednoduše uvede, v jakým systémových cestách jsou přítomné soubory, které nejsou importovány pomocí relativních cest (typicky se jedná o knihovny), a implementace nástroje zajistí, aby byly kompilátoru předány správné parametry nezávisle na verzi kompilátoru.

Proces kompilátoru *solc* je spouštěn pomocí standardní knihovny *asyncio* [112]. Díky tomu je možné snadno spustit více instancí kompilátoru najednou a využít tak potenciálu vícejádrového systému. I z tohoto důvodu je v této části kódu předpokládáno, že potřebná verze kompilátoru *solc* je na systému již nainstalována. Pokud by se požadovaná verze kompilátoru instalovala až nyní, mohlo by paralelně probíhat několik instalací stejné verze kompilátoru, což není žádoucí.

4.2.6.2 Analýza projektu před kompilací

Základní myšlenkou efektivní kompilace projektu pro účely LSP serveru i pro použití nástroje Woke jako samostatné aplikace je rozdělení Solidity souborů projektu na nejmenší, ne nutně disjunktní, podmnožiny, které je možné nezávisle na sobě zkompilovat. Tyto podmnožiny jsou v kódu nazývané kompilačními jednotkami (compilation units). Kompilačními jednotkami nejsou podmnožiny souborů, které by byly podmnožinou jiné kompilační jednotky. Díky kompilačním jednotkám je možné v případě úpravy kódu překompilovat jen nezbytně nutné soubory, což může vést ke značné úspoře času. Navíc rozdělením na kompilační jednotky vzniká více úloh pro kompilaci, které mohou být efektivně paralelně zpracovány na vícejádrovém systému. V přípravné fázi kompilace je cílem identifikace kompilačních jednotek.

Vstupem pro kompilaci je seznam cest k Solidity souborům, které mají být zkompilovány. Tento seznam nemusí být úplný, pro kompilaci mohou být potřeba další neuvedené soubory. Prvním krokem je identifikace všech potřebných souborů a vytvoření orientovaného grafu, kde uzly reprezentují jednotlivé Solidity soubory a hrana od uzlu `a.sol` k uzlu `b.sol` znamená, že je soubor `a.sol` importován souborem `b.sol`. Narozdíl od jiných programovacích jazyků mohou být importy cyklické, a přesto může kompilace proběhnout v pořádku. V generovaném grafu tedy mohou být cykly.

Před analýzou souborů předaných jako argument modulu pro kompilaci je potřeba vyhodnotit *source unit name* těchto souborů. Vyhodnocování řetězců *source unit name* probíhá přesně podle popisu v kapitole 2.2.1.5. V logice vyhodnocování *source unit name* seznam souborů předaný kompilačnímu modulu odpovídá argumentům příkazové řádky. Jejich *source unit name* se proto

vyhodnocuje jako relativní cesta od kořenového adresáře projektu. Soubory a jejich *source unit name* jsou přidány do fronty pro další zpracování.

Fronta zpracovává všechny soubory, které byly kompilačnímu modulu předány přímo, a také rekurentně všechny importované soubory. Modul pro parsování Solidity souborů umožňuje získat seznam textových řetězců reprezentujících názvy souborů v importech společně s BLAKE2b hashem obsahu souboru a seznamem vhodných verzí kompilátoru pro kompilaci daného souboru. Hash je společně se seznamem verzí uložen do uzlu stromu pro pozdější použití. Řetězce z importů musí být převedeny na konkrétní systémové cesty, a tedy konkrétní Solidity soubory. Je navíc nutné, aby tento převod probíhal přesně podle dokumentace kompilátoru *solc*, aby uživatel nástroje Woke mohl předpokládat stejné chování jako při přímém použití kompilátoru.

Převod importního řetězce na cesty k Solidity souborům je rozdělen do dvou částí. Nejprve je pro importní řetězec vyhodnocen *source unit name* s případným využitím *source unit name* importujícího souboru. Součástí vyhodnocení *source unit name* je také případná aplikace remappingu. Poté je z řetězce *source unit name* importovaného souboru vyhodnocena cesta k Solidity souboru. Toto vyhodnocení odpovídá postupu popsanému v kapitole 2.2.1.6. Řetězci *source unit name* importovaného souboru jsou postupně předřazeny cesty k adresářům specifikovaným v konfigurační možnosti `include_paths`. Také je předřazena cesta ke kořenovému adresáři projektu. Jestliže po předřazení žádná nebo více cest odkazuje na existující soubor, jedná se o chybu kompilace. V opačném případě je nalezena systémová cesta k Solidity souboru. Importovaný soubor je přidán do fronty ke zpracování, pokud již dříve nebyl zpracován. Do grafu je také přidána hrana mezi importující soubor a importovaný soubor. Jelikož je konkrétní Solidity soubor zpracován frontou nejvýše jednou, nemůže pro jeden soubor existovat více řetězců *source unit name* – je vždy použit ten první. Naopak pro více různých souborů by mohl existovat stejný řetězec *source unit name*. V takovém případě se jedná o chybu kompilace.

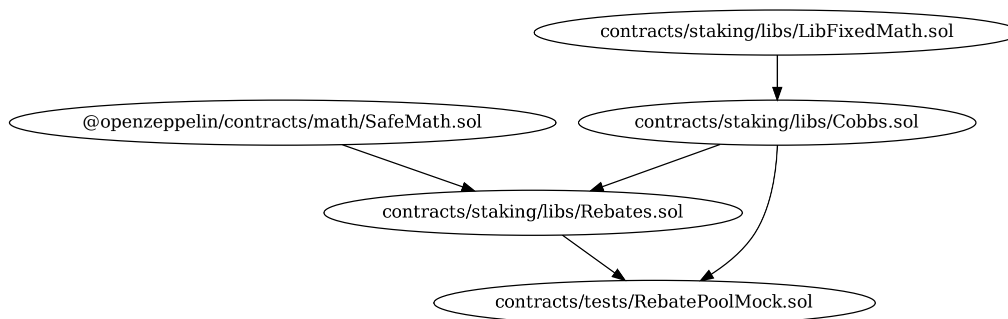
Po zpracování celé fronty jsou nalezeny všechny soubory, které jsou ke kompilaci potřeba. Ze souborů je vytvořen orientovaný graf na základě importních příkazů. V grafu je navíc uložen BLAKE2b hash obsahu souboru, *source unit name* souboru a seznam verzí kompilátoru *solc*, pomocí kterých je možné soubor zkompilovat.

Posledním krokem analýzy projektu je detekce kompilačních jednotek na základě vytvořeného grafu. Analýza kompilačních jednotek začíná vyhledáním uzlů nazývaných *stok*[7] (sink). Jedná se o soubory, které nejsou žádným jiným souborem importované. Každý z těchto uzlů reprezentuje jednu kompilační jednotku. Celá kompilační jednotka je nalezena postupným průchodem grafu od stoku v opačném směru hran (po vstupních hranách). Dále jsou v grafu detekovány cykly, pro něž platí, že součet počtů výstupních hran uzlů cyklu je roven počtu uzlů v cyklu. Lze si tento cyklus neformálně představit jako kružnici, do kterého mohou vést další hrany, ale žádné hrany nemohou vést mimo tuto kružnici. Tento typ cyklů je také základem pro kompilační jednotku. Celá kompilační jednotka je opět získána průchodem grafu od uzlů cyklu ve směru vstupních hran. Během detekce kompilační jednotky jsou také vyhodnoceny verze kompilátoru *solc*, pomocí kterých je možné kompilační jednotku zkompilovat. Jedná se o množinový průnik podporovaných verzí jednotlivých souborů. Na obrázku 4.1 je znázorněn příklad analýzy jedné kompilační jednotky nástrojem Woke. Text uvnitř uzlů je *source unit name* souborů.

4.2.6.3 Kompilace kompilační jednotky a artefakty sestavení

Po analýze kompilačních jednotek je pro každou jednotku určena verze kompilátoru, pomocí které bude jednotka kompilována. Jestliže uživatel zadal požadovanou verzi kompilátoru v konfiguračním souboru, jsou všechny jednotky kompilovány pomocí této verze. V opačném případě se pro každou kompilační jednotku použije poslední verze ze seznamu verzí, pomocí kterých je možné kompilační jednotku zkompilovat. Požadovaná verze kompilátoru je případně nainstalována pomocí SVM modulu.

[7] Jedná se o uzly, jejichž počet výstupních hran je roven nule.



Obrázek 4.1 Příklad analýzy kompilační jednotky nástrojem Woke

Pro každou kompilační jednotku je vytvořena asynchronní úloha. Jejím úkolem je použití předchozích artefaktů sestavení nebo zavolání *solc* frontendu a uložení nově vytvořených artefaktů sestavení (pokud jsou požadovány). Mezi artefakty sestavení patří veškerý výstup kompilátoru *solc*, aby bylo možné tento výstup znovu zreprodukovat, aniž by byl kompilátor volán. Dále jsou mezi artefakty sestavení ukládány informace, na základě kterých je možné určit, zda je možné při kompilaci použít předchozí artefakty sestavení, nebo ne. Pro každou kompilační jednotku je vypočítán BLAKE2b hash ze zřetězených hashů jednotlivých souborů seřazených podle jejich *source unit name*. Hash kompilační jednotky je první kontrolou, zda je možné předchozí artefakty sestavení použít. Tento hash zaručuje, že se obsah souborů od posledního sestavení nezměnil. Dále je testováno, zda se od poslední kompilace nezměnilo nastavení kompilátoru včetně argumentů příkazové řádky. V rámci tohoto testování jsou porovnávány řetězce *source unit name*, což ve výsledky zahrnuje kontrolu změny názvů souborů. Artefakty sestavení jednotlivých kompilačních jednotek jsou zapisovány paralelně. Po úspěšné kompilaci všech kompilačních jednotek je vytvořen jeden soubor popisující artefakty jednotlivých kompilačních jednotek.

