FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

# Zadání diplomové práce

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

## Pokyny pro vypracování

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

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

*Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 30. května 2022 v Praze.*

Diplomová práce

# ROZŠÍŘENÍ NÁSTROJE WOKE

**Bc. Jan Kuběna**

Fakulta informačních technologií
Katedra informační bezpečnosti
Vedoucí: Ing. Josef Gattermayer, Ph.D.
4. ledna 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Bc. Jan Kuběna. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.*

Odkaz na tuto práci: Kuběna Jan. *Rozšíření nástroje Woke.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

# Obsah

**Poděkování** viii

**Prohlášení** ix

**Abstrakt** x

**Seznam zkratek** xi

**Úvod** 1

**1 Ethereum** 3
1.1 Základní koncepty . . . 3
1.2 Ethereum Virtual Machine . . . 5
1.3 Smart kontrakty . . . 6
1.3.1 Nezměnitelnost kontraktů . . . 7
1.3.2 Proxy a implementační kontrakty a kolize selektorů . . . 8
1.3.3 Phishing a tx.origin . . . 8
1.4 Sítě určené pro vývoj . . . 9

**2 Solidity** 11
2.1 Přehled prvků jazyka . . . 11
2.2 Struktura souboru se zdrojovými kódy . . . 12
2.3 Typy . . . 12
2.4 Funkce . . . 13
2.5 Inline assembly . . . 14
2.6 Kontrakty . . . 14
2.6.1 Externí volání funkcí . . . 15
2.7 Výrazy a kontrolní struktury . . . 16
2.8 Kompilace a její artefakty . . . 16
2.8.1 Chyby způsobené kompilátorem . . . 18
2.9 Nasazování kontraktů a konstruktory . . . 18

**3 Hledání chyb a zranitelností** 21
3.1 Statická analýza . . . 21
3.1.1 Vnitřní reprezentace . . . 22
3.1.2 Abstraktní syntaktický strom . . . 22
3.1.3 Grafy toku řízení a dědičnosti . . . 22
3.2 Dynamická analýza . . . 23
3.2.1 Pokrytí kódu . . . 23
3.2.2 Testování na základě vlastností . . . 24
3.2.3 Fuzzing . . . 24

**4 Nástroje Slither a Woke** 27
4.1 Slither . . . 27
4.1.1 Statické detektory chyb a zranitelností . . . 27
4.1.2 Funkce pro výpis informací . . . 28
4.2 Woke . . . 29
4.2.1 Fuzzer . . . 29
4.2.2 Statické detektory chyb a zranitelností . . . 32
4.2.3 Provázání s IDE . . . 34

**5 Navrhovaná rozšíření a analýza požadavků** 35
5.1 Sběr dat o pokrytí kódu fuzz testy . . . 35
5.2 Vytvoření nových detektorů . . . 36
5.3 Funkční a obecné požadavky . . . 37
5.4 Případy užití . . . 37

**6 Návrh implementace** 39
6.1 Sběr pokrytí kódu fuzz testy . . . 39
6.1.1 Rozdělení kódu na větve . . . 40
6.1.2 Volání aplikovaných modifikátorů a konstruktorů rodičů . . . 41
6.1.3 Volání funkcí, modifikátorů a konstruktorů . . . 41
6.1.4 Mechanismus sběru pokrytí a dostupné API . . . 41
6.1.5 Nasazování kontraktu . . . 42
6.1.6 Zpracování informací o zkompilovaném kódu . . . 43
6.1.7 Unikátní pojmenování kontraktů . . . 43
6.1.8 Integrace do fuzzeru . . . 43
6.1.9 Integrace do CLI a soubory s výstupem . . . 44
6.2 Statické detektory . . . 44
6.2.1 Detekce nebezpečného použití tx.origin . . . 44
6.2.2 Detekce kolizí selektorů u proxy kontraktů . . . 44
6.2.3 Detekce funkcí nevracejících všechny návratové hodnoty . . . 45
6.2.4 Detekce chyby kompilátoru při kopírování prázdného pole . . . 45
6.2.5 Detekce nepoužitých kontraktů, knihoven a rozhraní . . . 45
6.2.6 Integrace detektorů . . . 46

**7 Implementace** 47
7.1 Sběr pokrytí kódu fuzz testy . . . 47
7.1.1 Třídy pro sběr pokrytí . . . 47
7.1.2 Sběr pokrytí a více procesů . . . 50
7.1.3 Využité artefakty a API . . . 50
7.1.4 Externí volání funkcí . . . 50
7.1.5 Nasazování kontraktu . . . 50
7.1.6 Modifikátory a konstruktory . . . 51
7.1.7 Integrace do CLI . . . 51
7.1.8 Výstup pro integraci do IDE . . . 51
7.2 Statické detektory . . . 53
7.2.1 Nebezpečné použití tx.origin . . . 53
7.2.2 Kolize selektorů u proxy kontraktů . . . 53
7.2.3 Funkce nevracející všechny návratové hodnoty . . . 54
7.2.4 Chyba kompilátoru při kopírování prázdného pole . . . 54
7.2.5 Nepoužité kontrakty, knihovny a rozhraní . . . 54

**8 Testování** **55**
8.1 Unit testy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
8.2 Systémové testování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**9 Závěr** **59**

**A Instalace nástroje Woke** **61**

**Obsah přiloženého média** **67**

# Seznam obrázků


1.1 Zjednodušený diagram EVM . . . 5
1.2 Přehled principu fungování smart kontraktů . . . 7
1.3 Ilustrace jednoduchého proxy a implementačního kontraktu . . . 8
1.4 Ilustrace rozdílu mezi `msg.sender` a `tx.origin` . . . 9

3.1 Ukázka CFG vygenerovaného nástrojem Woke pro funkci royaltyInfo z kontraktu ERC2981 projektu OpenZeppelin . . . 23
3.2 Ukázka grafu dědičnosti vygenerovaného nástrojem Woke pro kontrakt TransparentUpgradeableProxy projektu OpenZeppelin . . . 23

4.1 Ukázka detekce a poddetekce detektoru na re-entrancy nástrojem Woke v CLI . 33
4.2 Ukázka detekce nezpracování návratové hodnoty nástroje Woke v IDE VS Code . 34

7.1 Diagram tříd pro sběr pokrytí . . . 48
7.2 Ukázka prototypu integrace sběru pokrytí do IDE . . . 52
7.3 Diagram tříd pro detektory . . . 53

8.1 Ukázka detekce funkce která nevrací všechny návratové hodnoty . . . 58


# Seznam výpisů kódu


2.1 Ukázka možností importování souborů . . . 12
2.2 Ukázka definice funkce v Solidity . . . 13
2.3 Ukázka definice a použití modifikátorů funkcí v Solidity . . . 14
2.4 Ukázka externího volání funkce pomocí `call` přímo na adresu v Solidity . . . 15
2.5 Ukázka externího volání funkce na instanci jiného kontraktu . . . 15
2.6 Přiřazení více proměnných najednou . . . 16
2.7 Ukázka zkompilovaného kódu ve formě bytecode . . . 17
2.8 Ukázka stejného zkompilovaného kódu jako v výpisu 2.7, ale ve formě operačních kódů instrukcí . . . 17
2.9 Ukázka mapování instrukcí na zdrojový kód v Solidity pro stejný zkompilovaný kód jako ve výpisu 2.7 . . . 17
2.10 Ukázka způsobů nasazování nových kontraktů . . . 19
2.11 Ukázka způsobů volání konstruktorů . . . 19

3.1 Ukázka SlithIR nástroje Slither . . . 22

4.1 Ukázkový detektor funkcí obsahujících slovo „upgrade“ v názvu v nástroji Slither 28

4.2 Ukázka detekce slabého pseudonáhodného generátoru nástrojem Slither . . . . . 29
4.3 Ukázkový fuzz test v nástroji Woke . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.4 Detektor funkcí obsahujících slovo „upgrade“ v názvu v nástroji Woke . . . . . . 33

6.1 Ukázkový kód v Solidity s větvením . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.2 Ukázka výstupu RPC `debug_traceTransaction` . . . . . . . . . . . . . . . . . . 42
6.3 Bytecode ve formě instrukcí . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

7.1 Ukázka výstupu nástroje Woke s přepínačem `-v` pro výpis počtu volání funkcí do CLI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
7.2 Ukázka souboru s nasbíraným pokrytím určeného pro vizualizaci v IDE . . . . . 52

8.1 Pokrytí kódu unit testy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
8.2 Unit testy pro sběr pokrytí kódu . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
8.3 Unit testy pro detektory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
8.4 Spuštění testu sběru pokrytí aplikovaných modifikátorů . . . . . . . . . . . . . . . 57

*Rád bych touto cestou poděkoval Ing. Josefu Gattermayerovi, Ph.D. za jeho cenné rady, podporu a připomínky při vedení této práce. Dále bych chtěl poděkovat své rodině a přátelům, bez kterých bych studium nezvládl a zejména pak Ing. Michalu Převrátilovi za technické konzultace.*

## 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. ledna 2023 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Abstrakt

Cílem této práce je rozšířit open source nástroj Woke sloužící k analýze smart kontraktů v jazyce Solidity nasazených na Ethereu o sběr pokrytí kódu fuzz testy a nové statické detektory. Nejdříve je představen projekt Ethereum, poté programovací jazyk smart kontraktů Solidity a pak jsou popsány relevantní koncepty ze statické a dynamické analýzy programů. Následně jsou představeny nástroje Slither a Woke používané pro analýzu bezpečnosti smart kontraktů a je provedena analýza požadavků na rozšíření nástroje Woke. Dále je navržena implementace těchto rozšíření a tento návrh je realizován a otestován, přičemž důležité body této realizace a testování jsou popsány.



**Klíčová slova** Woke, fuzzing, pokrytí kódu, statická analýza, smart kontrakty, blockchain, Solidity, Ethereum



# Abstract

The aim of this thesis is to add fuzz test code coverage and new static detectors to an open source tool Woke that is used for analysis of smart contracts developed in Solidity and deployed on Ethereum. The first part of the thesis introduces project Ethereum, followed by the description of Solidity, a programming language for smart contracts. Then the description of relevant concepts from static and dynamic analysis of programs is provided. Nextly the existing tools for analysis of smart contracts Slither and Woke are described and requirements for new features that will be added to Woke are analyzed. Lastly, the implementation itself is proposed, realized and tested and then the important parts of this realizion and testing are covered.



**Keywords** Woke, fuzzing, code coverage, static analysis, smart contracts, blockchain, Solidity, Ethereum

# Seznam zkratek

| | |
|---|---|
| ABI | Application Binary Interface |
| API | Application Programming Interface |
| AST | Abstract Syntax Tree |
| CA | Contract Account |
| CFG | Control Flow Graph |
| CLI | Command Line Interface |
| EIP | Ethereum Improvement Proposal |
| EOA | Externally Owned Account |
| EVM | Ethereum Virtual Machine |
| FQN | Fully Qualified Name |
| HTTP | Hypertext Transfer Protocol |
| IDE | Integrated Development Environment |
| IPC | Inter-process Communication |
| IR | Intermediate Representation |
| JSON | JavaScript Object Notation |
| LSP | Language Server Protocol |
| PBT | Property-based testing |
| PC | Program Counter |
| PoS | Proof-of-Stake |
| PoW | Proof-of-Work |
| RPC | Remote Procedure Call |
| UTF | Unicode Transformation Format |

# Úvod

V roce 2008 byl představen Bitcoin a s ním i nová technologie blockchain jako veřejná a decentralizovaná databáze. Bitcoin však blockchain využíval jen omezeně na posílání jednoduchých transakcí převádějících finanční prostředky ve své síti. Novější projekt Ethereum představený v roce 2013 přidává turingovsky kompletní sadu instrukcí, které mohou být vykonávány v rámci transakcí probíhajících v síti. To umožňuje vytvářet širokou škálu aplikací využívajících Ethereum jako důvěryhodnou síť, přes kterou mohou aplikace komunikovat s uživateli nebo mezi sebou.

Tyto aplikace lze použít na obchodování, investování a půjčování finančních prostředků, které jsou převedeny do jejich virtuální podoby jako kryptoměny. Mají však využití i v řadě dalších oblastí, jako jsou volební systémy nebo třeba systémy pro sdílení klinických dat pacientů. Jsou realizovány pomocí tzv. smart kontraktů, což jsou programy operující na síti Ethereum. Ty umožňují automatizovaně zpracovávat přicházející transakce na základě jasně definovaných a známých podmínek, tedy podle toho, jak jsou naprogramovány.

Stejně jako v ostatních programech i ve smart kontraktech se objevují bezpečnostní chyby, které jsou zneužívány útočníky. V případě smart kontraktů toto zneužití může vést k odcizení finančních prostředků, se kterými operují, ale i ke ztrátě důvěry. Vývojáři smart kontraktů se tedy snaží chyby a zranitelnosti co nejdříve odhalit a napravit. I to je důvodem ke vzniku nových nástrojů specifických pro doménu blockchainu a smart kontraktů, které mají vývojářům se zabezpečením kontraktů pomoci. Jedním z takovým nástrojů je i open source nástroj Woke, který se zaměřuje na smart kontrakty naprogramované v jazyce Solidity. Ten využívá jak metod statické analýzy ve formě detektorů známých chyb a zranitelností, tak dynamické analýzy ve formě fuzz testování.

Cílem této práce je rozšířit nástroj Woke, a to jak v oblasti dynamické, tak statické analýzy. První rozšíření spočívá v přidání sběru pokrytí kódu fuzz testy. To pomůže jeho uživatelům určit, které části kódu jsou testovány a tedy i lépe zaměřit své fuzz testy na části kódu, které chtějí opravdu testovat. Druhým rozšířením je přidání nových statických detektorů, které uživatelům pomohou najít možné nedostatky a případné chyby a zranitelnosti ve smart kontraktech.

# Kapitola 1
# Ethereum

Projekt Ethereum vznikl v roce 2013 jako rozšíření tehdy již známého Bitcoinu. Jeho tvůrce, Vitalik Buterin, do Etherea přidal zásadní vylepšení, a to turingovsky kompletní jazyk umožňující vytvářet komplexní aplikace, tzv. smart kontrakty. Tyto smart kontrakty mezi sebou mohou navzájem komunikovat v blockchainové síti a přebírají tak veškeré výhody těchto sítí, mezi něž patří zejména globální uchování jejich stavu, možnost veřejného auditu a také kryptografická verifikace a s ní spojená důvěra[1].

Zejména v posledních letech nabývají smart kontrakty na popularitě a nalézá se pro ně stále širší uplatnění, s tím se ale také dostává na popředí zájmů jejich bezpečnost. Případné chyby a zneužití mohou totiž být pro daný smart kontrakt fatální, ať už díky ztrátě důvěry v něj, nebo z důvodu odcizení uložených finančních prostředků kontraktu. Jen v roce 2021 byly podle společnosti Chainalysis odcizeny kryptoměny v hodnotě 3,2 miliard dolarů a zároveň dlouhodobě narůstá podíl odcizených finančních prostředků díky chybám v kódu smart kontraktů[2].

V této kapitole budou popsány základní principy blockchainu a projektu Ethereum, jejichž znalost je předpokladem pochopení problémové domény a tedy i splnění cílů této práce.

## 1.1 Základní koncepty

Jedním z důležitých konceptů Etherea je blockchain, což je navazující řetěz bloků obsahujících transakce sdílený mezi uzly v síti. Typicky je od něj očekávána decentralizovanost, nezměnitelnost a veřejná auditovatelnost, ale nic z toho není nutně pravidlem. Nezměnitelnost znamená, že jakákoliv změna bude detekovatelná, což je zaručeno pomocí kryptograficky bezpečných hashů bloků. Ty zajišťují právě to, že žádný blok v blockchainu nebude bez povšimnutí změnitelný bez toho, aby byly změněny všechny následující bloky. Očekávání decentralizovanosti žádá, aby systém nebyl kontrolovatelný jedním člověkem, skupinou lidí nebo státem. Toho může být docíleno u veřejného blockchainu tím, že se do něj může kdokoliv zapojit a navrhovat a verifikovat další bloky na základě známého mechanismu konsenzu. Existují ale i privátní blockchainy, které mohou navrhování a verifikaci dalších bloků a i samotné zapojení do sítě omezovat[3]. Veřejná auditovatelnost je pak možná pokud je blockchain široce veřejně přístupný a lze se dostat k operacím na něm prováděným. Ethereum je případ veřejného blockchainu, který splňuje všechna tato očekávání, tedy je decentralizovaný, nezměnitelný a veřejně auditovatelný.

Důležitou součástí Etherea je jeho kryptoměna Ether, která je automaticky vyplácena za výběr nového bloku. Lze využít k platbě za transakce samotné, ale také s ní lze pracovat ve smart kontraktech, kde může být využit jako platidlo mezi uživatelem a smart kontraktem nebo mezi smart kontrakty samotnými. Ether lze dělit na menší jednotky, z nichž podstatné jsou nejmenší jednotka Wei s hodnotou $10^{-18}$ Etheru a jednotka Gwei s hodnotou $10^{-9}$.

Ethereum jako celek je pak stavový automat založený na transakcích, kde transakce reprezentuje validní přechod mezi stavy tohoto automatu[1]. Ethereum Virtual Machine (EVM) je označení pro virtuální zásobníkový stroj sloužící ke zpracovávání transakcí zajišťující právě onen přechod mezi zmíněnými validními stavy. Tento stroj je provozován uzly v síti vykonávající kód kontraktů a vybírající další transakce. Tyto uzly pak uchovávají globální stav, tedy mapování mezi adresami (160–bitové identifikátory) a stavem účtu, přičemž pro toto mapování se používá speciální datová struktura Merkle Patricia Trie[4].

U účtů je rozlišováno mezi externě vlastněnými účty (EOA – Externally Owned Account), tedy těmi uživatelskými, a účty kontraktů (CA – Contract Account). V účtu jsou uloženy následující informace:

- Množství Etheru, které je svázáno s účtem a s kterým může účet nakládat.
- Počet celkově odeslaných transakcí sloužící k ujištění, že každá transakce bude zpracována pouze jednou.
- Hash EVM kódu, který bude spuštěn v případě, že na účet přijde volání funkce. Pro EOA je tento hash kódu prázdný a pro CA je použit k získání daného EVM kódu, jeho spuštění a obsloužení volání funkce.
- Hash používaný k přístupu do paměti EVM se stavem účtu. EOA mají tento hash prázdný a CA má v paměti odkazované tímto hashem uloženy své stavové proměnné.

Založení EOA nestojí žádný Ether a tyto účty mohou iniciovat transakce a využívají veřejné a privátní klíče ke kontrole účtu a podpisu transakcí. Účty CA naopak stojí Ether při založení, protože obsahují kód, který je v síti uložen a právě samotné uložení dat stojí Ether. Smart kontrakty a tedy ani CA nemohou samy od sebe iniciovat transakce, to mohou provádět pouze uživatelé a jejich EOA. Kontrakty a tedy i EOA ale může v rámci transakce volat funkce jiných nasazených kontraktů[5].

Důležitou součástí blockchainu je mechanismus konsenzu, kterým je vybírán následující blok a který dělá blockchain decentralizovaný. V době psaní této práce již v Ethereu proběhl přechod z Proof-of-Work (PoW) na Proof-of-Stake (PoS). V mechanismu PoW těžaři hledají pro nový blok hodnotu nonce tak, aby hashe bloku s nalezenou hodnotou nonce s byl pod periodicky nastavovanou mezí, která tak určuje náročnost tohoto hledání. Na rozdíl od PoW jsou v PoS místo těžařů validátoři, což jsou uživatelé, kteří vložili určitý obnos Etheru, tzv. *stake*, do speciálního depozitního kontraktu a díky tomu participují s ostatními validátory na výběru validních bloků.[6]. Tato participace funguje tak, že každých 12 sekund je náhodně vybrán jeden validátor navrhující nový blok a minimálně 128 dalších validátorů sloužících jako komise volící o tom, zda je navrhovaný blok validní. Výhoda PoS je zejména v tom, že validátoři nepotřebují tak výkonný a energeticky náročný hardware, čímž se snižuje bariéra vstupu do tohoto procesu oproti PoW a tedy se i posiluje decentralizace celého mechanismu.

Oproti PoW je také PoS ekonomicky odolnější proti tzv. 51% útoku, kde v PoW stačilo získat kontrolu nad více než 50% hashrate sítě, tedy majoritní většinu výpočetního výkonu celé sítě. V takovém případě mohl útočník blokovat posílání transakcí, modifikovat již existující bloky v blockchainu a zvrátit své transakce, čímž mohl docílit dvojího utracení svého obnosu kryptoměny. V PoS tento útok lze provést také, ale útočník musí získat více než 50% celkového vloženého Etheru jako stake. To je pro útočníka ekonomicky nevýhodné, protože v případě útoku mohou ostatní validátoři provést fork, tedy začít ignorovat bloky validované útočníkem a vytvořit tak nový blockchain, který bude pro nové bloky nezávislý na blockchainu, na kterém je útočník. Pokud bude tento fork přijat komunitou, může útočník přijít o všechen svůj Ether vložený do speciálního depozitního smart kontraktu jako stake[7]. V případě PoW a 51% útoku zůstane útočníkovi hardware, s kterým by se o útok mohl pokusit znovu, ale v PoS o svou investici může přijít.

