FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Rozšíření nástroje Woke
Student:	Bc. Lukáš Böhm
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 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 Echidna, Brownie a Slither.
- 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. Lukáš Böhm

Katedra Informační bezpečnosti
Vedoucí práce: Ing. Josef Gattermayer, Ph.D.

4. května 2022

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce doktoru Josefu Gattermayerovi za jeho čas a cenné rady k vypracování této práce. Dále bych chtěl poděkovat vedoucímu týmu Woke Dominiku Teimlovi za ochotu a cenné rady týkající se praktické i implementační části práce.

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 4. května 2022 .....

Odkaz na tuto práci

Böhm, Lukáš. *Rozšíření nástroje Woke*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato práce se ve své první polovině věnuje technologii Ethereum, EVM a následně samotným smart kontraktům. Tyto informace jsou nezbytné pro analýzu nástrojů a návrh funkcionalit nástroje Woke. Jednotlivé nástroje jsou detailně popsány společně se způsobem jejich použití a nakonec jsou vytyčeny jejich silné a slabší stránky. Další kapitola se věnuje samotnému nástroji Woke. V kapitole je popsána motivace pro vytvoření nového nástroje, návrh funkcionalit a nedostatky, které oproti ostatním nástrojům má za cíl pokrývat. V poslední části je popsána implementace a testování podmnožiny navržených funkcionalit.

Klíčová slova Ethereum, Solidity, Smart Kontrakt, Bezpečnost, Server, Blockchain

Abstract

The first half of this thesis focuses on Ethereum, EVM and then smart contracts themselves. This information is necessary for the analysis of the tools and the design of the Woke tool functionalities. The individual tools are described in detail, along with how they are used, and finally their strengths and weaknesses are outlined. The next chapter focuses on the Woke tool itself. The chapter describes the motivation for the creation of the new tool, the design of the functionalities and the gaps it aims to cover compared to other tools. The last section describes the implementation and testing of a subset of the proposed functionalities.

Keywords Ethereum, Solidity, Smart Contract, Security, Server, Blockchain

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V posledních letech bylo snadné zaznamenat strmý nárůst v počtu technologií využívajících blockchainový [1] protokol. Několik let po představení Bitcoinu [1] začaly vznikat protokoly obohacující blockchainové jádro o komplexnější funkcionality, které mimo pouhého uchování záznamů transakcí umožnily také vytváření aplikací. Tyto aplikace nesou anglický název "smart contracts" (dále smart kontrakty, nebo pouze kontrakty).[2]

Nejsilnější stránkou blockchainu, ať už toho Bitcoinového nebo ostatních, je schopnost bezpečného uchování historie transakcí a dat s nimi spojených. Z toho důvodu se správa decentralizovaných financí jeví jako nejsmysluplnější využití těchto technologií. Tento fakt napomohl vzniku pojmu "Decentralizované Finance" (zkratka DeFi). DeFi aplikace - smart kontrakty jsou alternativou k běžnému finančnímu systému. Skrz tento druh softwaru, uloženého v blockchainové databázi, je možné směňovat jednotlivé tokeny a kryptoměny, využívat půjček, investovat, poskytovat likviditu s danou úrokovou sazbou nebo například využívat kontrakty jako spořící účty. Z běžného finančního světa byl také převzat pojem "kontrakt" neboli souhrn pravidel definující podmínky dané dohody. Na rozdíl od papírové smluvní formy jsou ve světě DeFi tato pravidla zapsána pomocí kódu smart kontraktů.

Není výjimkou, aby jednotlivé kontrakty, přesněji jejich adresy, uchovávaly kryptoměny a tokeny v hodnotách řádů desítek či stovek milionů dolarů, tedy i miliardy českých korun. Jediná zneužitelná chyba ve zdrojovém kódu může vést k přímé ztrátě aktiv v takto astronomických hodnotách. Tento fakt definuje novou éru v oboru kybernetické bezpečnosti, protože do příchodu DeFi neexistovala příležitost k přímé krádeži aktiv v hodnotě 624 milionů dolarů. [1] Za rok 2021 se celkové ztráty odhadují na 1.3 miliardy amerických dolarů [4]. Hlavním důvodem je laxní a neprofesionální přístup k bezpečnostnímu testování smart kontraktů. Přitom bezpečnost daných aplikací je to, co rozděluje

[1] V březnu roku 2022 došlo ke zneužití protokolu Ronin Network a bylo ukradeno přibližně 624 milionů amerických dolarů. Jedná se prozatím o největší hack v historii DeFi [3]

v blockchainovém světě produkty na úspěšné a neúspěšné.

Čím dál více vývojářů si uvědomuje hrozby, kterým čelí, a proto bezpečnost začínají stavět do popředí. Významným rozdílem mezi tradičním softwarem a smart kontraky je totiž fakt, že po nasazení kontraktu do reálného provozu již není možné kód jakkoliv upravovat, nebo službu pozastavovat. [2] Z toho důvodu se stávají standardem velké bezpečnostní audity, těsně před nasazením kontraktu na hlavní blockchainovou síť. Audity by měly být prováděny na finálním produktu, a to jak interním vývojářským týmem, tak ideálně i několika různými na sobě nezávislými experty, kteří se specializují na tuto konkrétní činnost.

Bezpečnostní audit smart kontraktu má dvě části. Bezpečnost kódu lze ověřit automatizovaným způsobem s využitím nástrojů k tomu určených a psaním testů, nebo manuální statickou analýzou kódu. V tuto chvíli se využívá jediný automatický detektor zranitelností, který je součástí nástroje Slither 3.2. Jedná se rychlý způsob, jak najít určité chyby v kódu, nicméně jak již bývá u automatizovaných detekcí zvykem, nelze se na ně stoprocentně spolehnout. Velká část vyobrazených chyb je ve stavu False-Positive (chybně detekováno jako zranitelnost) a mnoho chyb není možné detektorem vůbec zachytit, protože se často jedná o komplexní útočné vektory. Některé z nedetekovaných chyb je možné zachytit kvalitními testy, které pokrývají významnou část všech možných vstupů funkcí. K vytváření randomizovaných testů na míru daným kontraktům existuje specializovaný testovací nástroj Echidna 3.3. Ani automatická detekce s kvalitním testovým prokrytím není zárukou nalezení chyb vedoucích ke kompromitaci kontraktu a proto je manuální analýza kódu nevyhnutelnou částí bezpečnostního auditu . Nicméně i během manuální kontroly mohou asistovat některé nástroje. Brownie 3.1 umožňuje interakci s transakcemi a analýzu blockchainových adres, Slither je schopen vykreslit například graf volaných funkcí. Takto asistovaná manuální kontrola kódu se jeví být tím nejefektivnějším auditovacím postupem.

Práce byla vytvořena s motivací popsat základní konstrukty platformy EVM (Ethereum Virtual Machine), shrnout proces vývoje smart kontraktů a funkcionality nástrojů, které daný proces usnadňují a zrychlují. Z tohoto hlediska práce může sloužit pro hlubší porozumění Ethereum platformy a jako návod pro efektivnější využití dostupných nástrojů, které vývojářský, případně auditorský proces urychlují.

Druhou motivací pro tuto práci bylo využití poznatků z nastudovaných a zanalyzovaných nástrojů k návrhu vhodných funkcionalit pro vznikající nástroj Woke.

Cílem této práce je nastudování známých a hojně používaných nástrojů Echidna, Brownie a Slither. Pro lepší porozumění jejich vlastností je nejprve popsána platforma Ethereum, na které je software vyvíjen a pro kterou jsou

[2] Speciální návrh smart kontraktů umožňuje určitou manipulaci s již běžícím kódem, nicméně tato architektura s sebou nese další bezpečnostní hrozby.

zmíněné nástroje určeny. Během popisu nástrojů je kladen důraz na jejich jednotlivé funkcionality, kvalitu jejich výstupu a způsob použití. Následně jsou jednotlivé nástroje shrnuty, porovnány a vytyčeny jejich silné a slabší stránky. Na základě získaných vědomostí je definována motivace k vytvoření nově vznikajícího nástroje Woke a jeho požadované funkcionality. Zbytek práce se věnuje konkrétní podmnožině funkcionalit, vhodnému postupu jejich implementace a v posledním kroku také testování.

Práce je strukturována do čtyř hlavních kapitol. V první kapitole se práce věnuje projektu Etherum. Tato kapitola je nezbytná pro představení platformy, pro kterou je software vytvářen. Mimo krátké historie jsou zde popsány základní konstrukty a architektura virtuálního počítače Ehereum. Dále také termíny, které jsou pro tuto platformu unikátní, jako jsou adresy, transakce, smart kontrakty ad.

Ve třetí kapitole jsou postupně popsány nástroje Brownie, Slither a Echidna. Jejich popis vychází z autorových zkušeností s jejich používáním a z oficiálních dokumentací, případně z jejich zdrojových kódů. U každého z nástrojů jsou popsány jeho funkcionality a způsob, jakým je využívat. V poslední části kapitoly jsou nástroje zesumarizovány a vyhodnoceny jejich hlavní výhody a nedostatky.

Ve čtvrté kapitole je popis nástroje Woke a cíle, na základě kterých je vytvářen. Jsou zde vypsány jeho současné a budoucí funkcionality. Nakonec je popsán LSP server, který je součástí Woke, a hlavní důvody pro jeho implementaci. Práce pokračuje popisem implementace jazykového serveru a protokolu, který slouží ke komunikaci mezi serverem a klientem. V poslední řadě je implementace otestována.

Kapitola 1

Ethereum

Koncem roku 2008 byl pod pseudonymem Satoshi Nakamoto zveřejněn Bitcoinový whitepaper [1]. Tento dokument definoval systém decentralizované měny zvané Bitcoin, která zkombinovala primitiva pro správu vlastnictví pomocí algoritmů asynchronní kryptografie s algoritmem o důkazu práce (Proof Of Work) přiřazující jednotlivým adresám odpovídající počet mincí. Spojení těchto dvou algoritmů se stalo průlomovým v dané problematice. Dle Satoshiho Nakamota je cílem Bitcoinu stát se globální měnou a blockchainová "databáze", ve které jsou zaznamenané všechny transakce, slouží jako účetní kniha.

Bitcoin má definovaný vlastní skriptovací jazyk. Tento jazyk se využívá například při ověřování podpisů, případně vícenásobných podpisů na eliptických křivkách. Nicméně se nejedná o Turingově kompletní jazyk. Hlavním důvodem je bezpečnost Bitcoinového protokolu. Tato limitaca zamezuje Denial of Service útoku. Zmíněnou vlastnost Bitcoinu nelze považovat za jeho nedostatek, ale zbývá zde prostor pro nové technologie, jejichž cílem není vytvoření digitální měny, ale platformy pro nový druh softwaru.

V roce 2014 rusko-kanadský programátor Vitalik Buterin popsal ve svém whitepaperu [5] nový blockchainový protokol Ethereum s Turingově úplným skriptovacím jazykem. [3] Tato nová platforma umožňovala komukoliv vytváření komplexních programů interagujících s blockchainem a měnící jeho stav, vytváření specifických pravidel vlastnictví, automatizované zasílání a přijímání transakcí a to vše zabezpečené konsenzem definovaným Nakamotou.

V případě Bitcoinu se často setkáme s pojmem "distributed ledger". Pro tento kontext se jedná o decentralizovaně uchovávaný řetěz bloků transakcí. Díky takto uloženým a nezměnitelným informacím je možné určit přesný počet mincí na adresách a zabránit tak tzv. double-spend problému. Ethereum má svou vlastní nativní měnu Ether, ale také obsahuje komplexnější funkcionalitu smart kontraktů, proto Ethereum definuje pojem "distributed state machine".

[3] Přesnější definicí je Turingově pseudo-úplný stroj, protože protokol má specifické limitace, které nedovolují úplnost Turingova stroje.

1.1 Ethereum Virtual Machine

Distribuovaný virtuální stroj je sdílená datová struktura, která mimo adres a jejich zůstatků uchovává také "machine state". Jedná se o stav, ve kterém se stroj v daný moment nachází. Jednomu konkrétnímu času odpovídá vždy právě jeden konkrétní stav. Tento stav se může měnit po každém bloku transakcí na základě předdefinovaných pravidel, ale může také vykonávat na dotaz transakcí svůj kód.[6]

1.1.1 Stav EVM

Stavem v kontextu EVM je datová struktura zvaná "modified Merkle Patricia Trie", která udržuje všechny účty propojené pomocí jejich hašů redukovatelných na jediný kořenový haš uložený v blockchainu. Tento strom obsahuje veškeré informace, jako je počet mincí na adresách, stavy a data jednotlivých kontraktů, případně jakákoliv jiná data zaznamenatelná v digitální podobě. Etherem je tedy řetěz těchto stavů, nikoliv jen dat, jako je tomu v případě BTC.[6]

Modified Merkle Patricia Trie

Modifikovaný Merkle Patricia trie (strom) je jednou z hlavních datových struktur v síti Ethereum, která vznikla kombinací Patricia Trie a Merkle s několika optimalizačními modifikacemi. Jedná se o plně deterministickou strukturu mapující klíče na hodnoty, kde pro stejný pár (klíč, hodnota) je garantovaný vždy stejný strom. Nejsilnější stránkou této datové struktury je složitost $O(log(n))$ pro vkládání, náhled a mazání jednotlivých prvků. Ethereum využívá tuto datovou strukturu pro ukládání stavů EVM.

Patricia Trie používá cestu jako klíč, proto uzly, které sdílejí stejný prefix, mohou také sdílet stejnou cestu. Tato struktura je nejrychlejší při hledání společných prefixů a vyžaduje malou paměť. Proto je běžně používaná v systémech s omezenými paměťovými dispozicemi, jako jsou například routery.

Modifikovaný Merkle Patricia strom poskytuje trvalou datovou strukturu pro mapování libovolně dlouhých polí bajtů. Je definován jako datová struktura pro mapování dvojice 256bitového fragmentu a libovolně dlouhého bajtového pole. Hlavním požadavkem stromu je poskytnutí konkrétní hodnoty, která jednoznačně identifikuje danou dvojici klíč-hodnota.[7]

[Formálně definovaná vstupní hodnota J jako soubor párů bajtových sekvencí s unikátními klíči (klíč-hodnota)]

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

V modifikovaném Merkle Patricia stromu rodičovský uzel odkazuje na uzel potomka pomocí hashe (Keccak-256) podstromu s kořenem v uzlu potomka. Hash podstromu vzniká jako Keccak-256 hash serializovaných dat uložených v

uzlu. Součástí serializovaných dat mohou právě být i Keccak-256 hashe odkazující na uzly v hloubce stromu o jedna vyšší. V implementaci modifikovaných Merkle Patricia stromů se klíče ve formě sekvence bytů rozkládají na sekvence nibblů, kde jeden nibble je 4-bitová hodnota (jeden bajt jsou dva nibbly). Ve stromu se rozlišují tři typy uzlů. První typ uzlu je list, který je skutečně listem z pohledu stromové struktury grafu. V uzlu je uložena libovolná sekvence (podsekvence) nibblů a jedna hodnota odpovídající klíči, který odpovídá tomuto uzlu. Další typ uzlu je rozšíření. Rozšíření optimalizuje strom v tom, že umožňuje uložit libovolnou sekvenci (podsekvenci) nibblů klíče a odkazuje na jeden uzel v nižší hladině stromu. Poslední typ uzlu je větvení. Tento uzel obsahuje 17 položek, kde prvních 16 položek jsou odkazy (Keccak-256) hashe na podstromy modifikovaného Merkle Patricia stromu. Pořadí odkazu (0 až 15) odpovídá jednomu nibblu klíče. Poslední položka uzlu je hodnota uložená ve stromu, která odpovídá tomuto uzlu (nibblům vzniklým průchodem od kořene stromu k tomuto uzlu).[8]

1.1.2 Přechodová funkce EVM

Přechod z jednoho stavu do druhého funguje jako matematická funkce:

$$Y(S, T) = S'$$

kde pro jednu vstupní hodnotu produkuje jeden deterministický výstup. Pro původní validní stav (S) a soubor validních transakcí (T), přechodová funkce $Y(S, T)$ produkuje nový validní stav jako výstup S'. Tento přechod se dle definice nazývá "state transition function". Ethereum transakce 1.3 představují pomyslný most a tedy jediný způsob, jakým uskutečnit přechod mezi stavem (S) a stavem (S'). [6]

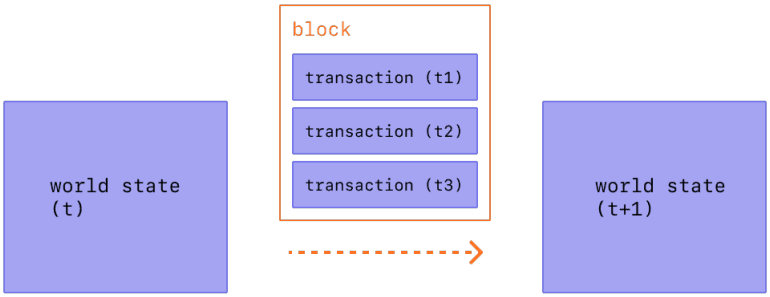
Aby byl přechod mezi stavy úspěšný musí být transakce validní a splňovat určité podmínky:

- transakce musí mít správný formát definovaný v Yellowpaperu [8],
- jmění Etheru odesílatele musí být větší nebo rovno množství posílaných etherů + požadovaných poplatků za exekuci transakce.