4.2.7 Parsování AST dat

Implementace modulu pro parsování dat v AST formátu nebyla součástí této práce – modul byl již implementován. Implementace využívá datového modelu knihovny *pydantic*. Model byl vytvořen na základě popisu AST formátu *npm* balíčkem *solidity-ast* [113]. Pro AST formát jazyka Solidity totiž neexistuje oficiální dokumentace. I z tohoto důvodu nemusí být datový model AST zcela správný a chyby jsou opravovány postupně během testování na různých projektech. Význam jednotlivých datových položek AST formátu je alespoň částečně vysvětlen v komentářích zdrojového kódu projektu *solc-typed-ast* [114].

4.2.8 Interakce s příkazovou řádkou

Pro práci s příkazovou řádkou bylo vybíráno z několika knihoven:

- *argparse* [115],
- *docopt* [116],
- *click* [117].

Knihovna *argparse* je standardním modulem jazyka Python. Umožňuje specifikaci jednotlivých argumentů příkazové řádky včetně očekávaného datového typu, počtu argumentů stejného

typu či textové nápovědy. Na základě nakonfigurovaných argumentů knihovna umožňuje vypsat nápovědu popisující všechny argumenty včetně jejich povinnosti, počtu a krátkého popisu.

Knihovna *docopt* volí opačný způsob. Programátor specifikuje textovou nápovědu podle stanoveného formátu ve formě dokumentačního textového řetězce jazyka Python. Knihovna tento dokumentační řetězec parsuje a na základě toho přiřazuje hodnoty předané na příkazové řádce konkrétním argumentům. Poslední commit knihovny *docopt* v době psaní této práce však pochází z roku 2018 a na základě historie repozitáře lze obecně usuzovat, že knihovna již není aktivně vyvíjena.

Knihovna *click* byla v nástroji Woke na základě analýzy všech tří knihoven použita. V době psaní této práce je stále aktivně vyvíjena. Podobně jako knihovna *argparse* umožňuje detailní specifikaci jednotlivých parametrů. Konfigurace parametrů je však o něco detailnější. Hlavní výhodou knihovny *click* je však rozdělení příkazů na jednotlivé podpříkazy ve formě skupin. Pro každou skupinu příkazů i každý samotný příkaz knihovna očekává funkci, která daný příkaz nebo skupinu příkazů zpracovává. Zparsované argumenty jsou předané programátorem implementované funkci. Knihovna také umožňuje velmi efektivní testování příkazů příkazové řádky.

Výpis 4.9 ukazuje použití knihovny pro definici příkazu `woke compile`. Příkaz očekává libovolný počet cest k souborům, pro které knihovna *click* dovoluje automaticky zkontrolovat, zda existují. Dále jsou specifikovány přepínače s hodnotou `True/False`, pro které je možné nastavit výchozí hodnotu. Knihovna také umožňuje předání kontextu od nadřazené skupiny příkazů. V tomto případě je kontext předán od funkce zpracovávající skupinu příkazů `woke`. Díky tomu je možné společné části kódu provést v nadřazené funkci.

Výpis kódu 4.9 Použití knihovny *click* v příkazu `woke compile`

```
1  @click.command(name="compile")
2  @click.argument("files", nargs=-1, type=click.Path(exists=True))
3  @click.option(
4      "--parse",
5      is_flag=True,
6      default=False,
7      help="Also try to parse the generated AST."
8  )
9  @click.option(
10     "--no-artifacts",
11     is_flag=True,
12     default=False,
13     help="Do not write build artifacts."
14 )
15 @click.option(
16     "--force",
17     is_flag=True,
18     default=False,
19     help="Force recompile the project without previous build artifacts.",
20 )
21 @click.pass_context
22 def run_compile(
23     ctx: Context, files: Tuple[str], parse: bool, no_artifacts: bool, force:
       ↪  bool
24 ) -> None:
25     """Compile the project."""
26     config = WokeConfig(woke_root_path=ctx.obj["woke_root_path"])
```

4.3 Testování implementace

Implementace nástroje Woke byla testována pomocí *pytest* frameworku [94]. Tento framework byl zvolen mimo jiné proto, že je využíván v nástroji *Brownie* [118], takže s ním má *Ethereum* komunita zkušenosti. Testy byly rozděleny do jednotlivých souborů podle testovaných funkcionalit. V každém souboru bylo vytvořeno několik testovacích případů.

Testy byly implementovány na základě požadavků uvedených v předchozích kapitolách (práce s konfiguračními soubory, správce verzí *solc*), podle dokumentace jazyka Solidity [6] (parsování importních příkazů a vyhodnocování cest souborů z těchto příkazů) a na základě analýzy a testů autora této práce (parsování verzovacích příkazů).

V případě testů kompilačního modulu byly zvoleny čtyři komplexní Solidity projekty, jejichž kód je volně dostupný na GitHubu. Projekty byly voleny tak, aby pokryly co nejvíce různých scénářů při kompilaci, aby obsahovaly značné množství Solidity souborů s různými funkcionalitami a aby se jednalo o populární projekty, u nichž je možné očekávat, že budou v budoucnu dále udržovány. Byly zvoleny tyto projekty:

- Uniswap v3 [119],
- The Graph [120],
- Trader Joe [121],
- Axelar [122].

V rámci každého ze čtyř testů projektů se pomocí nástroje *git* klonuje repozitář s kódem z hlavní vývojové větve. Dále se generuje konfigurační soubor nastavující adresář pro dodatečné knihovny pomocí konfigurační možnosti `include_paths = ["./node_modules"]`. Obvykle se jedná o jedinou nutnou konfigurační možnost pro kompilaci Solidity projektu. Dále se v naklonovaném repozitáři instalují případné *npm* balíčky pomocí příkazu `npm install`. Následně je testovaná samotná kompilace, kdy jsou nejprve vyhledány všechny Solidity soubory projektu. Poté se několikrát provádí kompilace. Při prvním spuštění je nutné celý projekt zkompilovat. Druhý příkaz využívá artefakty sestavení – je testováno jejich správné čtení a použití. Poslední kompilace především testuje volání příkazu `woke compile`. První dvě volání totiž přímo využívají funkce modulu pro kompilaci.

4.3.1 Vyhodnocení pokrytí implementace testy

Dále bylo zkoumáno pokrytí kódu implementace nástroje Woke pomocí testů. K tomu byl využit nástroj *pytest-cov* [123], který po provedení testů dokáže pro každý soubor jazyka Python vyhodnotit, jaké příkazy nebyly v žádném z testů vykonány. Nejzajímavějším údajem je procentuální pokrytí každého souboru testy.

Výpis 4.10 popisuje výstup nástroje *pytest-cov* při spuštění všech testů na Windows OS s funkčním síťovým připojením. Pro většinu souborů je toto pokrytí velmi dobré. V případě nepokrytých příkazů se většinou jednalo o části kódu, které nebylo možné rozumným způsobem pokrýt. Jednalo se o příkazy, jejichž výstup nelze zcela jistě predikovat, nebo příkazy, které byly určeny pro jinou platformu (Linux OS a macOS). Také se jednalo o příkazy ošetřující chybu, kterou je obtížné vyvolat (například špatné připojení k internetu). Vyhodnocení pokrytí kódu testy bylo záměrně provedeno na Windows OS, jelikož pro tuto platformu je v projektu nejvíce kódu závisejícího na platformě. Modul pro práci s příkazovou řádkou bylo možné snadno testovat díky použité knihovně *click*, která nabízí podporu pro testování implementace využívající právě této knihovny. Takto mohly být otestovány příkazy `woke compile` a `woke svm`. Naopak by bylo velmi obtížné otestovat příkaz `woke-solc`, který knihovnu *click* nevyužívá. To vysvětluje nízké pokrytí souboru `woke\x_cli__main__.py`. Z vyhodnocení pokrytí byly vyřazeny soubory modulu `woke.e_ast_parsing`, které nejsou součástí této práce.

Výpis kódu 4.10 Pokrytí kódu nástroje Woke testy

```
---------- coverage: platform win32, python 3.10.4-final-0 -----------
Name                                                   Stmts   Miss  Cover
--------------------------------------------------------------------------
woke\__init__.py                                           0      0   100%
woke\a_config\__init__.py                                  2      0   100%
woke\a_config\data_model.py                               56      0   100%
woke\a_config\woke_config.py                             106     11    90%
woke\b_svm\__init__.py                                     1      0   100%
woke\b_svm\abc.py                                         19      0   100%
woke\b_svm\exceptions.py                                   2      0   100%
woke\b_svm\svm.py                                        173     15    91%
woke\c_regex_parsing\__init__.py                           1      0   100%
woke\c_regex_parsing\solidity_import.py                   21      0   100%
woke\c_regex_parsing\solidity_parser.py                   61      0   100%
woke\c_regex_parsing\solidity_version.py                 440     24    95%
woke\core\__init__.py                                      0      0   100%
woke\core\enums.py                                        11      0   100%
woke\d_compile\__init__.py                                 3      0   100%
woke\d_compile\build_data_model.py                        19      0   100%
woke\d_compile\compiler.py                               266     52    80%
woke\d_compile\exceptions.py                               4      0   100%
woke\d_compile\solc_frontend\__init__.py                   3      0   100%
woke\d_compile\solc_frontend\exceptions.py                 2      0   100%
woke\d_compile\solc_frontend\input_data_model.py         166      0   100%
woke\d_compile\solc_frontend\output_data_model.py        148      0   100%
woke\d_compile\solc_frontend\solc_runner.py               43      3    93%
woke\d_compile\source_path_resolver.py                    22      5    77%
woke\d_compile\source_unit_name_resolver.py               61      1    98%
woke\l_lsp\__init__.py                                     0      0   100%
woke\utils\__init__.py                                     1      0   100%
woke\utils\cwd_changer.py                                 11      0   100%
woke\x_cli\__init__.py                                     0      0   100%
woke\x_cli\__main__.py                                    55     21    62%
woke\x_cli\compile.py                                     42      6    86%
woke\x_cli\console.py                                      2      0   100%
woke\x_cli\svm.py                                         81      1    99%
--------------------------------------------------------------------------
TOTAL                                                   1822    139    92%
```

4.3.2 Vyhodnocení rychlosti kompilace

Velmi zajímavým ukazatelem byla rychlost kompilace rozsáhlejších Solidity projektů oproti již existujícím nástrojům. Mezi porovnávané nástroje byly zařazeny:

- Brownie framework [118] ve verzi 1.18.1,
- Hardhat framework [124] ve verzi 2.5.0,
- Truffle framework [125] ve verzi 5.5.10,
- nástroj Foundry [126] ve verzi 0.2.0.