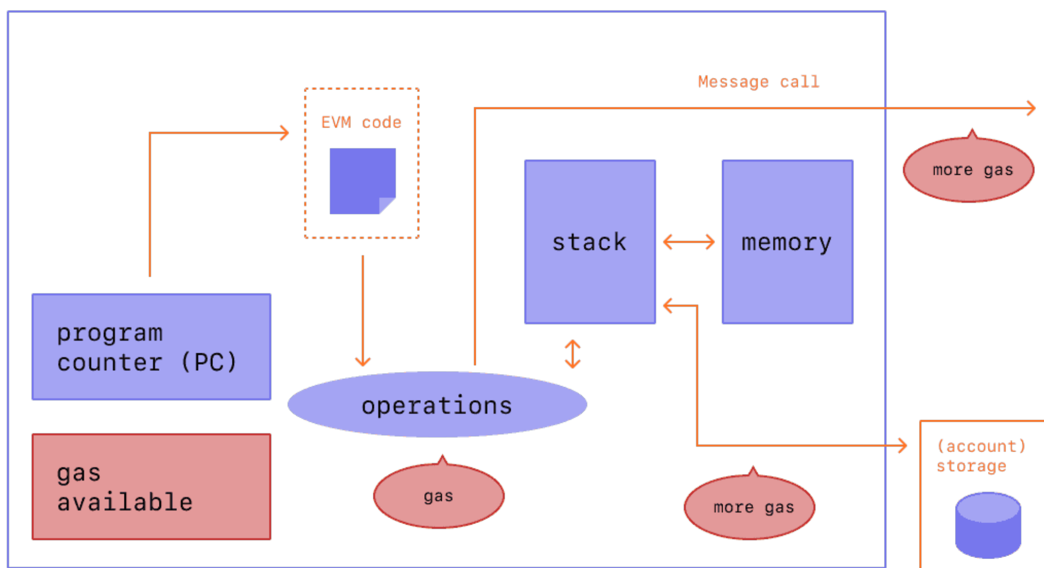
## 1.2 Ethereum Virtual Machine

Ethereum Virtual Machine je virtuální zásobníkový stroj, na kterém je spouštěn kód uložený pro adresu smart kontraktu. Tento kód se skládá z turingovsky kompletní sady instrukcí, které vykonávají validátoři sítě při zpracování transakce právě za pomoci konkrétní implementace EVM. Každá z instrukcí má stanovenou cenu v jednotce nazývané *gas* a validátor dostává odměnu za vykonané instrukce. Transakce pak mají stanovený limit gasu na jejich zpracování[8]. Cena gasu není pevně stanovena a je udávána v již zmíněné jednotce Gwei, která je k jejímu popisu ideálně velká.

V prostředí EVM a smart kontraktů existuje několik druhů pamětí, a to zásobník, *memory*, *storage* a *calldata*. Paměť *memory* se neuchovává mezi jednotlivými transakcemi a je dostupná pouze při vykonávání kódu samotného a její použití tak nevyžaduje uchovávání dat v blockchainu. Na rozdíl od ní je paměť *storage* součástí stavu a jedná se tedy o globální stav vázaný na konkrétní adresu a tedy i konkrétní smart kontrakt. Tato paměť je rozdělena po 32 bytech, tzv. slotech, kde má každý slot svůj index a smart kontrakt může využít libovolný index pro uložení svých dat[9]. Jedná se o mapu klíč-hodnota s $2^{256}$ možnými hodnotami klíčů. Paměť *calldata* slouží ke čtení parametrů volání funkce smart kontraktu.

Kód smart kontraktů je spojen s jejich adresami, kde je uložen hash pro přístup k tomuto kódu uloženého v blockchainové síti. Uložený kód je ve formě bytecode, tedy instrukcí ve formě bytů, a je většinou produktem kompilace z vyššího jazyka. Samotný smart kontrakt má nízkoúrovňové rozhraní ABI (Application Binary Interface), díky kterému je možné volat jeho funkce. Zjednodušená ilustrace fungování EVM lze vidět na obrázku 1.1.

**Obrázek 1.1** Zjednodušený diagram EVM, převzato z [4]




Instrukce EVM jsou zakódovány jedním bytem a pracují jen s daty v zásobníku. Speciální instrukcí je `PUSHX`, která slouží k přidání položky na vrchol zásobníku a za kterou následuje argument o velikosti $X$ bytů. Přístup k pamětem *memory* a *storage* je také prováděn pouze přes zásobník. Instrukce jsou volně rozděleny do skupin podle horní poloviny bytu jejich kódování následovně[1]:

**0x0. Aritmetické operace a `STOP`** – klasické aritmetické operace jako sčítání, násobení ale i instrukce pro počítání modulo a umocňování. Instrukce `STOP` zastavuje vykonávání kódu, což

se využívá v případě, že adresa nemá kód a je vrácena hodnota 0x00, což je právě instrukce `STOP`.

**0x1. Porovnání a logické operace** – jedná se o typické instrukce porovnání čísel, logických operací součtu, součinu a dalších a také bitové posuny.

**0x2. Instrukce pro Keccak-256** – instrukce pro hashovací funkci Keccak-256, která je v prostředí Etherea hlavní hashovací funkcí.

**0x3. Informace o prostředí** – tyto instrukce slouží k získávání informací o účtech, jejich stavech Etheru, informací o prováděné transakci, získávání vstupních dat prováděné transakce a dalších informací spojených s kódem a transakcí.

**0x4. Informace o blocích** – konkrétně se jedná o informace jako jsou hashe posledních bloků, získání identifikátoru blockchainu a detailní informace o aktuálním bloku jako jeho číslo, časová značka a podobně.

**0x5. Práce se zásobníkem, pamětí *storage* a *memory* a skokové instrukce** – tato sekce obsahuje instrukci `POP` pro odebrání položky z vrcholu zásobníku, instrukce pro práci s pamětmi, které ukládají a načítají z těchto pamětí přes zásobník a instrukce podmíněných i nepodmíněných skoků ovlivňující interní čítač instrukcí (PC – program counter).

**0x6. a 0x7. Operace `PUSH`** – těchto instrukcí je 32 ve formátu `PUSH`*X* kde *X* značí počet bytů, se kterým bude pracováno. Na vrchol zásobníku tedy lze vložit 1 až 32 bytů.

**0x8. Operace duplikace** – těchto instrukcí je 16 ve formátu `DUP`*X* kde *X* označuje 1. až 16. byte zásobníku, který bude duplikován vložením na vrchol zásobníku.

**0x9. Operace prohození** – těchto instrukcí je také 16 ve formátu `SWAP`*X* a umožňují prohazovat *X*-tý a *X+1*-tý byte na zásobníku.

**0xA. Protokoly** – tyto operace slouží k přidávání položek do protokolů v paměti *memory* a nemají vliv na stav EVM.

**0xF. Systémové operace** – jedná se o následující instrukce:

- `CREATE` a `CREATE2` sloužící k nasazování kontraktů.
- `CALL`, `CALLCODE` a `STATICCALL` pro volání funkcí jiných kontraktů. Liší se v detailech předávaných argumentů.
- `DELEGATECALL` pro volání funkce jiného kontraktu v kontextu volajícího kontraktu. Volaná funkce má přístup k paměti *storage* volajícího kontraktu a také má jeho hodnoty `msg.sender`, obsahující adresu volající původní funkci kontraktu (tedy před `DELEGATECALL`). Také má hodnotu `msg.value`, která obsahuje objem posílaného Etheru.
- `RETURN` zastaví provádění kódu a vrátí návratovou hodnotu a `REVERT`, která také zastaví provádění kódu, ale zahazuje modifikace stavu prováděné kódem do jejího zavolání.
- `SELFDESTRUCT` pro zastavení provádění kódu a registraci účtu ke smazání.
- `INVALID`, což je speciální instrukce označující nevalidní instrukci, která zahazuje modifikace stavu podobně jako `REVERT`.

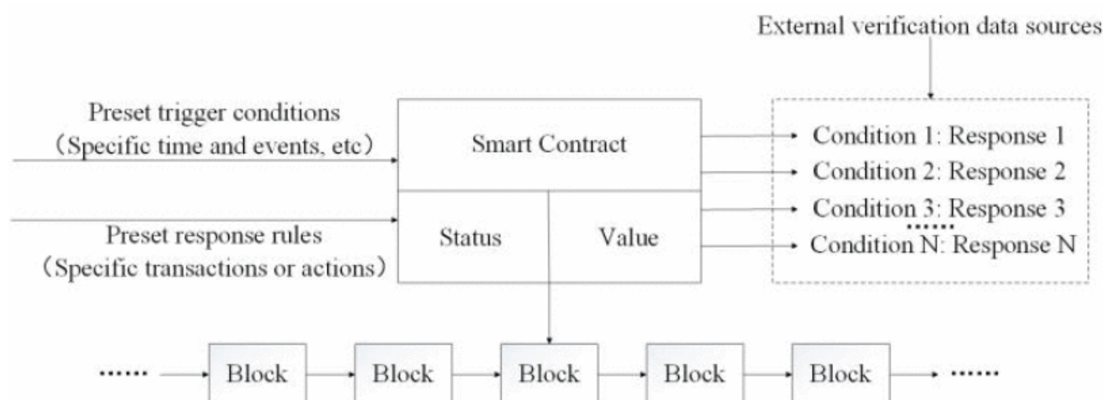
## 1.3 Smart kontrakty

Idea smart kontraktů byla formulována Nickem Szabem již v roce 1994 jako automatizovaný (ať už pomocí hardwaru nebo softwaru) transakční protokol, který vykonává kontrakt podle

jeho podmínek[10]. Smart kontrakty v Ethereu tuto definici dobře naplňují, protože automaticky hlídají dodržování podmínek stanovených kontraktem, tedy pokud je smart kontrakt dobře naprogramován. Díky decentralizované nátuře Etherea a možnosti veřejného auditu jsou pak uzavřené kontrakty nezávislé na globálních autoritách a jejich kód a transakce jsou veřejně auditovatelné. V případě nutnosti ověření z třetí strany je možné využít důvěrné externí zdroje verifikující informace mimo Ethereum[11].

Na ilustraci 1.2 lze vidět obecný princip fungování smart kontraktů. Ty čekají na externí vstup ve formě volání některé z jeho funkcí. Toto volání je následně smart kontraktem zpracováno podle jeho vnitřních podmínek. Kontrakty mohou využívat externí zdroje informací a případně volat funkce jiných kontraktů.

**Obrázek 1.2** Přehled principu fungování smart kontraktů, převzato z [11]




Smart kontrakty v Ethereu jsou typicky naprogramovány ve vyšších jazycích a poté kompilovány do kódu vykonávaným EVM. Nejpoužívanějšími a aktivně udržovanými jazyky jsou Solidity[1] a Vyper[2]. Solidity je vysokoúrovňový jazyk inspirovaný jazyky C++, Python a JavaScript a Vyper je jazyk snažící se přiblížit jazyku Python a záměrně nepodporuje některé pokročilé schopnosti Solidity za účelem bezpečnějších a jednodušších kontraktů[12]. Vzhledem k tomu, že Solidity je nejrozšířenějším jazykem pro vytváření smart kontraktů, budou konkrétní příklady uváděny v něm, nicméně obecné postupy a myšlenky lze často přenést i do jazyku Vyper.

### 1.3.1 Nezměnitelnost kontraktů

Jednou z důležitých vlastností smart kontraktů je jejich nezměnitelnost, která vyplývá z principu nezměnitelnosti validovaných bloků v blockchainu. Tato vlastnost má své výhody i nevýhody, kde hlavní výhodou nezměnitelnosti smart kontaktů je fakt, že se uživatelé mohou spolehnout na to, že autor smart kontraktu ani případný útočník nezmění kontrakt tak, aby to pro něj bylo nějakým způsobem výhodné. Nevýhodou však je, že pokud bude ve smart kontraktu nalezena chyba, není možné ji opravit. Jelikož se oba druhy chyb ve smart kontraktech objevují, může jejich zneužití způsobit značné finanční ztráty, jako tomu bylo ve známém útoku na projekt DAO, kde útočníci nalezli bezpečnostní slabinu v kódu smart kontraktu a podařilo se jim z kontraktu ukrást Ether v hodnotě asi 60 miliónů dolarů[13]. Po vrácení transakcí provedených útočníkem pomocí forku byly postupně vymyšleny mechanismy obcházející tuto vlastnost neměnitelnosti, tzv. *upgrade patterns*, které umožňují autorům nasazovat nové verze kontraktů[14].

[1] `https://docs.soliditylang.org/en/v0.8.17/`
[2] `https://vyper.readthedocs.io/en/stable/`

### 1.3.2 Proxy a implementační kontrakty a kolize selektorů

Upgrade patterns jsou dnes již poměrně standardizovaným způsobem, jak umožnit autorům smart kontraktů nasazovat nové verze jejich kontraktů. Populární metodou jsou tzv. proxy kontrakty dělící smart kontrakt na dva separátní kontrakty, a to proxy kontrakt a implementační kontrakt. Proxy kontrakt se stará o ukládání samotných dat kontraktu, proto se také někdy označuje jako tzv. storage kontrakt. Na implementační kontrakt jsou přesměrovány volání z proxy kontraktu, přičemž implementační kontrakt pracuje právě nad daty z proxy kontraktu a je také někdy označován jako tzv. logic kontrakt[15].

Samotné předání volání mezi proxy a implementačním kontraktem je prováděno pomocí speciální instrukce `DELEGATECALL`, která volá funkce jiného kontraktu stejně jako běžné volání s tím rozdílem, že kód volané funkce je prováděn v kontextu volajícího kontraktu. Tedy v případě proxy kontraktů je volán implementační kontrakt v kontextu proxy kontraktu a může tak přistupovat k datům v proxy kontraktu[16].

Z pohledu implementace proxy kontraktu se využívá speciální funkce Solidity `fallback`, která je zavolána, pokud žádná jiná funkce kontraktu neodpovídá signatuře volané funkce. O nasazení nového implementačního kontraktu se pak typicky stará již normální funkce, většinou pojmenovaná `upgrade`, která v proxy kontraktu přepíše adresu implementačního kontraktu, čímž efektivně změní implementaci. Tato adresa je většinou uložena v paměti *storage* na indexu, který výsledkem hashování funkcí Keccak-256 řetězců standardizovaných[3] v EIP-1967[17].

Použití proxy kontraktů pak pro programátora znamená buďto napsat veškerou logiku proxy kontraktů, čímž se vystavuje možnosti zanést do své implementace chyby, nebo využít některou z připravených implementací, například od projektu OpenZeppelin[4]. Implementace proxy a implementačních kontraktů pak může využívat jiné mechanismy lišící se zejména v přístupu k paměti *storage* a v konkrétním mechanismu nasazování nového implementačního kontraktu[18].

**Obrázek 1.3** Ilustrace jednoduchého proxy a implementačního kontraktu, převzato z [18]

Jedním z bezpečnostních problémů, který může u proxy a implementačních kontraktů nastat, jsou kolize tzv. selektorů. Volání funkce se v Solidity kontraktech řídí podle selektoru, což jsou první 4 byty z hashe pomocí Keccak-256 ze standardizovaného formátu signatury funkce[19]. Když neexistuje funkce odpovídající selektoru, je volání předáno funkci `fallback`. Může se ovšem stát, že se v proxy kontraktu objeví funkce se stejným selektorem jako v implementačním kontraktu. Místo aby se zavolala funkce `fallback`, která by toto volání předala dále implementačnímu kontraktu, bude vykonána funkce již v proxy kontraktu, což nemusí být chtěné. Tyto kolize mohou nastat například chybou programátora který přehlédne, že stejná funkce již existuje v proxy kontraktu.

### 1.3.3 Phishing a tx.origin

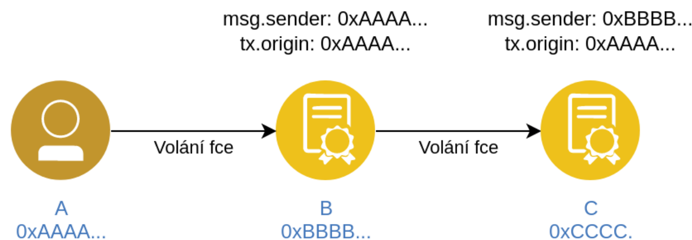
Jedním z důležitých mechanismů, které smart kontrakty využívají, je ověřování identity uživatele nebo kontraktu, který volá funkci. Ověření identity probíhá zpravidla pomocí adresy daného uživatele nebo kontraktu použitého při volání funkce kontraktu. Přístup k těmto informacím poskytují ve smart kontraktech speciální globální proměnné `msg.sender` obsahující adresu, ze

[3]Ethereum používá tzv. *EIP – Ethereum Improvement Proposals* k popsání svých standardů, podobně jako Python se svými *Python Enhancement Proposals*

[4]`https://openzeppelin.com`

které přišlo volání funkce, a `tx.origin` obsahující adresu, která iniciovala řetězec volání funkcí až k volané funkci v posledního smart kontraktu[20]. V praxi to znamená, že pokud uživatel **A** zavolá funkci kontraktu **B** a ten zavolá funkci kontraktu **C**, bude hodnota `msg.sender` obsahovat adresu kontraktu **B**, zatímco `tx.origin` bude obsahovat adresu uživatele `A`, což je znázorněno na obrázku1.4.

**Obrázek 1.4** Ilustrace rozdílu mezi `msg.sender` a `tx.origin`




Efekt neúmyslného použití `tx.origin` místo `msg.sender` může být zásadní, protože pokud cílový kontrakt ověřuje pouze `tx.origin`, je možné využít další metody, například phishing, k tomu, aby uživatel zavolal funkci z útočníkova kontraktu. Ten následně zavolá funkci z cílového kontraktu a tomuto kontraktu pak bude předána hodnota `tx.origin` s uživatelovou adresou. Přitom ale volání prováděl útočníkův kontrakt, který nyní může dělat stejné operace jako uživatel v cílovém kontraktu. I přes to, že nevhodnost používat `tx.origin` k autorizaci je obecně známá, je možné se setkat s kontrakty, které ji k tomuto účelu používají a tedy i s útoky na tyto kontrakty. V roce 2021 byl proveden útok na THORChain[5], což je síť umožňující převádět kryptoměny mezi různými blockchainovými sítěmi, při kterém byla napáchána škoda ve výši 80 000 dolarů[21]. Při převádění jsou využívány tzv. *tokeny*, které slouží k reprezentaci digitálních a fyzických aktiv[22]. Útok využil právě toho, že na THORChainu bylo používáno ověřování `tx.origin` místo `msg.sender`. Samotný útok pak probíhal následovně:

- Útočník poslal uživateli svůj token UniH.
- Uživatel se snažil token prodat nebo vyměnit a tuto transakci musel potvrdit.
- Potvrzením získal útočník uživatelovu adresu jako hodnotu `tx.origin` pro další volání.
- Útočník si nechal převést prostředky z adresy uživatele na svou adresu.

Od samotného použití `tx.origin` pro autorizaci je právě z těchto důvodů odrazováno[23] a dokonce od jeho použití obecně odrazuje i samotný tvůrce Etherea Vitalik Buterin. Ten v minulosti naznačoval, že `tx.origin` by v budoucnu nemusel zůstat použitelný[24].

## 1.4 Sítě určené pro vývoj

Při vývoji smart kontraktů jsou používány lokální vývojové blockchainy, někdy také označované jako tzv. *devchain*, na kterých lze rychle a jednoduše nasadit kontrakt a začít s testováním jeho funkcí. Tyto vývojové blockchainy již mají připravené adresy s dostatečným obnosem Etheru, nastavené vyšší limity na gas, rychlejší verifikace bloků a zpravidla lepší debuggovací rozhraní[25].

[5] `https://thorchain.org/`

Patří mezi ně například Ganache[6], Anvil[7] nebo Geth[8]. Rozdíly mezi jednotlivými implementacemi vývojových blockchainů jsou zejména v rychlosti vykonávání transakcí a v rozdílné podpoře některých nadstavbových API (Application Programming Interface) funkcí. Oba tyto parametry jsou při testování důležité. Rychlejší vykonávání transakcí umožní stihnout více testů a některé nadstavbové API funkce zase mohou být použity pro lepší přehled o stavu blockchainu samotného a kontraktů a transakcí na něm.

[6] https://trufflesuite.com/ganache/
[7] https://github.com/foundry-rs/foundry
[8] https://geth.ethereum.org/

# Kapitola 2
# Solidity

Solidity je objektově orientovaný jazyk pro implementaci Ethereum smart kontraktů inspirovaný jazyky C++, Python a Javascript. Je to staticky typovaný jazyk podporující vícenásobnou dědičnost, abstraktní kontrakty a metody, knihovny[26]. Jedná se o nejpoužívanější jazyk pro vytváření smart kontraktů[27], který je navíc stále v aktivním vývoji.

V době psaní této práce je nejvyšší verze Solidity 0.8.17 vydaná 8. září 2022 a tato verze bude také v rámci této kapitoly představena a popsána. Způsob verzování s 0 na začátku je použit jako upozornění na to, že Solidity je rychle se rozvíjejícím jazykem a často jsou přidávány nové funkce a změny které způsobují nekompatibilitu s dřívějšími verzemi. Kromě výjimečných případů jsou také bezpečnostní záplaty dostupné pouze pro aktuálně nejvyšší verzi oficiálního překladače *solc*.