V případě nesplnění některé z těchto podmínek dojde k selhání transakce a stav EVM bude vrácen do předchozího stavu. Výjimkou zůstávají poplatky 1.4, které budou odeslány těžaři 1.6 daného bloku. Pokud nedojde k vyčerpání všech poplatků předdefinovaných k exekuci transakce, dojde k jejich vrácení odesílateli transakce.[8]

1.1.3 Instrukce EVM

Virtuální stroj EVM exekuuje transakce jako zásobníkový počítač s hloubkou 1024 položek. Každá položka je 256bitové slovo. Tato bitová délka byla zvolena kvůli kompatibilitě s kryptografickými a podpisovými algoritmy EVM



Obrázek 1.1: Přechod ze stavu (t) do stavu $(t+1)$ [9]

(Keccak-256, secp256k1). Během exekuce je využívána přechodová paměť (pole bajtů adresovaných slov), která není mezi transakcemi zachována. Kontrakty mají svou paměť v podobě Merkle Patricia trie. Tato paměťová struktura spojuje adresu kontraktu s jeho současným globálním stavem.

Zkompilovaný smart kontrakt je exekuován jako sekvence EVM bajt kódových instrukcí (EVM opcodes). EVM obsahuje více než 140 instrukcí. Od základních logických a aritmetických operací až po instrukce specifické pro blockchain jako je například instrukce $CALLER$, která vrací adresu účtu, která je přímo zodpovědná za danou exekuci transakce.

1.2 Ethereum účty

Účty na síti Ethereum a jiných blockchainech si lze představit podobně jako IP adresy na webu. Jedná se o uzly, mezi kterými probíhá veškerá aktivita na blockchainu. EVM je deterministický stroj, který se v jeden daný moment nachází právě v jednom konkrétním *stavu*. Tento stav je definován objekty, které se nazývají účty (accounts). Každý tento objekt obsahuje tyto hodnoty:

- ***Adresa*** - Každý účet má svou originální adresu dlouhou 20 bajtů.
- ***Nonce*** - Nonce je továrně nastavena na hodnotu nula a po každé transakci je inkrementována. To zabezpečuje, že každá transakce bude provedena právě jednou. Aby účet mohl vytvořit kontrakt, jeho nonce musí být rovna nule.
- ***Zůstatek ETH*** - Proměnná znázorňující zůstatek mincí Etheru na daném účtu.

- ***Kód kontraktu*** - Pokud se jedná o účet smart kontraktu, nachází se v tomto poli jeho kód.
- ***Úložiště kontraktu*** - Továrně prázdné.

Na Ethereum existují dva druhy účtů.

- ***Externě vlastněný účet(EOA - Externally Owned Account)*** - Jedná se o běžné účty, které používá většina uživatelů, jsou kontrolovány pomocí privátních klíčů a neobsahují žádný kód. Účty drží Ether nebo jiné ERC20 tokeny. Tyto účty již nemohou vytvořit smart kontrakt.
- ***Účet kontraktu (CA - Contract Account)*** - Účty, které stojí za vytvořením smart kontraktu. Tento druh účtů není kontrolován externím uživatelem, ale o veškeré jeho aktivity se stará jeho zdrojový kód. Kód je exekuován vždy, když dojde k přijetí transakce. Smart kontrakt může číst z interního úložiště, posílat transakce, nebo dynamicky vytvářet nové smart kontrakty. Každá adresa může vytvořit, a tedy být vlastníkem právě jednoho kontraktu. O tuto podmínku se stará proměnná nonce. V jediném stavu, když je nonce rovna nule, je možné vytvořit smart kontrakt, poté se nonce inkrementuje o 1.

1.3 Ethereum transakce

Transakce je jedna kryptograficky podepsaná instrukce vytvořená externím uživatelem (EOA). Odesílatelem transakce nemůže být kontrakt. Yellowpaper definuje dva základní druhy transakcí. Ty, které končí voláním zprávy (message-call), a ty, které vedou k vytvoření smart kontraktu.

Kontrakty mají možnost posílaní zpráv (message). Zpráva je ve své podstatě to stejné jako transakce, s tím rozdílem, že je vytvořená kontraktem pomocí EVM CALL kódu, nikoliv pomocí EOA.

Ethereum transakce obsahuje následující parametry:

- ***type*** - Typ je roven 0 pro legacy transakce, typ je roven 1 dle definice EIP-2930 [10].
- ***nonce*** - Počet transakcí odeslaný odesílatelem.
- ***gasPrice*** - Počet Wei k zaplacení za jednu jednotku gas za výpočty, které budou následkem transakce.
- ***gasLimit*** - Maximální přípustný počet gas použitelný pro exekuci transakce.
- ***to*** - 160bitová adresa příjemce. Při vytvoření kontraktu je pole rovno 0.

- ***value*** - Počet posílaných Wei příjemci. Při vytváření kontraktu je částka přidělena nově vzniklému kontraktu.
- ***r, s*** - Kryptografické parametry podpisu určené k ověření odesílatele.
- ***init*** - Pole bajtů specifikující EVM kód pro vytváření kontraktu. Tento kód je exekuován pouze jednou při vzniku kontraktu.
- ***data*** - Volitelné pole bajtů specifikující vstupní data pro message call.

Parametry *to*, *(r, s)* a *value* jsou standardem pro všechny kryptoměny. Pole *data* nemá žádnou výchozí funkci, ale existuje EVM instrukce, která umožňuje kontraktu přístup k tomuto datovému poli, což umožňuje předávání aditivních dat mezi kontrakty. Pole *gasPrice* a *gasLimit* jsou hlavními limitujícími body Ethereum infrastruktury. Tyto hodnoty striktně omezují komplexitu a výpočetní složitost programů uložených v podobě smart kontraktů. Nicméně tyto omezující podmínky jsou nezbytné z několika důvodů:

- pro zamezení útok DoS (Denial of Service), ke kterému by stačila jedna nekonečná smyčka,
- pro zamezení obecného plýtvání výpočetním výkonem.

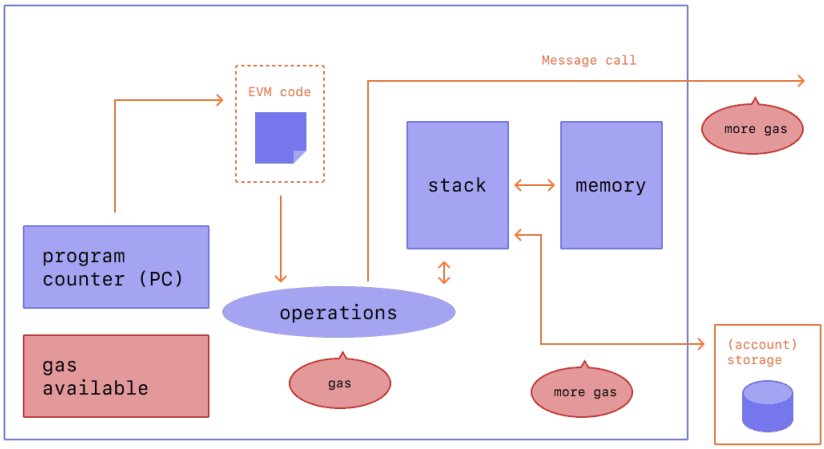
Každá transakce tedy musí definovat limit, kolik výpočetních kroků při provádění kódu může provést. K tomu účelu se používá již několikrát zmiňovaná jednotka *gas*.

1.4 Gas a měny

Poplatky jsou nezbytnou součástí blockchainových protokolů. Jedná se o jednotku měřící potřebné výpočetní úsilí k provedení operací na síti Ethereum. Protože každá transakce vyžaduje výpočetní zdroje k její exekuci, za každou transakci je potřeba zaplatit poplatek. Jednotka *Gas* označuje poplatek nutný k úspěšnému provedení transakce. Gas je odesílán uzlům (počítačům) sítě, které poskytují svůj výpočetní výkon pro její chod. Tyto uzly se nazývají *těžaři* 1.6 a Gas poplatky jsou motivací k propůjčení jejich HW.[11]

Nativní měna sítě Ethereum pro placení Gas je *Ether* (ETH). Částky jsou uváděny v jednotkách *gwei*, kde se jeden gwei rovná 0.000000001 neboli 10^{-9} ETH. Tato jednotka byla zavedena pro vyšší srozumitelnost při manipulaci a popisu malých ETH částek. Namísto zmatečné částky 0.0000001 ETH je častěji používáno 100 gwei. Slovo "gwei"vzniklo ze slova "giga-wei"[4], které značí 1 miliardu *wei*. Nejmenší jednotkou Ethereum sítě je proto *wei*.

[4]Jednotka wei je pojmenována po čínském počítačovém inženýrovi Wei Dai, který se podílel na vývoji měny b-money.[12]



Obrázek 1.2: Znázornění EVM operací, které pro jejich provedení vyžadují Gas [9]

Tabulka 1.1: Základní měny na síti Ethereum

Měnová jednotka	Relativní hodnota vůči Wei
Ether	10^{18}
GWei	10^{9}
Wei	10^{0}

1.5 Ethereum blok

Blok je kolekce relevantních informacích o daném bloku (hlavička bloku), společně s hlavičkami rodičovských bloků a informacemi o množině transakcí. Protože každý blok obsahuje haš hlavičky předchozího bloku, je zajištěna bezpečnost, a tedy nemožnost měnit informace bloků předchozích, aniž by se to projevilo na blocích navazujících.

V kapitole o přechodu mezi jednotlivými stavy 1.1.2 byl jako vstup do funkce, mimo původního stavu, soubor validních transakcí (T). Tímto souborem je právě blok. Bloky jsou vytvářeny přibližně každých 15 vteřin. Transakce odeslané do sítě čekají v "mempoolu"do okamžiku, než jsou jedním z těžařů vloženy do bloku, který je následně těžařem vytěžen.

Aby byla transakce vybrána k přiřazení do bloku a k následnému vytěžení, je těžařem ověřeno, že se jedná o validní transakci, která splňuje všechny požadavky pro její exekuci. Výběr a pořadí konkrétních transakcí udává koncept MEV (Maximal Extractable Value). MEV odkazuje na maximální možnou hodnotu, kterou lze získat z produkce bloku nad rámec odměny za jeho vytěžení a poplatků Gas [5]. Těžaři proto řadí transakce na základě nabízených poplatků.

[5]Řazení transakcí na základě nabízených poplatků vytváří na Ethereum síti vysoce kom-

Z tohoto důvodu se může čas vytěžení a zprocesování transakcí výrazně lišit a tento čas lze modifikovat nabídkou vyšších poplatků. Po ověření a zahrnutí transakcí do bloku jsou všechny transakce exekuovány na lokálním EVM. Po následném úspěšném ověření legitimity těžaře přes "proof-of-work", neboli těžbu 1.6, je blok rozeslán do zbytku sítě.[5]

Hlavičky bloku obsahují následující parametry[8]:

- ***parentHash*** - Keccak 256 haš hlavičky rodičovského bloku.
- ***ommersHash*** - Keccak 256 haš hlavičky příbuzného bloku.
- ***beneficiary*** - 160bitová adresa příjemce poplatků.
- ***stateRoot*** - Keccak 256 haš kořene stavového stromu po exekuci a finalizaci všech transakcí.
- ***transactionsRoot*** - Keccak 256 haš kořene stromu obsahující všechny transakce daného bloku.
- ***receiptsRoot*** - Keccak 256 haš kořene stromu obsahující příjemce transakcí daného bloku.
- ***logBloom*** - Bloomův filtr složený z indexovatelných informací obsažených v každém záznamu protokolu v seznamu transakcí.
- ***difficulty*** - skalární hodnota složitosti vytěžení daného bloku.
- ***number*** - skalární hodnota udávající pořadové číslo daného bloku.
- ***gasLimit*** - skalární hodnota udávající maximální přípustný gas pro exekuci daného bloku.
- ***gasUsed*** - skalární hodnota udávající gas spotřebovaný všemi transakcemi daného bloku.
- ***timestamp*** - skalární hodnota rovnající se Unixovému času při založení daného bloku.
- ***extraData*** - volitelné pole bajtového pole dat relevantních k danému bloku (max 32 bajtů).
- ***mixHash*** - 256bitový haš, který společně s hodnotou *nonce* prokazuje provedení dostatečného množství výpočtů.
- ***nonce*** - 64bitový haš, který společně s hodnotou *mixHash* prokazuje provedení dostatečného množství výpočtů.

petitivní prostředí pro arbitrážní obchodování a umožňuje takzvaný "Frontrunning", který může uživatelům značně znepříjemnit používání sítě. Problematikou se zabývá projekt Flash Boys.[13]

1.6 Těžba bloků

Aby byl EVM stav změněn 1.1.2 na základě vytvořené transakce, musí dojít k vytěžení bloku, který danou transakci obsahuje. Proces těžby bloků a zároveň bezpečný chod celé sítě obstarává protokol o důkazu práce (Proof-of-work). Protokol Proof-of-work byl definován v Bitcoinovém whitepaperu.[1] Základním konceptem protokolu je odměna v nativní měně daného blockchainu, která je přiřazena těžaři, jež daný blok vytěží.

Pro haš každého bloku jsou dána specifická kritéria, které se mění podle výpočetní síly celé sítě. Konkrétně se jedná o počet nul na začátku výsledného haše. Úkolem těžařů je náhodně tipovat hodnotu *nonce*, se kterou výsledný haš daného bloku splní kritéria. Vzhledem k výpočetní náročnosti se jedná o exponenciálně rostoucí funkci, jejíž správnost je zpětně velmi snadno ověřitelná. Toto ověření provádí zbytek sítě, se kterou těžař vytěžený blok sdílí. Poté co se ostatní uzly sítě shodnout na korektní validnosti bloku, je daný blok zařazen na konec blockchainového řetězce a globální stav EVM je aktualizován.[1]

Kapitola 2

Smart kontrakty

Protože se tato práce věnuje nástrojům usnadňujícím práci při vývoji a bezpečnostním auditu smart kontraktů, v této podkapitole budou shrnuty informace týkající se právě smart kontraků.

Smart kontrakt je program uložený a běžící na Ethereum blockchainu. Jedná se o množinu kódu a dat náležící specifické adrese. Smart kontrakt je jedním ze dvou druhů účtů 1.2, které v Ethereum síti existují. Účet kontraktu má vlastní 20bajtovou *adresu*, *Nonce* rovnající se jedné, ETH *zůstatek*, *kód* a své vlastní *úložiště*. Kontrakty nejsou kontrolovány uživatelem, ale veškeré jejich akce jsou definovány uloženým kódem.