Tabulka 4.1 Informace o Solidity projektech zvolených pro testování kompilace

	Uniswap v3	The Graph	Trader Joe	Axelar
Počet kompilačních jednotek	28	24	40	5
Počet Solidity souborů	62	69	72	24
Počet kompilovaných Solidity souborů	201	225	277	43
Počet kontraktů v projektu	62	95	134	24
Počet kompilovaných kontraktů	201	227	329	43

Nástroj Slither do porovnání nebyl zařazen, jelikož neposkytuje samostatnou funkcionalitu kompilace projektu. Měření již z principu nemohlo být provedeno zcela objektivním způsobem. Nástroj Woke při kompilaci využívá paralelizace, zatímco ostatní nástroje pomocí argumentů příkazové řádky možnosti paralelizace nenabízí. Každý nástroj může mít také výstupem kompilace jinou množinu dat uloženou v jinak strukturovaných souborech. Nástroj *Hardhat* navíc jako součást kompilace generuje datový model v jazyce TypeScript odpovídající jednotlivým Solidity kontraktům. Tuto dodatečnou funkcionalitu bylo však možné pro účely měření času přepínačem na příkazové řádce vypnout.

Pro porovnání časů kompilace bylo využito stejných projektů jako při testování kompilačního modulu:

- Uniswap v3 [119],
- The Graph [120],
- Trader Joe [121],
- Axelar [122].

V případě projektu *Axelar* bylo potřeba přejmenovat adresář `src` na `contracts`, jelikož nástroje na základě konvencí očekávají zdrojové soubory jazyka Solidity v adresáři s tímto názvem.

Tabulka 4.1 popisuje podrobnější informace o jednotlivých projektech z pohledu kompilace nástrojem Woke. Z tabulky je na první pohled patrné, že projekt *Axelar* není oproti ostatním projektům příliš rozsáhlý. Do testování byl zařazen především proto, že pro kompilaci vyžaduje kompilátor ve verzi `0.8.9`, což umožňuje otestování jiného postupu při kompilaci nástrojem Woke. Také je možné na tomto projektu vyhodnotit rychlost kompilace méně rozsáhlých projektů. Počet kompilačních jednotek přímo odpovídá počtu volání kompilátoru *solc*. Počet kompilovaných Solidity souborů je výrazně vyšší oproti počtu Solidity souborů v projektu. Je to dáno tím, že jeden soubor může být kompilován vícekrát v rámci více kompilačních jednotek. Obdobný vztah platí v případě počtu kompilovaných kontraktů a celkového počtu kontraktů v projektu. Z dat vyplývá, že každý soubor projektu je v průměru zařazen do tří až čtyř kompilačních jednotek. Každému kompilovanému Solidity souboru a každému kompilovanému kontraktu navíc odpovídá jeden soubor v artefaktech sestavení. Z tabulky lze tedy usuzovat, že je v artefaktech sestavení ukládáno „zbytečně“ trojnásobné množství potřebných dat. V praxi je ale deduplikace těchto dat komplikovaná, neboť soubory obsahují identifikátory, které závisí na konkrétním běhu kompilace. Jelikož jsou kompilační jednotky kompilovány nezávisle na sobě, identifikátory pro stejný soubor mohou být (a velmi často skutečně jsou) zcela rozdílné ve výstupech kompilací jednotlivých kompilačních jednotek. Sjednocení identifikátorů po kompilaci je pak obtížné. Po sjednocení identifikátorů by se však zjednodušila práce LSP serveru a také by se urychlila kompilace, jelikož by bylo zapisováno do méně souborů – artefaktů sestavení.

Tabulka 4.2 Porovnání rychlosti kompilace nástroje Woke s jinými nástroji v sekundách

	Uniswap v3	The Graph	Trader Joe	Axelar
Woke	$3{,}05 \pm 0{,}06$	$3{,}53 \pm 0{,}05$	$5{,}80 \pm 0{,}04$	$2{,}37 \pm 0{,}04$
Brownie	$5{,}88 \pm 0{,}04$	$9{,}74 \pm 0{,}08$	–	$4{,}45 \pm 0{,}07$
Hardhat	$4{,}00 \pm 0{,}05$	$7{,}27 \pm 0{,}06$	$10{,}90 \pm 0{,}07$	$1{,}42 \pm 0{,}01$
Truffle	$5{,}71 \pm 0{,}05$	$10{,}24 \pm 0{,}05$	–	$6{,}31 \pm 0{,}05$
Foundry	$1{,}75 \pm 0{,}01$	$3{,}40 \pm 0{,}07$	$6{,}28 \pm 0{,}05$	$2{,}03 \pm 0{,}00$

Měření bylo prováděno na notebooku s procesorem `AMD Ryzen 5 5600H`, 16 GB paměti RAM, operačním systémem `5.17.1-3-MANJARO #1 SMP PREEMPT x86_64 GNU/Linux`. Pro každou konfiguraci nástroj-projekt bylo provedeno pět měření. Z časů měření byl spočítán průměr a absolutní chyba měření. Kompilace byla prováděna tak, aby bylo zaručeno, že je při jednotlivých měřeních vždy odznovu kompilován celý projekt. Konkrétně byly jednotlivé nástroje spuštěny s těmito přepínači:

- `$ woke compile --force`,
- `$ brownie compile --all`,
- `$ npx hardhat compile --force --no-typechain`,
- `$ truffle compile --all`,
- `$ forge build --force`.

Před měřením každé konfigurace nástroj-projekt byla vždy několikrát spuštěna kompilace bez měření času. Tím bylo zaručeno, že se data projektu „ustálila“ v paměťovém subsystému a výsledky měření byly více stabilní.

Výsledky jsou znázorněny v tabulce 4.2. Z výsledků je především zajímavé porovnání časů kompilace s nástrojem Brownie, který je, stejně jako nástroj Woke, napsaný v jazyce Python [118]. Projekt Foundry je napsaný v jazyce Rust [126]. Jedná se kompilovaný jazyk zaměřený na vysoký výkon a bezpečnou práci s pamětí. Bylo tedy očekáváno, že oproti nástroji Woke bude kompilace nástrojem Foundry velmi rychlá. Z výsledků v tabulce je však patrné, že Woke dokáže tomuto nástroji konkurovat, což je dle názoru autora této práce zapříčiněno především paralelizací kompilace. Zbývající nástroje *Hardhat* a *Truffle* jsou napsané v jazyku TypeScript [124][125]. Projekt *Trader Joe* nebylo možné zkompilovat nástrojem Brownie, jelikož projekt obsahuje dva kontrakty se stejným názvem. Toto není nástrojem Brownie podporováno [127]. Stejný projekt nebylo možné zkompilovat ani nástrojem Truffle. Tento projekt totiž vyžaduje kompilaci pomocí alespoň dvou různých verzí kompilátoru *solc*. Nástroj Truffle se však pro tento typ kompilace nepodařilo nakonfigurovat.

Při měření časů kompilace bylo zjištěno, že zatímco ostatní nástroje používaly nativní sestavení kompilátoru *solc* pro Linux OS, nástroj Truffle pro kompilaci využíval kompilátor určený pro platformu WebAssembly. Toto může být důvod pro vysoké naměřené časy nástroje Truffle.

Závěr

Cílem této práce byla analýza stávajícího stavu implementace nástrojů Woke a Slither, rozšíření nástroje Woke o další funkcionality a otestování správnosti implementovaných funkcionalit. V práci jsem provedl analýzu nástroje Slither. Také jsem ve formě modulu nástroje Slither implementoval detektor, který může odhalit potenciální zranitelnost ve formě neupřesněné asociativity operace umocňování v programovacím jazyku Solidity. Dále jsem provedl analýzu současného stavu implementace nástroje Woke a na základě domluvy s vedoucím této práce navrhl funkcionality, o které jsem Woke rozšířil. V práci jsem popsal postup implementace funkcionalit, otestoval jsem správnost těchto funkcionalit a také jsem porovnal efektivitu implementace proti stávajícím řešením.

Po domluvě s vedoucím této práce jsem nástroj Woke rozšířil o moduly, které umožňují kompilaci libovolně rozsáhlého Solidity projektu. Při implementaci jsem kladl důraz na to, aby bylo možné moduly použít v LSP serveru i v nástroji Woke jako programu pro příkazovou řádku. Mezi implementované moduly patří modul pro čtení konfiguračních souborů, který zajišťuje validaci uživatelem zadaných dat a implementuje funkcionalitu, díky které může být z konfiguračního souboru načteno více dalších konfiguračních souborů, které mohou efektivně pozměňovat konfigurační hodnoty původního souboru. Dále jsem implementoval modul pro správu verzí kompilátoru *solc*. Funkce tohoto modulu jsou uživateli zpřístupněny pomocí příkazové řádky. Také jsem implementoval modul pro parsování Solidity souborů, který umožňuje extrakci informací o podporovaných verzích kompilátoru a importovaných souborech. Na základě těchto informací modul pro kompilaci detekuje všechny soubory projektu a vazby mezi nimi ve formě importních příkazů. Na základě orientovaného grafu reprezentujícího importní příkazy je projekt rozdělen na menší samostatné kompilační jednotky, které mohou být zkompilovány nezávisle na sobě pomocí paralelního zpracování. Součástí kompilačního modulu je také implementace zápisu a čtení artefaktů sestavení, což umožňuje znovupoužití informací z předcházející kompilace projektu.