## 2.1 Přehled prvků jazyka

Solidity je inspirováno několika jinými jazyky a v mnohém se jim tedy i podobá. Níže jsou uvedeny základní prvky jazyka, z nichž některé lze běžně potkat i v ostatních jazycích, ale jiné jsou uzpůsobené pro programování smart kontraktů v prostředí Etherea. Jedná se o tyto prvky:

**Kontrakty** – představují smart kontrakty později nasazenované na blockchain, pokud tedy nejsou označeny jako abstraktní. Mají své funkce, které mohou být veřejně volatelné uživateli a jinými kontrakty.

**Volné funkce** – jsou funkce definované mimo kontrakt a mohou být používány ostatními funkcemi, ale nikdy nebudou samostatně volatelné jako funkce kontraktu.

**Knihovny** – jsou podobné kontraktům, ale měly by být nasazeny pouze jednou na specifickou adresu a následně by měly být jejich funkce volány přes delegované volání, aby volající kontrakty vykonávaly kód z knihoven ve svém kontextu.

**Rozhraní** – jsou podobné abstraktním kontraktům, ale nemohou mít implementované žádné funkce.

**Události** – speciální zprávy sloužící na blockchainu jako veřejně dostupné notifikace ze smart kontraktů.

**Chybové typy** – umožňují definovat typ chyby lépe vysvětlující uživateli, co za chybu se stalo nad rámec chybového typu *Error*, který je definován přímo v Solidity. Mohou také obsahovat další data týkající se dané chyby.

**Konstantní proměnné** – není možné měnit jejich hodnotu a jsou označeny klíčovým slovem `constant`.

**Struktury** – jsou to uživatelem definované typy shlukující více proměnných různých typů kromě typu sama sebe.

**Výčty** – umožňují vytvořit uživatelem definovaný typ explicitně převoditelný na celé číslo. V Solidity mohou mít výčty až 256 hodnot.

## 2.2 Struktura souboru se zdrojovými kódy

Soubory se zdrojovými kódy by měly obsahovat licenci, přičemž doporučované jsou SPDX licence[1], které jsou strojově čitelné a pokud je SPDX licence použita, je tato informace přidána i do metadat nasazeného kontraktu.

Dále může soubor obsahovat tzv. pragmu, tedy příkazy určené pro kompilátor. Je jich hned několik, ale pro běžné potřeby postačuje `pragma solidity` *`verze`*, která kompilátoru říká, s jakou verzí Solidity kód půjde zkompilovat a `pragma abicoder` *`verze`*, která umožňuje vybrat mezi první verzí (hodnota `v1`) a druhou verzí (hodnota `v2`) kódování ABI, kde druhá verze podporuje více typů, ale může stát více gasu.

Do kódu lze také importovat jiné soubory pomocí příkazu `import`, jenž má více variant použití podle toho, co přesně by mělo být importováno.

**Výpis 2.1** Ukázka možností importování souborů

```
1 import "file.sol"
2 import * as symbol from "file.sol"
3 import {symbol1 as alias1, symbol2} from "file.sol"
```

V ukázce 2.1 budou v prvním případě importovány všechny globální symboly ze souboru `file.sol`, což ale může způsobovat problémy s přeplněním jmenného prostoru. Ve druhém případě budou všechny globální symboly v kódu přístupné skrze `symbol`, tedy `imported_symbol` bude přístupný jako `symbol.imported_symbol`. Poslední případ umožňuje importovat jen vybrané symboly a navíc jim i přiřazovat aliasy[28]. Dále v souboru samozřejmě mohou být již zmíněné prvky jazyka Solidity jako jsou kontrakty, rozhraní, chybové typy a další.

## 2.3 Typy

Solidity je staticky typovaný jazyk bez konceptu nulové hodnoty (null). Největší kategorií jsou typy hodnotové, přičemž důležité hodnotové typy jsou uvedeny v následujícím seznamu[29]:

- `bool`, `int`, `uint` a `float` – běžně používané typy z ostatních jazyků pro reprezentaci logické hodnoty, celých čísel s a bez znaménka a pro reprezentaci čísel s desetinnou čárkou.
- `address` a `address payable` – pro reprezentaci 20B adresy v Ethereu. Obě mají následující metody:
  - `balance` – pro zjištění, kolik je na adrese uloženo Etheru.
  - `code` a `codehash` – vrací kód a hash kódu pro danou adresu.

[1] `https://spdx.org/`

- `call`, `delegatecall` a `staticall` – volání na adresu při kterém se použije instrukce `CALL`, `DELEGATECALL` a nebo `STATICALL`.

  Pokud je adresa označena jako `payable`, má navíc ještě metody `transfer` a `send` pro odeslání Etheru, přičemž první z nich při neúspěchu vyvolá výjimku a druhá pouze vrátí hodnotu `false`[20].

- `contract` – typ reprezentující kontrakt.
- `bytes1`, `bytes2`, `...`, `bytes32` – fixní pole bytů o velikosti 1 až 32 bytů.
- `enum` – umožňuje definovat uživatelský typ.

Referenční typy se liší zejména tím, že na jejich hodnotu může odkazovat více proměnných, kdežto u hodnotových má každá proměnná svou vlastní hodnotu. V Solidity se rozlišuje několik typů pamětí, ve kterých jsou referenční typy ukládány a kterými může být označena proměnná při její definici:

- `memory` – v té jsou uloženy data po dobu trvání volání.
- `storage` – zde jsou data uloženy trvale po dobu životnosti kontraktu.
- `calldata` – obsahuje argumenty volání funkce smart kontraktu.

Mezi referenční typy pak patří dynamická pole bytů a textových řetězců v kódování UTF-8, v kódu jsou označené jako `bytes` a `string`. Mohou jimi být také struktury a mapy, které dovolují ukládat hodnoty pro klíče podobně jako hashovací tabulky. Tyto mapy jsou vždy ukládány do paměti *storage* a jsou tedy stavovými proměnnými. Používá se pro ně klíčové slovo `mapping`.

## 2.4 Funkce

Funkce se v Solidity dělí na funkce kontraktu a volné funkce. Jejich definice, která je vidět na ve výpisu 2.2, se skládá z klíčového slova `function`, jména, argumentů, viditelnosti, aplikovaných modifikátorů, indícií zda modifikují stav kontraktu a specifikace návratových hodnot. Vše, kromě klíčového slova, jména a argumentů, může být vynecháno.

**Výpis 2.2** Ukázka definice funkce v Solidity

```
1   function example_func(string memory c) internal test_mod pure returns (uint a, uint)
```

Volání funkcí se v Solidity dělí na interní a externí, přičemž pokud je psáno volání funkce obecně v kontextu smart kontraktů, je myšleno zpravidla volání externí. Interní volání jsou přeloženy jako jednoduché instrukce skoku a při interním volání tedy zůstává vykonávaný kód v kontextu kontraktu. Zabírají ale slot v zásobníku skoků, kterých je pouze 1024 a není tedy vhodné používat hluboko zanořenou rekurzi. Externí volání funkcí jsou prováděna na jiné nasazené kontrakty. Kontext se pro vykonávaný kód pak přesouvá pod tento jiný kontrakt, pokud ovšem není použit `DELEGATECALL`, který již byl zmíněn v minulé kapitole[30]. V případě Solidity je funkce jiného nasazeného kontraktu možno volat pomocí instance tohoto kontraktu nebo jeho adresy. Při volání u nich lze specifikovat maximální gas, který by měl být na jejich provedení spotřebován. Od těchto druhů volání je pak odvozeno i následující rozlišení mezi 4 druhy viditelnosti funkcí:[31].

- `external` – externí funkce, která může být volána pouze externě jiným kontraktem.

- `public` – veřejná funkce, která může být volána interně v kontraktu i externě jako `external`.
- `internal` – omezuje volání pouze na interní volání z kontraktu, ve kterém je definována a na kontrakty které ji od něj dědí.
- `private` – stejné omezení jako `internal`, ale nelze volat ani z dědících kontraktů.

Funkce mají indikátor toho, zda modifikují stav označením jako `view`, které stav nemodifikují a `pure`, které ho navíc ani nečtou. Jelikož tyto funkce nemodifikují stav, může být při jejich externím volání z EOA použit speciální způsob volání zvaný *eth_call*. Ten způsobí provedení volané funkce lokálně a bez vytvoření nové transakce. Díky tomu nestojí volání žádný Ether a není zaznamenáno v blockchainu. Funkce mohou být dále označeny indikátorem `payable`, pokud přijímají Ether.

Solidity má ještě speciální funkci *fallback*, kterou může mít každý kontrakt nejvíce jednou a to s modifikátorem `external`. Tato funkce slouží jako obslužná funkce pro případ, kdy je na kontraktu externě zavolána funkce, která není definována v ABI. Další speciální funkcí je *receive* sloužící k obsloužení volání posílající Ether. Ta nemůže vracet návratové hodnoty a musí být označena jako `external`.

Funkce také mohou mít uživatelsky definované modifikátory, které jsou používány k ověření podmínek nutných ke spuštění funkce, tedy například zda je adresa volající funkci uložená jako vlastník kontraktu. Tyto modifikátory jsou definovány podobně jako funkce a mohou přijímat argumenty, jak lze vidět ve výpisu 2.3. Modifikátor je na funkci aplikován jeho připsáním k deklaraci funkce za seznam argumentů. Kód modifikátoru obsahuje speciální příkaz `_` značící místo, na kterém bude v modifikátoru vykonána původní funkce a modifikátorem tedy lze definovat jak kód před, tak po vykonání funkce, na kterou je aplikován.

**Výpis 2.3** Ukázka definice a použití modifikátorů funkcí v Solidity

```
1  modifier onlyUser {
2      require(msg.sender == tx.origin);
3      _;
4  }
5
6  function example_func() public onlyUser {
7  ...
```

## 2.5 Inline assembly

Inline assembly je způsob, jakým se v Solidity dostat blíže k instrukcím EVM. Je k tomu použit jazyk Yul, který je těmto instrukcím velice blízko. Ten je zapisován do bloku `assembly { }` přímo mezi ostatní kód v Solidity, podobně jak je tomu v jazyce C s blokem `asm { }`. Je možné ho využít k optimalizaci kódu tam, kde nestačí optimalizace od kompilátoru nebo k robustnějšímu využití speciálních instrukcí, jakou je `CREATE2`, která slouží nasazení nového kontraktu. Kód napsaný v Yul má také přímý přístup k externím proměnným a externím funkcím[32].

## 2.6 Kontrakty

Kontrakty jsou jedním ze základních stavebních prvků v Solidity a jsou obdobou objektů v ostatních programovacích jazycích. V Solidity jako takový reprezentuje smart kontrakt, který

bude posléze nasazen na blockchain, pokud tedy není definován jako abstraktní pomocí klíčového slova `abstract`. Tím musí být kontrakt označen, pokud alespoň jedna z jeho funkcí není implementována nebo pokud nespecifikují argumenty pro všechny konstruktory svých rodičů.

Kontrakty mají své stavové proměnné kontraktu definované v jeho těle na stejné úrovni jako jeho funkce, které umožňují kontraktu ukládat si svůj stav v EVM. Tyto proměnné mají své vlastní modifikátory viditelnosti podobně jako funkce.

- `public` – pro tyto stavové proměnné jsou automaticky generovány getter funkce, které mohou být používány ostatními kontrakty ke čtení stavu kontraktu.
- `internal` – omezuje přístup k proměnným na kontrakt, ve kterém jsou definovány, a jeho dědicům.
- `private` – stejné omezení jako `internal`, ale proměnnou nemohou používat ani dědící kontrakty.

### 2.6.1 Externí volání funkcí

Pokud jsou funkce kontraktu označené jako `external` nebo `public`, mohou být volány externě, tedy jinými nasazenými smart kontrakty. To samé platí pro gettery funkcí stavových proměnných označených jako `public`. Tyto funkce mají v ABI vygenerovaný tzv. selektor, který je používán k identifikaci volané funkce. Tento selektor je uložen jako první 4 byty ve speciální paměti *calldata* obsahující argumenty externího volání funkce. Funkce lze v Solidity volat dvěma způsoby, a to přímo s konkrétní adresou pomocí metody `call` nebo přímo na instanci jiného kontraktu. U volání s adresou a metodou `call` je nutné specifikovat volanou funkci a její parametry, aby mohl být vytvořen její selektor v ABI, jak lze vidět ve výpisu 2.4. Při volání na instanci jiného kontraktu vše zařídí Solidity. Tento přístup ale vyžaduje, aby byl kód kontraktu předem znám a při kompilaci byl dostupný ve zdrojových souborech volajícího kontraktu. Ukázka takového volání je vidět ve výpisu 2.5.

**Výpis 2.4** Ukázka externího volání funkce pomocí `call` přímo na adresu v Solidity

```
1  addr.call(abi.encodeWithSignature("some_func(address,uint256)", addr, num_arg)
```

**Výpis 2.5** Ukázka externího volání funkce na instanci jiného kontraktu

```
1  contract Called {
2      function fn_called(uint x) {
3  ...
4
5  contract Callee {
6     function fn_call(address addr, uint x) {
7        Called c = Called(addr);
8        c.fn_called(x);
9  ...
```

## 2.7 Výrazy a kontrolní struktury

Kontrolní struktury v Solidity jsou podobné jako v jazycích C a JavaScript a jedná se o `if`, `else`, `while`, `do`, `for`, `break`, `continue` a `return`, která může vracet hned několik hodnot najednou. Pro externí volání funkcí a nasazování kontraktů je možné použít `try/catch` a zachytávat výjimky (v Solidity reprezentovány chybovými typy), které jsou vyvolány příkazem `revert` nebo z jiných důvodů přímo EVM[30].

Zajímavou vlastností Solidity je možnost přiřazovat hodnoty více proměnným najednou, podobně jako v jazyce Python, ale s tím rozdílem, že v Solidity musí být proměnné v takovémto přiřazení uzávorkovány jednoduchými závorkami, jak je vidět ve výpisu 2.6 ve funkci *f*. Solidity umožňuje vracet více hodnot najednou, jak je také vidět ve výpisu 2.6, ale ve funkci *g*. U této funkce lze pozorovat, že návratové proměnné mohou a nemusí být pojmenovány. K jejich vrácení nemusí být použit příkaz `return` a stačí, že jsou všechny návratové proměnné nastaveny.

**Výpis 2.6** Přiřazení více proměnných najednou

```
1  contract C {
2      function f() public view returns (uint x, uint y) {
3          (x, y) = g();
4          (x, y) = (y, x);
5      }
6
7      function g() public view returns (uint a, uint) {
8          a = 3;
9          return (a, 2);
10     }
11 }
```

## 2.8 Kompilace a její artefakty

Ke kompilaci je zpravidla volen oficiální kompilátor *solc*[2], jehož verze se drží verzí Solidity. Lze ho používat buďto z příkazové řádky nebo přes JSON (JavaScript Object Notation) API, které je doporučeno pro komplexnější a automatizované použití[33]. Tento kompilátor má řadu funkcionalit jako jsou optimalizace, automatické dohledávání importovaných souborů a knihoven nebo nastavení specifické verze EVM. Nejdůležitějším výstupem je dvojice bytecode, tedy zkompilovaných kódů kontraktu ve formě bytů. V rámci další analýzy bude rozlišení konkrétního typu bytecode prováděno pouze, pokud to nebude jasné z kontextu. Jedná se o tuto dvojici bytecode:

- *nasazovaný bytecode* – zkompilovaný kód ve formě bytů, který se posílá při nasazování kontraktu a typicky obsahuje kód konstruktoru a všech funkcí, které konstruktor používá a také obsahuje parametry pro konstruktor.
- *nasazený bytecode* – zkompilovaný kód ve formě bytů kontraktu po nasazení obsahující všechny funkce, které mohou být na kontraktu volány a všechen další kód, který tyto funkce používají.

Ke každému bytecode jsou také dostupné další pomocné informace, ze kterých stojí za pozornost zejména následující:

- Zkompilovaný kód v lidsky čitelné formě operačních kódů instrukcí EVM, tzv. *p*.

[2] `https://github.com/ethereum/solc-bin/`

- Mapování bytecode na zdrojový kód v Solidity, díky kterému lze zjistit, které části zdrojového kódu odpovídají kterým instrukcím v bytecode.

Ukázku zkompilovaného kódu ve formě bytecode je možné vidět ve výpisu 2.7 a ukázku stejného zkompilovaného kódu ale ve formě operačních kódů instrukcí je možné vidět ve výpisu 2.8.

**Výpis 2.7** Ukázka zkompilovaného kódu ve formě bytecode

```
1  6080604052600043610601c5760003560e01c80630be96c9c146021575b600080fd5b6027603d565b60405180828152602
   ↪ 00191505060405180910390f35b600060aa90509056fea2646970667358221220580829d3d4735c5fcd9abbdfd1268
   ↪ 5b163ede762eeb20777dcc92011a2e98e37b64736f6c63430007060033
```

**Výpis 2.8** Ukázka stejného zkompilovaného kódu jako v výpisu 2.7, ale ve formě operačních kódů instrukcí

```
1  PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x1C JUMPI PUSH1 0x0 CALLDATALOAD
   ↪ PUSH1 0xE0 SHR DUP1 PUSH4 0xBE96C9C EQ PUSH1 0x21 JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT
   ↪ JUMPDEST PUSH1 0x27 PUSH1 0x3D JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1
   ↪ 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH1 0x0 PUSH1
   ↪ 0xAA SWAP1 POP SWAP1 JUMP INVALID LOG2 PUSH5 0x6970667358 0x22 SLT KECCAK256 PC MUL SWAP14
   ↪ RETURNDATASIZE SELFBALANCE CALLDATALOAD 0xC5 0xFC 0xD9 0xAB 0xBD REVERT SLT PUSH9
   ↪ 0x5B163EDE762EEB2077 PUSH30 0xCC92011A2E98E37B64736F6C634300070600330000000000000000000000
```

Mapování bytecode na zdrojový kód je důležité pro celou řadu nástrojů od statických analyzátorů přes debuggery, až po nástroje provádějící dynamickou analýzu. Formát zápisu mapování klade důraz na kompaktnost, jelikož kompilovaný kód může být poměrně rozsáhlý a opakovat všechny hodnoty pro všechny instrukce není vhodné.

**Výpis 2.9** Ukázka mapování instrukcí na zdrojový kód v Solidity pro stejný zkompilovaný kód jako ve výpisu 2.7

```
1  23:103:0:-:0;;;;;;;;;;;;;;;;;;;;;45:79;;;::i;:::-;;;;;;;;;;;;;;;;;;;;90:4;113;106:11;;45:79;:::o
```

Tyto informace, jejichž ukázku lze vidět ve výpisu 2.9, jsou interpretovány podle následujících pravidel[34]:

- Jednotlivé instrukce jsou v mapování odděleny znakem `;`, tedy záznamy jsou uvedeny pro všechny instrukce.
- Pro každou instrukci se jedná o pětici hodnot oddělených znakem `:` a lze je popsat písmeny jako `s:l:f:j:m`, které znamenají:
  - `s` – počet bytů od začátku souboru se zdrojovým kódem v Solidity pro kód přidružený k instrukci.
  - `l` – počet bytů od `s` pro které je daná instrukce přidružená. (`s`: `s+l`) tvoří interval (*od: do*) v bytech v souboru se zdrojovým kódem.

  - `f` – index souboru se zdrojovým kódem, který je nastaven na `-1` pokud index není přidružen k žádnému zdrojovému souboru. Seznam zdrojových souborů lze zjistit z dalších artefaktů kompilace.
  - `j` – pokud je instrukcí skok, nabývá hodnot `i`, `o` nebo `-` podle toho zda se jedná o skok do funkce, návrat z funkce nebo součást smyčky.
  - `m` – určuje zanoření při vykonávání instrukcí modifikátoru funkce.
- Pokud by se některá z hodnot pětice měla opakovat s předchozími hodnotami mapování, bude vynechána, právě když se po ní všechny další hodnoty z pětice také opakují s hodnotami předchozího mapování.

Například záznam `45:78` z ukázky 2.9, pro který je část hodnot stejná jako pro předchozí záznamy, lze pomocí uvedených pravidel interpretovat následovně:

- Přidružený zdrojový kód k instrukci začíná na 45. bytu Solidity zdrojového kódu.
- Odkazovaná část zdrojového kódu je 79 bytů dlouhá.
- Jedná se o soubor s indexem 0.
- Pokud se jedná o skok, tak je součástí smyčky.
- Instrukce není součástí modifikátoru funkce.

### 2.8.1 Chyby způsobené kompilátorem

Chyby způsobené kompilátorem jsou kategorií chyb, které tvůrci smart kontraktů mohou jen těžko ovlivnit. Tím nejlepším, co tvůrci smart kontraktů mohou udělat, je kompilovat své kontrakty s nejnovějšími verzemi kompilátorů a sledovat upozornění na nové chyby, které se v těchto kompilátorech objeví. Přehled takovýchto chyb pro oficiální kompilátor *solc* je dostupný ve formě článků na oficiálním blogu jazyka Solidity[3].

Jedním z dobrých příkladů je chyba opravená ve verzi 0.7.4 se střední závažností, která může způsobit, že proměnná v paměti *storage* bude neinicializovaná, tedy nebude nastavena na hodnotu `0`, jak by programátor očekával[35]. Tato chyba byla způsobena snahou snížit množství gasu potřebné k uložení stavové proměnné typu pole s délkou menší než 1 slot (tedy 32 bytů) a následným nesprávně vygenerovaným kódem, který neošetřoval případ, kdy je délka pole 0.

## 2.9 Nasazování kontraktů a konstruktory

Nasazování kontraktů lze provádět dvěma způsoby, pně nebo z již nasazeného kontraktu. Při externím nasazování je poslána transakce s kódem *nasazovaného bytecode* kontraktu určeného k nasazení z EOA s prázdnou adresou příjemce. Na základě této prázdné adresy je transakce rozpoznána jako žádost o nasazení nového kontraktu.