Pro vývoj jsou v tuto chvíli používány především dva programovací jazyky - Solidity a Vyper, které nabízejí syntaxi velmi podobnou známým programovacím jazykům jako je C++ a Python. Pokročilejší vývojáři využívají v kritických částech kódu jazyk Yul, který se podobá operačnímu kódu EVM. Jedná se low-level jazyk, díky kterému je možné interagovat přímo s operačním kódem a tím docílit například efektivnějších výpočetních operací.

2.1 Solidity

Dominantní zastoupení mezi programovacími jazyky pro Ethereum má především jazyk Solidity. Solidity je objektově orientovaný vysoko-úrovňový jazyk vyvíjený Ethereum Foundation a je určený přímo pro implementaci Ethereum smart kontraktů. Jazyk je staticky typovaný, podporuje dědičnost, využití knihoven, využití vlastních datových typů. Syntaxe jazyka je silně ovlivněna jazykem C++, ale využívá i koncepty jazků Python a Javascript. Současná verze jazyka nese označení **v0.8.13**. Soubory se zdrojovým kódem jazyka Solidity se ukládají s příponou **.sol**.

Ve světě smart kontraktů je zvykem veřejně přístupný zdrojový kód. To zejména kvůli transparentnosti, která je pro alternativní finanční svět žádaná a vzbuzuje větší důvěru uživatelů. Z toho důvodu je doporučeno na první řádek zdrojového kódu uvádět licenci ve tvaru:

2.1: Formát licence ve smart kontraktu

```
// SPDX-License-Identifier: MIT
```

Kompilátor jazyka Solidity nevaliduje platnost licence. V případě, že kód není veřejně přístupný, nebo si autor kódu nepřeje specifikovat licenci, je doporučeno uvést klíčové slovo *UNLICENSED*, které zakazuje použití zdrojového kódu. Toto klíčové slovo může být snadno zaměnitelné za jiné klíčové slovo *UNLICENSE*, které dává komukoliv práva na využití zdrojového kódu.

2.1.1 Pragma

Klíčové slovo **pragma** je používáno k definování verze kompilátoru pro daný zdrojový kód. Použití klíčového slova není vyžadováno, ale je silně doporučeno. Při nedefinování klíčového slova je kód automaticky kompilován nejnovější verzí kompilátoru, který nemusí být kompatibilní se zastaralými postupy zdrojového kódu. Zdrojový kód může být kompatibilní s několika verzemi kompilátoru, proto není nutné specifikovat jednu konkrétní. Definici lze tedy deklarovat několika způsoby s použitím operátorů a tím definovat různé intervaly verzí. Několik příkladů pro klíčové slovo pragma:

2.2: Ukázka příkladného použití klíčového slova *pragma*

```
pragma solidity 0.8.1;
pragma solidity ^0.8.1;
pragma solidity 0.7.5-0.8.2;
pragma solidity >=0.7.5<=0.8.2;
pragma solidity 0.8.x;
pragma solidity >0.8.0;
```

Druhým využitím klíčového slova **pragma** je výběr verze kódování ABI (Application Binary Interface). Továrně je použita verze **v2** pro verzi Solidity 0.8.0 a vyšší.

2.3: Formát slova pragma pro nastavení verze kódování

```
pragma abicoder v1;
pragma abicoder v2;
```

Posledním a vzácným využitím klíčového slova **pragma** je SMT (Satisfiability Modulo Theories) ověřování kódu během kompilace. Pro využití této funkcionality musí být použit speciální kompilátor pro režim SMT.

2.4: Formát slova pragma pro funkcionalitu SMT

```
pragma experimental SMTChecker;
```

2.1.2 Importy

Solidity podporuje příkaz **import** nazývaný *import path* (importní cesta), který napomáhá k modularizaci zdrojového kódu.

2.5: Formát importování souborů v jazyce Solidity

```
import "filename";
```

Tento příkaz importuje všechny globální symboly ze zdroje *filename* do aktuálního globálního prostředí. Podobně jako v jazyce Python je možné naimportovat pouze konkrétní funkce/proměnné, nebo globální symbol pojmenovat.

2.6: Alternativní způsob importování souborů v jazyce Solidity

```
import x from "filename";
import "filename" as fl;
```

Aby bylo možné podporovat reprodukovatelné sestavení na všech platformách, musí překladač Solidity odfiltrovat ze zdrojových cest znaky specifické pro jednotlivé operační systémy. Z tohoto důvodu neodkazují cesty přímo na soubory v souborovém systému. Místo toho kompilátor udržuje interní databázi (VFS), kde je každé zdrojové jednotce přiřazen jedinečný název sloužící jako identifikátor. Importní cesta zadaná v příkazu importu je přeložena na jedinečný název a použita k vyhledání odpovídající zdrojové jednotky v této databázi.

2.1.3 Komentáře

Solidity používá pro jednořádkové komentáře znaky "`//`"a pro víceřádkové komentáře "`/*...*/`". Jazyk podporuje i formát NatSpec [6], který je používán k anotaci funkcí. V případě NatSpec existují pro jednořádkové komentáře znaky "`///`"a pro víceřádkové komentáře "`/**...*/`".

2.7: Ukázka NatSpec formátu

```
/// @param x value to add
/// @dev perform adding operation to stored value y and input x
```

2.1.4 Struktura kontraktu

Klíčové slovo **contract** značí instanci kontraktu v Solidity, která se podobá třídám z objektově orientovaných jazyků. Každý kontrakt může obsahovat *State Variables, Functions, Function Modifiers, Events, Errors, Struct Types and Enum Types*. Kontrakty od sebe mohou dědit a existují speciální kontrakty *Libraries* a *interfaces*.

Globální proměnné

Globální proměnné jsou deklarovány mimo funkce a jsou permanentně uloženy v úložišti smart kontraktu. Pro číselné datové typy Solidity využívá pouze INT a UINT. V případě potřeby typu s přesností desetinné čárky se využívají konstanty, kterými jsou typy INT, případně UINT vynásobeny a tímto způsobem

[6] Formát komentování kódu určený pro anotaci aplikačního rozhraní aplikace

je nasimulována větší přesnost. Specifickým typem pro Solidity je **address**. Jedná se o 160bitovou adresu Ethereum účtu.[11]

Funkce

Funkcí se rozumí spustitelná část kódu, může vyžadovat vstupní data a vracet výstupní data. Funkce může měnit stav kontraktu nebo pouze číst jeho data. Každá funkce musí být definována pomocí jednoho z klíčových slov definující její viditelnost.

- ***public*** - Funkce je přístupná z ostatních funkcí kontraktu, z kontraktů potomků i ze vně kontraktu.
- ***internal*** - Funkce je přístupná z ostatních funkcí kontraktu a z kontraktů potomků.
- ***external*** - Funkce je přístupná pomocí transakcí z externích adres, nebo jiných kontraktů.
- ***private*** - Funkce je přístupná z ostatních funkcí kontraktu, nikoliv pro potomky kontraktu.

Modifiers funkcí

Důležitým bezpečnostním aspektem smart kontraktů jsou takzvané **Function Modifiers**. Modifier slouží jako podmínka pro přístup k dané funkci. Jedná se o speciální funkci, která definuje určitá pravidla a podmínky. Modifier je přiřazen ke standardní funkci stejně jako ostatní klíčová slova. Při volání funkce s přiřazeným Modifierem se jako první vykoná kód Modifieru. Jsou-li podmínky splněny, je následně volána funkce samotná. Pokud podmínky nejsou úspěšně splněny, přístup k funkci je odepřen.

V ukázkovém kusu kódu je zobrazen modifier *OnlyOwner* který povoluje volání funkce pouze adrese, která stojí za vytvořením kontraktu. Modifier *OnlyWhitelisted* opravňuje k volání pouze adresy z mappingu *whitelist*. Symbol _ definuje pozici, kde má být funkce, ke které je modifier přiřazen, volána.

```
contract Storage {
    mapping(address => uint256) public storedToken;
    mapping(address => bool) public whitelist;
    address owner;

    constructor() {
        owner = msg.sender
    }

    modifier OnlyOwner() {
        require(owner == msg.sender, "Not␣the␣owner");
        _;
    }
```

```
    modifier OnlyWhitelisted() {
        require(whitelist[msg.sender], "Not␣in␣whitelist");
        _;
    }

    function IncreaseToken() external OnlyWhitelisted {
        storedTokenp[msg.sender] += 1;
    }

    function updateWhitelist(address newMember) external
        OnlyOwner {
        whitelistp[newMember] == true;
    }

}
```

Události

Události (events) představují abstrakci nad funkcí logování EVM. Externí aplikace se mohou přihlásit k odběru těchto událostí a poslouchat je prostřednictvím rozhraní RPC klienta Ethereum. Události jsou dědici kontraktů. Při jejich volání se argumenty uloží do speciální datové struktury transakce. Záznamy o datech událostí nejsou přístupné zevnitř kontraktů.

Error

Chyby umožňují definovat a popsat údaje pro případy selhání kontraktu. Chyby lze použít v příkazech revert. Ve srovnání s popisnými řetězci (textový řetězec v require objektu) jsou chyby mnohem levnější a umožňují kódovat další údaje. K popisu chyby pro uživatele můžete použít NatSpec.

Typ Struct

Pomocí typu *Struct* (struktura), lze v jazyce Solidy definovat vlastní komplexnější datové typy. Pomocí struktur je možné sjednotit více datových typů do jednoho.

2.8: Ukázka použití datového typu Struct

```
contract Storage {
    struct UserData {
        address addr;
        string name;
        uint256 date;
    }
    mapping(address => UserData) public storedData;
    function getDate() exetrnal view returns (uint256) {
        return storedData[msg.sender].date;
    }
```

```
}
```

Ke konkrétnímu prvku struktury se přistupuje pomocí tečky. Struktury je možné používat stejně jako ostatní datové typy například v mapování, jako je znázorněno v ukázce v výše.

Typ Enum

Enumy jsou jedním ze způsobů, jak v Solidity vytvořit uživatelsky definovaný typ. Jsou explicitně převoditelné ze všech celočíselných typů, ale implicitní konverze není povolena. Pomocí enumu je možné omezit daný datový typ pouze na předem specifikované hodnoty.

2.9: Ukázka použití datového typu Enum

```
contract Storage {
    enum StorageSize{ SMALL, MEDIUM, LARGE }
    StorageSize ram;

    function setLarge() public {
        ram = StorageSize.LARGE;
}
```

Datové lokace

Každý referenční typ má další anotaci, "data location", definující, kde je uložen. Existují tři datová umístění.

- ***memory*** - Proměnné jejichž čas je limitován externím voláním funkce.
- ***storage*** - Lokace, kde jsou uloženy globální stavové proměnné, jejichž čas je limitován existencí kontraktu.
- ***calldata*** - Speciální datové lokace pro argumenty funkcí.

2.2 Vytvoření kontraktu

Smart kontrakt může být vytvořen dvěma různými způsoby. Pomocí EVM funkce *CREATE* a *CREATE2*. Při vytváření kontraktu se používají následjicí parametery:

- ***(s) sender*** - Odesílatel transakce.
- ***(o)original transactor*** - Originální odesílatel se může lišit od (s) v případě, že se jedná o message call, nebo v případě, že vytvoření kontraktu není exekuováno transakcí, ale přímo EVM kódem.
- ***(g) available gas*** - Gas 1.4 dostupný na adrese odesílatele.

- ***(g) gas price*** - Cena za Gas.
- ***(v) endowment*** - Počet Wei 1.1 odeslaný na adresu kontraktu s polem bajtů volitelné délky.
- ***(i) init*** - Inicializační EVM kód zodpovědný za vytvoření kontraktu.
- ***(e)*** - Hloubka zásobníku transakce, případně message-callu.
- ***(ζ) salt*** - Sůl (v případě použití EVM kódu $CREATE\ \zeta = \theta$).
- ***(w)*** - Povolení k provádění stavových modifikací.

Funkce pro vytváření kontraktu Λ 2.2 s výše definovanými vstupy společně se stavem σ a vznikajícím stavem A je rovna souboru hodnot obsahující nový stav σ', zbývající Gas, podstav transakce A', status kód z a výstup o.

[Formálně definovaná funkce pro vytváření kontraktu Λ]

$$(\sigma', g', A', z, o) \equiv \Lambda(\sigma, A, s, o, g, p, v, i, e, \zeta, w)$$

Adresa nově vznikajícího kontraktu je získána z nejvíce vpravo uložených (nejméně významných) 160 bitů výstupu funkce:

$$Keccak256(RLP(sender, nonce)$$

V případě *CREATE2* 160 nejméně významných bitů výstupu funkce, kde "$\cdot$"značí operaci zřetězení polí bajtů:

$$Keccak256(0xff \cdot sender \cdot \zeta \cdot Keccak265(i))$$

Proměnná *nonce* je nastavena na hodnotu 1, ETH zůstatek kontraktu na hodnotu v, úložiště kontraktu je prázdné ($Trie(\theta)$), haš kódu je roven $Keccak256()$ a od zůstatku adresy (s) je odečtena odeslaná hodnota (v). Následně probíhá exekuce inicializačního EVM kódu (i) kontraktu. Tato exekuce může ovlivnit několik neinterních stavových událostí. Může dojít ke změně úložiště kódu, k vytvoření dalších účtů nebo k volání zpráv. Exekuční funkce Ξ s přechodným stavem σ^*, hodnotou *Gas* g, nově transakčním podstavem A^* a s parametry exekučního prostředí I je rovna souboru hodnot $(\sigma^{**}, g^{**}, A^{**}, o)$ obsahujícímu nový stav, zbývající *Gas*, výsledný podstav transakce a kód účtu. Exekuční prostředí I obsahuje sadu parametrů nezbytných k úspěšné exekuci transakce. Pokud jakýkoliv z parametrů nesplní exekuční podmínky, transakce bude přerušena a vrácena do původního stavu.

Po úspěšném provedení exekuční transakce je zaplacen poplatek c za vytvoření kontraktu a za jeho uložení, který je úměrný velikosti vytvořeného smart kontraktu. Při nedostatečné hodnotě *Gas* ($\sigma^{**} < c$) bude vyovolána out-of-gas podmínka. V takovém případě bude zbývající *Gas* roven nule, tj. pokud transakce pro vytvoření kontraktu byla přijata, tak není ovlivněna platba za

tvorbu kontraktu. Nicméně data transakce nejsou odeslána na adresu kontraktu, a proto kód není uložen. [7] Pokud podmínka není vyvolána, zbývající *Gas* je navrácen originálnímu odesílateli transakce a nově vytvořený stav nabude povolení k jeho přetrvání.

Během vykonávání inicializačního existuje adresa nově vznikajícího kontraktu, nicméně adresa neobsahuj žádný kód. Proto v tuto chvíli nedojde v exekuci žádného kódu při přijetí jakékoliv zprávy. Pokud inicializační kód obsahuje instrukci *SELFDESTRUCT*, účet bude smazán ještě před dokončením transakce. [8]

2.3 Interakce s kontraktem

Interagovat s kontraktem lze dvěma různými způsoby.

- ***Call*** - Jedná se o lokální exekuci funkce smart kontraktu, která není vytěžena a její stav není rozesílán do Ethereum sítě. Operace při volání pomocí *Call* jsou určeny pouze ke čtení dat a nestojí žádný *Gas*. Volání simuluje reálnou transakci, nicméně po získání dat jsou všechny změny stavu vráceny do stavu původního. Funkce smart kontraktu, které tento formát dotazu umožňují, jsou deklarovány s klíčovými slovy *view, pure, constant*. Tato klíčová slova obecně definují, že nedojde ke změně stavu smart kontraktu.