V práci jsem také popsal přípravu vývojového prostředí, která umožnila mnohem efektivnější vývoj díky konzistentnímu formátování kódu, typové kontrole a automatickému spouštění testů modulů. Dále jsem popsal metodiku testování správnosti modulů včetně způsobu výběru Solidity projektů pro otestování nejvýznamějšího modulu – modulu pro kompilaci projektu. Pomocí vybraných projektů jsem otestoval efektivitu implementace oproti Ethereum komunitou běžně používaným nástrojům. Na základě výsledků usuzuji, že nástroj Woke může z pohledu rychlosti kompilace konkurovat nejefektivnějším nástrojům.

Navazující práce

Během implementace modulů nástroje Woke jsem identifikoval další funkcionality, které by zásadním způsobem rozšířily stávající možnosti nástroje. Některé z těchto funkcionalit jsou popsány v příloze B.

Příloha A

Použití nástroje Woke

A.1 Instalace

Nástroj Woke v době psaní této práce není distribuován prostřednictvím žádného balíčkovacího systému. Je tedy nutné provést instalaci ze zdrojového kódu. Nástroj Woke je možné používat na libovolné z platforem Linux OS, Windows OS a macOS. Podporované verze jazyka Python jsou 3.7, 3.8, 3.9 a 3.10. Zdrojový kód je dostupný na adrese `https://github.com/Ackee-Blockchain/woke/tree/0d27de25720142beb9619a89619b7a94c3556af1`. Po stažení kódu je nutné přepnout pracovní adresář do adresáře `woke` uvnitř projektu[1]. Tento adresář obsahuje veškerý zdrojový kód, který je součástí této práce.

Při instalaci je výrazně doporučeno využití nástroje *virtualenvwrapper* [128] nebo Pythonem vestavěného modulu *venv* [129]. Díky použití virtuálního prostředí může být zamezeno případným konfliktům s jinými již nainstalovanými balíčky.

V případě použití nástroje *virtualenvwrapper* na *nix systému probíhá příprava prostředí pomocí následujících příkazů[2]:

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv woke
```

Pro pozdější přepnutí do virtuálního prostředí je možné využít příkaz `workon`:

```
$ workon woke
```

V případě použití modulu *venv* je příprava prostředí jednodušší:

```
$ python -m venv venv
$ source venv/bin/activate
```

Druhý z uvedených příkazů je možné použít kdykoliv později pro přepnutí do virtuálního prostředí.

[1] Adresářová struktura projekt je poměrně komplexní. Správný adresář `woke` lze rozeznat tak, že obsahuje soubor `setup.py`.

[2] Návod vychází z dokumentace nástroje *virtualenvwrapper*. Příkazy může být nutné upravit v závislosti na konkrétním systému.

Instalace probíhá z adresáře `woke` obsahujícího soubor `setup.py`. Instalace je spuštěna tímto příkazem:

```
$ pip install -e ".[tests,dev]"
```

Po instalaci by měl být dostupný příkaz `woke`:

```
$ woke
Usage: woke [OPTIONS] COMMAND [ARGS]...

Options:
  --woke-root-path PATH  Override Woke root path.
  --debug / --no-debug
  --help                 Show this message and exit.

Commands:
  compile  Compile the project.
  config   Print loaded config options in JSON format.
  svm      Run Woke solc version manager.
```

A.2 Použití správce verzí kompilátoru *solc*

Pro správu různých verzí kompilátoru *solc* slouží příkaz `woke svm`:

```
$ woke svm
Usage: woke svm [OPTIONS] COMMAND [ARGS]...

  Run Woke solc version manager.

Options:
  --help  Show this message and exit.

Commands:
  install  Install the latest solc version matching the given version range.
  list     List installed solc versions.
  remove   Remove the target solc version.
  switch   Switch to the target version of solc.
  use      Install the target solc version and use it as the global version.
```

Dodatečné možnosti jednotlivých příkazů je možné zobrazit přepínačem `--help`:

```
$ woke svm install --help
Usage: woke svm install [OPTIONS] [VERSION_RANGE]...

  Install the latest solc version matching the given version range.

Options:
  --force  Reinstall the target version if already installed.
  --help   Show this message and exit.
```

Pro instalaci a zvolení konkrétní verze kompilátoru *solc* je nejvhodnější příkaz `woke svm use`:

```
$ woke svm use '0.7'
Using woke-solc version 0.7.6.
$ woke-solc --version
solc, the solidity compiler commandline interface
Version: 0.7.6+commit.7338295f.Linux.g++
```

Příkaz může přijímat rozsah verzí. V takovém případě je použita poslední verze tohoto rozsahu. Příkaz `woke svm use` kombinuje příkazy `woke svm install` a `woke svm switch`.

Po změně verze *solc* pomocí příkazu `woke svm switch` nebo `woke svm use` je daná verze k dispozici pod příkazem `woke-solc`.

Pro výpis nainstalovaných verzí kompilátoru slouží příkaz `woke svm list`:

```
$ woke svm list
- 0.5.17
- 0.7.4
- 0.7.6
- 0.8.0
```

Nainstalovanou verzi kompilátoru je možné smazat příkazem `woke svm remove`:

```
$ woke svm remove '0.5.17'
Removed solc version 0.5.17.
```

A.3 Kompilace projektu

Cílem nástroje Woke je umožnit efektivní kompilaci libovolného Solidity projektu. Tento návod ukazuje postup kompilace projektu SushiSwap [130] se zdrojovým kódem dostupným na adrese `https://github.com/sushiswap/sushiswap/tree/56cedd0`.

Nástroj Woke předpokládá, že je spouštěn v kořenovém adresáři projektu a že zdrojové soubory jazyka Solidity jsou umístěny v podadresáři `contracts`. Také je možné příkazu `woke compile` předat seznam cest k Solidity souborům. Příkaz automaticky instaluje potřebné verze kompilátoru.

V případě spuštění příkazu `woke compile` v kořenovém adresáři projektu SushiSwap dochází k chybě:

```
$ woke compile
...
CompilationResolveError: Unable to find
↪  @openzeppelin/contracts/token/ERC20/IERC20.sol in the project root dir or
↪  include paths.
```

Výpis výstupu příkazu je zkrácen. Součástí výstupu příkazu je okno obsahující úryvky zdrojového kódu reprezentující jednotlivé rámce zásobníku Pythonu. Příkazem `woke --debug compile` je možné získat podrobnější informace. Součástí výstupu je pak tabulka s hodnotami lokálních proměnných.

Solidity knihovny jsou velmi často distribuovány pomocí balíčkovacího nástroje *npm*. Projekt SushiSwap takto distribuované knihovny využívá. Pro jejich instalaci je nutné v kořenovém adresáři projektu provést následující příkaz:

```
$ npm install
```

Poté již kompilace pomocí příkazu `woke compile` proběhne v pořádku. Součástí výstupu příkazu je také seznam varování vygenerovaných kompilátorem *solc*.

Příloha B

Navrhované budoucí funkcionality

B.1 Automatické formátování kódu

Na základě doporučení v kapitole 2.1.1 lze nástroj Woke rozšířit o modul pro formátování Solidity zdrojového kódu. Předpokládá se, že by nástroj převáděl AST reprezentaci do Solidity kódu. Každý AST node by bylo možné převést na Solidity kód. Tento způsob převodu by musel být závislý na kontextu – tedy na AST nodech, které jsou v hierarchii níže i výše.

Nástroj by byl využíván LSP serverem pro formátování kódu v IDE. Také by bylo možné spustit formátovací nástroj pomocí příkazové řádky, což by mohlo být využíváno například v git hoocích.

B.2 Interpret a symbolické vykonávání jazyka Solidity

Během psaní této práce se ukázalo, že může být velmi užitečné nástroj Woke rozšířit také o funkcionality dynamické analýzy. Toto je znatelné například v implementovaných detektorech nástroje Slither, pomocí kterých je možné detekovat jen určitou množinu potenciálních zranitelností s omezenou přesností.

Symbolické vykonávání kódu je technika, kdy je kód vykonáván (interpretován) s tím, že některé proměnné mohou nabývat symbolických (tedy libovolných) hodnot namísto konkrétních číselných výrazů [131]. Implementace si obvykle uchovává seznam prozkoumávaných větví programu společně s podmínkami symbolických hodnot, za kterých je daná větev dosažitelná. K větvení může typicky docházet v podmínkových příkazech. V některých případech může docházet ke značnému větvení programu. V takových případech je často nemožné dokončit analýzu v rozumném čase. Příkladem může být přítomnost symbolické hodnoty v podmínce cyklu. Množství větví je možné ořezávat zkoumáním dosažitelnosti větví. Toho je možné docílit zkoumáním splnitelnosti podmínek symbolických proměnných v dané větvi. Jedná se o zobecněný problém splnitelnosti boolovské formule, SAT problému, který je NP-úplný [132]. Přesto existují nástroje, které pro určitou množinu vstupů dokážou otázkou splnitelnosti zodpovědět v rozumném čase. Tyto nástroje se nazývají *SMT solvery* a patří mezi ně například nástroj *Z3* [133].

Symbolické vykonávání kódu EVM by však mohlo v některých případech dokončit analýzu v rozumném času. Oproti tradičním programům jsou kontrakty platformy Ethereum mnohem jednodušší vzhledem k velikosti programu. Je to způsobené především tím, že za každou vykonávanou instrukci je nutné platit ve formě Gasu. Nástroj Woke by tedy mohl pomocí symbolické analýzy zkoušet vyhodnocovat jednotlivé funkce kontraktů. Pokud by vykonávání funkce trvalo příliš dlouho nebo by byla detekována symbolickým vykonáváním nepodporovaná funkcionalita, byl by výpočet ukončen. V opačném případě by však výsledky symbolické analýzy

umožnily vznik mnohem sofistikovanějších detektorů zranitelností a chyb. Mohlo by se jednat například o detektory dělení nulou či přetečení a podtečení. Díky symbolickému vykonávání by navíc součástí výstupu byl rozsah hodnot, pro které k chybě v kódu může dojít. Tento problém by mohl být vhodným tématem disertační práce.

Autor této práce možnosti použití symbolického vykonávání na Ethereum platformě dále analyzoval. Symbolickou analýzu nad EVM bytekódem již implementuje nástroj *Manticore* [134]. Symbolickou implementaci EVM také nabízí nástroj *hevm* [135]. Z výsledků nástroje *Manticore* je však patrné, že ve velkém množství případů je kompletní analýza EVM bytekódu časově neúnosná a musí být ukončena [136].

Symbolické vykonávání (intepretace) Solidity kódu by však mohlo větvení při analýze kódu dále zredukovat. V době psaní této práce není autorovi známa žádná implementace symbolického vykonávání nebo interpretace Solidity kódu. Spojení interpretu jazyka Solidity a symbolického vykonávání přitom může být velmi užitečné. Díky této kombinaci by mohl vzniknout debugger jazyka Solidity, který by umožňoval zadání symbolických vstupních hodnot funkce. V případě větvení symbolického vykonávání by mohl být uživatel dotázán, jaká výpočetní větev má být nadále sledována. Také by mohla být implementována funkcionalita, která by zbývající výpočetní větve „uložila na zásobník“, což by umožňovalo se k nim později vrátit. Tímto způsobem by symbolické vykonávání Solidity kódu nalezlo využití i v případě, kdy by symbolické vykonání celé funkce bylo výpočetně nepřijatelné. Uživatel nástroje by také měl plnou kontrolu nad podmínkami vstupních symbolických hodnot, což by efektivně umožňovalo omezení množství výpočetních větví. Interpretace jazyka Solidity by navíc nevyžadovala nasazení kontraktů na blockchain, což by usnadnilo ladění kódu, který je často upravován.

Nevýhodou symbolického vykonávání jazyka Solidity by zcela jistě bylo omezené množství podporovaných funkcionalit jazyka. Bez vykonávání odpovídajícího EVM bytekódu by nebylo možné pracovat s Gasem. Také by nemohl být podporován inline assembly kód v jazyce *Yul* [137]. Data blockchainu, jako je aktuální číslo bloku nebo adresa odesílatele transakce, by bylo možné nakonfigurovat uživatelem nebo pseudonáhodně vygenerovat.

Autor této práce nad rámec zadání začal pracovat na modulu, který by implementoval symbolický interpret jazyka Solidity. Předpokládá se, že tento modul bude v budoucnu dále vyvíjen. Implementace spoléhá na AST reprezentaci Solidity kódu. Nutnými vstupními parametry je funkce, která se má symbolicky vykonávat, a její (konkrétní nebo symbolické) parametry. Pomocí AST reprezentace je možné procházet tělo funkce po jednotlivých příkazech. V implementaci je pro každý typ AST node vytvořena funkce, která daný typ AST záznamu zpracovává. Mezi tyto funkce patří například funkce pro zpracování podmínek, jednotlivých typů cyklů nebo volání funkce. Dále jsou implementovány funkce pro zpracování výrazů a podvýrazů jazyka Solidity. Pro každý výraz je určen jeho datový typ, což umožňuje nad tímto výrazem v budoucnu správně provádět další operace. Také je díky typu možné hodnotu výrazu vypsat ve správném formátu uživateli.

Aby bylo možné funkcionality symbolického interpretu ladit, je interpret implementován ve formě debuggeru. Při zpracování každého AST node jsou pomocí generátorů jazyka Python (ve formě `yield` příkazů) označena místa, kdy má být vykonávání kódu pozastaveno. Díky tomu je možné debugger krokovat na úrovní jednotlivých Solidity příkazů.

Výpis B.1 představuje funkci implementující symbolické zpracování podmínkového příkazu `if` jazyka Solidity. Funkce na řádce 1 vytváří pomocí dekorátoru jazyka Python nový blok, do kterého jsou ukládány všechny nově deklarované lokální proměnné. Po opuštění této funkce je blok smazán, což efektivně ruší také deklarované proměnné v rámci `if` bloku. Funkce přijímá dva argumenty. Prvním argumentem je kontext symbolického vykonávání, který poskytuje obecné informace, jako je aktuální vykonávaná funkce či dostupné stavové a lokální proměnné. Druhým argumentem je AST node reprezentující `if` příkaz jazyka Solidity. Na řádce 5 je `yield` příkaz, který efektivně způsobuje pozastavení vykonávání kódu. Tento příkaz je proveden před zpracováním samotné podmínky, což je typické chování pro debugger. Příkaz také předává parametry, kde první parametr je AST node reprezentující výraz uvnitř podmínky. Každý AST node

reprezentující část kódu jazyka Solidity obsahuje přesné souřadnice (ve formě offsetu od začátku souboru a délky) do zdrojového kódu. Díky předání parametru `_if.condition` na řádce 5 je kód volající symbolické vykonávání schopný uživateli graficky znázornit přesnou pozici v kódu, kde je debugger pozastaven. Toto grafické znázornění zároveň reprezentuje příkaz nebo část příkazu, která bude v dalším kroku debuggeru provedena. Parametr `context` je na řádce 5 předán proto, aby kód volající symbolické vykonávání mohl vypsat seznam stavových a lokálních proměnných včetně jejich hodnot. Kód na řádce 6 vytváří `with` blok jazyka Python s implementovaným `resolve_postfix_operations` context managerem. Během vykonávání všech příkazů uvnitř `with` bloku jsou ukládány postfixové operace (`++` a `--`). Po opuštění `with` bloku jsou uložené postfixové operace aplikovány na konkrétní proměnné. Tímto způsobem je řešena sémantika postfixových operací. Postfixové operace totiž nemusí (a často nemohou) být vyhodnoceny uvnitř funkce jazyka Python, ve které jsou zpracovány. Příkaz na řádce 7 vyhodnocuje hodnotu výrazu uvnitř příkazu `if`. Na základě této hodnoty jsou provedeny odpovídající příkazy těla podmínky `if`.

Výpis kódu B.1 Funkce symbolického vykonávání příkazu `if` jazyka Solidity

```
1  @creates_execution_block
2  def _process_solc_if_statement(
3      context: AstExecutionContext, _if: SolcIfStatement
4  ) -> StepContext:
5      yield _if.condition, context
6      with resolve_postfix_operations(context):
7          result = yield from process_solc_expression(context, _if.condition)
8      if result:
9          yield from _process_solc_statement(context, _if.true_body)
10     elif _if.false_body is not None:
11         yield from _process_solc_statement(context, _if.false_body)
```

Symbolický debugger z uživatelského pohledu je znázorněn na obrázku B.1. Na obrázku je zachycen vykonávaný Solidity kód. V kódu je označena řádka 15, na které je debugger aktuálně pozastaven. Výraz `i < y` je podtržen, což značí, že je debugger pozastaven na tomto příkazu. Zároveň se jedná o příkaz, který bude v dalším kroku debuggeru vykonán. Dále jsou v obrázku znázorněny dvě tabulky s lokálními a stavovými proměnnými. Červené značení představuje proměnné, které byly v posledním kroku debuggeru změněny. Z tabulky lokálních proměnných je patrné, že funkci byla předána konkrétní adresa jako parametr `addr` a symbolická hodnota `b` pro parametr s názvem `b`. Symbolická hodnota `b` se během vykonávání kódu zanesla také do hodnoty proměnných `result` a `x`. Ve spodní části obrázku je vidět konzole, pomocí které jsou debuggeru předávány příkazy. Příkazy mohou ovládat krokování debuggeru nebo detailně vypisovat hodnoty proměnných.

V době psaní této práce je modul symbolického interpretu/debuggeru rozpracován a dokáže zpracovat jen některé části syntaxe jazyka Solidity. Kód je dostupný v repozitáři `https://github.com/Ackee-Blockchain/woke` v adresáři `woke/woke/f_symbolic_execution/ast` ve větvi `feat/x-ast-execution`. V budoucnu však může být kód této větve začleněn do hlavní vývojové větve.