Druhým způsobem nasazení nového kontraktu je využití nízkoúrovňových instrukcí `CREATE` a `CREATE2` nebo za pomoci klíčového slova `new`, jak lze vidět ve výpisu 2.10.

Výhodou přímého využití instrukce `CREATE2` je možnost nasadit kontrakt na předem určenou adresu. Instrukce `CREATE` se používá obdobně, ale neumožňuje specifikovat parametr `_salt`, který je součástí výpočtu adresy nově nasazeného kontraktu. Samotné `new` také používá instrukce `CREATE` a `CREATE2`, ale použití druhé zmíněné instrukce podporuje až od verze 0.8[30].

Každý kontrakt může definovat právě jeden konstruktor, který je volán při nasazování tohoto kontraktu. Argumenty konstruktoru jsou přidávány za *nasazovaný bytecode* kontraktu. Pokud

[3] `https://blog.soliditylang.org/category/security-alerts/`

**Výpis 2.10** Ukázka způsobů nasazování nových kontraktů

```
contract ExampleContract {
    string public name;
    constructor(string _name) {
        name = _name
    }
...
ExampleContract c = new ExampleContract("ContractViaNew");
...
assembly {
    new_addr := create2(callvalue(), add(bytecode, 0x20), mload(bytecode), _salt)
}
```

kontrakt dědí od jiných kontraktů, může konstruktorům těchto jiných kontraktů předat parametry dvěma způsoby, které lze vidět v ukázce 2.11. První ze způsobů v kontraktu *Child* volá rodičovský konstruktor s argumentem podobně jako modifikátor. Druhý způsob specifikuje argument konstruktoru již při dědění od rodičovského kontraktu.

**Výpis 2.11** Ukázka způsobů volání konstruktorů

```
contract Parent {
    string public name;
    constructor(string memory _name) {
        name = _name;
    }
}

contract Child is Parent {
    constructor(string memory _name) Parent(_name) { }
}

contract Child2 is Parent("parent construtor argument") {
    constructor() { }
}
```

## Kapitola 3

# Hledání chyb a zranitelností

Hledání chyb a případně zranitelností v programech je netriviální úkon, který lze praktikovat jak při vývoji software, tak po jeho dokončení v rámci oficiálních auditů nebo potenciálními útočníky za účelem zneužití chyb. Jsou k němu používány různé techniky, které se obecně řadí mezi statickou nebo dynamickou analýzu. Při statické analýze jsou hledány chyby bez spuštění programu samotného pouze z jeho kódu, který může být kódem zdrojovým nebo již zkompilovaným. U dynamické analýzy je naopak program plně nebo z části spouštěn a je pozorováno jeho chování při různých vstupech. Techniky z obou oblastí mají své výhody a nevýhody. Statická analýza je sice rychlejší, ale typicky má vyšší podíl hlášení chyb, které však chybami nejsou. Dynamická analýza je zase pomalejší a často náročnější na nastavení, ale dokáže objevit chyby, které statická analýza z principu objevit nemůže. Některé typy chyb jsou však jednoduše detekovatelné statickou analýzou a těžko detekovatelné dynamickou. Tyto techniky pak lze kombinovat a využívat informace získané ze statické analýzy při analýze dynamické[36].

Nástroje pro statickou a dynamickou analýzu pak používají uživatelé, na kterých je zpracování výstupu těchto nástrojů. Rozhodují tak o tom, zda chybné chování zjištěné analýzou odpovídá reálnému chování analyzovaného programu a zda se tedy jedná o validní nález. Pokud se jedná o validní nález, mohou doporučit způsob nápravy a případně ji přímo implementovat, nebo se mohou pokusit chybu zneužít, pokud se jedná o útočníky. V případě, že se nejedná o validní nález, mohou daný nástroj pro statickou nebo dynamickou analýzu upravit dle potřeby, zejména pokud takových nevalidních nálezů bylo více. Tento problém se ale objevuje zejména při statické analýze, která je na nevalidní nálezy náchylnější než analýza dynamická.

V této kapitole budou zavedeny konkrétní pojmy z oblasti statické a dynamické analýzy o které se následně bude opírat analýza požadavků cílů této práce, návrh jejich implementace a implementace samotná.

## 3.1 Statická analýza

Statická analýza kódu je dobře škálovatelná metoda hledání chyb a případných zranitelností z nich plynoucích. Jedná se o analýzu kódu jako takového bez jeho spouštění, díky čemuž je rychlejší a méně náročná než analýza dynamická, která kód spouští. Nasazení statické analýzy kódu je také zpravidla snadnější, než u dynamické analýzy, protože nevyžaduje kompletní a ani plně funkční kód, který by měl být testován, a zároveň lze statickou analýzu použít na velký objem kódu najednou bez konkrétního zacílení a předchozí analýzy možných problémových částí. Obecně ale statická analýza není schopná najít všechny zranitelnosti kvůli nemožnosti vyhodnotit veškerý kód bez jeho vykonání[37].

Zejména u statických detektorů chyb je tak často nutné pracovat s falešně pozitivními a falešně

negativními výsledky. Falešně pozitivní detekce znamená, že statický detektor hlásí nález chyby, i když v analyzovaném kódu tato chyba není. Falešně negativní detekce nastane, pokud chyba v kódu je, ale detektor ji není schopen detekovat. Obecně lze říct, že falešně pozitivní detekce je lepší než falešně negativní, protože analytik ověřující detekci má možnost falešně pozitivní detekci prověřit a případně ji označit za falešnou, kdežto o falešně negativní detekce analytik neví. Míra falešných detekcí by ale měla být přiměřená a detektor by měl být naprogramován dostatečně citlivě tak, aby analytika nezahltil falešně pozitivními detekcemi, ale zároveň by neměl každý kód prohlásit za bezchybný, i když v něm chyby jsou[38].

### 3.1.1 Vnitřní reprezentace

Vnitřní reprezentace (IR – Intermediate Representation) obecně slouží k reprezentaci programu bez ztráty dat o něm tak, aby s ním šlo lépe pracovat v závislosti na konkrétních požadavcích od IR a její následného použití. IR lze generovat jak ze zdrojového kódu, tak z již kompilované binární podoby[39]. V závislosti na požadované sofistikovanosti se může se jednat o strukturované stromy podobné AST, ale i komplexní kód s vlastními strukturami, který pak lze také nazvat mezijazykem[40]. Ukázku IR používaného ke statické analýze lze vidět ve výpisu 3.1.

**Výpis 3.1** Ukázka SlithIR nástroje Slither

```
1  Contract Callee
2      Function Callee.call(uint256,address) (*)
3          Expression: a == 1
4          IRs:
5              TMP_0(bool) = a == 1
6              CONDITION TMP_0
7          Expression: Called(_addr).receive_A()
8          IRs:
9              TMP_1 = CONVERT _addr to Called
10             TMP_2(uint256) = HIGH_LEVEL_CALL, dest:TMP_1(Called), function:receive_A,
               ↪  arguments:[]
11         Expression: a == 2
12         IRs:
```

### 3.1.2 Abstraktní syntaktický strom

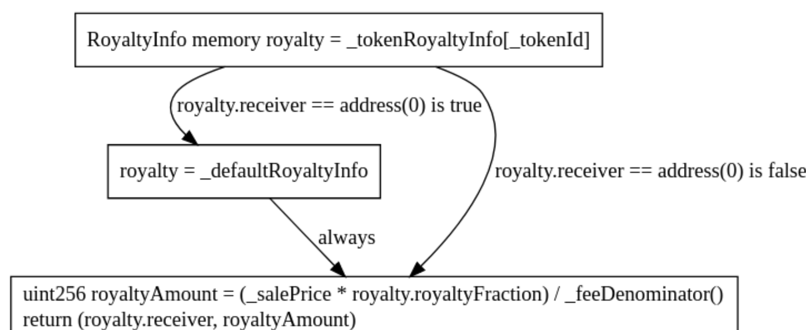
Abstraktní syntaktický strom (AST – Abstract Syntax Tree) je datová struktura popisující strukturu kódu. Může obsahovat další informace o jednotlivých uzlech tohoto stromu, například o datových typech proměnných, se kterými se v daných uzlech pracuje[41]. AST lze při statické analýze využít třeba jeho procházením a porovnáváním prošlých struktur s předdefinovanými vzorci a pravidly. Tímto lze odhalit celou řadu informací o kódu, jakou může být sběr obecných metrik, analýza komplexity, detekce neadekvátní struktury nebo kontrola správného sledu vybraných příkazů[42].

### 3.1.3 Grafy toku řízení a dědičnosti

Graf toku řízení (CFG – Control Flow Graph) je reprezentace všech cest, které kód může projít při svém běhu ve formě orientovaného grafu s hranami orientovanými podle směru průchodu kódu. Kód je v něm rozdělen na maximálně možné velké bloky podle řídících struktur a příkazů

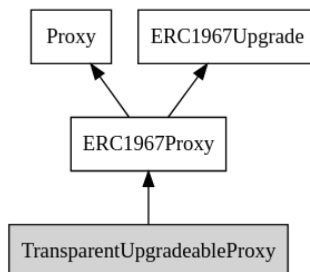
měnící tok kódu. Mezi jednotlivými bloky je pak možné přecházet jen pomocí hran v grafu. CFG mají dva speciální typy uzlů, a to uzel začátku, který značí místo kde kód začíná a koncový uzel, který značí místo, kde kód končí[43]. Ukázku tohoto grafu lze vidět na obrázku 3.1

**Obrázek 3.1** Ukázka CFG vygenerovaného nástrojem Woke pro funkci royaltyInfo z kontraktu ERC2981 projektu OpenZeppelin




Dalším užitečným grafem je graf dědičnosti, který znázorňuje vztah dědičnosti tříd v programu. Jedná se také o orientovaný graf, jehož hrany jsou orientovány od třídy, která dědí, ke třídě, od které je děděno. Oba tyto grafy jsou využívány jak nástroji provádějícími optimalizace a statickou analýzu, tak programátory a analytiky pro lepší pochopení a orientaci v kódu samotném. Ukázku tohoto grafu lze vidět na obrázku 3.2.

**Obrázek 3.2** Ukázka grafu dědičnosti vygenerovaného nástrojem Woke pro kontrakt TransparentUpgradeableProxy projektu OpenZeppelin




## 3.2 Dynamická analýza

Dynamická analýza spočívá ve spuštění programu nebo jeho části a následném sledování chování programu. Její výstupy se vyznačují větší spolehlivostí než u analýzy statické[44]. Systematické dynamické testování programů pak může pomoct odhalit celou řadu chyb a takto nalezené chyby jsou jednoduše reprodukovatelné, protože jsou známy veškeré vstupy, které program přijal. Dynamická analýza může využívat výstupy statické analýzy jako podklad.

### 3.2.1 Pokrytí kódu

Pokrytí kódu je důležitou metrikou při dynamickém testování programů. Dává totiž informaci o tom, jak velká část programu je testována a může být použita k dohledání toho, které části to jsou, a nebo jak moc jsou některé kusy kódu testovány oproti jiným[45]. Pokrytí kódu lze měřit na více úrovních, konkrétně na úrovni:

**Příkazů** – Jedná se o jednotlivé příkazy jazyka ve kterém je kód k dispozici. Vypovídající hodnota tohoto pokrytí spočívá zejména v určení, jak velká část kódu samotného je testována. Tato informace samotná ale většinou neposkytuje dostatečně komplexní pohled na problém testování celého programu. Je totiž možné dosáhnout velkého procentuálního pokrytí kódu a i tak vynechat některé kusy kódu vyžadující specifické podmínky, aby byly vykonány.

**Větví kódu** – Určení pokrytí větví kódů vyžaduje rozdělení kódu na větve, které je prováděno podle řídících struktur a příkazů ovlivňujících tok kódu. Jedná se tedy o uzly CFG. Typickým příkladem řídících struktur jsou podmínky, které mohou definovat kód pro různé situace, které mohou v kódu nastat v závislosti na vstupních datech. Informace o pokrytí podle větví kódu poskytuje lepší přehled o tom, kolik možných situací, které v kódu mohou nastat, bylo skutečně otestováno.

**Cest v kódu** – Oproti pokrytí větví kódu rozlišuje i možné kombinace posloupnosti procházení větví[46].

### 3.2.2 Testování na základě vlastností

Testování na základě vlastností (PBT – Property-based testing) spoléhá na nalezení obecně platných vlastností testovaného programu nebo jeho částí. Pro takovéto vlastnosti jsou naprogramovány tzn. invarianty, které ověřují jejich validní stav během testování. Velké množství invariantů, které projdou dostatečně rigorózním testováním bez jejich porušení, demonstruje, že vlastnosti testovaného programu zůstávají takové, jaké mají být. Naopak nalezení invariantu, který byl porušen, napovídá, že v programu může být chyba, která může způsobit neočekávaný stav a chování programu. Příkladem vlastnosti a invariantu může být situace, kdy program dělá symetrické operace kódování a dekódování. Jako vlastnost lze označit požadavek, že pokud jsou provedeny operace zakódování a dekódování nějakého vstupu, bude po těchto operacích výstup stejný jako vstup. Invariant by tedy ověřoval, zda předchozí vlastnost platí[47].

Program popsaný invarianty je testován automatizovaně za pomoci generátorů vstupů, které mohou být náhodné, ale mohou také sledovat různé strategie výběru generovaných hodnot, jenž lépe prozkoumávají prostor možných vstupů.

### 3.2.3 Fuzzing

Fuzzing je technikou dynamického testování programů spočívající ve spouštění daného programu se vstupními daty, které mohou vznikat buďto generováním nových vstupů, mutací původních vstupů nebo kombinací obojího. Následně je při fuzzingu sledováno chování tohoto programu za běhu a jsou detekovány případné vzniklé chyby. Je to efektivní a široce používaná technika testování software a existuje celá řada projektů a fuzzerů, tedy programů provádějících fuzzing, zaměřujících se na různé úrovně programů[48]. Příkladem může být následující výběr fuzzerů počínaje fuzzerem syzkaller[1], zaměřujícím se Linuxový kernel, Afl++[2], primárně pro programy v C/C++, Atheris[3], kterým lze testovat kód v Pythonu a syntribos[4], který slouží k fuzzování webových API.

Generováním vstupů je myšleno jejich generování podle předem dané specifikace, kde jsou zároveň záměrně generovány částečně nevalidní vstupy. Tento způsob generování bývá používán pro protokoly nebo soubory se specifickým formátem. Příprava této specifikace je ale časově náročná a vyžaduje mnohem větší úsilí uživatele, který fuzzing provádí. Mutace vstupů pracuje nad uživatelem dodaným vstupem, na který aplikuje transformační pravidla většinou náhodného charakteru. Jednoduché mutační fuzzery mohou vstup modifikovat prohozením bytů nebo

[1] `https://github.com/google/syzkaller`
[2] `https://github.com/AFLplusplus/AFLplusplus`
[3] `https://github.com/google/atheris`
[4] `https://github.com/openstack-archive/syntribos`

vygenerováním náhodné sekvence bytů a nahrazením části vstupu. Pokročilejší mutační fuzzery pak umějí odhalit strukturu vstupu na základě zpětné vazby z fuzzeru a podle toho provádět mutace na části vstupu tak, aby byl vstup z větší části validní. Zpětná vazba lze ale také použít k mutacím tak, aby byly maximalizovány určité cíle, které se navíc mohou v průběhu kampaně měnit. Na začátku kampaně, tedy specifického spuštění fuzzeru na testovaném programu, může být cílem se dostat do co nejvíce větví testovaného programu a pamatovat si části vstupu, které k němu vedly. Na konci kampaně se cíl může změnit na snahu v každé nalezené větvi vyzkoušet co největší objem mutovaných vstupních dat. Problémem mutačních fuzzerů obecně jsou například kontrolní součty, které mutace nedokáží správně měnit a také složité formáty vstupů[49].

Samotný fuzzer může buďto pracovat pouze se zkompilovaným kódem bez znalosti zdrojového kódu nebo se znalostí zdrojového kódu a možností tento kód zkompilovat, což usnadňuje tzv. instrumentaci, tedy modifikace které fuzzer může provádět na testovaném programu za účelem efektivnějšího fuzzingu. U sledování programu za běhu lze například získávat právě pokrytí testovaného kódu, tedy které části kódu programu jsou vykonávány s danými vstupy. S pokrytím kódu je spojená i tzv. *taint analysis*, která se snaží odhalit závislosti mezi strukturou vstupu a tím, které větve kódu budou v programu vykonány. Podle toho, jaký program je testován, pak lze sledovat specifičtější chování programu, kterým může být například sledování paměti využívané programem. V kontextu Etherea tímto chováním mohou být interakce mezi smart kontrakty nebo objem Etheru, který je ve smart kontraktu uložen. Detekce vzniklých chyb je závislá na typu testovaného programu a může se jednat o detekci pádů a neošetřených výjimek, ale i o komplexní detekci nesprávného chování programu například pomocí invariantů z PBT[50].

Příkladem takového PBT fuzzeru je Echidna[5] jakožto fuzzer pro Ethereum smart kontrakty. Echidna generuje transakce a volá funkce smart kontraktů na základě jejich ABI a snaží se maximalizovat pokrytí kódu díky zpracování zpětné vazby. Při testování ověřuje invarianty, které jsou naprogramovány přímo v Solidity, což umožňuje jednoduše zjišťovat a ověřovat správnost vnitřního stavu kontraktu.

[5] `https://github.com/crytic/echidna`

# Kapitola 4

# Nástroje Slither a Woke

V oblasti bezpečnosti Ethereum smart kontraktů začaly již po prvních úspěšných útocích vznikat nástroje, jejichž účelem je pomoct vývojářům smart kontraktů jejich kontrakty zabezpečit. V této kapitole bude představen nástroj Slither, který disponuje širokou škálou statických detektorů, a nástroj Woke, který se snaží nástroj Slither nahradit a kromě statických detektorů disponuje i fuzzerem pro smart kontrakty.

## 4.1 Slither

Slither[1], vytvořený společností Trail of Bits, je open source nástroj pro statickou analýzu smart kontraktů naprogramovaných v Solidity. V době psaní této práce má Slither 80 statických detektorů schopných detekovat nedoporučované praktiky v kódu smart kontraktů a často se vyskytující chyby a zranitelnosti. Mimo statické detektory má také 18 funkcí pro výpisy, tzv. *printers*, které slouží k získání a prezentaci celé řady informací o kontraktech, ať už jsou to informace o struktuře kontraktů z pohledu dědičnosti, vytvoření a export grafu volání nebo výpis reprezentace SlithIR. Autoři o něm také tvrdí, že je schopen správně pracovat s 99.9% veřejného kódu v Solidity a slibují nízkou míru falešně pozitivních detekcí[51].

Kromě zmíněných vestavěných detektorů umožňuje framework jednoduché rozšíření o uživatelské detektory v jazyce Python díky své vlastní vnitřní reprezentaci Solidity kódu nazvané SlithIR (Slither Intermediate Representation). Tato reprezentace sjednocuje některé prvky Solidity, tedy třeba různé způsoby jak zavolat funkci jiného kontraktu a pomáhá tak při precizní analýze smart kontraktů a detekci komplexních vzorů. Slither má také Python API pomocí kterého mohou uživatelé automatizovat použití Slitheru a používat ho ve svých skriptech[52].

### 4.1.1 Statické detektory chyb a zranitelností

Vestavěné detektory v nástroji Slither se dělí do 3 kategorií, a to detektorů zranitelností, čistě informačních a možných optimalizací. Mezi detektory zranitelností patří například:

- Překrývání proměnných, tedy třeba lokální proměnná překrývající proměnnou globální.
- Nechráněná metoda `upgrade` pro nasazení nové verze kontraktu.
- Detekce slabého pseudonáhodného generátoru.

[1] `https://github.com/crytic/slither`

Informační detektory slouží zejména k upozornění na porušení doporučených postupů, používání zastaralých funkcí a dalších informací podobného typu. Detektory možných optimalizací jsou pouze dva, a to detektor proměnných, které by měly být deklarovány jako konstanty a detektor funkcí, které by měly být deklarovány jako externí.

**Výpis 4.1** Ukázkový detektor funkcí obsahujících slovo „upgrade“ v názvu v nástroji Slither

```
1  // SPDX-License-Identifier: GPL-3.0
2  from slither.detectors.abstract_detector import AbstractDetector, DetectorClassification
3
4  class UpgradeFunctionDetector(AbstractDetector):
5      ARGUMENT = 'upgrade_fn_detector'
6      HELP = 'Help printed by slither'
7      IMPACT = DetectorClassification.INFORMATIONAL
8      CONFIDENCE = DetectorClassification.HIGH
9
10     WIKI = 'link to wiki'
11     WIKI_TITLE = 'Function with "upgrade" in its name'
12     WIKI_DESCRIPTION = 'Detects function with "upgrade" in its name'
13     WIKI_EXPLOIT_SCENARIO = 'This could be part of detection for proxy and implementation
       ↪  contracts'
14     WIKI_RECOMMENDATION = 'empty'
15
16     def _detect(self):
17         results = []
18         for contract in self.slither.contracts_derived:
19             for fn in contract.functions:
20                 if "upgrade" in fn.name.lower():
21                     results.append(self.generate_result(["Function with \"upgrade\" in its name
                       ↪  found", fn, "\n"]))
22         return results
```

Jednoduchý detektor pro detekci funkce obsahující řetězec „upgrade“ v názvu lze vidět ve výpisu 4.1. Detektory dědí od abstraktní třídy *AbstractDetector* a musí implementovat metodu `_detect` vracející nalezené výsledky. Také lze detektoru nastavit odpovídající dopad detekované zranitelnosti a důvěryhodnost detekce pro případy, kdy detektor neumí nebo nemůže jistě říct, zda skutečně detekoval to, co měl[53].