- ***Transaction*** - Standardní transakce, která mění stav EVM, je vytěžena a výsledný stav je rozeslán do Ethereum sítě. Transakce volá funkce smart kontraktu, jejichž exekuce vede ke změně stavu kontraktu, který je nutné aktualizovat v celé síti.

[7] Část Ethereum Yellowpaper [8], popisující akce po vyvolání podmínky out-of-gas, nebyla jednoznačně popsána. Autor práce proto navrhl úpravu textu s explicitnějším popisem akcí, které nastanou po vyvolání dané podmínky. Úprava byla přijata a začleněna. [14]

[8] EVM instrukce *SELFDESTRUCT* slouží ke smazání smart kontraktu. Pokud je v kódu kontraktu implementována funkce obsahující danou instrukci, kontrakt je při zavolání smazán a zůstatek účtu je odeslán na zvolenou adresu

Kapitola 3

Nástroje pro Ethereum

Vývoj decentralizovaných aplikací se značně liší oproti tvorbě běžnějších softwarových produktů. Důvodem je unikátní architektura virtuálního počítače EVM1.1, pro který je software vyvíjen. Právě tato architektura přináší výhody využití blockchainu, ale také několik limitací a bezpečnostních rizik, které je třeba mít na paměti. Nástroje určené k vývoji produktu mají tedy značně jiné požadavky než ty, které se používají ve standardním IT odvětví.

Vývoj kontraktu - Prvním bodem je vývoj samotného smart kontraktu. Během implementace kódu hraje důležitou roli jednotka Gas 1.4. Gas vyjadřuje cenu za vykonání transakce. Každá ze 141 EVM instrukcí má přiřazený Gas na základě její paměťové a výpočetní náročnosti. Gas kontraktu je tedy součtem zkompilovaných bajtkódových instrukcí, které je třeba vykonat a zaplatit pomocí $ETH.

Tento fakt značně limituje a zároveň udávává směr vývoje decentralizovaných aplikací. Mimo vysoké výpočetní ceny za vykonání strojového kódu je zde také druhý limitují faktor - velikost kódu. 24.576 kb je maximální povolená paměť pro jeden smart kontrakt. Při překročení tohoto limitu dojde k 'revertu' transakce a kód nebude možné nikdy vykonat. Přestože existují programovací jazyky, jako je Solidity, nebo Vyper, je vhodné při vývoji uvažovat na úrovni EVM pro dosažení maximální efektivity výpočetních operací.[15]

Testování kódu - Testování hraje velkou roli při implementaci jakéhokoliv softwaru. Ale na rozdíl od toho běžného, je kód uložený na blockchainu v podobě smart kontraktu ve stavu, kdy jej není možné jakkoliv měnit či mazat (až na speciální případy). Z toho důvodu tento bod hraje kritickou roli při vytváření projektu na platformě Ethereum.

Testnet - Testnetem rozumíme veřejný blockchain, který funguje téměř stejným způsobem jako Mainnet s tím rozdílem, že je určen pro testování a vývoj kontraktů. Veškeré poplatky se platí měnou, která je zdarma k

dostání pomocí faucetu.[9] Tento blockchain funguje pro simulaci aplikací v reálném prostředí. S kontraktem je možné interagovat stejně, jako kdyby byl uložený na Mainnetu. Je dobrým zvykem hotový a otestovaný kód nasadit na testnet a nasimulovat co nejvíce reálných situací pro otestování správnosti implementace.

Nasazení na Mainnet - Mainnet je primární veřejný Ethereum blockchain, na kterém se uchovávají reálné distribuované aplikace. Nasazení na Mainnet je finálním krokem ke spuštění produktu. Tento krok s sebou nese určitá rizika. Například v mnohých kontraktech hraje klíčovou roli adresa, jež kontrakt na Mainnet nasadila. Správce této adresy mívá nadstandardní práva, díky kterým má často moc kontrakt totálně znehodnotit.

Aby byl celý výše zmíněný proces stejně komfortní, jako je tomu zvykem z vývoje tradičního softwaru, vznikají neustále nové nástroje. Ty se zaměřují buď na jednotlivé části, jako je třeba testování, nebo na celý proces tvorby kontraktů. Protože se jedná o poměrně mladý obor, je kvalita těchto nástrojů často zpochybnitelná. V následujících podkapitolách budou představeny nástroje Brownie, Slither a Echidna. Jedná se o jedny z nejvyužívanějších nástrojů v Ethereum ekosystému. Každý z nástrojů se specializuje na jiné funkcionality z výše uvedeného procesu tvorby produktu.

3.1 Brownie

Brownie je vývojové a testovací rozhraní napsané v jazyce Python, zaměřené na EVM smart kontrakty. Autorem nástroje je Ben Hauser, který se mimo Brownie také podílí na vývoji decentralizované směnárny a poskytovatele likvidity pro obchod se stablecoiny s názvem Curve. [10] Přestože je software implementovaný a udržovaný jediným vývojářem, je poměrně hojně využívaný. V komunitě ETH vývojářů se těší oblibě zejména díky jazyku Python, kterým je napsán. Jako "konkurentní"nástroje je možné považovat Hardhat a Truffle. V tuto chvíli se jedná o jediné tři kompletní vývojové prostředí, které uživateli asistují během všech zmiňovaných 3 fází vývoje smart kontraktu. Dokumentace Brownie deklaruje následující funkcionality:

- podpora a kompilace jazyků Solidity a Vyper,
- testování pomocí pythoní testovací knihovny pytest,
- testování pomocí hypothesis,

[9] Faucet (česky kohoutek) je webová stránka, do které je zadána Testnet adresa, na kterou jsou odeslány testovací ETH.

[10] *Stablecoin* je kryptoměna vázaná na hmotné aktivum (například americký dolar.

- debugovací nástroje,
- Brownie konzole,
- podpora package manažeru ethPM.

3.1.1 Kompilace

Při volání příkazu:

3.1: Brownie příkaz ke kompilaci

```
$ brownie compile
```

dojde ke kompilaci všech souborů v podadresáři *./contracts*. Při každém spuštění kompilace se provede porovnání haše zdrojového kódu kontraktu s haši existujících, již zkompilovaných verzí. Pokud nedošlo k žádné změně v kódu, haše se rovnají, a tím pádem nedojde k opětovné zbytečné kompilaci.[16] Nicméně je možné vynutit kompletní rekompilaci všech zdrojových souborů příkazem:

3.2: Brownie příkaz ke kompletní rekompilaci

```
$ brownie compile --all
```

Při neúspěšné kompilaci jednoho nebo více kontraktů Brownie vypíše zdroj kompilačních chyb.

Jak již bylo výše uvedeno, Brownie podporuje dva programovací jazyky:

1. **Solidity (.sol)** verze $>= 0.4.22$
2. **Vyper (.vy)** verze $>= 0.1.0 - beta.16$

Kompilátor

Kompilátor je volen podle přípony souboru a jeho nastavení lze upravit v konfiguračním souboru $brownie - config.yaml$. Tento soubor není vytvořen automaticky při inicializaci projektu. Když dojde ke kompilaci bez vytvořeného .yaml souboru, použijí se tovární hodnoty:

3.3: Ukázka struktury konfiguračního souboru

```
compiler:
    evm_version: null
    solc:
        version: null
        optimizer:
            enabled: true
            runs: 200
    vyper:
        version: null
```

V takovém případě Brownie zvolí verzi kompilátoru podle pragma verze 2.1.1 každého z kontraktů. Pokud daná verze není lokálně k dispozici, dojde k její instalaci.

Při nastavení verze kompilátoru v konfiguračním souboru jsou všechny kontrakty kompilovány danou verzí. Pokud daná verze kompilátoru již není nainstalována, dojde opět k jejímu stažení a instalaci.

V továrním nastavení je automaticky zapnuta optimalizace. Opět je možné její parametry ladit pomocí konfiguračního souboru pod hodnotami

compiler.solc.optimizer

. [17]

Verze EVM

Verze kompileru automaticky nastaví EVM verzi 1.1 podle následujících kritérií:

- **byzantium** pro Solidity <= 0.5.4 ,
- **petersburg** pro Solidity >= 0.5.5 <= 0.5.12,
- **istanbul** pro Solidity >= 0.5.12 a Vyper.

Mezi platné EVM verze, které lze nastavit manuálně, patří byzantium, constantinople, petersburg, istanbul a ETH Classic verze atlantis a agharta. ETH classic verze jsou převedeny na jejich ETH ekvivalenty.

Mapování

Ve smart kontraktech se často setkáme s využitím knihoven třetích stran pomocí importů. Mezi ty nejčastější patří knihovny od firmy Openzeppelin . Tyto knihovny mohou být lokálně nainstalovány různými způsoby, proto je k úspěšné kompilaci potřeba namapovat systémové cesty. Tyto cesty lze vypsat do konfiguračního souboru do pole *compiler.sol.remappings*. Remapping je string ve formátu $prefix = path$. Důležitou poznámkou je fakt, že Brownie není schopné detekovat změny v importovaných souborech, které se nachází mimo kořenový adresář projektu. Projekt je tedy nutné manuálně rekompilovat.

3.1.2 Práce s kontrakty a adresami

Aby bylo s blockchainovým kontraktem možné interagovat, Brownie využívá takzvaný "local chain". Tím je myšlena lokální instance blockchainu určená pro vývoj. To umožňuje pracovat s Ethereum adresami, provádět transakce a interagovat s kontraktem. Zmíněné funkcionality je možné využívat třemi různými způsoby:

1. pomocí Brownie console,
2. ve skriptech určených pro automatizaci procesu nasazení kontraktu na Mainnet,
3. při testování.

Brownie konzole

Používání Brownie konzole se využívá pro přímou interakci s kontrakty Brownie konzole je aktivována příkazem:

3.4: Brownie příkaz ke spuštění interaktivní konzole

```
$ brownie console
```

Konzole je vytvořena pomocí Python interpretru, proto interakce s ní se velmi podobá klasické Python konzoly. Při aktivaci konzole dojde automaticky ke kompilaci kontraktů a jejich nasazení na lokální testovací instanci blockchainu. Poté dojde ke zobrazení příkazového řádku a je možné začít interagovat s kontrakty na základě funkcionalit definovaných Brownie API [16]:

- **brownie**
  - brownie
  - brownie.exceptions
  - brownie._config
  - brownie._singleton
- **brownie.convert**
  - brownie.convert.main
  - brownie.convert.datatypes
  - brownie.convert.normalize
  - brownie.convert.utils
- **brownie.network**
  - brownie.network.main
  - brownie.network.account
  - brownie.network.alert
  - brownie.network.contract
  - brownie.network.event
  - brownie.network.gas
  - brownie.network.multicall

  - brownie.network.state
  - brownie.network.rpc
  - brownie.network.transaction
  - brownie.network.web3

- **brownie.project**
  - brownie.project.main
  - brownie.project.build
  - brownie.project.compiler
  - brownie.project.ethpm
  - brownie.project.scripts
  - brownie.project.sources

- **brownie.test**
  - brownie.test.fixtures
  - brownie.test.strategies
  - brownie.test.stateful
  - brownie.test.plugin
  - brownie.test.manager
  - brownie.test.output
  - brownie.test.coverage

- **brownie.utils**
  - brownie.utils.color

Psaní skriptů

Mimo konzole je možnost psaní skriptů pro testování a automatizování procesu nasazení aplikace. Skripty jsou uloženy v adresáři *./scripts/*. Skripty využívají standardní Python syntaxi. Nadstandardním prvkem je využití Brownie API objektů, ke kterým lze dostat přístup pomocí jejich importování. Nejjednodušší je kompletní import objektů, se kterými se pak pracuje stejným způsobem jako v Brownie konzoly 3.1.2:

3.5: Import brownie balíčku do testovacího souboru

```
from brownie import *
```

Druhou možností je import konkrétních objektů pomocí jejich explicitního vyjmenování:

3.6: Import objektu *accounts* z brownie balíčku

```
from brownie import accounts
```

Skripty jsou exekuovány voláním příkazu:

3.7: Příkaz pro exekuci skriptu

```
$ brownie run <script> [function]
```

Nebo z Brownie konzole:

3.8: Exekuce skriptu z Brownie konzole

```
>>> run('token_script')
```

Tímto způsobem se zavolá *main*() funkce ze souboru $scripts/token_script.py$.

3.1.3 Testování

K psaní unit testů je využívána známá Python knihovna pytest. Výhodou pytest knihovny je přímočarý a snadno pochopitelný kód testů. Pytest automaticky lokalizuje testovací funkce, které musí být v adresáři uloženy podle následujících dvou pravidel:

- testy musí být uloženy v adresáři *tests/*,
- jména souborů ve tvaru $test\_ * .py$ nebo $* \_test.py$.

Následně jsou spuštěny funkce, které splňují podmínky:

- funkce mimo třídu s prefixem *test*,
- metody třídy s prefixem *test*, kde jméno třídy obsahuje prefix *Test* a neobsahuje $\_\_init\_\_$ metodu.

Do testovacího souboru je nutné importovat balíček *brownie*, pomocí kterého bude možné přistupovat k instancím, jako jsou například jednotlivé účty *accounts*. Důležitým prvkem testů jsou *Fixtures*. Jedná se o funkce, které se aplikují na jednu nebo více testovacích funkcí a jsou volány před provedením každého testu. Fixtures se používají k nastavení počátečních podmínek potřebných pro test.

3.9: Ukázka využití *Fixtures* funkce

```
@pytest.fixture
def token(accounts, staking):
    return xToken.deploy(staking, {'from': accounts[0]})
```

V příkladu je uvedena *Fixture*, která nasazuje na lokální blockchain instanci *xTokenu*. Tato operace je nezbytná pro jakoukoliv interakci s *xTokenem*. Z toho důvodu se vytvoří tato speciální funkce, jejíž název je předáván jako vstupní parametr do testovací funkce, kde je třeba mít funkční token. Ke spuštění testů projektu dojde po zadání jednoduchého příkazu:

3.10: Příkaz pro spuštění Brownie testů

```
$ brownie test
```

Příkaz *test* může následovat řada volitelných voleb.

- –**update (-U)** - Spustí pouze testy, které byly pozměněny.
- –**coverage (-C)** - Vypočítá a v tabulce vyobrazí testové pokrytí kontraktů.
- –**interactive -I** - V případě selhání testu se otevře interaktivní Brownie konzole, ve které lze pomocí debugovacích nástrojů hledat zdroj chyby a interagovat s aktuálním stavem kontraktu.
- –**stateful** [**true,false**] - Volba pro ne/spuštění stavových výstupů.
- –**failfast** - Vynucení rychlejšího ukončení testů, při jejich neúspěchu.
- –**revert-tb -R** - Zobrazení detailního trasování k transakci, která vedla k revertu.
- –**gas -G** - Zobrazení konzumace Gas pro jednotlivé funkce a kontrakty.
- –**network** [**name**] - Výběr specifické sítě.
- –**showinternal** - Aditivní informace o trasování chybových transakcí.

3.1.4 Debugovací nástroje

Pojem ”debugování”není úplně trefný pro funkcionality, které dokumentace Brownie popisuje. Jedná se především o výpis informací o provedených transakcích. K tomuto účelu Brownie poskytuje objekt:

$$brownie.network.transaction.TransactionReceipt$$

Instanci provedené transakce lze uložit do tohoto objektu, který může následně volat například metodu $.info()$, která vypíše základní informace o transakci.