```
 1 pragma solidity ^0.8.0;
 2
 3 contract Example {
 4     mapping (address => uint) public myMapping;
 5
 6     function sum(uint[3] memory x) public returns(uint) {
 7         return x[0] + x[1] + x[2];
 8     }
 9
10     function test(address addr, uint b) public returns(uint) {
11         uint y = myMapping[addr];
12         uint x = sum([uint(1), y * b, 3]);
13
14         uint result;
❯ 15       for (uint i = 0; i < y; i++)
16             result += x;
17
18         return result;
19     }
20 }
21
```

Local variables

Name	Type	Value
addr	address	0xd1755356d356c9b6992ceb591da0…
b	uint256	b
i	uint256	1
result	uint256	4*b + 4
x	uint256	4*b + 4
y	uint256	4

State variables

Name	Type	Value
myMapping	mapping(address => uint256)	{0xd1755356d356c9… 4, 0x0918a91d5d8642b… 10}

```
------------------------------------------------------------------------
next
next
next
next
next
next
next
print myMapping
AstMappingExpression{AstAddressExpression(0xd1755356d356c9b6992ceb591da03d8c237531b6): AstUint256Expression(4),
AstAddressExpression(0x0918a91d5d8642bae243861a2877cc14cc2a105d): AstUint256Expression(10)}
print result
AstUint256Expression(4*b + 4)
>>>
```

Obrázek B.1 Terminálové rozhraní symbolického debuggeru jazyka Solidity

Bibliografie

1. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008 [cit. 2022-04-29]. Dostupné z: `https://bitcoin.org/bitcoin.pdf`.
2. WOOD, Gavin. Ethereum: A secure decentralised generalised transaction ledger. BERLIN VERSION 4b05e0d. 2022-03-09.
3. BEOSIN. *Monthly Recap: More than 21 Typical Security Incidents Occurred in April 2022* [online]. 2022 [cit. 2022-04-29]. Dostupné z: `https://medium.com/@Beosin_com/monthly-recap-more-than-21-typical-security-incidents-occurred-in-april-2022-7f853aed53c7`.
4. CERTIK. *$4.3 Million Lost to Another Bridge Hack* [online]. 2022 [cit. 2022-04-29]. Dostupné z: `https://medium.com/@certik/4-3-million-lost-to-another-bridge-hack-db9b028a3c28`.
5. BUTTERIN, Vitalik. *Hard Fork Completed* [online]. 2016 [cit. 2022-04-30]. Dostupné z: `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`.
6. ETHEREUM DEVELOPERS. *Solidity* [online]. 2022 [cit. 2022-04-15]. Dostupné z: `https://docs.soliditylang.org/en/v0.8.13/`.
7. ANTONOPOULOS, Andreas; WOOD, Gavin. *Mastering Ethereum*. Sebastopol, CA: O'Reilly Media, 2018.
8. BITCOIN COMMUNITY. *Script* [online]. 2021 [cit. 2022-04-10]. Dostupné z: `https://en.bitcoin.it/wiki/Script`.
9. ETHEREUM DEVELOPERS. *INTRO TO ETHEREUM* [online]. 2022 [cit. 2022-03-25]. Dostupné z: `https://ethereum.org/en/developers/docs/intro-to-ethereum/`.
10. CHANG, Shu-jen; PERLNER, Ray; BURR, William; TURAN, Meltem Sönmez; KELSEY, John; PAUL, Souradyuti; BASSHAM, Lawrence. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. 2012-11. Tech. zpr., NISTIR 7896. NIST.
11. WARD, Chris. *Ethash* [online]. 2020 [cit. 2022-04-18]. Dostupné z: `https://eth.wiki/en/concepts/ethash/ethash`.
12. BEREGSZASZI, Alex. *Replace SHA3 opcode with KECCAK256* [online]. 2021 [cit. 2022-04-12]. Dostupné z: `https://github.com/ethereum/yellowpaper/commit/fcf1d13`.
13. ETHEREUM DEVELOPERS. *Ether* [online]. 2016 [cit. 2022-03-27]. Dostupné z: `https://ethdocs.org/en/latest/ether.html`.
14. ETHEREUM DEVELOPERS. *GAS AND FEES* [online]. 2022 [cit. 2022-03-27]. Dostupné z: `https://ethereum.org/en/developers/docs/gas/`.

15. BECZE, Martin; JAMESON, Hudson et al. „EIP-1: EIP Purpose and Guidelines," *Ethereum Improvement Proposals*. In: [online]. 2015 [cit. 2022-04-04]. Č. 1. Dostupné z: `https://eips.ethereum.org/EIPS/eip-1`.
16. BITCOIN DEVELOPERS. *Bitcoin Improvement Proposals* [online]. 2022 [cit. 2022-04-12]. Dostupné z: `https://github.com/bitcoin/bips`.
17. WARSAW, Barry; HYLTON, Jeremy; GOODGER, David; COGHLAN, Nick. *PEP 1 – PEP Purpose and Guidelines* [online]. 2022 [cit. 2022-04-12]. Dostupné z: `https://peps.python.org/pep-0001/`.
18. VOGELSTELLER, Fabian; BUTERIN, Vitalik. „EIP-20: Token Standard," *Ethereum Improvement Proposals*. In: [online]. 2015 [cit. 2022-04-09]. Č. 20. Dostupné z: `https://eips.ethereum.org/EIPS/eip-20`.
19. ENTRIKEN, William. „EIP-2228: Canonicalize the name of network ID 1 and chain ID 1," *Ethereum Improvement Proposals*. In: [online]. 2019 [cit. 2022-04-10]. Č. 2228. Dostupné z: `https://eips.ethereum.org/EIPS/eip-2228`.
20. *The "Yellow Paper": Ethereum's formal specification* [online]. 2022 [cit. 2022-04-04]. Dostupné z: `https://github.com/ethereum/yellowpaper`.
21. PŘEVRÁTIL, Michal. *Fix ambiguous empty brackets in Appendix D* [online]. GitHub, 2022 [cit. 2022-04-07]. Dostupné z: `https://github.com/ethereum/yellowpaper/pull/852`.
22. THE GO-ETHEREUM AUTHORS. *Official Go implementation of the Ethereum protocol* [online]. GitHub, 2022 [cit. 2022-04-08]. Dostupné z: `https://github.com/ethereum/go-ethereum/blob/9fd8825/trie/node_enc.go`.
23. ZHANG, Leo. *A simplified golang implementation of Ethereum's Modified Patricia Trie.* [Online]. GitHub, 2020 [cit. 2022-04-09]. Dostupné z: `https://github.com/zhangchiqing/merkle-patricia-trie/tree/e8e18ce`.
24. THOMAS, Lee. *Ethereum Modified Merkle-Patricia-Trie System* [online]. 2016 [cit. 2022-04-09]. Dostupné z: `https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture/6413`.
25. ETHEREUM DEVELOPERS. *NODES AND CLIENTS* [online]. 2022 [cit. 2022-04-12]. Dostupné z: `https://ethereum.org/en/developers/docs/nodes-and-clients/`.
26. POMERANTZ, Ori. *MERKLE PROOFS FOR OFFLINE DATA INTEGRITY* [online]. 2021 [cit. 2022-04-09]. Dostupné z: `https://ethereum.org/en/developers/tutorials/merkle-proofs-for-offline-data-integrity/`.
27. ETHEREUM DEVELOPERS. *PROOF-OF-STAKE (POS)* [online]. 2022 [cit. 2022-04-12]. Dostupné z: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`.
28. ETHEREUM DEVELOPERS. *PROOF-OF-WORK (POW)* [online]. 2022 [cit. 2022-04-12]. Dostupné z: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/`.
29. BUTERIN, Vitalik. „EIP-198: Big integer modular exponentiation," *Ethereum Improvement Proposals*. In: [online]. 2017 [cit. 2022-04-08]. Č. 198. Dostupné z: `https://eips.ethereum.org/EIPS/eip-198`.
30. REITWIESSNER, Christian. „EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," *Ethereum Improvement Proposals*. In: [online]. 2017 [cit. 2022-04-08]. Č. 196. Dostupné z: `https://eips.ethereum.org/EIPS/eip-196`.
31. VERCAUTEREN, Frederik. Optimal Pairings. *IEEE Transactions on Information Theory* [online]. 2010, roč. 56, č. 1, s. 455–461 [cit. 2022-04-17]. Dostupné z DOI: `10.1109/TIT.2009.2034881`.

32. BUTERIN, Vitalik; REITWIESSNER, Christian. „EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128," *Ethereum Improvement Proposals*. In: [online]. 2017 [cit. 2022-04-08]. Č. 197. Dostupné z: https://eips.ethereum.org/EIPS/eip-197.
33. HESS, Tjaden; LUONGO, Matt; DYRAGA, Piotr; HANCOCK, James. „EIP-152: Add BLAKE2 compression function 'F' precompile," *Ethereum Improvement Proposals*. In: [online]. 2016 [cit. 2022-04-08]. Č. 152. Dostupné z: https://eips.ethereum.org/EIPS/eip-152.
34. ZOLTU, Micah. „EIP-2718: Typed Transaction Envelope," *Ethereum Improvement Proposals*. In: [online]. 2020 [cit. 2022-04-09]. Č. 2718. Dostupné z: https://eips.ethereum.org/EIPS/eip-2718.
35. BUTERIN, Vitalik; SWENDE, Martin. „EIP-2930: Optional access lists," *Ethereum Improvement Proposals*. In: [online]. 2020 [cit. 2022-04-11]. Č. 2930. Dostupné z: https://eips.ethereum.org/EIPS/eip-2930.
36. JOHNSON, Nick. „EIP-658: Embedding transaction status code in receipts," *Ethereum Improvement Proposals*. In: [online]. 0217 [cit. 2022-04-10]. Č. 658. Dostupné z: https://eips.ethereum.org/EIPS/eip-658.
37. BLOOM, Burton H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*. 1970, roč. 13, s. 422–426.
38. COMITY LABS. *An interactive reference to Ethereum Virtual Machine Opcodes* [online]. 2022 [cit. 2022-04-15]. Dostupné z: https://www.evm.codes.
39. ETHEREUM DEVELOPERS. *Contract ABI Specification* [online]. 2021 [cit. 2022-04-07]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/abi-spec.html.
40. BEREGSZASZI, Alex. „EIP-2488: Deprecate the CALLCODE opcode," *Ethereum Improvement Proposals*. In: [online]. 2019 [cit. 2022-04-12]. Č. 2488. Dostupné z: https://eips.ethereum.org/EIPS/eip-2488.