Detekce pak využívá instance třídy *Slither* dědící od třídy *SlitherCore*, kde jsou z pohledu detektorů důležité třídní proměnné `contracts` a `contracts_derived` se seznamem kontraktů, nad kterými může detektor provádět analýzu. Kontrakty reprezentované třídou *Contract* pak již mají všechny potřebné informace jako je jejich název, typ kontraktu, definované výčty, funkce a proměnné, přičemž všechny jmenované jsou reprezentovány svými vlastními třídami s potřebnými informacemi[54]. Uvedený detektor tedy projde všechny kontrakty, jejich funkce a zjistí, jestli v názvu funkce není řetězec „upgrade“. Výstup detektoru nástroje Slither je možné vidět ve výpisu 4.2.

### 4.1.2 Funkce pro výpis informací

Slither disponuje funkcionalitou umožňující výpis nejrůznějších informací o kontraktech, jejich funkcích, dědičnosti a dalších. Také ale umožňují nahlédnout na kontrakty pohledem reprezentace Slitheru nebo pomocí CFG. Na rozdíl od detektorů nejsou tyto funkce pro výpis informací přímo

**Výpis 4.2** Ukázka detekce slabého pseudonáhodného generátoru nástrojem Slither

```
$ slither weak_prng.sol --detect weak-prng
Compilation warnings/errors on weak_prng.sol:
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding
↪  a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use
↪  "SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see https://spdx.org
↪  for more information.
--> weak_prng.sol

Game.guessing() (weak_prng.sol#7-9) uses a weak PRNG: "reward_determining_number =
↪  uint256(blockhash(uint256)(10000)) % 10 (weak_prng.sol#8)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
weak_prng.sol analyzed (1 contracts with 1 detectors), 1 result(s) found
```

určeny k rozšiřování uživatelem. Mají zpravidla jeden ze dvou formátů výstupu, a to textový a DOT[2][55]. Jednou z jeho funkcí je například vypsání IR SlithIR, které bylo využito pro vytvoření ukázky 3.1 v předchozí kapitole.

## 4.2 Woke

Woke[3] je poměrně nový open source nástroj a testovací framework pro smart kontrakty vytvořený společností Ackee Blockchain. Mezi jeho hlavní funkcionality patří statické detektory zranitelností, PBT fuzzer a provázání s IDE (Integrated Development Environment) pomocí LSP (Language Server Protocol) serveru.

Nástroj je vytvořen v jazyce Python a podporuje jeho verze 3.7 a vyšší. Využívá tzv. *type hints*[4], tedy možnosti specifikovat používané typy v kódu. Ty jsou ve Woke následně kontrolovány nástrojem pyright[5]. Kromě něj je při vývoji využíváno formátování nástrojem Black[6], který sjednocuje podobu kódu v projektu. Projekt je částečně pokryt testy, které jsou spouštěny nástrojem pytest[7]. Jednotlivé funkcionality jsou odděleny do svých vlastních balíčků (například `woke.cli` pro CLI, `woke.lsp` pro LSP server nebo `woke.ast.ir` pro IR Solidity).

### 4.2.1 Fuzzer

Woke disponuje svým vlastním PBT fuzzerem, který je výjimečný zejména tím, že fuzz testy jsou vytvářeny v Pythonu. Samotný fuzzer je poměrně jednoduchý bez užití pokročilých technik jako je navádění podle pokrytí kódu nebo automatické mutace vstupů podle zpětné vazby. Fuzz testy jsou totiž vytvářeny tak, že přímo uživatel vybírá funkce, které budou testovány a jejich sled a parametry, do kterých budou dosazována data, která mohou být náhodně generována. Tento způsob testování umožňuje lepší kontrolu nad tím, které části kódu budou testovány, ale vyžaduje větší zapojení uživatele při jejich vytváření a lepší pochopení kódu, který bude testován. Woke také podporuje spouštění fuzz testů na více procesech najednou, nicméně tyto procesy spolu nijak nekomunikují a nesdílejí mezi sebou žádné informace ani si dynamicky nerozdělují práci.

[2] DOT je formát běžně používaný pro popis grafů
[3] `https://github.com/Ackee-Blockchain/woke`
[4] `https://peps.python.org/pep-0484/`
[5] `https://github.com/microsoft/pyright`
[6] `https://github.com/psf/black`
[7] `https://docs.pytest.org/en/7.2.x/`

Generování vstupů má na starosti několik vestavěných metod popsaných níže a v případě potřeby sofistikovanějších generátorů vstupů je na uživateli, aby si tyto generátory vytvořil. Již hotovými funkcemi pro generování dat jsou:

- `random_account(lower_bound, length, predicate, chain)` – pro výběr jednoho či více náhodných existujícího účtů v testing frameworku.
- `random_address()` – pro generování náhodné adresy.
- `random_int(min, max)` – pro vygenerování náhodného čísla s upraveným pravděpodobnostním rozložením zaručující častější generování minima a maxima a 0, pokud je v předávaných mezích. Pro zbytek čísel je použit generátor náhodných čísel Pythonu `random`.
- `random_bool()` – vracející náhodně hodnoty `True` nebo `False`.
- `random_string(min, max, alphabet, predicate)` – generující řetězec o dané minimální a maximální délce se znaky z dané abecedy.
- `random_bytes(min, max, predicate)` – generující náhodnou sekvenci bytů o specifikované minimální a maximální délce.

Argument `predicate` ve výše zmíněných metodách umožňuje předat funkci, která bude volána na generované hodnoty a umožňuje tak bližší specifikaci, které hodnoty by měly být při generování vynechány. Toho může být využito například při výběru náhodného účtu, kdy je možné vynechat některé konkrétní účty, jako je účet samotného kontraktu.

Ukázku jednoduchého fuzz testu lze vidět ve výpisu 4.3. Jako první je spuštěna metoda `test_fuzz`, ta provádí nastavení blockchainu, na kterém bude probíhat testování, a také vytváří instance třídy *Campaign*, která je následně spuštěna s třídou *Sequence*. Z této třídy jsou pak vybrány metody označené dekorátorem `flow`, které budou dále v textu označovány jako *flow*, z nichž je vytvořena sekvence spouštění těchto metod. Samotné flow popisuje sled akcí s kontraktem a blockchainem, které chce uživatel nasimulovat. Na konci sekvence volání flow jsou pak spuštěny metody označené dekorátorem `invariant`, které mají stejný účel jako invarianty v PBT, a ve kterých může uživatel kontrolovat stav kontraktu a porovnávat ho s předpokládaným chováním. Metoda `run` třídy *Campaign* má 2 argumenty, kde prvním je počet sekvencí a druhým počet flows. To znamená, že pokud má být spuštěno 10 sekvencí po 100 flow, bude celkově spuštěno 1000 flow. Mezi jednotlivými sekvencemi je vývojový blockchain vyresetován, takže lze simulovat jak interakce s kontraktem, který je nasazen krátce, tak s kontraktem, který je nasazen déle.

Nasazení totiž probíhá v konstruktoru *Sequence* a tedy na začátku každé sekvence. Pokud při vykonávání flow nebo invariantů nastane výjimka, je provádění zastaveno a uživateli jsou vypsány informace o výjimce a čeká se na jeho pokyn, jestli se má pokračovat v testu nebo zda má být spuštěna instance interaktivního debuggeru pdb, pomocí kterého může uživatel zjistit další informace o stavu testu, testovaného kontraktu a blockchainu. Samotné generování sekvencí flow pak lze ovlivnit pomocí těchto dekorátorů:

- `weight(x)` – nastavuje váhu flow při generování. Výchozí hodnota váhy flow je 100 a pokud tedy budou existovat 2 flow, jedno z nich bude mít váhu 100 a druhé 200, bude v průměru spuštěno první flow v 1/3 z celkových spuštění a druhé flow ve 2/3 celkových spuštění.
- `max_times` – umožňuje specifikovat maximální počet spuštění daného flow.
- `ignore` – slouží hlavně pro testování a debugging. Flow jím označené bude při generování sekvence flows ignorováno. Tento dekorátor lze použít i pro invarianty.

**Výpis 4.3** Ukázkový fuzz test v nástroji Woke

```
from woke.testing import default_chain
from woke.testing.campaign import Campaign
from woke.testing.decorators import flow, weight, invariant
from woke.testing.random import random_address

from pytypes.contracts.Counter import Counter

class Sequence:
    counter: Counter
    value: int

    def __init__(self) -> None:
        self.counter = Counter.deploy()
        self.value = 0

    @flow
    @weight(150)
    def flow_increment(self) -> None:
        self.counter.increment()
        self.value += 1

    @invariant
    def invariant_count(self) -> None:
        assert self.counter.count() == self.value


def test_fuzz():
    default_chain.default_account = random_address()
    default_chain.gas_price = 0

    campaing = Campaign(Sequence)
    campaing.run(10, 100)
```

#### 4.2.1.1 Spouštění fuzzeru a kampaní

Implementace fuzzeru se v projektu nachází v balíčku `woke.testing` v modulu `fuzzer` a samotný fuzzer je spouštěn metodou `fuzz`. Ta podle nastavených parametrů spustí požadovaný počet procesů a následně se v hlavním procesu stará o přijímání výjimek z ostatních procesů a případnou interakci s uživatelem a spuštění debuggeru. Samotné procesy jsou spouštěny pomocí balíčku `multiprocessing` a jeho třídy *Process*, a komunikace mezi procesy je realizována pomocí tříd *Pipe* a *Event* ze stejného balíčku. Každý proces je spuštěn s funkcí `_run`, která slouží k rozlišení, kam proces bude posílat svůj výstup a k přípravě ošetření výjimek v procesu a jejich poslání hlavnímu procesu. Následně je spuštěna funkce `_run_core`, která procesu spustí jeho vlastní vývojový blockchain, na kterém probíhá testování a předá požadované parametry spolu s funkcí, která je použita ke spuštění kampaně.

Samotné spouštění přes CLI obstarává metoda `run_fuzz` modulu `fuzz` balíčku `woke.cli`. Tento balíček obecně slouží k obsluze uživatelských příkazů z CLI a v případě fuzzeru načte soubory s fuzz testy, vyhledá metody začínající řetězcem `test` a spustí pro každou z nich fuzzer.

Pro práci s CLI je používán balíček Click[8].

#### 4.2.1.2 Záznamy testování

Záznamy o průběhu testování ve formě jednotlivých flow a posílaných transakcí, které byly zvoleny ke spuštění, je možné najít v adresáři `.woke-logs/fuzz`, kde se nacházejí adresáře pojmenované podle času a data kdy byl fuzz test spuštěn. Také zde existuje speciální adresář `latest` odkazující na adresář posledního spuštění fuzzeru. V adresářích se pak nacházejí soubory se záznamy pro každý z procesů. Tyto soubory obsahují speciální znaky obarvující text a je možné je prohlížet například příkazem `less -r`, který tyto znaky interpretuje.

#### 4.2.1.3 Testovací framework

Testovacím frameworkem je v rámci fuzzeru nástroje Woke myšlen framework, který se stará o vše spojené s komunikací s blockchainovou sítí, reprezentací adres, kontraktů, transakcí, eventů a všeho dalšího na ní. V době psaní této práce probíhá v nástroji Woke přechod z frameworku Brownie[9] na vlastní framework. Brownie je vývojový a testovací framework umožňující jednoduchou práci s účty, kontrakty a blockchainy. Využívá frameworku pytest a umožňuje psát testy pro smart kontrakty. Hlavním důvodem pro náhradu Brownie za vlastní testovací framework je zejména rychlost samotného Brownie a špatná podpora rychlejších vývojových blockchainů, tedy například Anvil. Samotné Brownie má také celou řadu závislostí, které komplikovaly instalaci Woke jako takového a neumí například pracovat s uživatelsky definovanými chybami ve smart kontraktech.

Nový testovací framework je zaměřen na rychlost a podporu rychlých lokálních vývojových blockchainů právě kvůli tomu, aby fuzzer mohl otestovat co nejvíce kombinací vstupů v krátkém čase. Také ale přináší příjemnější psaní fuzz testů díky předgenerovaným typům smart kontraktů v Pythonu, ve Woke označovaných jako *pytypes*. Díky nim funguje v IDE našeptávání pro funkce smart kontraktů, parametry jejich funkcí a další. Nový testovací framework implementuje vlastní komunikaci s API vývojového blockchainu přes IPC (Inter-process Communication), HTTP (Hypertext Transfer Protocol) a technologii websocket.

Dále umožňuje vynutit volání funkcí označených `pure` a `view` pomocí transakce místo *eth_call*, které lze použít při externím volání z EOA. Při volání funkcí je možné specifikovat argument `request_type="transaction"`, díky kterému testovací framework nepoužije volání *eth_call*, který jinak používá. Ve výchozím stavu je použito volání *eth_call* a vynucení volání jako transakce, které je pomalejší, je ponecháno na uživateli.

### 4.2.2 Statické detektory chyb a zranitelností

Podobně jako nástroj Slither má Woke statické detektory využívající své vlastní IR kódu Solidity. Woke přímo nerozlišuje typ detektoru, ani nedává možnost určit hodnotu spolehlivosti detektoru a závažnosti detekce jako Slither. V době psaní této analýzy má 8 hotových detektorů, s příklady detektorů popsanými dále.

- Nezpracování návratové hodnoty po volání funkce.
- Nedostatečně chráněná metoda s příkazem `selfdestruct`.
- Delegované volání nebezpečného kontraktu.

Jednoduchý detektor funkce obsahující řetězec „upgrade“ v názvu lze vidět ve výpisu 4.4. Detektory dědí od třídy *DetectorAbc* a implementují metodu `report`, která vrací seznam objektů

[8] `https://click.palletsprojects.com/en/8.1.x/`
[9] `https://github.com/eth-brownie/brownie`

**Výpis 4.4** Detektor funkcí obsahujících slovo „upgrade“ v názvu v nástroji Woke

```python
from typing import List, Set
from woke.analysis.detectors import DetectorAbc, DetectorResult, detector
from woke.ast.ir.declaration.function_definition import FunctionDefinition

@detector(-1000, "upgrade_fn")
class UpgradeFunctionDetector(DetectorAbc):
    """
    Detects functions with "upgrade" in name
    """
    _detections: Set[DetectorResult]

    def __init__(self):
        self._detections = set()

    def report(self) -> List[DetectorResult]:
        return list(self._detections)

    def visit_function_definition(self, node: FunctionDefinition):
        if "upgrade" in node.name:
            self._detections.add(DetectorResult(node, "Function name with \"upgrade\" detected",
            ↪ related_info=()))
```

*DetectorResult* s detekcemi. Tyto detekce jsou vázané na konkrétní uzel IR spolu se zprávou, co bylo detekováno a s případným dalším seznamem detekcí v `related_info`. Díky těmto informacím navíc lze hlavní detekci doplnit o jisté poddetekce, které mohou blíže specifikovat, proč došlo k hlavní detekci. Ukázku výpisu těchto poddetekcí lze vidět na obrázku 4.1. Třída *DetectorAbc* však předepisuje další překrytelné funkce ve tvaru `visit_node` kde *node* je typ uzlu, tedy například uzel s definicí kontraktu, definicí funkce, podmínkou a dalšími. Výhodou tohoto přístupu oproti nástroji Slither je, že detektor může reagovat jen na konkrétní uzly IR a vyhne se tedy procházení velké části uzlů, které by musel procházet, pokud by postupoval přes všechny kontrakty, jejich funkce, proměnné, a další.

**Obrázek 4.1** Ukázka detekce a poddetekce detektoru na re-entrancy nástrojem Woke v CLI

```
┌ Possible re-entrancy in `Reentrancy.func` ──────────────────────────┐
│   7                                                                  │
│   8     function func() public {                                     │
│   9         require(!state);                                         │
│ ❯ 10        msg.sender.call{value: 1}("");                           │
│  11         state = true;                                            │
│  12     }                                                            │
│  13                                                                  │
└ contracts/reentrancy.sol ────────────────────────────────────────────┘
└── ┌ Exploitable from `Reentrancy.func`, address is safe: False, state modified: MOD┐
    │   7                                                              │
    │   8     function func() public {                                 │
    │   9         require(!state);                                     │
    │ ❯ 10        msg.sender.call{value: 1}("");                       │
    │  11         state = true;                                        │
    │  12     }                                                        │
    │  13                                                              │
    └ contracts/reentrancy.sol ────────────────────────────────────────┘
```

#### 4.2.2.1 Vnitřní reprezentace

Důležitou součástí podpory pro vytváření statických detektorů nástroje Woke je jeho vlastní IR kódu Solidity. Ta začíná v balíčku balíčku `woke.ast`, ve kterém se nachází následující 3 moduly.

- `types` – reprezentující typy Solidity.
- `enums` – obsahující výčty sloužící k popisu vlastností reprezentovaných uzlů, například tedy výčet s viditelnostmi funkcí nebo výčet s možnými paměťovými lokacemi.
- `nodes` – sdružující typy reprezentovaných uzlů.

V balíčku `woke.ast.ir` je modul `abc` s obecnou třídou *IrAbc* sloužící jako rodičovská třída pro všechny uzly reprezentované IR. Ve stejném balíčku jsou pak další balíčky reprezentující jednotlivé uzly podle typu, tedy například `woke.ast.ir.statement` nebo `woke.ast.ir.declaration`. Některé reprezentované uzly pak mají v IR užitečné funkcionality, jako například dohledávání všech rodičovských kontraktů u definice kontraktu.

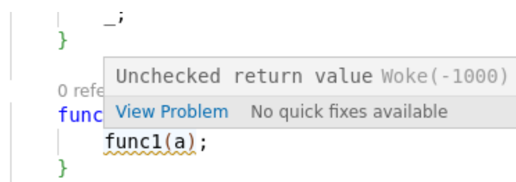
#### 4.2.2.2 Spouštění detektorů

Detektory mohou být spouštěny buď z CLI podobně jako fuzzer, tj. z balíčku `woke.cli`, ale modulem `detect` nebo přímo z IDE, kde je spouštění zajišťováno LSP serverem, tedy balíčkem `woke.lsp` a modulem `lsp_compiler`. Samotné detektory a metoda `detect`, která je spouští, jsou v balíčku `woke.analysis.detectors` v souboru `api.py`. Tato metoda poté pro vybrané uzly spouští překryté metody jednotlivých detektorů a sbírá jejich výstupy.

### 4.2.3 Provázání s IDE

Nástroj Woke má v sobě zabudovaný LSP server umožňující integraci jeho funkcionalit do IDE podporující LSP. To umožňuje přímo v IDE využívat funkcí Woke a tedy například spouštět detektory a označit uživateli detekci přímo v kódu, generovat CFG, graf dědičnosti a další. Pro integraci do IDE VS Code[10] existuje rozšíření Tools for Solidity[11] také od Ackee Blockchain, které LSP server nástroje Woke využívá. Toto rozšíření obohacuje VS Code o možnost hledání referencí, přechodu na definici a již zmíněné generování různých grafů.

**Obrázek 4.2** Ukázka detekce nezpracování návratové hodnoty nástroje Woke v IDE VS Code




[10] https://code.visualstudio.com/
[11] https://marketplace.visualstudio.com/items?itemName=AckeeBlockchain.tools-for-solidity

Kapitola 5

# Navrhovaná rozšíření a analýza požadavků

Po seznámení s nástrojem Woke a následné diskuzi s jeho vývojáři a vedoucím práce byla navržena rozšíření, která budou v této kapitole popsána a budou analyzovány požadavky na ně. Těmito navrhovanými rozšířeními je vylepšení fuzzeru nástroje Woke přidáním sběru pokrytí testovaného kódu a dále vytvoření několika nových statických detektorů.

Sběr pokrytí kódu fuzz testy je žádanou funkcionalitou proto, že v současné chvíli nedostávají uživatelé nástroje dostatečnou zpětnou vazbu o tom, do jakých částí kódu se jejich testy dostanou. Musí se tedy spolehnout buď na to, že správně nastavili všechny parametry, které jejich flow odesílají tak, aby se dostali do částí kódu, které chtějí testovat nebo si kód smart kontraktu musejí upravit tak, aby tuto zpětnou vazbu získali například pomocí výjimky. To však není vyhovující stav a sběr pokrytí a jeho následné promítnutí do Solidity kódu by ho mělo napravit.

Rozšíření o další statické detektory je dlouhodobým cílem nástroje Woke, protože se snaží nahradit nástroj Slither, který má detektorů zatím mnohem více. Konkrétně bylo navrženo rozšíření o 5 nových detektorů, z nichž se 2 snaží přímo detekovat zranitelnosti kontraktů, další 2 odhalují chyby, které na zranitelnosti mohou vést, a 1 je čistě informační s účelem zlepšit čistotu kódu.