3.11: Ukázkový výpis informací o transakci

```
>>> tx = Token[0].transfer(accounts[1], 1e18, {'from':
    accounts[0]})
>>> tx.info()
  Transaction was Mined
  ---------------------
  Tx Hash: 0xd5479a987ccb684002ab3957e9cc73243a5225...
  From: 0x4C54640854FE357AdBdB4C6C37161D6bff9296C8
  To: 0xCEeDd135A7bBd899BE730a905C71E453D8d647a1
  Value: 0
  Function: Token.transfer
  Block: 3
```

```
  Gas Used: 51019 / 151019 (33.8%)

  Events In This Transaction
  --------------------------
  Transfer
      from: 0x4c54640854fe357adbdb4c6c37161d6bff9296c8
      to: 0xfae9bc8a468ee0d8c84ec00c8345377710e0f0bb
      value: 1000000000000000000
```

V případě neúspěšné transakce je k dispozici metoda $.revert_msg$, která vrací error string. Případně $.traceback()$, která vypíše zdroj chyby stejným způsobem, jak je zvykem v jazyce Python.

3.12: Ukázkový výpis trasovacích informací o transakci

```
>>> tx = Token[0].transfer(accounts[1], 1e18, {'from':
    accounts[0]})
>>> tx.traceback()
  Traceback for '0xd33fcc14dcf7e4a942793cb...bed4':
  Trace step 132, program counter 3103:
      File "contracts/Token.sol", line 56, in Token.transfer:
      transfer(msg.sender, [msg.sender, to], value);
  Trace step 3420, program counter 4326:
      File "contracts/Token.sol", lines 90-94, in Token.
          _transfer:
      addr = checkTransfer(
          authID,
          id,
          address
      );
  Trace step 3092, program counter 7032:
      File "contracts/Token.sol", line 36, in Token.
          checkTransfer:
      require(balances[addr[SENDER]] >= value, "Insufficient␣
          Balance");
```

3.1.5 Manager ethPM

Ethereum Package Manager je decentralizovaný správce balíčků, který se používá k distribuci smart kontraktů a EVM projektů. Balíček ethPM je ve své podstatě objekt JSON obsahující ABI, zdrojový kód, bajtový kód, údaje o nasazení a veškeré další informace, které dohromady tvoří kompletní myšlenku smart kontraktu. Specifikace ethPM definuje schéma pro ukládání všech těchto dat ve strukturovaném formátu JSON, což umožňuje rychlý a efektivní přenos nápadů na chytré kontrakty mezi nástroji a frameworky, které tuto specifikaci podporují.[16]

K získání balíčku ethPM je potřeba znát jeho název i adresu jeh registru. Tyto informace se sdělují prostřednictvím URI registru. Příkladná instalace známého matematického balíčku od firmy OpenZeppelin vypadá pomocí Brownie následovně:

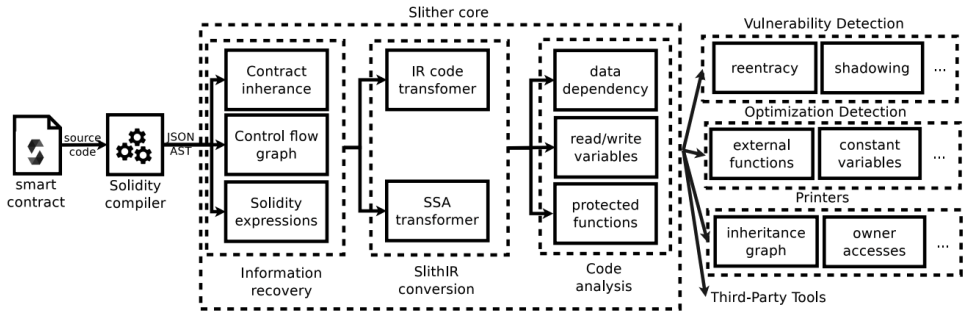
3.13: Instalace balíčku *math* pomocí Brownie balíčkového manageru

```
$ brownie pm install ethpm://zeppelin.snakecharmers.eth:1/math@1
    .0.0
```

3.2 Slither

Nástroj Slither je statický analyzátor a detektor chyb jazyku Solidity. Zdrojový kód nástroje je napsaný v jazyce Python. Mimo detekování chyb umí také vypisovat, případně vyobrazovat aditivní informace o smart kontraktech. Autorem nástroje je bezpečnostní americká firma Trail of Bits.[18] Jedná se o open-source software dostupný na GitHubu. [11] Mezi hlavní 4 vlastnosti Slitheru patří:

- automatická detekce zranitelností,
- automatická detekce optimalizačních příležitostí,
- výpomoc k detailnějšímu pochopení smart kontraktu,
- asistence pro statickou analýzu kódu.



Obrázek 3.1: Architektura nástroje Slither [18]

Slither nabízí několik konfiguračních argumentů, přes které lze nastavit nástroje, které Slither využívá ke kompilaci a sestavení projektu. Jedná se o nástroje Waffle, Hardhat, případně NPX a známý kompilátor Solc. Dále je možné pomocí Slitheru analyzovat zdrojový kód uložený na konkrétní aktivní Ethereum adrese a upravovat formát výstupních dat. Tyto funkcionality nejsou pro tuto práci klíčové, protože se nejedná o konkrétní analytické funkcionality nástroje, ale pouze o způsoby, jakými je zdrojový kód přeformátován do stavu připraveného k analýze.

[11] https://github.com/crytic/slither

3.2.1 Detektor zranitelností

Slither v tuto chvíli obsahuje více než 70 detektorů zranitelností. K hledání chyb je využívána statická analýza, která na rozdíl od té dynamické nespouští zdrojový kód. Detektor chyby je spuštěn nad speciální reprezentací kódu zvanou SlithIR. Každý uzel řídícího grafu kódu může obsahovat maximálně jeden výraz Solidity, který je převeden na sadu instrukcí SlithIR. Tato reprezentace usnadňuje implementaci analýz, aniž by došlo ke ztrátě kritických sémantických informací obsažených ve zdrojovém kódu Solidity.

Proces detekce je možné v tomto konkrétním případě rozdělit do 3 hlavních po sobě jdoucích fází.

1. Vstupními parametry je abstraktní syntaktický strom (AST - Abstract Syntax Tree), který je získáván kompilátorem ze zdrojového Solidity kódu. V první fázi Slither obnoví důležité informace, jako je graf dědičnosti (inheritance graph), graf řídícího toku kódu (CFG - Control Flow Graph) a list výrazů (expressions).

2. V dalším kroku je kód kontraktu transformován do vlastního reprezentativního jazyka SlithIR. Ten používá používá speciální datovou reprezentaci (Slither Intermediate Representation) pro snazší analýzu kódu.

3. Během třetí fáze provádí Slither analýzu jednotlivých předdefinovaných zranitelností.

Pro spuštění kompletní analýzy celého projektu je zadán jednoduchý příkaz, který sám projekt zkompiluje a na terminálový výstup vypíše nalezené zranitelnosti.

3.14: Příkaz pro spuštění Slitheru nad celým projektem

```
$ slither .
```

Pro analýzy konkrétního souboru je namísto znaku `.` zadána cesta k cílovému souboru. Továrně jsou aktivovány všechny detektory Slitheru. Seznam detektrotů je vypsán pomocí příkazu:

3.15: Příkaz pro výpis detektorů

```
$ slither --list-detectors
```

Poté je možné pomocí příkazu `--detect` a názvu konkrétního detektoru analyzovat kód pouze jedním konkrétním detektorem. Slither také nabízí možnost vynechat výpis sady zranitelností na základě jejich závažnosti (*high*, *medium*, *low*, *informational*, *optimization*, *dependencies*).

3.16: Příkaz pro vynechání zrantielností vážnosti *low*

```
$ slither . --exclude-low
```

3.2.2 Detektor optimalizačních příležitostí

Na začátku této kapitoly byly zmiňovány specifické limitace během vývoje smart kontraktů na rozdíl od jiného druhu softwaru. Jednou z nich byla omezená velikost kódu a poplatky 1.4 za vykonání výpočetních operací na virtuálním Ethereum stroji. Automatický detektor Slitheru napomáhá vývojářům v detekování proměnných, které by měly být deklarovány jako konstanty, a funkcí, jejichž viditelnost by měla být nastavena jako externí (v jazyce Solodity existují internal, public a exetrnal funkce 2.1.4, přičemž public je zároveň external a private). Tyto jednoduché změny docílí nižší konzumaci Gas a umožní kompilátoru lépe optimalizovat kód.

3.2.3 Asistovaná analýza a porozumění kódu

Mimo detektorů Slither obsahuje několik funkcionalit, které vývojáři, případně auditorovi pomohou s pochopením kódu, nebo s jeho analýzou. Ke snažšímu porozumění, jak kontrakty fungují a jakým způsobem spolu komunikují slouží funkcionalita PRINTERS, která umožňuje:

- pomocí grafů vyobrazovat specifické vlastnosti smart kontraktů (příkladem může být graf dědičnosti, nebo graf volání funkcí),
- vypsat souhrn informací o kontraktu v čitelné podobě (počet chyb, komplexita, modifiers 2.1.4),
- vypsat shrnutí autorizačního přístupu ke kontraktům (modifiers 2.1.4) a metody, které mohou být díky tomuto přístupu volány.

Pro výpis všech možných argumentů, které přijímá příkaz *printers*, je třeba zadat příkaz:

3.17: Příkaz pro výpis printerů

```
$ slither --list-printers
```

který nám v tuto chvíli [12] vypíše do terminálu následující tabulku:

Pro následné spuštění jednoho z nich:

3.18: Příkaz pro výpis zvoleného printeru

```
$ slither --print SELECTED_PRINTER
```

Pro asistovanou analýzu kódu Slither poskytuje vestavěné rozhraní Python API. Díky tomu má uživatel možnost vytváření vlastních skriptů, které mohou využívat jednotlivých funkcionalit Slitheru, případně je rozšiřovat.

[12] Detektory postupně přibývají. V době tvorby této práce Slither nabízí detektory uvedené v tabulce 3.2.3.

Num	Printer	What it Does
1	call-graph	Export the call-graph of the contracts to a dot file
2	cfg	Export the CFG of each functions
3	constructor-calls	Print the constructors executed
4	contract-summary	Print a summary of the contracts
5	data-dependency	Print the data dependencies of the variables
6	echidna	Export Echidna guiding information
7	evm	Print the evm instructions of nodes in functions
8	function-id	Print the keccack256 signature of the functions
9	function-summary	Print a summary of the functions
10	human-summary	Print a human-readable summary of the contracts
11	inheritance	Print the inheritance relations between contracts
12	inheritance-graph	Export the inheritance graph of each contract to a dot file
13	modifiers	Print the modifiers called by each function
14	require	Print the require and assert calls of each function
15	slithir	Print the slithIR representation of the functions
16	slithir-ssa	Print the slithIR representation of the functions
17	variable-order	Print the storage order of the state variables
18	vars-and-auth	Print the state variables written and the authorization of the functions

Tabulka 3.1: Tabulka print možností

3.3 Echidna

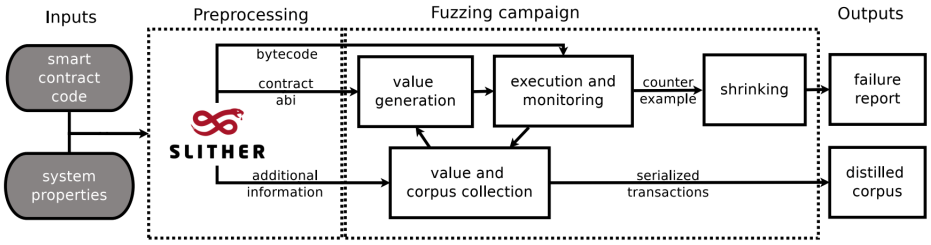
Nástroj Echidna je určený k designování a pouštění (property-based) fuzz testů. Stejně jako v případě Slitheru, je autorem Echidny firma Trail of Bits a cílí pouze na Ethereum smart kontrakty. Jedná se o open-source nástroj dostupný na githubu. [13] Echidna byla vyvinuta v jazyce Haskell a zároveň využívá Slither pro extrakci užitečných informací. [19] Hlavní inspirací k vývoji tohoto nástroj je software QuickCheck.[19] Funkcionality Echidna nástroje lze rozdělit do několika bodů:

- generování vstupních dat určených k testování funkcí,
- zobrazení testového pokrytí,
- extrakce užitečných dat o smart kontraktu pomocí nástroje Slither,
- report o využití Gas,
- textové uživatelské rozhraní.

3.3.1 Fuzzing testování

Fuzzing je známá bezpečnostní testovací technika, která generuje vstupní parametry na základě předem definovaných pravidel. Cílem této techniky je testování krajních podmínek pro nalezení chyb a zranitelností. Mezi nejznámější

[13] [https://github.com/crytic/echidn]



Obrázek 3.2: Architektura nástroje Echidna [19]

fuzzovací nástroje pro tradiční software patří například AFL nebo LibFuzzer. Parametry můžou být plně náhodné a randomizované, nebo přizpůsobené daným testovaným funkcím. Existuje několik strategií ke generování vhodných testovacích vstupních dat [20].

- Kolekce dat z předchozích testů. Vede-li sekvence dat k objevení cesty vedoucí k pádu programu, budou data zaznamenána a na základě nich vytvořena nová testovací sada.
- Použít platná vstupní data jako seed (semínko) ke generování nových vstupů.
- Generování vstupu respektující strukturální omezení programu.

Echidna využívá takzvané property-based fuzz testy. Jádrem funkcí systému Echidna je spustitelný soubor s názvem *echidna-test*. Tento soubor přijímá jako vstup kontrakt a seznam **invariantů** (podmínek, které by měly vracet logickou hodnotu *True*). Pro každý invariant generuje náhodné posloupnosti volání kontraktu a kontroluje, zda invariant platí. Pokud najde nějaký způsob, jak invariant zfalšovat, vypíše posloupnost volání, která tak učiní. Pokud to nedokáže, testy dopadly úspěšně. Ve smart kontraktech jsou **invarianty** podmínky funkcí Solidity, které mohou reprezentovat jakýkoli nesprávný nebo neplatný stav, do kterého může kontrakt dospět včetně:

- nesprávné kontroly přístupu (útočník se stane vlastníkem kontraktu),
- nesprávného stavu EVM (tokeny mohou být odeslány během pozastavení kontraktu),
- aritmetických chyb (uživatel má přístup k více tokenům než by měl mít).

Property jsou Solidity funkce, pro které platí:

- funkce nesmějí mít žádný vstupní argument,

- funkce vracejí booleovskou hodnotu *True*, pokud dopadnou úspěšně,
- jména funkcí musí začínat předponou *echidna_*.

Echidna poté:

- automaticky generuje transakce pro testování property funkcí,
- reportuje transakce, které vedly u property funkce k vrácení hodnoty *False*, nebo k vygenerování chybové hlášky,
- vrací hodnoty proměnných do původního stavu, pokud došlo k jejich změně uvnitř property funkce.

Jednoduchý případ testu ověřující, že adresa volající funkci má v poli *tokens* přiřazenou hodnotu větší nebo rovno 500:

3.19: Ukázka jednoduché testovací funkce

```
function echidna_tokens_under_500() public view returns(bool){
         return tokens[msg.sender] <= 500;
 }
```

Všechny testy musí obsahovat návratovou hodnotu typu *bool*, která udává zda test dopadl úspěšně, nebo se vyskytla chyba.

Testování je spuštěno následujícím jednoduchým příkazem:

3.20: Příkaz pro spuštění testu

```
  $ echidna-test contract.sol
```

3.4 Shrnutí nástrojů

Nejvíce funkcionalit nabízí nástroj Brownie. Jedná se především o software zaměřený na vývoj kompletního projektu. Brownie vývojářům pomůže s inicializací adresáře projektu a s jeho následnou kompilací. Pomocí své interaktivní konzole umožňuje interakci se smart kontrakty běžícími na lokálním, nebo veřejném blockchainu. Silnou funkcionalitou je integrace testovacího prostředí pomocí známé Python knihovny *pytest*, díky které je psaní testů stejně snadné jako je zvykem v jazyce Python, a zároveň pomocí naimportovaného balíčku *brownie* je možné jednoduše interagovat s Ethereum účty a provádět transakce. Transakce je poté možné jednoduše analyzovat a hledat zdroje chyb pomocí "debuggovacích"funkcionalit Brownie. Přestože Brownie nazývá některé ze svých funkcionalit jako debugovací, nejedná se o skutečný debugger.