41. BUTERIN, Vitalik; REITWIESSNER, Christian. „EIP-214: New opcode STATICCALL," *Ethereum Improvement Proposals*. In: [online]. 2017 [cit. 2022-04-12]. Č. 214. Dostupné z: https://eips.ethereum.org/EIPS/eip-214.
42. ETHEREUM DEVELOPERS. *EVENTS AND LOGS* [online]. 2022 [cit. 2022-04-24]. Dostupné z: https://ethereum.org/en/developers/docs/smart-contracts/anatomy/#events-and-logs.
43. ETHEREUM DEVELOPERS. *MINING* [online]. 2022 [cit. 2022-04-28]. Dostupné z: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/mining/.
44. ROSENFELD, Meni. *Analysis of Hashrate-Based Double Spending* [online]. arXiv, 2014 [cit. 2022-04-27]. Dostupné z DOI: 10.48550/ARXIV.1402.2009.
45. ETHEREUM DEVELOPERS. *CONSENSUS MECHANISMS* [online]. 2022 [cit. 2022-04-28]. Dostupné z: https://ethereum.org/en/developers/docs/consensus-mechanisms/.
46. BUTTERIN, Vitalik. *Meta: cap total ether supply at 120 million* [online]. GitHub, 2018 [cit. 2022-04-27]. Dostupné z: https://github.com/ethereum/EIPs/issues/960.
47. SAYEED, Sarwar; MARCO-GISBERT, Hector. Assessing Blockchain Consensus and Security Mechanisms against the 51% Attack. *Applied Sciences* [online]. 2019, roč. 9, č. 9, s. 1788 [cit. 2022-04-27]. Dostupné z DOI: 10.3390/app9091788.
48. ETHEREUM DEVELOPERS. *The Merge* [online]. 2022 [cit. 2022-04-27]. Dostupné z: https://ethereum.org/en/upgrades/beacon-chain/.
49. ETHEREUM DEVELOPERS. *Language Influences* [online]. 2022 [cit. 2022-04-07]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/language-influences.html.

50. ETHEREUM DEVELOPERS. *Language Grammar* [online]. 2022 [cit. 2022-04-15]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/grammar.html.
51. ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. *PEP 8 – Style Guide for Python Code* [online]. 2022 [cit. 2022-04-15]. Dostupné z: https://peps.python.org/pep-0008/.
52. ETHEREUM DEVELOPERS. *Style Guide* [online]. 2022 [cit. 2022-04-05]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/style-guide.html.
53. ETHEREUM DEVELOPERS. *Types* [online]. 2020 [cit. 2022-04-17]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/types.html.
54. ZOS GLOBAL LIMITED. *OpenZeppelin Contracts is a library for secure smart contract development.* [Online]. GitHub, 2022 [cit. 2022-04-18]. Dostupné z: https://github.com/OpenZeppelin/openzeppelin-contracts.
55. BUTERIN, Vitalik; SANDE, Alex Van de. „EIP-55: Mixed-case checksum address encoding," *Ethereum Improvement Proposals*. In: [online]. 2016 [cit. 2022-04-19]. Č. 55. Dostupné z: https://eips.ethereum.org/EIPS/eip-55.
56. ETHEREUM DEVELOPERS. *Layout of a Solidity Source File* [online]. 2022 [cit. 2022-04-18]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/layout-of-source-files.html.
57. SPDX WORKGROUP. *The Software Package Data Exchange® (SPDX®)* [online]. 2021 [cit. 2022-04-07]. Dostupné z: https://spdx.dev.
58. NPM, INC. *The semantic versioner for npm* [online]. 2021 [cit. 2022-04-17]. Dostupné z: https://docs.npmjs.com/cli/v6/using-npm/semver.
59. ETHEREUM DEVELOPERS. *SMTChecker and Formal Verification* [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/smtchecker.html.
60. ETHEREUM DEVELOPERS. *NatSpec Format* [online]. 2022 [cit. 2022-04-18]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/natspec-format.html.
61. ETHEREUM DEVELOPERS. *Structure of a Contract* [online]. 2021 [cit. 2022-04-29]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/structure-of-a-contract.html.
62. ETHEREUM DEVELOPERS. *Contracts* [online]. 2021 [cit. 2022-04-29]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/contracts.htm.
63. NAKAMURA, Tasuku. *Function Modifier* [online]. 2022. Dostupné také z: https://solidity-by-example.org/function-modifier.
64. ETHEREUM DEVELOPERS. *Using the Compiler* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/using-the-compiler.html.
65. ETHEREUM DEVELOPERS. *Import Path Resolution* [online]. 2021 [cit. 2022-04-17]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/path-resolution.html.
66. ETHEREUM DEVELOPERS. *Using the Compiler* [online]. 2017 [cit. 2022-04-07]. Dostupné z: https://docs.soliditylang.org/en/v0.4.10/using-the-compiler.html.
67. ETHEREUM DEVELOPERS. *Using the Compiler* [online]. 2017 [cit. 2022-04-08]. Dostupné z: https://docs.soliditylang.org/en/v0.4.11/using-the-compiler.html.
68. ETHEREUM DEVELOPERS. *Using the Compiler* [online]. 2018 [cit. 2022-04-08]. Dostupné z: https://docs.soliditylang.org/en/v0.5.0/using-the-compiler.html.
69. ETHEREUM DEVELOPERS. *Using the Compiler* [online]. 2020 [cit. 2022-04-08]. Dostupné z: https://docs.soliditylang.org/en/v0.6.9/using-the-compiler.html.

70. ETHEREUM DEVELOPERS. *Import Path Resolution* [online]. 2021 [cit. 2022-04-10]. Dostupné z: https://docs.soliditylang.org/en/v0.8.5/path-resolution.html.

71. ETHEREUM DEVELOPERS. *Import Path Resolution* [online]. 2021 [cit. 2022-04-10]. Dostupné z: https://docs.soliditylang.org/en/v0.8.8/path-resolution.html.

72. ŚLIWAK, Kamil. *Search path for libraries (path spec v3)* [online]. GitHub, 2021 [cit. 2022-04-23]. Dostupné z: https://github.com/ethereum/solidity/issues/11409.

73. ETHEREUM DEVELOPERS. *Solidity, the Smart Contract Programming Language* [online]. GitHub, 2022 [cit. 2022-04-15]. Dostupné z: https://github.com/ethereum/solidity.

74. ŚLIWAK, Kamil. *Better path-related error messages (path spec v3)* [online]. GitHub, 2021 [cit. 2022-04-23]. Dostupné z: https://github.com/ethereum/solidity/issues/11413.

75. ŚLIWAK, Kamil. *Detect platform-dependent paths in default file loader (path spec v3)* [online]. GitHub, 2021 [cit. 2022-04-23]. Dostupné z: https://github.com/ethereum/solidity/issues/11412.

76. ŚLIWAK, Kamil. *Import path normalization (path spec v3)* [online]. GitHub, 2021 [cit. 2022-04-23]. Dostupné z: https://github.com/ethereum/solidity/issues/11411.

77. ŚLIWAK, Kamil. *Disallow absolute import paths (path spec v3)* [online]. GitHub, 2021 [cit. 2022-04-23]. Dostupné z: https://github.com/ethereum/solidity/issues/11410.

78. TRAIL OF BITS. *Static Analyzer for Solidity* [online]. GitHub, 2022 [cit. 2022-04-18]. Dostupné z: https://github.com/crytic/slither.

79. TRAIL OF BITS. *Adding a new detector* [online]. GitHub, 2020 [cit. 2022-04-15]. Dostupné z: https://github.com/crytic/slither/wiki/Adding-a-new-detector.

80. THE GRAPHVIZ AUTHORS. *DOT Language* [online]. 2022 [cit. 2022-04-16]. Dostupné z: https://graphviz.org/doc/info/lang.html.

81. ALLEN, Frances E. Control flow analysis. *ACM SIGPLAN Notices* [online]. 1970, roč. 5, č. 7, s. 1–19 [cit. 2022-04-15]. Dostupné z DOI: 10.1145/390013.808479.

82. ENTRIKEN, William; SHIRLEY, Dieter; EVANS, Jacob; SACHS, Nastassia. „EIP-721: Token Standard,“ *Ethereum Improvement Proposals*. In: [online]. 2018 [cit. 2022-04-20]. Č. 721. Dostupné z: https://eips.ethereum.org/EIPS/eip-721.

83. DAFFLON, Jacques; BAYLINA, Jordi; SHABABI, Thomas. „EIP-777: Token Standard,“ *Ethereum Improvement Proposals*. In: [online]. 2017 [cit. 2022-04-20]. Č. 777. Dostupné z: https://eips.ethereum.org/EIPS/eip-777.

84. RADOMSKI, Witek; COOKE, Andrew; CASTONGUAY, Philippe; THERIEN, James; BINET, Eric; SANDFORD, Ronan. „EIP-1155: Multi Token Standard,“ *Ethereum Improvement Proposals*. In: [online]. 2018 [cit. 2022-04-20]. Č. 1155. Dostupné z: https://eips.ethereum.org/EIPS/eip-1155.

85. ZOS GLOBAL LIMITED. *Upgrading smart contracts* [online]. 2021 [cit. 2022-04-27]. Dostupné z: https://docs.openzeppelin.com/learn/upgrading-smart-contracts.

86. TRAIL OF BITS. *Contract upgrade anti-patterns* [online]. 2018 [cit. 2022-04-27]. Dostupné z: https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/.

87. ETHEREUM FOUNDATION. *Remix IDE* [online]. 2022 [cit. 2022-04-14]. Dostupné z: https://remix.ethereum.org.

88. TRAIL OF BITS. *Manage and switch between Solidity compiler versions* [online]. 2022 [cit. 2022-04-21]. Dostupné z: https://github.com/crytic/solc-select.

89. PŘEVRÁTIL, Michal. *Incorrect use of SHA3 instead of keccak256* [online]. GitHub, 2022 [cit. 2022-04-22]. Dostupné z: https://github.com/crytic/solc-select/issues/89.

90. MICROSOFT. *Code editing. Redefined.* [Online]. 2022 [cit. 2022-04-16]. Dostupné z: https://code.visualstudio.com.
91. MICROSOFT. *Language Server Protocol* [online]. 2021 [cit. 2022-04-19]. Dostupné z: https://microsoft.github.io/language-server-protocol/.