## 5.1 Sběr dat o pokrytí kódu fuzz testy

Jedním z těchto vylepšení je sběr dat o pokrytí kódu fuzz testy, který by měl uživatelům používající nástroj Woke jako fuzzer usnadnit orientaci v tom, které části kódu jejich fuzz testy pokrývají. Této informace pak uživatel může využít pro úpravu fuzz testů tak, aby otestoval méně navštěvované části kódu vyžadující specifické vstupní parametry volaných funkcí nebo stav dat uložených v blockchainu. Sběr pokrytí kódu fuzz testy by tedy měl být navázaný na zdrojový kód v Solidity, na který se Woke zaměřuje. Nebude se tedy jednat jen o sběr na úrovni instrukcí, ale i o mapování prováděných instrukcí zpět do kódu v Solidity. Konkrétně bylo zvoleno pokrytí na úrovni větví kódu, které tak pomůže uživateli zjistit, které větve byly a nebyly testovány a také jak moc byly testovány. Toto pokrytí větví by mělo odpovídat větvím v kódu Solidity a ne větvení ve zkompilovaném kódu, které se v určitých případech může lišit. Data o pokrytí by navíc měla být zaznamenávána jak při normálním běhu nasazeného kontraktu, tak při jeho samotném nasazování. Cílem tedy bude zjistit počet spuštění následujících prvků:

**Volání funkcí, modifikátorů a konstruktorů** – pro funkce, modifikátory a konstruktory využívané v kontraktu, tedy i volné funkce a funkce jeho rodičovských kontraktů. Součástí by měl být jak počet volání celkově, tak počet volání jednotlivých větví kódu.

**Volání modifikátorů aplikovaných na funkci** – zde je myšlen počet spuštění jako přispění aplikovaného modifikátoru na funkci k celkovému počtu spuštění kódu modifikátoru. Jako příklad lze uvažovat situaci, kde budou spouštěny dvě funkce se stejným modifikátorem. První z funkcí bude spuštěna 2x a druhá funkce 3x a u kódu modifikátoru bude tedy 5 spuštění. Pro aplikaci modifikátoru na první funkci budou evidovány 2 spuštění a u aplikace modifikátoru na druhou funkci 3 spuštění. Funkce může mít i více aplikovaných modifikátorů, přičemž z těchto dat půjde zjistit, kolikrát byl který modifikátor funkce spuštěn, což umožní rozlišit případy, kdy například první modifikátor zamezil vykonávání funkce a tedy i vykonání kódu druhého modifikátoru, který bude mít menší počet spuštění než modifikátor první u dané funkce.

**Volání rodičovského konstruktoru jako modifikátoru** – jedná se o situaci, kde konstruktor volá konstruktor rodičovského kontraktu jako modifikátor. Princip a důvod k počítání počtu spuštění je stejný jako u modifikátorů aplikovaných na funkci.

Uživatel spouští fuzz testy přes CLI, kde by měl mít možnost dostávat základní informace o průběhu fuzz testů, konkrétně o počtu spuštění testovaných funkcí. Detailnější pokrytí, tedy pro jednotlivé větve kódu, počty volání funkcí, modifikátorů a konstruktorů a jejich aplikovaných modifikátorů a volání rodičovských konstruktorů bude exportováno do souboru s formátem vhodným pro použití již existujícím pluginem Tools for Solidity. Tento soubor tedy bude sloužit jako podklad pro zobrazení informací o pokrytí kódu formou podbarvení a uvedením nasbíraného počtu spuštění pokrytého kódu v IDE VS Code. Samotná úprava pluginu a následná vizualizace v IDE bude realizována vývojáři tohoto pluginu.

## 5.2 Vytvoření nových detektorů

Druhým vylepšením nástroje Woke je vytvoření nových statických detektorů. Důvodem je potřeba rozšířit počet detektorů Woke, jelikož oproti konkurenčnímu nástroji Slither, který má nahradit, má detektorů mnohem méně. Není však cílem pouze zkopírovat detektory nástroje Slither a konkrétní výběr nových detektorů byl proveden po diskuzi a na základě požadavků od vývojářů nástroje Woke tak, aby reagoval na skutečně se vyskytující chyby v kontraktech. Cílem tedy bude vytvořit těchto 5 nových detektorů:

**Nebezpečné použití `tx.origin`** – použití `tx.origin` není doporučováno a může u kontraktů vést k jejich zneužití za pomoci phishingu. I tak má ale několik validních použití, například k ověření, zda transakci odeslal uživatel nebo smart kontrakt porovnáním `tx.origin` s `msg.sender`, proto je požadavkem detekovat pouze nebezpečné použití.

**Kolize selektorů v proxy kontraktu a jeho implementačním kontraktu** – tento detektor bude detekovat kolize selektorů funkcí a stavových proměnných proxy kontraktu a jeho implementačního kontraktu. K detekci těchto kolizí bude potřeba detekovat proxy a jejich implementační kontrakty.

**Funkce nenastavující všechny nebo žádné návratové hodnoty** – to může mít za následek nespecifikované chování volající funkce, která dostane neočekávanou návratovou hodnotu. Tato detekce je sice přítomná v novějších verzích Solidity kompilátoru, není však úplná a v některých případech nedetekuje nenastavení návratové proměnné.

**Známou chybu kompilátoru při kopírování prázdného pole** – jedná se o chybu při kopírování prázdného pole do paměti *storage*. Díky nezměnitelnosti kontraktů toto může být problém pokud už je kontrakt nasazen a není možnost provést u něj aktualizaci.

**Nepoužité kontrakty a knihovny** – tento detektor je spíše informačního charakteru a měl by přispět k větší čistotě kódu smart kontraktu.

Detektory jsou spouštěny buďto automaticky z IDE nebo ručně z CLI. Výstupem detektoru je umístění v kódu a hláška, co bylo detekováno a případně další informace ve formě detekcí jednotlivých indicií, které vedly k finální detekci.

## 5.3 Funkční a obecné požadavky

Na základě požadované funkcionality byly vytvořeny následující funkční požadavky. Ty specifikují konkrétní očekávané schopnosti nově implementovaných vylepšení. Tyto požadavky jsou následující:

**F01** – uživatel spustí fuzz test a bude sbíráno pokrytí funkcí (počet volání funkce a jejich větví) a na ně aplikovaných modifikátorů (včetně volání konstruktoru jako modifikátoru) a to přes více procesů najednou.

**F02** – uživatel spustí detektory a uvidí případný výstup daných detektorů.

Pro navrhovaná vylepšení byly také specifikovány následující obecné požadavky, které by měla vylepšení splňovat. Jedná se o následující požadavky:

**NP01** – Sběr pokrytí musí být integrován přímo do nástroje Woke.

**NP02** – Detektory musí být integrovány přímo do nástroje Woke.

**NP03** – Vývoj bude probíhat ve veřejném repozitáři nástroje.

## 5.4 Případy užití

Na základě požadavků byly vypracovány případy užití pro obě navrhovaná rozšíření, které podrobně popisují interakci uživatelů s navrhovanými rozšířeními a výsledky této interakce. V nástroji Woke není rozlišováno mezi jednotlivými uživateli a v případech užití tedy vystupuje pouze jeden uživatel. Pokud u případu užití sběru pokrytí není specifikováno jinak, je myšlen výstup do výstupního souboru s nasbíraným pokrytím kódu testy. Těmito případy užití jsou:

**UC01 – Počet volání funkcí, modifikátorů a konstruktorů v CLI** – uživatel spustí fuzz test a v CLI uvidí základní přehled o testovaných funkcích, modifikátorech a konstruktorech a počtu jejich volání.

**UC02 – Počet volání funkcí, modifikátorů a konstruktorů** – uživatel spustí fuzz test a ve výstupním souboru s pokrytím uvidí jednotlivé funkce, modifikátory a konstruktory a jejich počet volání.

**UC03 – Počet volání větví kódu funkcí** – uživatel spustí fuzz test a ve výstupním souboru s pokrytím uvidí počet volání jednotlivých větví.

**UC04 – Počet volání aplikovaných modifikátorů funkce** – uživatel spustí fuzz test a ve výstupním souboru s pokrytím uvidí u každé funkce její modifikátory a počet volání každého z modifikátorů.

**UC05 – Počet volání rodičovských konstruktorů jako modifikátorů** – uživatel spustí fuzz test a ve výstupním souboru s pokrytím uvidí u každého konstruktoru jednotlivá volání rodičovského konstruktoru a jejich počet.

**UC06 – Detekce nebezpečného použití `tx.origin`** – uživatel spustí detektor a dostane informaci o nebezpečném použití `tx.origin` v kontraktech.

**UC07 – Detekce kolizí selektorů** – uživatel dostane informaci, zda pro detekované proxy kontrakty a k nim přidruženým implementačním kontraktům existuje kolize selektorů.

**UC08 – Detekce funkcí nevracejících návratové hodnoty** – uživatel spustí detektor a dostane informaci o funkcích nevracející hodnoty ve všech svých větvích.

**UC09 – Detekce známé chyby kompilátoru při kopírování prázdného pole** – uživatel spustí detektor a dostane informaci, zda může dojít v některé z funkcí kontraktů ke zkopírování prázdného pole.

**UC10 – Detekce nepoužitých kontraktů, knihoven a rozhraní** – uživatel spustí detektor a dostane informaci o nepoužitých kontraktech, knihovnách a rozhraních.

## Kapitola 6

# Návrh implementace

V předchozích kapitolách byla provedena rešerše relevantních technologií a analýza schopností stávajícího řešení a požadavků na navrhovaná rozšíření. V této kapitole bude navržena implementace rozšíření nástroje Woke tak, aby splňovala specifikované funkční i obecné požadavky a bude také navržena jejich integrace do něj.

## 6.1 Sběr pokrytí kódu fuzz testy

V rámci projektu, na kterém jsou prováděny fuzz testy, se zpravidla objevuje více kontraktů, každý se svými funkcemi a používající funkce z rodičovských kontraktů nebo z knihoven. Kód těchto funkcí je při kompilaci přibalen přímo ke kontraktu, který je používá. V mapování pak mají instrukce funkcí z jiných kontraktů a volných funkcí jiný index zdrojového souboru, pokud se kód nacházel v jiném souboru. Při sběru pokrytí se pak může stát, že více kontraktů bude mít stejný rodičovský kontrakt a bude využívat jeho funkci a je očekáváno, že pokrytí této funkce bude součtem pokrytí kontraktů, které ji používají.

Další situací, která může nastat, je externí volání funkce jiného kontraktu (tedy je volána funkce jiného nasazeného kontraktu). V tom případě je nutné znát zdrojový kód a mapování po kompilaci tohoto jiného kontraktu v Solidity, aby bylo možné sbírat pokrytí pro jeho zdrojový kód. Vzhledem k těmto požadavkům bude sběr pokrytí vyžadovat, aby byly používané kontrakty známé a zkompilovány v rámci fuzz testu. Stejně tak může nastat situace, kdy tyto kontrakty využívají stejné společné funkce nebo se dokonce jedná o stejný kontrakt nasazený dvakrát a pokrytí těchto funkcí tedy bude napočítáno z obou nasazených kontraktů. V případě stejného kontraktu nasazeného několikrát však není nutné tyto kontrakty z pohledu sběru pokrytí rozlišovat, protože budou mít stejný bytecode i mapování. Konkrétní pokrytí kontraktu tedy bude určeno kombinací zdrojového souboru a jména kontraktu v něm, které se nemůže opakovat.

Pro získání výsledného pokrytí tedy bude nutné sečíst pokrytí všech kontraktů. Vzhledem k tomu, že nástroj Woke dokáže pracovat s více procesy, bude součet prováděn i přes více procesů. Jednotlivé procesy pracují samostatně a zároveň provádějí stejný fuzz test, tedy testování mezi procesy se liší jen v náhodně generovaných hodnotách. Z tohoto důvodu bude přidána možnost sbírat pokrytí jen přes několik z těchto procesů, protože samotný sběr pokrytí bude fuzz testování zpomalovat. Pokrytí pouze části procesů pořád může sloužit jako ilustrace toho, co je v kontraktech testováno, i když toto pokrytí nemusí plně odpovídat reálně testovanému kódu z ostatních procesů, které mohly vygenerovat takové hodnoty vstupů, díky kterým se dostaly do jiných částí kódu.

Pro správný sběr pokrytí bude doporučeno, aby byla kompilace prováděna bez optimalizací, které mohou výrazně modifikovat výsledný kód kontraktu. Další limitací bude požadavek, aby

externí volání funkcí *pure* a *view* z EOA nebyly prováděny přes *eth_call*, ale vytvářely novou transakci. Uživatel se tedy bude tak bude moct rozhodnout, zda pro tyto funkce chce sbírat pokrytí za cenu jejich pomalejšího volání pomocí transakcí.

### 6.1.1 Rozdělení kódu na větve

Rozdělení kódu na větve je jeden ze základních požadavků plynoucích z analýzy požadavků, které umožní uživatelům rozpoznat, které části kódu byly spuštěny. K rozdělení kódu na větve bude využito IR nástroje Woke na úrovni Solidity a Yul. Solidity kód bude rozdělen podle následujících reprezentací v IR:

- `IfStatement` – příkaz pro podmínku.
- `ForStatement`, `WhileStatement` a `DoWhileStatement` – příkazy pro cykly.
- `ExpressionStatement` s výrazem typu `FunctionCall` – příkaz pro zavolání funkce, kterou by se mohlo vykonávání kódu ukončit.
- `PlaceholderStatement` – příkaz užívaný v modifikátorech pro označení místa, kde by se měl spustit kód funkce, na kterou je modifikátor aplikován.
- Ostatních příkazů, které v sobě obsahují `FunctionCall` – například volání funkce v případě přiřazení.

Příkladem budiž ukázkový kód funkce v Solidity s větvením z ukázky 6.1, který obsahuje jak podmínky, tak volání funkce. Ve funkci `func1` by přitom mohl být příkaz `revert`, který by zabránil dalšímu provádění těla původní funkce. Z toho důvodu jsou musí být funkce rozdělena i podle volání funkcí.

**Výpis 6.1** Ukázkový kód v Solidity s větvením

```
1    function func2(uint256 a) test1 test2 public returns (uint) {
2        uint x=func1(a);
3        if (a < 6)
4        {
5            x=func1(a);
6        }
7        else if (a > 5)
8            func1(a);
9        return x;
```

Kód v Solidity může obsahovat bloky s kódem Yul v `InlineAssembly`, které také obsahuje vlastní řídící struktury a může vracet hodnoty a ukončovat funkci. Kód Yul tedy bude také rozdělen podle následujících reprezentací v IR:

- `If` – příkaz pro podmínku.
- `ForLoop` – příkaz pro cykly.
- `Switch` – příkaz umožňující definovat kód pro různé hodnoty porovnávané proměnné.
- `Break`, `Continue` a `Leave` – příkazy pro zastavení cyklu, přeskočení cyklu a ukončení funkce.
- `ExpressionStatement` s výrazem typu `FunctionCall` – příkaz volání funkce, kterou by se mohlo vykonávání kódu ukončit.

Reprezentace samotné pak mohou obsahovat informace o případném kódu, který je vykonáván například u příkazů pro cykly. Pro detekci navštívení větve pak bude využito mapování mezi původním kódem kontraktu a kompilovaným kódem, kde pro každou větev bude nalezena první instrukce z dané větve a podle té bude určován počet volání dané větve. Implementace sběru těchto dat bude řešit část případů užití UC03.

### 6.1.2 Volání aplikovaných modifikátorů a konstruktorů rodičů

Sběr pokrytí těchto volání je komplikovaný v tom, že při kompilaci ze Solidity může být výsledný kompilovaný kód v kódu kontraktu hned několikrát, pokud má více funkcí aplikovaný stejný modifikátor. Kompilovaný kód modifikátoru je totiž nakopírován přímo před funkci, na kterou byl aplikován a to jen pokud nejsou zapnuty optimalizace. Pokud jsou optimalizace zapnuty, zkompilovaný kód modifikátorů může a nemusí být v kódu kontraktu několikrát. Obecně ale nelze z mapování instrukcí na kód Solidity jednoznačně říct, které instrukce patří ke které aplikaci modifikátoru na funkci. Samotné instrukce patřící aplikaci modifikátoru budou dohledány na základě volání z funkce, na které jsou aplikovány. Pokud nebude možné dohledat instrukce patřící aplikaci modifikátoru na funkci, bude uživatel na tuto skutečnost upozorněn a bude mu doporučeno provést kompilaci s vypnutými optimalizacemi. Sběr těchto dat bude řešit část případů užití UC04 a UC05.

### 6.1.3 Volání funkcí, modifikátorů a konstruktorů

Dalším požadavkem bylo zaznamenávání počtu volání funkcí, modifikátorů a konstruktorů. Počet volání bude určen jako maximum z počtu volání pro první větev kódu a počtu volání aplikovaných modifikátorů a rodičovských konstruktorů. Pro funkce bez modifikátorů bude tedy maximum počet volání první větve a pro funkce s modifikátory to bude počet volání prvního modifikátoru. Sběr těchto dat bude řešit část případů užití UC01 a UC02.

### 6.1.4 Mechanismus sběru pokrytí a dostupné API

Sběr pokrytí bude probíhat periodicky po několika provedených transakcích, aby bylo pokrytí aktualizováno v přiměřené míře. Mechanismus sběru bude takový, že bude ukládáno číslo posledního zpracovaného bloku s transakcemi a pokud budou existovat novější bloky, budou zpracovány a číslo posledního zpracovaného bloku bude aktualizováno.

K získání čísla posledního bloku bude využita metoda `eth_blockNumber`, která vrací číslo posledního bloku. Podle čísla bloku lze následně použít metodu `eth_getBlockByNumber`, která vrací nejen informace o bloku, ale v závislosti na vstupních parametrech může vracet i seznam všech provedených transakcí. Pro každou ze získaných transakcí pak bude volána metoda `debug_traceTransaction`, která sice není v oficiální JSON-RPC (JSON Remote Procedure Call) API specifikaci, ale je podporována všemi vývojovými blockchainy, které nástroj Woke používá. Sběr dat o pokrytí bude využívat zejména položku `pc`, tedy hodnotu PC, při které byla instrukce vykonávána. Dále v práci je hodnotou PC instrukce myšlena právě hodnota čítače instrukcí, při které bude zpracovávána daná instrukce. Tato metoda pak bude volána bez výpisu paměti *storage*, ale bude vracet aktuální obsah zásobníku a paměti *memory*. To bude nutné proto, že v případě instrukcí volání funkce jiného kontraktu je na zásobníku uložena adresa tohoto kontraktu. Tu bude potřeba získat, aby bylo možné začít sbírat pokrytí pro kontrakt, ve kterém se nachází volaná funkce. V případě nasazení kontraktu instrukcemi *CREATE* a *CREATE2* bude zjištěno, který kontrakt je nasazován z paměti *memory* a pro ten bude sbíráno pokrytí kódu. Ukázku výstupu této metody lze vidět ve výpisu 6.2.

**Výpis 6.2** Ukázka výstupu RPC `debug_traceTransaction`, převzato z [56]

```
1  > debug.traceTransaction("0x2059dd53ecac9827faad14d364f9e04b1d5fe5b506e3acc886eff7a6f88a696a")
2  {
3    gas: 85301,
4    returnValue: "",
5    structLogs: [{
6        depth: 1,
7        error: "",
8        gas: 162106,
9        gasCost: 3,
10       memory: null,
11       op: "PUSH1",
12       pc: 0,
13       stack: [],
14       storage: {}
15   },
16     /* snip */
17   {
18       depth: 1,
19       error: "",
20       gas: 100000,
21       gasCost: 0,
22       memory: ["0000000000000000000000000000000000000000000000000000000000000006",
         ↪  "0000000000000000000000000000000000000000000000000000000000000000",
         ↪  "0000000000000000000000000000000000000000000000000000000000000060"],
23       op: "STOP",
24       pc: 120,
25       stack: ["00000000000000000000000000000000000000000000000000000000d67cbec9"],
26       storage: {
27         0000000000000000000000000000000000000000000000000000000000000004:
           ↪  "8241fa522772837f0d05511f20caa6da1d5a3209000000000000000400000001",
28         0000000000000000000000000000000000000000000000000000000000000006:
           ↪  "0000000000000000000000000000000000000000000000000000000000000001",
29         f652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d3f:
           ↪  "00000000000000000000000002e816afc1b5c0f39852131959d946eb3b07b5ad"
30       }
31   }]
```

### 6.1.5 Nasazování kontraktu

Nasazování kontraktu bude rozpoznáno podle prázdného parametru `to` v transakci při zpracovávání transakcí. Na rozdíl od ostatních transakcí, kde je volána funkce již nasazeného kontraktu, se však v této transakci používá jiný bytecode. Kvůli tomuto případu budou zpracována a uchovávána data umožňující sběr pokrytí pro oba bytecode z artefaktů po kompilaci, tedy *nasazovaný bytecode*, který se bude využívat při nasazování nového kontraktu i *nasazený bytecode*, který bude využit v ostatních případech. Pro oba tyto bytecode tak bude zpracováno i jejich mapování instrukcí na zdrojový kód v Solidity, které je také dostupné jako artefakt kompilace. Implementace sběru těchto dat bude řešit případy užití UC01 a UC02, které vyžadují sběr volání kontruktorů.