Hlavní funkcionalitou nástroje Slither je automatická detekce zranitelností. Slither je orientován primárně na bezpečností automatizovanou analýzu Solidity kódu. Slither je tradičně využívaný při bezpečnostní analýze a téměř

v každém případě dokáže zaznamenat určité chyby. [14] Slovo "chyby"je zde záměrně použito, protože se většinou nejedná o skutečné zranitelnosti kódu, ale spíše o jeho syntaktické nedostatky a optimalizační příležitosti. Velká část detekovaných chyb je ve stavu False-Postive. Výpis informací o jednotlivých kontraktech, případě vykreslování grafů, může sloužit ke srozumitelnějšímu pochopení kódu, nebo celého projektu. Nedostatkem je formát vykreslování grafů, který často nabývá nesrozumitelné a zmatečné podoby. Automatický detektor zranitelností je značně omezen statickou analýzou kódu. Přestože je pomocí statické analýzy schopný najít určité druhy chyb, některé zranitelnosti není teoreticky, ani prakticky schopný rozpoznat, aniž by byla využita dynamická analýza. Několik detektorů je navíc zastaralých a tím pádem jejich výstup neaktuální.

Nástroj Echidna je cíleně orientován pouze na psaní testů. Testovací funkcionality nástroje Echidna mohou být silným nástrojem pro hledání zranitelností. Vytváření Fuzzy testů a celková práce s Echidnou ovšem není triviální, čemuž nenapomáhá ani jazyk Haskell, ve kterém je nástroj implementován. Testy jsou psané stejně jako smart kontrakty v jazyce Solidity. Část komunity tento přístup podporuje, ale obecně je známo, že programovací jazyk Solidity neumožňuje stejně kvalitní psaní testů jako jazyk Python. Echidna není schopná detekovat selhání testu v interní transakci [**?**].

Nástroj	Využití	Výhody	Nevýhody
Brownie	vývoj, psaní testů	intuitivní použití, pytest testy, analýza transakcí	chybí debugger, nekompiluje stejnojmenné kontrakty
Slither	bezpečnostní analýza	snadné použití, automatická detekce, extrakce informací	detekce pouze na základě statické analýzy, zastaralé detektory, nepraktický výstup nástroje
Echidna	psaní testů	efektivní testovací funkcionality	komplikované použití, testy v jazyce Solidity, nepřesné zachycení podmínek při interních transakcích

Tabulka 3.2: Shrnutí nástrojů Brownie, Slither a Echidna

[14] Tento fakt je formulován na základně osobní zkušenosti autora této práce.

Kapitola 4

Nástroj Woke

Woke je nově vznikající nástroj určený primárně k bezpečnostní analýze smart kontraktů psaných v jazyce Solidity. Protože se jedná o poměrně mladý obor, je zde nedostatek kvalitních nástrojů, které jsou zvykem ve světě standardního softwaru. Motivací pro vytvoření nástroje je pokrytí nedostatků stávajících nástrojů a zavedení nových funkcionalit. Jak již bylo zmiňováno, jde především o bezpečnostní nástroj, jehož funkcionality usnadní práci auditorům Solidity kódu na platformě EVM. S touto myšlenkou byly funkcionality nástroje od začátku navrhovány. Kód nástroje Woke je psaný v jazyce Python a je rozdělený do několika hlavních modulů.

4.1 Návrh funkcionalit

Návrh celkové architektury nástroje, včetně jeho funkcionalit, probíhal v době vytváření zadání této práce. Původním záměrem bylo vylepšení nástroje Slither 3.2, který je nejvyužívanějším nástrojem pro bezpečnostní analýzu smart kontraktů. Přestože Slither nabízí řadu užitečných funkcionalit, práce s nástrojem není optimální. Hlavním důvodem jsou statické výstupní informace, se kterými není možné dále interagovat. Nástroj Woke je od počátku zamýšlen jako nástroj, který bude poskytovat informace v podobě, ve které mohou být vstupem do dalších navazujících funkcionalit. Může se jedna o integraci s vývojovým prostředím, nebo podpora interaktivního shellu iPython.

Funkcionality nástroje Woke je možné rozdělit do tří základních skupin. První skupinou je sada funkcionalit a modulů, které ze zdrojového kódu připraví prostředí pro následnou analýzu a interakci se smart kontraktem. V první řadě je potřebný kompilační modul, jež je jádrem celého programu. Prvním krokem ke kompilaci souboru je parsování zdrojového Solidity kódu pro určení importních závislostí a verze (pragma 2.1.1) vhodného kompilátoru. Pokud systém, na kterém je nástroj spouštěn, neobsahuje danou verzi kompilátoru, dojde k jejímu stažení. Protože nové verze jazyka Solidity vznikají téměř každý měsíc, je toto automatické stažení a volba vhodného kompilátoru znatelným

ulehčením práce v porovnání s ostatními nástroji, kde si uživatel musí spravovat verze sám. Pro kompilaci je využíván kompilátor *solc*. Mimo rychlosti je po kompilačním modulu požadováno rozdělení souboru projektu na několik disjunktních podmnožin, které je možné kompilovat samostatně. Při návrhu této funkcionality se bere v potaz spojení s LSP serverem. Při komunikaci LSP serveru s vývojovým prostředím bude server přijímat informace o změnách v Solidity souboru. Dojde li ke změně jedné řádky kódu, není efektivním řešením kompilace kompletního projektu, ale pouze dané kompilační jednotky.

Druhou skupinou jsou moduly, které využívají data získaná při kompilaci projektu a statické analýze kódu. Tato skupina modulů je v průběhu práce stále ve stavu návrhu. Mezi těmito moduly se mohou objevit automatizované detektory zranitelností, nástroje určené k automatické anotaci zdrojového kódu a další analytické a bezpečnostní funkcionality.

Třetí skupina modulů zajišťuje interpretaci a interakci s výstupem předchozích dvou skupin. Jedná se o interakci s příkazovou řádkou, terminálové výstupy jednotlivých funkcionalit, interaktivní konzole pro lokální simulaci blockchainu, nebo LSP server, díky kterému bude možné přenést veškeré výstupy nástroje Woke do jakéhokoliv vývojového prostředí dle preference uživatele.

4.2 Language Server Protocol

Protokol jazykového serveru, ve zkratce LSP, je protokol sloužící ke komunikaci mezi editorem zdrojového kódu a jazykovým serverem. LSP byl vytvořen firmou Microsoft s cílem definovat standard pro protokol, pomocí kterého budou komunikovat jazykové servery a vývojová prostředí. V tuto chvíli už je LSP protokol podporován několika dalšími významnými firmami, včetně Red Hat a Sourcegraph. Protokol začíná podporovat čím dál vyšší počet editorů a jazykových komunit.

Jazykový server poskytuje funkcionality, které napomáhají ke komfortnějšímu a rychlejšímu vývoji softwaru v daném programovacím jazyce. Mezi tyto funkcionality může patřit barevné rozlišení syntaxe, našeptávání, nebo například vyobrazení hierarchie volání. Každé vývojové prostředí, nebo textový editor má své vlastní unikátní aplikační rozhraní, pomocí kterého je možné naprogramovat jednotlivé funkcionality. Z toho důvodu je nezbytné dané funkcionality implementovat pro každý nástroj zvlášť. Řešením je *jazykový server* určený k poskytování jazykově specifických funkcí a ke komunikaci s vývojářskými nástroji. Komunikace je obstarávána protokolem umožňujícím synchronizaci procesů na obou stranách.

Myšlenkou Language Server Protocol (LSP) je standardizovat protokol, přes který jazykové servery a vývojářské nástroje komunikují. Tento standard umožňuje použití jednoho jazykového serveru s různými nástroji. V opačném případě umožňuje nástrojům využívat různé jazykové servery pro odlišné programovací jazyky. Z tohoto důvodu je LSP výhodný pro obě strany.

4.2.1 JSON-RPC protokol

Jazykový server běží na systému jako separátní proces, který komunikuje s nástroji pomocí JSON-RPC. Tento protokol je navržen s důrazem na jednoduchost a kompatibilitu a je určen na vzdálené volání systémových procedur. Obecný mechanismus spočívá v tom, že dvě rovnocenné strany navážou datové spojení. Během doby trvání spojení mohou strany volat metody poskytované druhou stranou. Pro zavolání vzdálené metody se odešle požadavek (request). Pokud se jedná o oznámení (notification), není nutná odpověď druhé strany. V případě požadavku musí být navrácena odpověď (response).

JSON-RPC nespecifikuje konkrétní transportní vrstvu. Doporučuje se použití TCP/IP síťové vrstvy. Serializované objekty se přes TPC/IP protokol posílají pomocí proudů bajtů. Požadavky 4.2.1 a odpovědi 4.2.1 mohou být mezi stranami zasílány v jakýkoliv čas. Požadovaný je povinný odpovědět na každý požadavek vyjma notifikace 4.2.1. Nevalidní dotazy nebo odpovědi musí vést k vyvolání výjimky a ukončení spojení.

JSON

JavaScript Object Notation (JSON) je "lehký" datový formát, který je založen na standardu ECMA-404 4.2.1. Hlavní výhodou je mimo jeho jednoduchosti snadná čitelnost a jeho snadné parsování a generování pro systémy. JSON je textový jazykově nezávislý formát. Jeho konvenční struktura připomíná jazyky jako je C, JAVA, Python ad. JSON je tvořen dvěma základními strukturami.

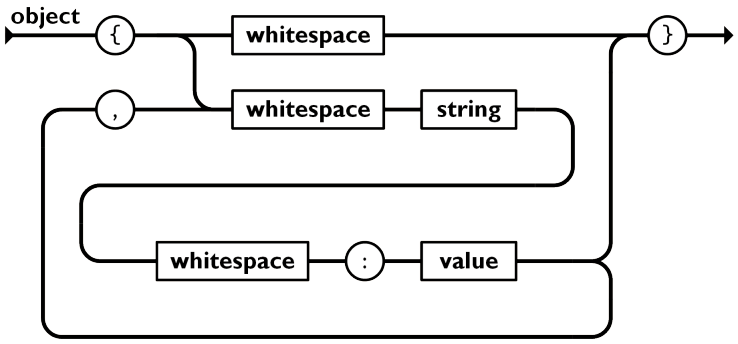
- ***Objekt*** 4.1 - Kolekce párů název-hodnota, které například Python nazývá slovníkem. Objekt začíná levou složenou závorkou "{"a končí pravou složenou závorkou "}". Každý název je následován mezerou a dvojtečkou ":", po které následuje hodnota. Jednotlivé páry název-hodnota jsou odděleny čárkou. Přípustné typy pro hodnoty jsou: textový řetězec (string), číslo, objekt, pole, booleovské hodnoty a typ null.

- ***Pole*** 4.2 - Seřazený list hodnot. V programovacích jazycích často známý jako pole nebo vektor. Jednotlivé prvky v poli jsou oddělené čárkou a uzavřené v hranatých závorkách [...].

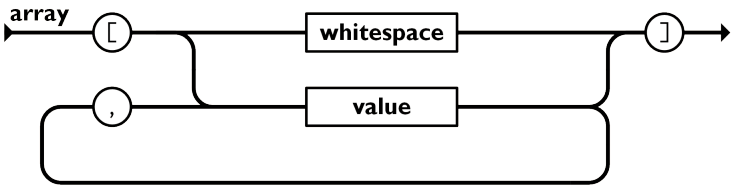
Právě kvůli využití standardních a univerzálních datových struktur, které jsou známé pro většinu programovacích jazyků, je JSON široce využívaným a snadno použitelným formátem.

RPC

Vzdálené volání procedur (Remote Procedure Call) je pojem z oboru distribuovaných výpočetních systémů. RPC označuje volání jednoho systému, jehož následek je spuštění procedury na jiném adresním prostoru. Může jít o jiný



Obrázek 4.1: Formát JSON objektu [21]



Obrázek 4.2: Formát JSON pole [21]

stroj na stejné síti, stroj mimo síť, nebo pouze jiný port (jako v případě jazykového serveru). Volání je prováděno formou interakce klient-server, kde klient je volající a server je spouštějící danou proceduru. Interakce nejčastěji probíhá formou dotazů a odpovědí, jako je tomu známo z HTTP protokolu. Po dobu exekuce procedury je klient serverem blokován a musí počkat na odpověď po vykonání dané procedury. Výjimkou je, pokud klient zasílá své dotazy asynchronním způsobem.

Požadavek protokolu (request)

Procedura na vzdáleném systému je vyvolána pomocí odeslání požadavku. Požadavek je objekt se strukturou JSON 4.2.1 a obsahuje tři základní parametry.

- ***Method*** - Metoda ve fomrátu string, kterou chce klient na serveru spustit.
- ***Params*** - Pole objektů obsahující argumenty nezbytné ke spuštění metody.
- ***Id*** - ID daného požadavku, ke kterému je později přiřazena odpověď.

Odpověď protokolu (request)

Po úspěšném provedení metody, tázané pomocí požadavku, musí být odeslána odpověď. Odpověď je objekt se strukturou JSON 4.2.1 a obsahuje tři základní parametry.

- ***Result*** - Objekt vrácený provedenou metodou. V případě chyby je objekt roven *null*.
- ***Error*** - V případě chyby a tím pádem hodnoty *result* rovnající se *null*, je zde Error objekt popisující chybu. Je-li metoda provedena úspěšně, tento objekt se rovná *null*.
- ***Id*** - ID odpovědi, které se musí rovna ID požadavku volající tuto metodu.

Oznámení protokolu (notification)

Oznámení nebo také notifikace je speciální druh požadavku, který nevyžaduje odpověď. Notifikace je objekt se strukturou JSON 4.2.1 a obsahuje stejné parametry jako požadavek 4.2.1 vyjma parametru ID, který musí být roven *null*.

4.2.2 JSON-RPC v LSP

Protokol LSP se skládá ze dvou částí. První část je hlavička (Header) a druhá je část s daty nazývaná "content part". Obě části jsou odděleny prázdným řádkem neboli ASCII znaky "`\r\n`"Tato struktura odpovídá HTTP dotazům.

Hlavička protokolu LSP

V hlavičkové části, která je kódována pomocí ASCII, se nachází několik hlavičkových polí a odpovídá tedy sémantice HTTP protokolu. Každé pole obsahuje dvojici název-hodnota oddělených pomocí dvojtečku a mezery ("`: `"). Každé pole hlavičky je ukončeno znaky "`\r\n`". Protože alespoň jedno pole je povinné, každé pole je ukončeno "`\r\n`"a celá hlavička je ukončena "`\r\n`", pak dvě sekvence těchto znaků ("`\r\n\r\n`") značí konec hlavičky. LSP v tuto chvíli podporuje dvě hlavičkové pole 4.2.2.

Název pole	Typ	Popis
Content-length	celé číslo	Povinné pole definující délku datové části v bajtech.
Content-type	string	Typ obsahu s výchozí hodnotou `application/vscode-jsonrpc; charset=utf-8`.