92. PYTHON SOFTWARE FOUNDATION. *typing — Support for type hints* [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://docs.python.org/3.10/library/typing.html.
93. COLVIN, Samuel. *Data parsing and validation using Python type hints* [online]. GitHub, 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/samuelcolvin/pydantic/.
94. KREKEL, Holger et al. *pytest: helps you write better programs* [online]. 2022 [cit. 2022-04-18]. Dostupné z: https://docs.pytest.org/en/7.1.x/.
95. LANGA, Łukasz; WILLING, Carol; MEYER, Carl; ZIJLSTRA, Jelle; NAYLOR, Mika; DOLLENSTEIN, Zsolt; LEES, Cooper; SI, Richard; HILDÉN, Felix; TASKAYA, Batuhan. *The uncompromising code formatter* [online]. 2022 [cit. 2022-04-18]. Dostupné z: https://black.readthedocs.io/en/stable/.
96. MICROSOFT CORPORATION. *Static type checker for Python* [online]. 2022 [cit. 2022-04-22]. Dostupné z: https://github.com/microsoft/pyright.
97. GITHUB, INC. *GitHub Actions* [online]. 2022 [cit. 2022-04-22]. Dostupné z: https://docs.github.com/en/actions.
98. CHACON, Scott; STRAUB, Ben. *Pro Git.* 2. vyd. Berlin, Germany: APress, 2014.
99. BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C. M. *Extensible Markup Language (XML) 1.0* [online]. 1998 [cit. 2022-04-25]. Dostupné z: https://www.w3.org/TR/1998/REC-xml-19980210.
100. PALMER, Rob; SHARMA, Ujjwal; TERLSON, Brian. *ECMA-404: The JSON data interchange syntax* [online]. 2013 [cit. 2022-04-25]. Dostupné z: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.
101. BEN-KIKI, Oren; EVANS, Clark; NET, Ingy döt. *YAML™ Specification Index* [online]. 2001 [cit. 2022-04-25]. Dostupné z: https://yaml.org/spec/history/2001-03-30.html.
102. PRESTON-WERNER, Tom. *TOML: [Tom's Obvious Minimal Language]* [online]. 2013 [cit. 2022-04-25]. Dostupné z: https://toml.io/en/v0.1.0.
103. YAML LANGUAGE DEVELOPMENT TEAM. *YAML Ain't Markup Language (YAML™) version 1.2: Indentation Spaces* [online]. 2021 [cit. 2022-04-29]. Dostupné z: https://yaml.org/spec/1.2.2/#61-indentation-spaces.
104. HUKKINEN, Taneli; JAIN, Shantanu. *PEP 680 – tomllib: Support for Parsing TOML in the Standard Library* [online]. 2022 [cit. 2022-04-10]. Dostupné z: https://peps.python.org/pep-0680/.
105. PYTHON SOFTWARE FOUNDATION. *pathlib — Object-oriented filesystem paths* [online]. 2022 [cit. 2022-04-21]. Dostupné z: https://docs.python.org/3.10/library/pathlib.html.
106. MCGUGAN, Will. *Rich is a Python library for rich text and beautiful formatting in the terminal.* [Online]. GitHub, 2022 [cit. 2022-04-21]. Dostupné z: https://github.com/Textualize/rich.
107. HERMAN, David; WAGNER, Luke; ZAKAI, Alon. *asm.js* [online]. 2014 [cit. 2022-04-12]. Dostupné z: http://asmjs.org/spec/latest/.
108. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.* [Online]. 2022 [cit. 2022-04-12]. Dostupné z: https://webassembly.org.

109. PRESTON-WERNER, Tom. *Semantic Versioning 2.0.0* [online]. 2020 [cit. 2022-04-21]. Dostupné z: https://semver.org/spec/v2.0.0.html.

110. BARROIS, Raphaël. *Semantic version comparison for Python* [online]. GitHub, 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/rbarrois/python-semanticversion/blob/7dcc42d/semantic_version/base.py.

111. PARPART, Christian. *Utilities to handle semantic versioning* [https://github.com/ethereum/solidity/blob/55467c1/liblangutil/SemVerHandler.cpp]. GitHub, 2021 [cit. 2022-04-08].

112. PYTHON SOFTWARE FOUNDATION. *Subprocesses* [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://docs.python.org/3.10/library/asyncio-subprocess.html.

113. ZOS GLOBAL LIMITED. *Solidity AST Types* [online]. 2022 [cit. 2022-04-17]. Dostupné z: https://www.npmjs.com/package/solidity-ast.

114. ZVEREV, Pavel; BOUNOV, Dimitar. *A TypeScript package providing a normalized typed Solidity AST along with the utilities necessary to generate the AST (from Solc) and traverse/manipulate it.* [Online]. 2022 [cit. 2022-04-21]. Dostupné z: https://github.com/ConsenSys/solc-typed-ast/.

115. PYTHON SOFTWARE FOUNDATION. *argparse — Parser for command-line options, arguments and sub-commands* [online]. 2022 [cit. 2022-04-22]. Dostupné z: https://docs.python.org/3.10/library/argparse.html.

116. KELESHEV, Vladimir. *Pythonic command line arguments parser, that will make you smile* [online]. GitHub, 2018 [cit. 2022-04-22]. Dostupné z: https://github.com/docopt/docopt.

117. PALLETS. *Python composable command line interface toolkit* [online]. 2022 [cit. 2022-04-22]. Dostupné z: https://click.palletsprojects.com/en/8.1.x/.

118. HAUSER, Ben. *A Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.* [Online]. GitHub, 2022 [cit. 2022-04-22]. Dostupné z: https://github.com/eth-brownie/brownie.

119. UNISWAP LABS. *Core smart contracts of Uniswap v3* [online]. GitHub, 2022 [cit. 2022-04-19]. Dostupné z: https://github.com/Uniswap/v3-core.

120. THE GRAPH FOUNDATION. *The Graph Protocol* [online]. GitHub, 2022 [cit. 2022-04-19]. Dostupné z: https://github.com/graphprotocol/contracts.

121. TRADER JOE. *Main contracts for Trader Joe* [online]. GitHub, 2022 [cit. 2022-04-19]. Dostupné z: https://github.com/traderjoe-xyz/joe-core.

122. AXELAR NETWORK. *Axelar cross-chain gateway protocol solidity implementation* [online]. GitHub, 2022 [cit. 2022-04-19]. Dostupné z: https://github.com/axelarnetwork/axelar-cgp-solidity.

123. MĂRIEȘ, Ionel Cristian; SCHLAICH, Marc et al. *Coverage plugin for pytest* [online]. 2022 [cit. 2022-04-21]. Dostupné z: https://pytest-cov.readthedocs.io/en/latest/.

124. NOMIC LABS LLC. *Hardhat is a development environment to compile, deploy, test, and debug your Ethereum software.* [Online]. GitHub, 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/NomicFoundation/hardhat.

125. TRUFFLE BLOCKCHAIN GROUP, INC. *A tool for developing smart contracts. Crafted with the finest cacaos.* [Online]. GitHub, 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/trufflesuite/truffle.

126. KONSTANTOPOULOS, Georgios. *Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust.* [Online]. GitHub, 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/foundry-rs/foundry.

127. BROWNIE COMMUNITY. *Support multiple namespace* [online]. 2022 [cit. 2022-04-20]. Dostupné z: https://github.com/eth-brownie/brownie/issues/1404.

128. HELLMANN, Doug. *virtualenvwrapper 5.0.0* [online]. 2020 [cit. 2022-04-23]. Dostupné z: https://virtualenvwrapper.readthedocs.io/en/5.0.0/.

129. PYTHON SOFTWARE FOUNDATION. *venv — Creation of virtual environments* [online]. 2022 [cit. 2022-04-23]. Dostupné z: https://docs.python.org/3.10/library/venv.html.

130. SUSHISWAP DEVELOPERS. *Be a DeFi Chef with Sushi.* [Online]. 2022 [cit. 2022-04-21]. Dostupné z: https://www.sushi.com.

131. KING, James C. Symbolic execution and program testing. *Communications of the ACM* [online]. 1976, roč. 19, č. 7, s. 385–394 [cit. 2022-04-26]. Dostupné z DOI: 10.1145/360248.360252.

132. COOK, Stephen A. The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71* [online]. ACM Press, 1971 [cit. 2022-04-26]. Dostupné z DOI: 10.1145/800157.805047.

133. MOURA, Leonardo de; BJØRNER, Nikolaj. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 337–340 [cit. 2022-04-26]. ISBN 978-3-540-78800-3. Dostupné z DOI: 10.1007/978-3-540-78800-3_24.

134. TRAIL OF BITS. *Symbolic execution tool* [online]. 2022 [cit. 2022-04-26]. Dostupné z: https://github.com/trailofbits/manticore.

135. BROCKMAN, Mikael; LUNDFALL, Martin. *hevm: Ethereum virtual machine evaluator* [online]. 2022 [cit. 2022-04-26]. Dostupné z: https://hackage.haskell.org/package/hevm.

136. MOSSBERG, Mark; MANZANO, Felipe; HENNENFENT, Eric; GROCE, Alex; GRIECO, Gustavo; FEIST, Josselin; BRUNSON, Trent; DINABURG, Artem. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts* [online]. arXiv, 2019 [cit. 2022-04-26]. Dostupné z DOI: 10.48550/ARXIV.1907.03890.

137. ETHEREUM DEVELOPERS. *Yul* [online]. 2022 [cit. 2022-04-30]. Dostupné z: https://docs.soliditylang.org/en/v0.8.13/yul.html.

Obsah přiloženého média

- `readme.txt` .... stručný popis obsahu média
- `src`
  - `woke` .... zdrojové kódy implementace nástroje Woke
  - `slither_detector` .... zdrojové kódy vlastního detektoru nástroje Slither
  - `thesis` .... zdrojová forma práce ve formátu LaTeX
- `text` .... text práce
  - `thesis.pdf` .... text práce ve formátu PDF