### 6.1.6 Zpracování informací o zkompilovaném kódu

Z mapování zkompilovaného kódu pro bytecode bude určen zdrojový soubor, ze kterého pochází kompilované instrukce, díky čemuž následně bude přiřazeno pokrytí ke správnému souboru. Mapování dostupné jako artefakt kompilace však samo o sobě není dostačující, protože mapuje pouze instrukci na zdrojový kód. V samotném mapování instrukce na zdrojový kód není konkrétní PC pro tu kterou instrukci zřejmý, protože některé instrukce mohou mít argumenty a budou tedy větší než 1 byte. Při získávání informací o vykonaných instrukcích je totiž dostupná právě hodnota PC vykonané instrukce. Pro určení PC k instrukci v mapování bude využit artefakt, kterým je bytecode ve formě instrukcí. Zkompilovaný kód ve formě instrukcí lze vidět v ukázce 6.3. Samotná délka argumentu instrukce pak bude získána z názvu instrukce `PUSHX`. Ta má totiž argument, jehož velikost je zakódována v jeho názvu a který je vkládán na zásobník. V mapování ale není rozlišováno mezi různě dlouhými instrukcemi, tedy pokud by byla první instrukce `PUSH2` a další instrukce `ADD`, v mapování budou hned za sebou, tedy na indexech 0 a 1, ale PC budou mít 0 a 3. Na základě toho bude určen PC všech instrukcí, které mohou být vykonány.

**Výpis 6.3** Bytecode ve formě instrukcí

```
1   "opcodes": "PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1
    ↪  REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH2 0x7D JUMPI PUSH1 0x0 CALLDATALOAD PUSH1
    ↪  0xE0 SHR DUP1 PUSH4 0x9C4BE147 GT PUSH2 0x5B JUMPI DUP1 PUSH4 0x9C4BE147 EQ PUSH2 0xD6 JUMPI
    ↪  DUP1 PUSH4 0xE50B310B EQ PUSH2 0x106 JUMPI DUP1 PUSH4 0xEFB98BCF EQ PUSH2 0x124 JUMPI DUP1
    ↪  PUSH4 0xF4AB9ADF EQ PUSH2 0x142 JUMPI PUSH2 0x7D JUMP JUMPDEST DUP1 PUSH4 0x667867F EQ PUSH2
    ↪  0x82 JUMPI DUP1...
```

Mapování pro instrukci na zdrojový kód nemusí být vždy na konkrétní Solidity příkaz, ale může pokrývat větší i menší část kódu, tedy například celou funkci nebo jen přiřazení, které může být součástí větší větve. Při hledání mapování instrukce na větev bude tedy nalezen takový interval mapování instrukce na zdrojový kód, aby byl nalezen interval začínající co nejblíže začátku větve a pokrývající co nejmenší část kódu. Tedy pokud bude potřeba pokrýt větev s intervalem $(10, 30)$ a budou k dispozici instrukce pokrývající $(20, 30)$, $(10, 20)$ a $(10, 30)$, bude využito instrukce pokrývající interval $(10, 20)$, protože tato instrukce bude nejspecifičtější prvnímu příkazu větve.

### 6.1.7 Unikátní pojmenování kontraktů

V rámci sběru pokrytí se mohou objevit kontrakty, které jsou stejně pojmenované a liší se pouze v cestě ke zdrojovému souboru. Cesta ke zdrojovému souboru musí být tedy použita jako součást identifikátoru určující, ke kterému kontraktu a jeho funkcím se které pokrytí kódu vztahuje. Testing framework nástroje Woke takovýto identifikátor obsahuje a nazývá ho jako *FQN*, tedy *Fully Qualified Name*. Tento identifikátor se skládá z cesty a jména souboru a názvu kontraktu v něm a testing framework jej dokáže určit z bytecode používaného při nasazování kontraktu a nebo z adresy již nasazeného známého kontraktu.

### 6.1.8 Integrace do fuzzeru

Pro úspěšnou integraci do fuzzeru budou implementovány nové třídy umožňující sběr pokrytí a to na úrovni kontraktu. Samotné pokrytí bude počítáno pomocí mapování PC na počet spuštění. Ještě před vytvořením procesů provádějících testování bude zpracován kód smart kontraktů v

projektu, který je předmětem fuzz testu. Pro ten budou vytvořeny instance těchto nových tříd pro sběr pokrytí. Tyto instance budou předány do procesů, které jednou za několik transakcí provedou aktualizaci pokrytí, což bude znamenat příslušná volání na API vývojového blockchainu a jejich zpracování. Samotné pokrytí také bude periodicky odesíláno hlavnímu procesu, který jej posbírá od všech procesů, pokrytí sloučí a následně zobrazí uživateli a uloží jej do výstupních souborů.

### 6.1.9 Integrace do CLI a soubory s výstupem

Pro integraci do CLI bude využito již existující rozhraní, které nástroj Woke používá a budou přidány přepínače ovládající sběr pokrytí a výpis informací požadovaných UC01 na CLI. Budoucí integrace do IDE bude zajištěna výstupními soubory ve formátu umožňující jejich snadné použití pro vizualizaci pokrytí kódu. Bude se jednat o soubor s pokrytím pro funkce, jejich aplikované modifikátory a větve. U každého z těchto prvků bude označena část kódu, ke které se pojí, tedy u funkce její název, u aplikovaného modifikátoru název modifikátoru v definici funkce a u větve celá větev. Větev jako taková může pokrývat více řádků a v takovém případě nebudou rozsahy děleny na jednotlivé řádky, ale bude obarven celý tento rozsah od prvního příkazu až do posledního včetně mezer. Implementace této integrace bude důležitá pro splnění všech případů užití týkajících se sběru pokrytí.

## 6.2 Statické detektory

Všechny statické detektory dědí v nástroji Woke od třídy `DetectorAbc` s abstraktní metodou `report`, která slouží k vrácení výsledků detektoru. Mimo to pak třída `DetectorAbc` nabízí další metody k překrytí, které dědící detektory využívají ke specifikaci uzlů, na které chtějí reagovat a provádět nad nimi detekci. Volba překrytých metod tedy bude záviset na konkrétním detektoru a jeho potřebách.

### 6.2.1 Detekce nebezpečného použití tx.origin

Jak už bylo zmíněno, v `tx.origin` je uložena adresa, která původně započala řetězec volání, a není doporučeno ji používat pro autorizaci z bezpečnostních důvodů a do budoucna není jisté ani zachování její původní funkce. Vzhledem k tomu budou všechna použití `tx.origin` kromě specifických případů detekována. Tyto specifické případy bude možné v budoucnu přidávat, pokud by detektor vykazoval velké množství detekcí stejného typu, které nebudou vyhodnoceny jako nebezpečné. Z předběžné analýzy použití `tx.origin` vyšly najevo následující výjimky:

- Při určování, zda zdroj transakce neposílá mnoho transakcí v krátkém čase. V tomto případě je chtěné, aby byl opravdu porovnáván původní zdroj transakce, jinak by limitování počtu transakcí bylo možné obcházet například přeposíláním transakce přes speciální smart kontrakt k tomu naprogramovaný.
- Porovnání `tx.origin` s hodnotou `msg.sender`, kterým se zjistí, zda byla transakce poslána bez prostředníka.

Implementace tohoto detektoru bude řešit případ použití UC06.

### 6.2.2 Detekce kolizí selektorů u proxy kontraktů

Detekce kolizí selektorů nutně vyžaduje detekci proxy a implementačních kontraktů. Ta bude postavena na několika předpokladech vycházejících z běžného použití a implementace proxy kontraktů. Jedná se o následující indicie:

- Každý proxy kontrakt má speciální metodu `fallback`.
- Metoda `fallback` nebo metody, které volá, provádí delegované volání.
- Delegované volání využívá slotu s uloženou adresou implementačního kontraktu.

Implementační kontrakty nemají žádnou speciální metodu, podle které by je šlo poznat, využívají však stejného implementačního slotu jako proxy kontrakt. Detekce implementačního kontraktu tedy bude záviset na detekci proxy kontraktu, a to tak, že pokud bude detekován proxy kontrakt s implementačním slotem a jiný kontrakt, který tento implementační slot využívá (typicky v metodě sloužící k výměně implementačního kontraktu), bude tento kontrakt detekován jako implementační. Samotný implementační slot je běžně ve formě stavové proměnné a jedná se o hash spočtený funkcí Keccak-256.

Jakmile bude identifikován proxy kontrakt a jeho implementační kontrakt, budou porovnány selektory funkcí a stavových proměnných těchto kontraktů a kolize budou nahlášeny. Implementace tohoto detektoru bude řešit případ použití UC07.

### 6.2.3 Detekce funkcí nevracejících všechny návratové hodnoty

V Solidity existují dva způsoby vracení hodnot z funkce. Prvním z nich je použití příkazu `return` a druhým je pojmenování návratových proměnných a jejich následné nastavení. Tyto způsoby pak mohou být kombinovány, tedy některé větve kódu mohou vracet hodnoty pomocí příkazu `return` a jiné mohou nastavovat návratové hodnoty přímo. Detekce tedy bude fungovat tak, že se v poslední větvi ověří přítomnost příkazu `return` a pokud ve větvi nebude, nastane kontrola, zda byly ve všech předcházejících větvích nastaveny návratové hodnoty. Implementace tohoto detektoru bude řešit případ použití UC08.

### 6.2.4 Detekce chyby kompilátoru při kopírování prázdného pole

Detekce tohoto známého problému bude probíhat detekováním sledu akcí popsaných v bezpečnostním upozornění[35] od týmu vyvíjejícího jazyk Solidity pro kontrakty, které mají specifikovanou verzi menší než 0.7.14.

- Prázdná proměnná typu pole je zkopírována z **memory** nebo **calldata** do **storage**.
- Takto překopírované proměnné ve **storage** je zvýšena délka za pomoci vložení prázdného prvku metodou `push` bez argumentů.
- Bez dalšího zápisu je proměnná používána, což může mít za následek to, že přečtená hodnota nebude nulová.

Implementace tohoto detektoru bude řešit případ použití UC09.

### 6.2.5 Detekce nepoužitých kontraktů, knihoven a rozhraní

Tento detektor bude zjišťovat, zda jsou všechny kontrakty, knihovny a rozhraní, které Woke zkompiloval, někde použity. U kontraktů a rozhraní bude jako použití chápána situace, kde jakýkoliv jiný kontrakt dědí od kontraktu nebo rozhraní. V případě knihoven bude ověřeno, zda je v knihovně alespoň jedna funkce použita v ostatních kontraktech. Implementace tohoto detektoru bude řešit případ použití UC10.

### 6.2.6 Integrace detektorů

Integrace samotných detektorů do nástroje bude realizována samostatnými moduly pro každý detektor. Bude se jednat o vytvoření 5 modulů, tedy samostatných souborů, s detektory a jejich přidání do souboru `__init__.py` balíčku `woke.analysis.detectors`. Přidáním detektorů do tohoto souboru bude detektory možné použít jak ručně z CLI, tak automaticky z IDE.

Kapitola 7

# Implementace

V této kapitole je popsána realizovaná implementace podle návrhu ve formě pohledu na změny oproti původnímu stavu pro obě vylepšení, tedy sběr pokrytí kódu fuzz testy a vytvoření nových detektorů. Veškeré změny kódu, které byly provedeny, jsou dostupné i na přiloženém fyzickém médiu v adresáři `src`. Ten obsahuje adresář `impl` se soubory, které byly v rámci implementace vytvořeny nebo změněny. Tyto soubory jsou zasazeny ve stejné stromové struktuře, jakou má kód nástroje Woke. Dále obsahuje adresář `impl_blame`, který obsahuje ty samé soubory ale s anotacemi příkazem `git-blame`, které pro každý řádek vypisují commit a autora poslední změny. Jelikož na nástroji pracují i další vývojáři a vývoj probíhal během přechodu z frameworku Brownie na nový testovací framework, byl tento formát výstupu dodán pro jasnější znázornění autorova příspěvku do nástroje, hlavně pak v souborech, které nebyly nově přidány. Zároveň je dodána kopie celého repozitáře Woke v adresáři `exe/woke` s historií verzování nástrojem git. Posledním commitem učiněným v rámci této práce je `de1ce97490f096e502a51d4e89eda1442a030758`.

## 7.1 Sběr pokrytí kódu fuzz testy

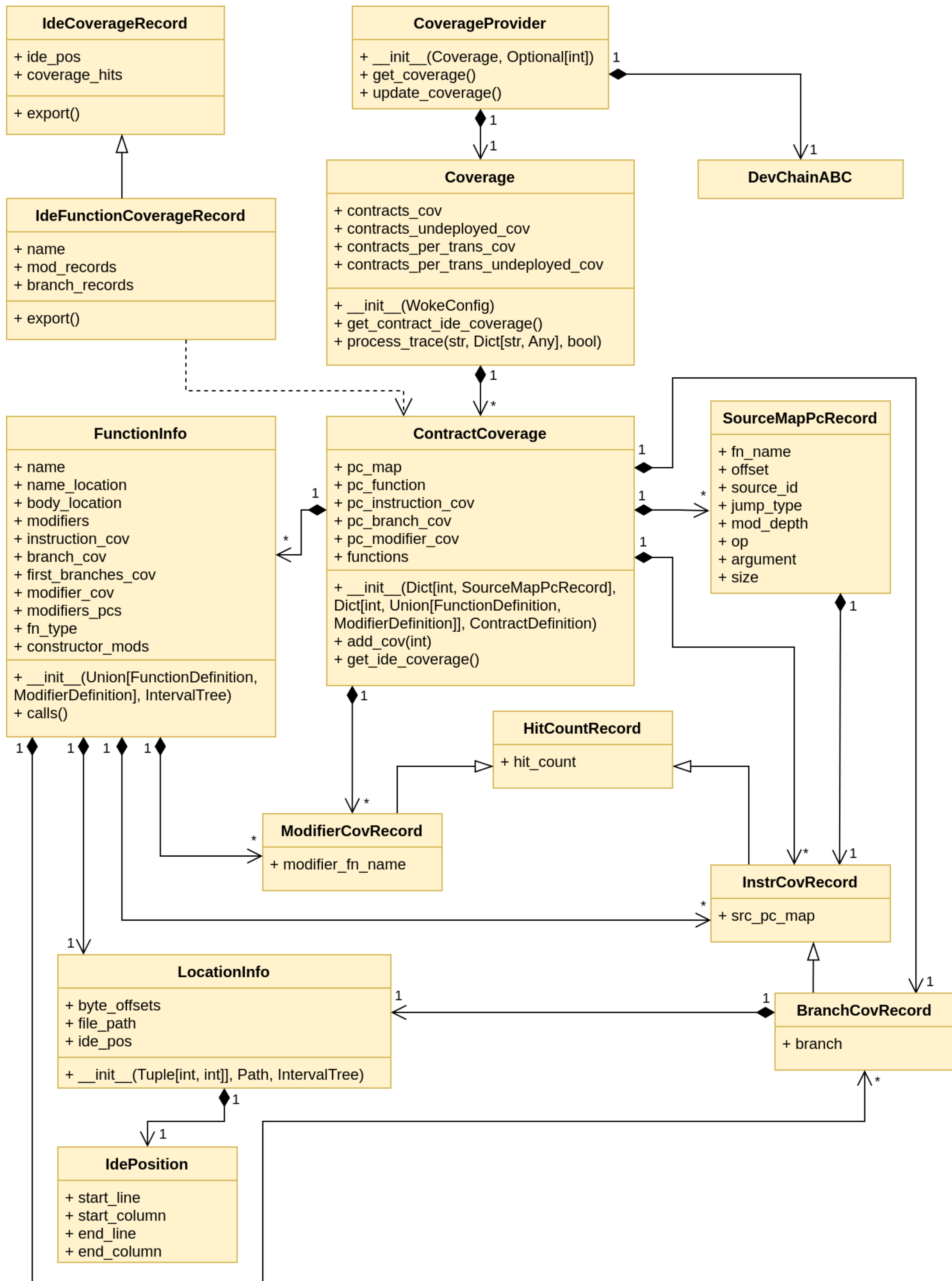
Sběr pokrytí kódu fuzz testy vyžadoval zásah zejména v části fuzzeru, tedy `woke.testing`, kde byl přidán nový modul `coverage` obsahující nové třídy a logiku pro sběr pokrytí. Také byly změněny moduly `fuzzer` a `campaign` ve stejném balíčku, do kterých byl sběr pokrytí integrován. Implementace sběru pokrytí kódu fuzz testy je funkční s novým testovacím frameworkem a využívá jeho funkcionalit. Pro správné fungování sběru pokrytí kódu je vhodné vypnout při kompilaci optimalizace, které mohou způsobovat nepřesné měření v závislosti na provedených optimalizacích. Také je nutné ve fuzz testech vynucovat volání *pure* a *view* funkcí jako transakcí, pokud má být v těchto funkcích prováděn sběr pokrytí.

### 7.1.1 Třídy pro sběr pokrytí

Implementované třídy a vztahy mezi nimi lze vidět na diagramu 7.1. Třída *ContractCoverage* slouží pro záznam pokrytí k jednomu konkrétnímu kontraktu, což ale znamená i ke všem funkcím, které dědí nebo které jsou volné. Tato třída obsahuje několik slovníků, kde je klíčem PC s následujícími hodnotami:

- `pc_map` – hodnotami jsou instance třídy *SourceMapPcRecord* obsahující data získaná z mapování instrukcí na zdrojový kód.
- `pc_instruction_cov` – hodnotami jsou instance třídy *InstrCovRecord* obsahující data o počtu spuštění dané instrukce a také odkaz na přidruženou instanci třídy*SourceMapPcRecord*.

**Obrázek 7.1** Diagram tříd pro sběr pokrytí



- `pc_branch_cov` – hodnotami jsou instance třídy *BranchCovRecord* obsahující počet spuštění dané větve a také odkaz na instanci třídy *SourceMapPcRecord* první instrukce větve, která je také použita jako klíč do tohoto slovníku, a také počet spuštění dané instrukce.

- `pc_modifier_cov` – hodnotami jsou instance třídy *ModifierCovRecord*, která obsahuje název modifikátoru a počet spuštění tohoto modifikátoru. Jedná se o modifikátory ve smyslu modifikátorů aplikovaných na funkci a tento slovník tedy pokrývá i volání rodičovského konstruktoru jako modifikátoru konstruktoru.

- `pc_function` – hodnotami jsou instance třídy *FunctionInfo*, která uchovává informace o jednotlivých funkcích a blíže je popsána dále.

Slovníky `pc_instruction_cov`, `pc_branch_cov` a `pc_modifier_cov` jsou využívány metodou `add_cov(pc)` přidávající pokrytí do záznamů v nich uložených. Dále je ve třídě *ContractCoverage* slovník `functions`, jehož klíčem je název funkce a hodnotou instance třídy *FunctionInfo*. V rámci tohoto slovníku a `pc_function` se jedná o stejné instance třídy *FunctionInfo*.

Třída *FunctionInfo* obsahuje informace o funkci, jako jsou jméno, pozice jména ve zdrojovém souboru pro zpracování pro IDE reprezentovanou třídou *LocationInfo* a aplikované modifikátory. Také má ale vlastní slovníky pro mapování PC na záznamy s počty spuštění stejně jako třídy *ContractCoverage*, ale obsahuje klíče a hodnoty pouze pro PC, které ji náleží. Každá funkce tedy zná své PC a také jejich počet spuštění a tedy pokrytí. Tato třída má metodu `calls`, která je používána pro zjištění počtu volání funkce, které je získáno z maxima pokrytí první větve a volání modifikátorů. Zmíněná třídy *LocationInfo* obsahuje informace jako cesta k souboru a pozice ve zdrojovém souboru. K ukládání pozic ve zdrojovém souboru pro zpracování IDE slouží třída *IdePosition* obsahující řádek a sloupec začátku a konce oblasti v IDE.

Instance *ContractCoverage* jsou vytvářeny při vytváření třídy *Coverage*, která obsahuje 4 slovníky, jejichž klíči je FQN kontraktu a hodnotou instance *ContractCoverage*. Tyto slovníky mají následující účel:

- `contracts_cov` – pro ukládání pokrytí kontraktů.

- `contracts_undeployed_cov` – pro ukládání pokrytí kontraktů při jejich nasazování.

- `contracts_per_trans_cov` – pro ukládání pokrytí kontraktů ale s PC započítanými pouze jednou za transakci.

- `contracts_undeployed_per_trans_cov` – pro ukládání pokrytí kontraktů při jejich nasazování, ale s PC započítanými pouze jednou za transakci.

Tato třída má veřejnou metodu `get_contract_ide_coverage` sloužící k získání dat o pokrytí ve formátu vhodném pro další zpracování IDE. Konkrétně tato metoda vrací slovník, kde je klíčem název souboru se zdrojovým kódem a hodnotou je další slovník, kde je klíčem pozice dané funkce ve zdrojovém kódu a hodnotou instance třídy *IdeFunctionCoverageRecord*. Ta sama o sobě obsahuje informace o počtu spuštění funkce a také obsahuje další slovníky s klíči *IdePosition* a hodnotami pokrytím pro aplikované modifikátory, resp. pro větve v podobě instancí třídy *IdeCoverageRecord* obsahující počet spuštění. Další její veřejnou metodou je `process_trace`, která se stará o zpracování informací o provedených instrukcích jednotlivých transakcí.

Třída *CoverageProvider* slouží ke spojení tříd *Coverage* a *DevChainABC* a stará se se svou metodou `update_coverage` o interakci s vývojovým blockchainem, tedy o procházení bloků a jejich transakcí. Samotné zpracování těchto informací ale nechává na instancích třídy `Coverage` a její metodě `process_trace`.