Datová část protokolu LSP

V originálním znění "Content Part"obsahuje tělo dotazu, případně odpovědi. Část s obsahem je ve formátu JSON-RPC 4.2.1 a je zakódována použitím znaků podle hodnoty v hlavičkovém poli *Content-type* 4.2.2. Výchozí hodnotou a zároveň jedinou podporovanou je *utf-8*. V případě dotazu s jinou hodnotou je očekávána odpověď s chybovou hláškou. V přechozích verzích LSP byla používána hodnota *8*, která neodpovídá standardům. Proto je doporučeno zpracovávat řetězec *8* jako *utf-8*. Příkladný inicializační dotaz klienta k serveru může mít následující podobu:

4.1: Ukázka formátu zprávy v LSP protokolu

```
Content-Length: 92\r\n
Content-type: application/vscode-jsonrpc; charset=utf-8
\r\n
{
        "jsonrpc": "2.0",
        "id": 1,
        "method": "initialize",
        "params": "uri:example:6432/path/file.sol"
}
```

4.2.3 Struktury zpráv LSP

LSP specifikace popisuje několik základních datových struktur, které jsou využívány napříč celým LSP protokolem. V první řadě jsou definovány číselné typy:

- ***integer*** - číslo v intervalu $\langle -2^{31}; 2^{31} - 1 \rangle$
- ***uinteger*** - číslo v intervalu $\langle 0; 2^{31} - 1 \rangle$
- ***decimal*** - desetinné číslo v intervalu $\langle 0; 1 \rangle$

Dalším datovým typem jsou zprávy 4.2.1. Protokolem je definována abstraktní datová třída **Message**, od které poté dědí všechny ostatní druhy zpráv určených ke komunikaci mezi klientem a serverem.

4.2: Objekt *Message*

```
abstract class Message{
    json_rpc: string
}
```

Jediným společným parametrem pro všechny zprávy je **json_rpc**. LSP protokol vždy pro tento parametr používá hodnotu "2.0". Jakákoliv jiná hodnota končí chybovým hlášením.

Třída **RequestMessage** dědí vlastnosti třídy **Message**. Mezi další parametry této třídy patří **id** sloužící k přiřazení odpovědi k požadavku, **method**

nesoucí název metody, jejíž spuštění je požadováno, a volitelný parametr **params**, pomocí kterého se předávají aditivní argumenty k dané metodě.

4.3: Objekt *RequestMessage* dědící od *Message*

```
class RequestMessage(Message){
    id: integer | string
    method: string
    ?params: array | object
}
```

Třída **ResponseMessage** opět dědí vlastnosti třídy **Message** a obsahuje přiřazovací parametr **id**. Novým volitelným parametrem je zde **result**, který v případě úspěšného vykonání požadavku musí vracet potřebná data, nebo *null*. V případě neúspěšného požadavku nesmí být tento parametr v odpovědi zahrnut, ale může být vrácen volitelný parametr **error**, který obsahuje číselný kód chybové hlášky, popis chybové hlášky a případně aditivní data určená k popisu vzniklého problému.

4.4: Objekt *ResponseMessage* dědící od *Message*

```
class ResponseMessage(Message){
    id: integer | string
    ?result: string | number | bool | object | null
    ?error: (code: int, message: string, ?data: any)
}
```

Třída **NotificationMessage** oproti třídě **RequestMessage** neobsahuje parametr *id*, protože v případě notifikace není vyžadována odpověď.

4.5: Objekt *NotificationMessage* dědící od *Message*

```
class NotificationMessage(Message){
    method: string
    ?params: array | object
}
```

4.2.4 Datové struktury LSP

V předchozí podkapitole jsou uvedeny struktury zpráv, pomocí kterých LSP protokol komunikuje. Lze si všimnout pole ***method***, které textovým polem definuje funkcionalitu, která by měla být příjemcem zprávy vyvolána. Tyto metody je možné rozdělit do několika základních skupin podle jejich vlastností, které vykonávají. (některé z metod jsou požadavky, jiné pouze notifikace4.2.3.)

- **Základní** - Jedná se o metody využívané k synchronizaci vývojového prostředí s jazykovým serverem a k ukončení spojení.
- **Okna** - Metody oken umožňují zobrazování informací v oknu klienta pomocí vyskakovacích zpráv a logů.

- **Telemetrie** - Jediná metoda (notifikace) slouží k oznamování události vyvolané uživatelem.
- **Registrační** - Metody sloužící k dynamické registraci a odregistraci jazykových metod.
- **Práce s adresářem** - Metody pro práci s adresářem projektu.
- **Práce se soubory** - Metody k ohlašování změn v jednotlivých souborech.
- **Diagnóza** - Notifikační metoda pro ohlašování diagnózy.
- **Jazykové** - Sada komplexních jazykových metod sloužících k usnadnění práce při vývoji nebo analýze kódu.
- **Speciální** - Metody, které jsou závislé na konkrétní implementaci protokolu a nemohou být použity ve všech LSP serverech.

Téměř všechny metody vyžadují pro ně specifická data, která jsou zpracována klientem nebo serverem. Tato data jsou předávány zprávami pomocí pole ***params***. Dle specifikace protokolu LSP pole ***params*** může obsahovat typ ***array*** nebo typ ***object***. Obecně protokol jazykového serveru podporuje zprávy JSON-RPC, nicméně podle oficiální Microsoft dokumentace základní protokol používá takovou konvenci, že případné parametry předávané ve zprávách by měly být typu ***object***. Ve vlastně definovaných zprávách je možné i použití typu ***array***. LSP server definuje přes 140 datových struktur, které jsou nezbytné ke korektní funkčnosti všech metod. Data jsou serverem uchovávána ve vlastních datových třídách a zprávami předávána ve formátu JSON 4.2.1.

Datové struktury lze rozdělit do tří skupin. První skupinou jsou struktury pro samotné zprávy, které byly definovány v předchozí podkapitole 4.2.3. Druhou skupinou jsou základní struktury, které slouží k popisu elementů jednotlivých souborů, případně celých adresářů. Jedná se o referenční datový model, kde jednotlivé struktury obsahují jiné, méně komplexní datové konstrukce. Díky tomu vznikají čím dál více komplexní třídy, pomocí kterých je možné definovat velmi konkrétní stavy jednotlivých dokumentů. Jednoduchým příkladem je struktura ***DiagnosticRelatedInformation*** sloužící k uchování části informace pro diagnózu.

```
class DiagnosticRelatedInformation{
        location: Location
        message: string
}
```

Třída ***DiagnosticRelatedInformation*** obsahuje zprávu datového typu ***string*** popisující určité diagnostické informace. Druhý parametr pojmenovaný ***location*** s datovým typem stejného jména odkazuje na konkrétní pozici dokumentu, ke které se daná diagnóza vztahuje.

4.6: Objekt *Location*

```
class Location{
        uri: DocumentUri
        range: Range
}
```

K určení konkrétního místa v adresáři projektu slouží datová struktura ***Location***, která obsahuje dva parametry. První parametr ***uri*** definující URI adresu souboru je datový typ ***DocumentUri***, který je definovaný pouze jako textový řetězec. Druhým parametrem je ***range*** se stejnojmenným datovým typem.

4.7: Objekt *Range*

```
class Range{
        start: Position
        end: Position
}
```

Aby bylo možné určit konkrétní pozici, případně rozsah, ve zdrojovém kódu jsou zapotřebí dvě informace. Jedná se o začátek a konec vymezeného rozsahu. Pro tento účel struktura ***Range*** obsahuje dva parametry: ***start*** a ***end*** typu ***Position***.

4.8: Objekt *Position*

```
class Position{
        line: int
        character: int
}
```

Struktura ***Position*** udává konkretní místo ve zdrojovém kódu, které je definované pomocí čísla řádky (***line***) a pořadového čísla znaku na řádce. Příklad referenčního datového modelu znázorňuje, jakým způsobem se uchovávají a předávají jednotlivé informace v LSP protokolu. Tyto informace jsou nezbytné pro vykonávání požadovaných funkcionalit. Třetím typem datových struktur jsou právě struktury přiřazené k jednotlivým funkcionalitám protokolu. U každé jednotlivé metody je jasně definovaná datová struktura, která bude obsahem parametru ***params***. Například pro publikování diagnostického reportu dokumentu je serverem přijat dotaz 4.2.1 s metodou ***textDocument/diagnostic***. Po jejím úspěšném vykonání je vyžadována odpověď 4.2.1 s parametrem datového typu ***FullDocumentDiagnosticReport***, který obsahuje mimo dalších parametrů také pole objektů ***Diagnostic***. Protože se jedná o pole, je možné odeslat do vývojářského nástroje několik diagnóz najednou. V následujícím okomentovaném pseudokódu je vyobrazený objekt ***Diagnostic***, ve kterém lze vidět i volitelné pole výše definovaného objektu ***DiagnosticRelatedInformation***.

4.9: Objekt *Diagnostic*

```
class Diagnostic{
        range: Range
        // The range at which the message applies
        ?severity: DiagnosticSeverity
        // The diagnostic's severity.
        ?code: int | string
        // The diagnostic's code.
        ?code_description: CodeDescription
        // An URI to open with more information
        // about the diagnostic error.
        ?source: string
        // A human-readable string describing of
        // the source of this diagnostic
        message: string
        // The diagnostic's message
        ?tags: DiagnosticTag[]
        // Additional metadata about the diagnostic.
        ?related_information: DiagnosticRelatedInformation[]
        // An array of related diagnostic information,
        ?data: Any
        // A data entry field that is preserved between
        // a `textDocument/publishDiagnostics` notification
        // and `textDocument/codeAction` request.
}
```

Příkladná ukázka datového modelu nebyla náhodná. Právě diagnostická funkcionalita je jednou z původních motivací pro implementaci LSP serveru. Jádro programu Woke obsahuje kompilační modul, který může být spouštěn LSP serverem. Při neúspěšné kompilaci budou pomocí LSP protokolu předána data vývojovému prostředí pro konkrétní vyobrazení zdroje kompilačního neúspěchu.

4.3 Implementace

Při tvorbě datového modelu hraje důležitou roli Python knihovna *pydantic*. Běhové prostředí jazyka Python nevynucuje anotace typů funkcí a proměnných. Tento fakt může značně znepříjemňovat vývoj komplexnějšího projektu, jako je nástroj Woke. Knihovna *pydantic* umožňuje anotaci typů jazyka Python a jejich ověřování. Knihovna vynucuje typové poznámky za běhu a poskytuje uživatelsky přívětivé chyby, pokud jsou data neplatná.

4.3.1 Protokol

JSON-RPC je protokol, jehož cílem je posílání a přijímání dat, dle kterých jsou spouštěny procedury na vzdáleném systému. Formát zpráv protokolu popisuje kapitola 4.2.1. Pro protokol byla vytvořena samostatná třída ***RPCProtocol*** s funkcemi pro čtení, zápis a formátování zpráv protokolu. Tato třída ve svých funkcích využívá knihovnu *pydantic* s naimplementovaným datovým modelem přímo pro LSP server. Díky tomu je možné z přečtených zpráv v podobě sek-

vence bajtů přímo vracet konkrétní datové třídy. Díky této architektuře LSP server vždy s jistotou ví, jaká data mu třída protokolu navrátí, a v případě chyby server zná konkrétní zdroj chyby. Tok kódu pro přijímání zprávy protokolu je následující:

1. Příjem zprávy v podobě sekvence bajtů.
2. Přečtení hlavičky zprávy.
3. Přečtení těla zprávy.
4. Rozřazení na dotaz , nebo 4.2.1.
5. Vytvoření odpovídajícího datového objektu.
6. Předání objektu jazykovému serveru.

V případě zachycení chyby v jakémkoliv z výše uvedených kroků je vytvořena datová struktura s 15 různými chybovými hláškami, které jsou implementovány na základě oficiální LSP specifikace [22].

Při odesílání zprávy LSP server zavolá jednu z metod:

- `send_rpc_response`
- `send_rpc_request`
- `send_rpc_error`
- `send_rpc_notification`

Metoda bude serverem vybrána na základě typu zprávy, která má být odeslána, nebo chyby, která během vykonávání programu nastala. Vstupem do metod je odpovídající *pydantic* datový objekt. Každá z těchto metod vytvoří z datového objektu slovník a volá privátní metodu `_send`, která je zodpovědná za formátování slovníku do JSON objektu, vytvoření odpovídající hlavičky 4.2.2 a zapsání výstupních bajtů. Tok kódu tedy lze shrnout v po sobě jdoucích bodech:

1. Zvolení metody podle typu zprávy.
2. Předání odpovídající datové struktury.
3. Převedení datové struktury do slovníku.
4. Formátování do objektu JSON.
5. Vytvoření hlavičky zprávy.
6. Zapsání JSON v podobě bajtů na výstup.

4.3.2 Server

Serverový submodul je nejdůležitější částí modulu LSP serveru. Tento submodul spravuje logiku přijetí zpráv, volání příslušné metody, odesílání zpráv a práci s chybami.

Sever obsahuje několik konfiguračních proměnných.

- `protocol` - Instance protokolu použitého ke komunikaci s klientem. V tomto případě se jedná o JSON-RPC protokol 4.2.1.
- `thread` - Počet vláken použitých pro příjem zpráv od klienta.
- `cleint_capabilities` - Jazykové metody podporované vývojovým prostředím (klientem).
- `server_capabilities` - Jazykové metody podporované serverem
- `running` - Booleovská proměnná definující stav serveru. Automaticky nastavena na hodnotu True. Proměnná nabude hodnoty False při přijetý metody evokující ukončení spojení mezi klientem a serverem.
- `init_request_recieved` - Booleovská proměnná definující, zda proběhla inicializační výměna informací mezi klientem nebo serverem. Továrně False.
- `LSPContext` - Datová třída sloužící k předávání informací mezi serverem a jádrem Woke.

Po aktivaci serveru se tok kódu dostane do nekonečné while smyčky s podmínkou running is True. Nekonečná smyčka je nezbytná pro souběžnou komunikaci s klientem vývojového prostředí. Pro přijetí příchozí zprávy je volána instance protokolu, která umožňuje čtení vstupních dat a vrací odpovídající datový objekt. Díky využití knihovny *pydantic* je možné ověřit, o jaký objekt se jedná a tomu přizpůsobit jeho následující zpracování. V této fázi mohou nastat 4 okolnosti:

- Na vstupu se nenachází žádná data k přečtení a následnému zpracování. V tomto případě server pokračuje ve čtení vstupů, dokud ze data neobjeví.
- Na vstupu jsou přijata data, která ovšem neodpovídají specifikaci LSP protokolu. V tom případě je vrácena chybová hláška a server pokračuje ve čtení.
- Je přijat dotaz odpovídající specifikaci LSP 4.2.1. Protokol přečte data a vytvoří z nich odpovídající pydantic datovou strukturu *RequestMessage*.

- Je přijata notifikace odpovídající specifikaci LSP 4.2.1. Protokol přečte data a vytvoří z nich odpovídající pydantic datovou strukturu *NotificationMessage*.

Protože notifikace, na rozdíl od klasického dotazu, nevyžaduje odpověď, je vhodné logiku jejich zpracování oddělit. Ke zpracování dotazů (request) slouží funkce `_handle_request`. Pro započetí úspěšné komunikace a výměny dat mezi klientem a serverem je potřeba provést úvodní inicializaci (podobně jako handshake u TCP-IP protokolu). Požadavek pro inicializaci musí být odeslán jako první dotaz 4.2.1 klienta na server. V případě, že server obdrží od klienta jiný typ dotazu nebo notifikaci, měl by reagovat následujícím způsobem:

- Při přijetí jiného dotazu by měla být odeslána odpověď s chybovým kódem **-32002** a libovolnou chybovou hláškou.
- Při přijetí notifikace by měla být zpráva jednoduše zahozena. Výjimkou je notifikace ukončující spojení se serverem (*exit*).

Dokud server neodpoví na inicializační požadavek objektem *InitializeResult*, nesmí klient posílat serveru žádné další požadavky nebo oznámení. Kromě toho nesmí ani server posílat klientovi žádné požadavky a notifikace. Výjimkou jsou notifikace určené k vypisování zpráv a logů, které slouží ve vývojovém nástroji pro ohlašování neočekávaných událostí. Pole dotazu pro inicializaci obsahuje řetězec ***initialize*** a jeho parametr je datová struktura *InitializeParams*.

4.10: Objekt *InitializeParams* dědící od *WorkDoneProgressParams*

```
class InitializeParams(WorkDoneProgressParams){
    processId: int | Null
    rootUri: DocumentUri | Null
    capabilities: ClientCapabilities
    ?clientInfo: (name: string, ?version: string
    ?locale: string
    ?rootPath: string | Null
    ?trace: TraceValue
    ?workspaceFolders: WorkspaceFolder[] | Null
}
```

V této struktuře jsou nejdůležitější první 3 nevolitelné parametry. Parametr *processId* udává ID nadřazeného procesu, který spustil server. Pokud server nebyl spuštěn jiným procesem, je ID rovno *Null*. Pokud ID není nulové, ale rodičovský proces není aktivní, měl by se také ukončit proces serveru. Parametr *rootUri* obsahuje URI adresu dokumentu v otevřeném adresáři. Pokud není otevřený žádný soubor, parametr se rovná *Null*. Parametr *capabilities* uchovává seznam jazykových metod, které jsou klientem podporovány.

Odpovědí serveru je objekt typu *InitializeResult*, který uchovává seznam jazykových metod podporovaných serverem.

4.11: Objekt *InitializeResult*

```
class InitializeResult{
        capabilities: ServerCapabilities;
    ?serverInfo: (name: string, ?version: string
}
```

Po úspěšné výměně inicializačních zpráv klient pošle serveru jednoduchou notifikaci initialized, která pouze oznamuje serveru úspěch operace. Od této chvíle server ví, jaké jazykové metody jsou klientem podporovány a je připraven na příjem dotazů.

Po přečtení přijatého dotazu, případně notifikace, protokol vrátí serveru strukturu požadavku, nebo notifikace 4.2.1 . Podle textového parametru *params* uvnitř struktury server vybírá metodu, která provede odpovídající operaci požadovanou klientem.

4.3.3 Výběr metod

Datová třída **RequestMethodEnum** obsahuje veškeré názvy metod podle specifikace LSP, které jsou přiřazeny stejnojmenným proměnným.

4.12: Datová třída *RequestMethodEnum* pro uchování názvů metod

```
class RequestMethodEnum(StrEnum):
    # General
    INITIALIZE = 'initialize'
    INITIALIZED = 'initialized' # Notification
    SHUTDOWN = 'shutdown'
    EXIT = 'exit' # Notification
    ...
```

Tato třída je nezbytná pro možnost dynamického výběru metody. Na základě těchto proměnných je vytvořena mapovací struktura, která pomocí slovníku přiřazuje každé proměnné danou volatelnou funkci. Příkladem je část struktury mapující názvy metod na konkrétní funkce.

4.13: Mapovací struktura pro dynamické volání funkcí

```
notification_mapping: Dict[str, Callable[[LSPContext, Any], None
    ]] = {
    RequestMethodEnum.INITIALIZED: lsp_initialized,
    RequestMethodEnum.EXIT: lsp_exit,
    RequestMethodEnum.WINDOW_SHOW_MESSAGE:
        lsp_window_show_message,
    RequestMethodEnum.WINDOW_LOG_MESSAGE: lsp_window_log_message
        ,
        ...
    RequestMethodEnum.DID_CLOSE: lsp_did_close,
    RequestMethodEnum.PUBLISH_DIAGNOSTICS:
        lsp_publish_diagnostic,
    RequestMethodEnum.LOG_TRACE_NOTIFICATION:
        lsp_log_trace_notification,
```

```
    RequestMethodEnum.SET_TRACE_NOTIFICATION:
        lsp_set_trace_notification,
}
```

Aby bylo možné hodnoty ve slovníku volat jako metody, musí být typu *Callable*. Jednotlivé funkce jsou připravené v separátním souboru společně s mapovací strukturou, která je importována serverem a následně s parametrem *params* volána. Ve stejném formátu je vytvořen separátní soubor pro dotazy.

Jednotlivé metody jsou předdefinovány, ale nejsou spojeny s jádrem nástroje Woke. V době tvorby této práce nebyl hotový kompilační a parsovací modul nástroje Woke, které by umožňovaly kompletní implementaci jazykových metod. Pro server je připravená paralelizace pro síťovou komunikaci s využitím knihovny *socketserve*. Paralelizae bude využita v následném vývoji, až bude potřebné řešit větší množství dotazů.

4.4 Testování

Testování softwarového produktu by mělo vždy zahrnovat manuální a automatizovanou část. Manuální část je vhodná zejména pro testování uživatelského rozhraní a celkové interakce uživatele s programem. Automatizované testy jsou speciálně napsané metody, které volají funkce vyvíjeného softwaru a ověřují, zda se jejich výstupy rovnají očekávaným hodnotám. Pro testování byl zvolen opět jazyk Python a známý testovacího frameworku *pytest*. Tradičním postup při testování, je nejprve návrh testů pro nejmenší programové jednotky. Takzvané Unit testy slouží k testování jednotlivých funkcí daných submodulů. Po vyhotovení Unit testů se přechází na integrační testování, které slouží k ověření komunikace mezi submoduly. Finální krokem jsou systémové testy, ověřující korektnost a funkčnost kompletní implementace.

V případě síťových aplikací není implementace automatizovaných testů triviální úlohou. V kontextu modulu LSP nástroje Woke je zde několik problémových částí. V první řadě je problematické určení vstupu a výstupu většiny funkcí, bez zahrnutí výstupů ostatních submodulů. Manuální příprava dat pro jednotlivé testované funkce se nejeví jako efektivní řešení. Druhou problematickou částí jsou finální výstupy, které nevracejí konkrétní hodnoty, ale rovnou je odesílají pomocí protokolu směrem k síťovému klientovi.

Z výše uvedených důvod probíhalo testování méně tradičním způsobem. Prvním krokem bylo vytvoření simulace reálného prostředí. Byl implementován testovací klient v jazyce python s použitím síťové knihovny *socket*. Testovací klient obsahuje funkce pro vytvoření spojení, odesílání formátovaných dotazů a jejich přijímání.

4.14: Ukázka testovacího klienta využívající knihovnu *socket* pro odesílání testovacích dotazů serveru

```
class Client:
    def __init__(self, sock=None):
```

```
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def _create_packet(self, data):
        return f"Content-Length: {len(data)}\r\nContent-Type: 
            application/vscode-jsonrpc; charset=utf8\r\n\r\n{
            data}"

    def connect(self):
        self.sock.connect((HOST, PORT))

    def send_request(self, msg):
        sent = self.sock.send(str.encode(msg))
        chunk = self.sock.recv(2048)
        if sent == 0:
            raise RuntimeError("socket connection broken")
        return chunk.decode()

    def send_notification(self, msg):
        sent = self.sock.send(str.encode(msg))
        if sent == 0:
            raise RuntimeError("socket connection broken")
```

Dále byla připravena sada dotazů odpovídají požadavkům LSP, a zároveň sada s nevalidní strukturou parametrů. Byly také implementovány funkce pro zanesení chyb do formátu zpráv. Byly nasimulovány chyby v hlavičkách souborů, nekompletní názvy parametrů hlaviček, prázdné parametry, kompletně prázdné hlavičky dotazů, nebo hlavičky obsahující více než povolený počet parametrů. V druhém bodě byly testovány dotazy s kompletními hlavičkami, ale s nevalidními těly zpráv. Byly testovány případy, kdy mají těla dotazů nevalidní strukturu, gramatické chyby v parametrech, případně těla obsahují neodpovídající parametry. V poslední řadě byly testovány reakce LSP protokolu na konkrétní dotazy. Způsob, jakým server komunikuje s klientem bez inicializační výměny údajů, či jakým způsobem reaguje na vícenásobnou inicializaci. Poslední testovací strategií bylo ověřit vytváření a předávání korektních datových typů na základě přijatého dotazu a zda jsou volány odpovídající funkcionality definované v požadavku.

Výsledná permutace dotazů a notifikací byla použita pro systémové testování. Testování probíhalo poloautomatizovaným způsobem, kdy byly serveru odesílány různé dotazy, případně sady zpráv. Tímto postupem se podařilo nalézt několik funkcionálních nedostatků implementace

4.5 Pokračování vývoje Woke

Nástroj Woke je stále v počáteční fázi vývoje. Následujícími kroky bude spojení LSP serveru s jádrem programu a implementace rozšíření pro vybrané vývojové prostředí. Souběžně vyvíjené moduly pro kompilaci Solidity kódu a debugger budou spojeny s LSP serverem a integrovány do vývojových prostředí, přes které bude možné využívat jejich funkcionalit.

Při spojení kompilační a parsovací jednotky s modulem LSP bude Woke schopný okamžitě reagovat na změny v souborech, které budou rychle kompilovány díky rozdělení souboru na nezávislé kompilační jednotky. V tu chvíli bude mít Woke všechny potřebné informace o zdrojovém kódu. Následující plánované funkcionality se věnují extrakci těchto informací k bezpečnostní analýze. Plánem je automatický detektor chyb založený na statické i dynamické analýze. Další funkcionality jsou zatím ve fázi návrhu.

Závěr

Prvním cílem této práce bylo nastudovaní tří známých nástrojů určených k vývoji a bezpečnostní analýze smart kontraktů na platformě Ethereum. Druhý cíl práce se týkal nástroje Woke, pro který měly být navrženy vhodné funkcionality na základě vědomostí získaných z předchozí části práce. Posledním cílem bylo implementovat a otestovat množinu navržených funkcionalit.

Práce splnila každý z vytyčených cílů a navíc důkladně popsala platformu Ethereum. Část věnující se Ethereum nebyla součástí zadání, nicméně pro splnění cílů této práce bylo nezbytné definovat pojmy týkající se platformy, pro kterou byly studované nástroje konstruovány a pro kterou vzniká nástroj Woke. V této části 1 byly popsány základní konstrukty sdíleného virtuálního stroje EVM, druhy Ethereum účtů a struktura transakcí. Následovalo seznámení čtenáře se smart kontrakty 2 neboli s programy uloženými na Ethereum blockchainu. Byl popsán jazyk Solidity, který je nejrozšířenějším jazykem v daném ekosystému.

Po uvedení čtenáře do problematiky, práce pokračovala zkoumáním jednotlivých funkcionalit nástrojů Brownie, Slither a Echidna. Každý nástroj byl nejdříve obecně popsán. Dále se práce věnovala jednotlivým funkcionalitám nástrojů a způsobu jejich použití a využití. Na konci kapitoly 3 byly nástroje zesumarizovány. Zde bylo definovány jejich zaměření, silné stránky a také nedostatky.

Poslední část práce se věnovala samotnému nástroji Woke. Nejdříve byla popsána motivace pro vývoj nového bezpečnostního nástroje. Na základě informací a zkušeností nabytých z předchozích kapitol, proběhl návrh funkcionalit pro nástroj Woke. Primárním cílem nástroje byla stanovena integrace bezpečnostních nástrojů přímo do vývojových prostředí. Tento cíl vzešel z faktu, že studované nástroje poskytují především statické terminálové výstupy, které nenabízí takový komfort při bezpečnostní analýze. Nástroj Woke by měl promítat tyto výstupy přímo do zdrojového kódu a tím docílit zřetelnější prezentace informací. Jednou z funkcionalit byl proto stanoven Language Server Protocol (LSP), který definuje standard pro výměnu informací mezi jazy-

kovým serverem a vývojovým prostředí. Podle protokolového standardu byl implementován jazykový server a komunikační kanál, který umožní jednoduché spojení mezi ostatními moduly nástroje Woke a libovolným vývojovým prostředím, které podporuje standard LSP. V posledním kroku byla implementace protokolu a serveru testována. Protože ostatní moduly vznikaly souběžně s modulem LSP, nebylo možné testovat kompletní kompatibilitu s jádrem nástroje.

Literatura

[1] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 2008: str. 21260. Dostupné z: https://bitcoin.org/bitcoin.pdf

[2] Scott, B.; Loonam, J.; Kumar, V.: Exploring the rise of Blockchain Technology: Towards Distributed Collaborative Organisations. *Strategic Change*, ročník 26, 2017, doi:10.1002/jsc.2142.

[3] News, R.: Ronin hack. online, 2022. Dostupné z: https://rekt.news/ronin-rekt/

[4] Certik: The State of DeFi Security 2021. online, 2021. Dostupné z: https://www.businesswire.com/news/home/20220113005054/en/

[5] Buterin, V.; aj.: A next-generation smart contract and decentralized application platform. *white paper*, ročník 3, č. 37, 2014: s. 2–1.

[6] Buterin, V.; aj.: Ethereum virtual machine. online, 2022. Dostupné z: https://ethereum.org/en/developers/docs/evm/

[7] Chris ward and others: Merkle patricia Tree. online, 2022. Dostupné z: https://eth.wiki/fundamentals/patricia-tree

[8] Wood, G.; aj.: Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, ročník 151, č. 2014, 2014: s. 1–32.

[9] T, T.: Ethereum EVM illustrated - exploring some mental models and implementations. online, 2019. Dostupné z: https://takenobu-hs.github.io/downloads/ethereum$_evm_illustrated.pdf$

[10] Vitalik Buterin, M. S.: EIP-2930: Optional access lists. Online, 2020. Dostupné z: https://eips.ethereum.org/EIPS/eip-2930

[11] Dannen, C.: *Introducing Ethereum and solidity*, ročník 1. Springer, 2017.

[12] Tardi, C.: EIP-2930: Optional access lists. Online, 2022. Dostupné z: https://www.investopedia.com/terms/g/gwei-ethereum.asp

[13] Daian, P.; Goldfeder, S.; Kell, T.; aj.: Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[14] Böhm, L.: Add more clear explanation of exception. GitHub, 2022. Dostupné z: https://github.com/ethereum/yellowpaper/pull/856

[15] Waas, M.: Downsizing contracts to fight the contract size limit. Online, 2020. Dostupné z: https://ethereum.org/en/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit/

[16] Hauser, B.; aj.: Official Brownie documentation. online, 2020. Dostupné z: https://eth-brownie.readthedocs.io/en/stable/

[17] developer, E.: Using the Compiler. Online, 2021. Dostupné z: https://docs.soliditylang.org/en/latest/using-the-compiler.htmlinput-description

[18] Feist, J.; Grieco, G.; Groce, A.: Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019. Dostupné z: https://arxiv.org/abs/1908.09878

[19] Grieco, G.; Song, W.; Cygan, A.; aj.: Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, s. 557–560. Dostupné z: https://agroce.github.io/issta20.pdf

[20] team, C.; other: Introduction to property-based fuzzing. GitHub, 2022. Dostupné z: https://github.com/crytic/building-secure-contracts/blob/master/program-analysis/echidna/fuzzing-introduction.md

[21] json.org: Introducing JSON. online, 2014. Dostupné z: https://www.json.org/json-en.html

[22] Microsoft: Language Server Protocol Specification. online, 2020. Dostupné z: https://microsoft.github.io/language-server-protocol/specification

Příloha A

Seznam použitých zkratek

ETH Platící jednotka Ether

ABI Aplikační binární rozhraní

ASCII American Standard Code for Information Interchange

BTC Bitcoin

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

EVM Ethereum Virtual Machine

JSON JavaScript Object Notation

LSP Language Server Protocol

MEV Minex Extractable Value

RLP Recursive Length Prefix

RPC Remote Procedur Call

TCP Transmission Control Protocol

SPDX Software Package Data Exchange

VFS Virtual Filesystem

Příloha **B**

Obsah přiloženého média

- *README.md* ..... stručný popis obsahu a instalace
- *requirements.txt* ..... instalační požadavky
- *setup.py* ..... instalační skript
- *test_client.py* ..... testovací klient
- *test_data.py* ..... testovací data
- `LSP` ..... zdrojové kódy práce
  - *__init__.py*
  - *__main__.py*
  - *basic_structures.py*
  - *context.py*
  - *enums.py*
  - *methods_impl.py*
  - *methods.py*
  - *notification_impl.py*
  - *protocol_structures.py*
  - *RPC_protocol.py*
  - *server.py*
- `text` ..... text práce
  - *thesis.pdf* ..... text práce ve formátu PDF
  - `src` ..... zdrojoví kód práce