### 7.1.2 Sběr pokrytí a více procesů

Nástroj Woke dokáže fuzz test spustit na více procesech najednou, přičemž procesy samotné jsou vytvářeny a spuštěny ve funkci `fuzz` v souboru `woke/testing/fuzzer.py`. Ještě před spuštěním těchto procesů je vytvořena instance třídy *Coverage*, která zkompiluje smart kontrakty v daném projektu a následně vyextrahuje informace podstatné pro sběr pokrytí. Při spouštění dalších procesů se pak tato instance nakopíruje do dalších procesů, díky čemuž je nutné zpracovávat smart kontrakty v projektu pouze jednou. Až samotný proces vytváří instanci třídy *CoverageProvider*, protože až tehdy je spuštěný vývojový blockchain pro tento proces.

Posílání dat o pokrytí zpět hlavnímu procesu probíhá přes `multiprocessing.Queue`, díky které nejsou procesy blokovány. Hlavní proces po přijetí nových dat o pokrytí provede sloučení aktuálních dat pokrytí od všech procesů a následně aktualizuje výpis pro uživatele na CLI, pokud je takovýto výpis zapnut, a zapíše data o pokrytí do výstupních souborů pro IDE.

### 7.1.3 Využité artefakty a API

Při vytváření instancí tříd *ContractCoverage* je přímo využito 2 artefaktů kompilace. Jedná se o mapování instrukcí na kód v Solidity v kombinaci s instrukcemi ve formě operačních kódů, ze kterého je vytvářen slovník s klíčem PC a hodnotou jako záznamem z mapování. Vygenerovaný bytecode a jeho nasazování obstarává testovací framework. Metody API popsané v návrhu jsou využívány skrze testovací framework, který obstarává komunikaci s API vývojového blockchainu a zpracovává výstupy těchto metod.

### 7.1.4 Externí volání funkcí

V případě externího volání funkcí je nutné rozlišovat instance *ContractCoverage* tak, aby se pokrytí přičítalo správnému kontraktu. K tomuto účelu je v metodě `process_trace` třídy `Coverage` zjišťováno, zda nebyly vykonány instrukce `CALL`, `CALLCODE`, `DELEGEATECALL` nebo `STATICALL` a pokud ano, je ze zásobníku získána adresa volaného kontraktu. Z této adresy je za pomoci testing frameworku získáno FQN volaného kontraktu, které je následně použito jako index do slovníků v instanci třídy `Coverage`. Takovýchto volání může být hned několik, a proto jsou FQN kontraktů ukládány do zásobníku, odkud jsou vyndávány v případě vykonání instrukcí `RETURN`, `STOP` nebo `REVERT`.

### 7.1.5 Nasazování kontraktu

Kvůli podpoře sběru pokrytí při nasazování kontraktu je při vytváření instance `Coverage` u každého kontraktu zpracován zvlášť *nasazovaný bytecode* a *nasazený bytecode*. Nasazování kontraktů je rozpoznáno v metodě `update_coverage` třídy *CoverageProvider* podle prázdné cílové adresy. V případě detekce nasazování kontraktu je metoda `process_trace` třídy `Coverage` volána s argumentem `is_from_deployment` nastaveným na hodnotu `True`. Při detekci vykonávání instrukcí `CREATE` a `CREATE2` zjištěno FQN nasazovaného kontraktu s kterým je dále nakládáno stejně jako při volání externí funkce.

Kvůli chybě[1] v testovacím blockchainu Anvil, který je ve Woke výchozí volbou, není sběr pokrytí při nasazování kontraktu přímo ze Solidity funkční s tímto blockchainem. Anvil totiž od jisté verze nevrací v API metodě `debug_traceTransaction` instrukce vykonávané při nasazování kontraktu z jiného kontraktu, tedy pomocí instrukcí `CREATE` a `CREATE2`. Pro sběr pokrytí při nasazování kontraktu je tedy nutné využít alternativní testovací blockchain Ganache, což lze udělat přidáním parametru `--network ganache` při spouštění fuzz testů.

[1] `https://github.com/foundry-rs/foundry/issues/3819`

### 7.1.6 Modifikátory a konstruktory

Modifikátory a konstruktory samotné jsou brány jako funkce, tedy je pro ně sbíráno pokrytí jejich modifikátorů a případně volání rodičovského konstruktoru a také pokrytí větví. U modifikátorů může nastat problém, že instrukce mohou být zdvojené, pokud je modifikátor aplikován na více funkcí. To samo o sobě nevadí, protože tyto zdvojené instrukce mají rozdílné PC a pouze se pro stejné části kódu v Solidity sbírá pokrytí z více PC.

Pokrytí pro aplikované modifikátory na funkci však musí rozlišovat, ke které funkci patří. Implementace tohoto rozlišování využívá toho, že zavolání modifikátoru provádějí instrukce přiřazené k funkci, na které je modifikátor implementován. Je tedy detekována posloupnost instrukcí vedoucí k internímu volání modifikátoru, přičemž tyto instrukce jsou označeny za instrukce náležící aplikaci modifikátoru na danou funkci. V případě více modifikátorů na sebe volání modifikátorů navazuje, nicméně samotný způsob volání zůstává stejný, takže princip získání PC instrukcí více modifikátoru aplikovaných na jednu funkci je stejný jako u jednoho samostatného modifikátoru.

### 7.1.7 Integrace do CLI

Integrace do CLI vyžadovala změny v balíčku `woke.cli` v modulu `fuzz`. Při spouštění fuzzeru je nově možné specifikovat počet procesů, na kterých bude probíhat sběr pokrytí pomocí přepínače `-c`/`--cov-proc-count`, který má výchozí hodnotu nastavenou pro sběr pokrytí na 1 procesu. Také byl přidán přepínač pro výpis počtu volání jednotlivých metod `-v`/`--verbose-coverage`, které ve výchozím nastavení není zapnuto. Ukázku výstupu fuzz testu s tímto přepínačem lze vidět ve výpisu 7.1, kde `-n` značí počet procesů.

**Výpis 7.1** Ukázka výstupu nástroje Woke s přepínačem `-v` pro výpis počtu volání funkcí do CLI

```
1  $ woke fuzz tests/op.py -n 1 -c 1 -v
2  Found 'test_counter_fuzz' function in 'tests.test_parents_fuzz' file.
3
4
5
6  Fuzzing 'test_parents_fuzz' in 'tests.test_parents_fuzz'.
7  Using seed '26d500bf51f85154' for process #0
8    Finished, 0 processes remaining
9    Parent:function func1(uint a) public returns (uint): 150
10   Child:function func2(uint256 a) public returns (uint): 50
11   Parent:constructor(string memory a) public: 10
12   Child:constructor(string memory _test) public Parent(_test): 5
```

### 7.1.8 Výstup pro integraci do IDE

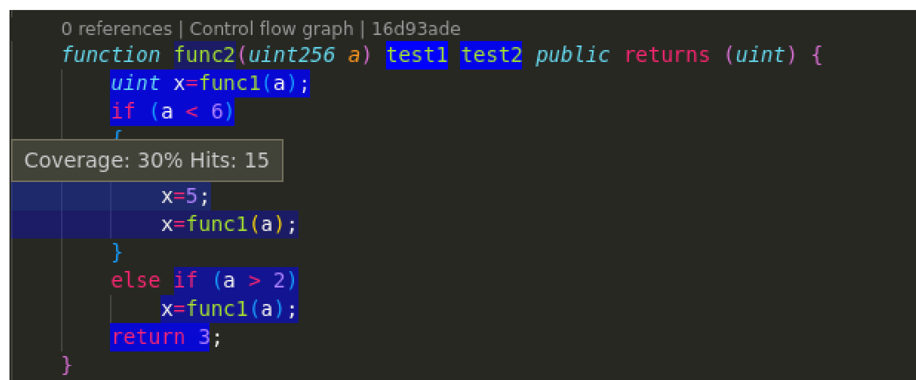
Budoucí integrace do IDE je zajištěna přes 2 výstupní soubory s pokrytím `woke-coverage.cov` a `woke-coverage-per-trans.cov`. U těchto souborů je rozlišováno mezi pokrytím, kde může být instrukce na PC započítána víckrát a pokrytím, kde je instrukce na PC započítávána pouze jednou v rámci transakce. Výstupní soubory jsou ve formátu JSON a jsou určeny ke zpracování rozšířením Tools for Solidity. Soubory obsahují pokrytí pro více souborů se zdrojovými kódy najednou a pro každý soubor mají záznamy pro jednotlivé funkce, jejich modifikátory a větve. Ukázku formátu tohoto souboru lze vidět ve výpisu 7.2. Záznamy pro „modRecords“ obsahují

stejná data jako pro „branchRecords“. Ukázku z budoucí integrace do IDE skrze plugin lze vidět na obrázku 7.2. Tato integrace zatím nebyla oficiálně přidána do rozšíření Tools for Solidity a jedná se pouze o pracovní verzi integrace.

**Výpis 7.2** Ukázka souboru s nasbíraným pokrytím určeného pro vizualizaci v IDE

```
{
    "path/to/contract/code.sol": [
        {
            "branchRecords": [
                {
                    "coverageHits": 10,
                    "endColumn": 23,
                    "endLine": 28,
                    "startColumn": 8,
                    "startLine": 28
                },
                ...
            ],
            "modRecords": [
                {
                    ...
                },
                ...
            ]
            "coverageHits": 10,
            "name": "Contract:function covered_func_name() public",
            "endColumn": 23,
            "endLine": 25,
            "startColumn": 13,
            "startLine": 25
        }
    ],
    ...
}
```

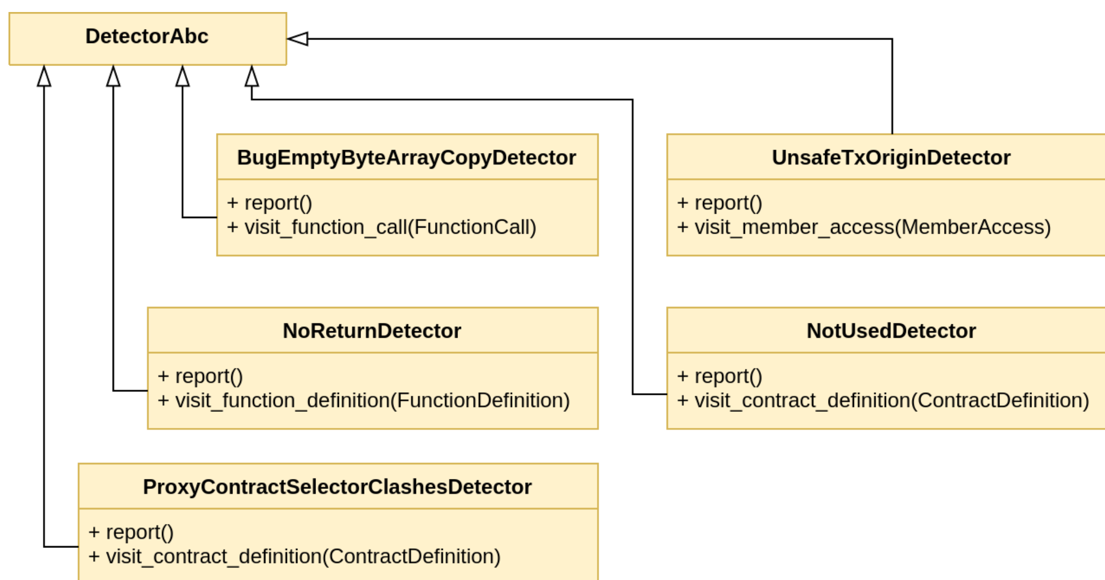
**Obrázek 7.2** Ukázka prototypu integrace sběru pokrytí do IDE



## 7.2 Statické detektory

Implementace statických detektorů spočívala ve vytvoření 5 modulů obsahující příslušné třídy detektorů. Ty dědí od třídy *DetectorAbc*, implementující metodu `report` a překrývající potřebné metody pro navštívení uzlu a konečně samotnou detekci. Třídy detektoru, které byly implementovány, lze vidět na diagramu 7.3. Integrace nových detektorů do nástroje Woke je provedena jejich přidáním do souboru `__init__.py` balíčku `woke.analysis.detectors`.

**Obrázek 7.3** Diagram tříd pro detektory




### 7.2.1 Nebezpečné použití tx.origin

Detektor nebezpečného použití `tx.origin` je implementován třídou `UnsafeTxOriginDetector` a k detekci využívá metody `visit_member_access`. Ta je určena pro implementaci detekce při navštívení uzlu reprezentovaného třídou `MemberAccess`, jejíž význam je přístup k členské proměnné, což je přesně to, co se děje při přístupu k `tx.origin`. Následně je tedy detekováno, že je skutečně přistoupeno k `tx.origin` a pokud ano, je detekováno nebezpečné použití tak, jak bylo vymezeno v návrhu, tedy se dvěma případy, kdy je použití bráno jako legitimní.

### 7.2.2 Kolize selektorů u proxy kontraktů

Detektor kolize selektorů funkcí proxy a implementačního kontraktu je implementován třídou `ProxyContractSelectorClashDetector` v modulu `proxy_contract_selector_clashes`. Detekci kolize selektorů předchází detekce implementačního a proxy kontraktu a detektor tedy překrývá metodu `visit_contract_definition`. Pracuje tedy nad celým kontraktem, resp. jeho IR třídou *ContractDefinition*. Proxy kontrakt je detekován podle toho, že implementuje speciální funkci `fallback` a že používá instrukci `DELEGATECALL`, kde je argumentem implementační slot. Implementační slot je rozpoznám podle toho, že se jedná o stavovou proměnnou s hodnotou, která je výstupem hashovací funkce Keccak-256. Následně jsou detekovány implementační kontrakty jako kontrakty využívající implementační slot v některé ze svých funkcí. Samotná detekce kolize pak spočívá v procházení dvojic proxy a smart kontraktů a porovnání selektorů všech jejich funkcí navzájem.

### 7.2.3 Funkce nevracející všechny návratové hodnoty

Detektor funkcí nevracejících všechny návratové hodnoty je implementován třídou `NoReturnDetector` v modulu `no_return_detector`. Překrývá metodu `visit_function_definition` a pracuje nad IR definice funkce, tedy třídou `FunctionDefinition`. Pro tyto funkce si detektor vytvoří CFG dané funkce (dostupné v IR) a následně prochází tento graf od koncového uzlu a zjišťuje, zda jsou ve všech možných cestách splněny podmínky toho, že je buď cesta ukončena příkazem **`return`**, který musí obsahovat všechny návratové hodnoty, nebo že jsou nastaveny všechny proměnné návratových hodnot. Pokud je objevena cesta, ve které nejsou nastaveny všechny tyto návratové hodnoty, detektor tento fakt nahlásí.

### 7.2.4 Chyba kompilátoru při kopírování prázdného pole

Detekce kontraktu, který je zkompilován starou verzí kompilátoru způsobující chybné chování při kopírování prázdného pole, je implementována třídou `BugEmptyByteArrayCopyDetector` v modulu `bug_empty_byte_array_copy` a využívá překrytou metodu `visit_function_call`. Pracuje tedy nad IR volání funkce reprezentovaného třídou `FunctionCall`, kterou používá k detekci funkce **`push`** nad polem a následně postupuje podle popsaného sledu akcí v návrhu detektoru.

### 7.2.5 Nepoužité kontrakty, knihovny a rozhraní

Detekce nepoužitých kontraktů, knihoven a rozhraní je implementována třídou `NotUsedDetector` v modulu `not_used_detector` a překrývá metodu `visit_contract_definition` a dále pracuje nad IR definicí kontraktu reprezentovanou třídou `ContractDefinition`. Tato třída reprezentuje jak kontrakty, tak knihovny a rozhraní. U kontraktů detektor zjišťuje, zda jsou abstraktní a pokud ano, tak zda je od těchto kontraktů děděno a to samé detektor provádí i u rozhraní. U knihoven zase prochází její metody a zjišťuje, zda jsou někde použité.

Kapitola 8

# Testování

Řešení popsané v předchozí kapitole je otestováno jak automatizovanými tak manuálními testy, jejichž konkrétní podoba je popsána v této kapitole. Při vývoji bylo také využíváno tzv. smoke testování pro rychlé ověření funkčnosti. Pro implementované celky byly vytvořeny unit testy ověřující jejich funkčnost i v případě budoucích změn. Po integraci do nástroje Woke byly provedeny manuální systémové testy pro ověření celkové funkčnosti implementovaných vylepšení.

## 8.1 Unit testy

Implementovaný kód byl pokryt unit testy jak pro část sběru pokrytí kódu testy tak pro nové detektory. Pro sběr pokrytí bylo vytvořeno 13 unit testů a pro nové detektory 5 unit testů. Detektory pro unit testy pokrývají jak případy, kdy by měla proběhnout detekce, tak případy, kdy by detekce proběhnout neměla.

Pokrytí kódu těmito unit testy lze vidět ve výpisu 8.1 vygenerovaného nástrojem pytest-cov[1]. Do pokrytí je zahrnut i soubor `woke/analysis/detectors/utils.py`, do kterého byla přidána funkce z detektoru nevrácení všech návratových hodnot. Soubor ale obsahuje i další funkce, což je důvod pro jeho menší pokrytí testy. Výstup nástroje pytest pro obě sady detektorů je vidět ve výpisu 8.2 a 8.3.

**Výpis 8.1** Pokrytí kódu unit testy

```
1  Name                                                           Stmts   Miss  Cover
2  ---------------------------------------------------------------------------------
3  woke/testing/coverage.py                                         739    105    86%
4  woke/analysis/detectors/bug_empty_byte_array_copy.py              64      5    92%
5  woke/analysis/detectors/no_return_detector.py                     60      2    97%
6  woke/analysis/detectors/not_used_detector.py                      21      0   100%
7  woke/analysis/detectors/proxy_contract_selector_clashes.py       287     53    82%
8  woke/analysis/detectors/unsafe_tx_origin.py                       76      5    93%
9  woke/analysis/detectors/utils.py                                 193     74    62%
```

[1] https://pypi.org/project/pytest-cov/

**Výpis 8.2** Unit testy pro sběr pokrytí kódu

```
$ pytest -v tests/test_coverage.py
===================================== test session starts =====================================
...
tests/test_coverage.py::TestContractCoverage::test_add_random_coverage PASSED             [  7%]
tests/test_coverage.py::TestContractCoverage::test_add_random_branch_coverage PASSED      [ 15%]
tests/test_coverage.py::TestContractCoverage::test_get_ide_branch_coverage PASSED         [ 23%]
tests/test_coverage.py::TestContractCoverage::test_get_ide_modifier_coverage PASSED       [ 30%]
tests/test_coverage.py::TestContractCoverage::test_get_ide_function_calls_coverage PASSED [ 38%]
tests/test_coverage.py::TestContractCoverage::test_parsing_modifiers PASSED               [ 46%]
tests/test_coverage.py::TestContractCoverage::test_parsing_branches PASSED                [ 53%]
tests/test_coverage.py::TestCoverage::test_get_contract_ide_coverage PASSED               [ 61%]
tests/test_coverage.py::TestCoverage::test_process_trace PASSED                           [ 69%]
tests/test_coverage.py::TestCoverage::test_parent_coverage PASSED                         [ 76%]
tests/test_coverage.py::TestCoverageProvider::test_basic_coverage PASSED                  [ 84%]
tests/test_coverage.py::TestCoverageProvider::test_calls_coverage PASSED                  [ 92%]
tests/test_coverage.py::TestCoverageProvider::test_constructor PASSED                     [100%]

===================================== 13 passed in 18.99s =====================================
...
```

**Výpis 8.3** Unit testy pro detektory

```
$ pytest -v tests/test_detectors.py
===================================== test session starts =====================================
...
tests/test_detectors.py::TestNoReturnDetector::test_sources PASSED                        [ 20%]
tests/test_detectors.py::TestBugEmptyByteArrayCopy::test_sources PASSED                   [ 40%]
tests/test_detectors.py::TestUnsafeTxOrigin::test_sources PASSED                          [ 60%]
tests/test_detectors.py::TestNotUsed::test_sources PASSED                                 [ 80%]
tests/test_detectors.py::TestProxyContractSelectorClashes::test_sources PASSED            [100%]

====================================== 5 passed in 6.07s ======================================
...
```

## 8.2 Systémové testování

Správnost implementace a její integrace do nástroje Woke byla ověřována manuálním systémovým testováním. Při tomto druhu testování je testován celý systém a je zjišťováno, zda jsou nově přidané funkcionality správně integrovány. Systémové testování vyžaduje nainstalovaný nástroj Woke, jehož instalace je popsána v příloze A.

Pro systémové testy sběru pokrytí kódu byly vytvořeny 4 fuzz testy ověřující požadované funkcionality a nachystán konfigurační soubor `woke.toml` nástroje Woke vypínající optimalizace. Tyto fuzz testy jsou umístěny v adresáři `exe/coverage_tests`, kde je připraven projekt se smart kontrakty v adresáři `contracts`. Jednotlivé testy jsou v adresáři `tests` a lze je spustit pomocí příkazu `woke fuzz tests/`*`zvoleny_test.py`* `-n 2 -c 2 -v`. Konkrétně se jedná o soubory:

- `test_calls.py` – pro ověření funkčnosti sběru pokrytí mezi voláními 2 kontraktů. Využívá

kontrakty v souborech call_contract_coverage.sol a call_contract_coverage2.sol.

- test_parents.py – pro ověření funkčnosti výpočtu pokrytí v případě že jeden kontrakt dědí od druhého. Využívá kontrakty v souboru parents_contract_coverage.sol.
- test_modifiers.py – pro ověření funkčnosti sběru pokrytí s modifikátory funkce. Využívá kontrakt v souboru modifiers_contract_coverage.sol.
- test_deploy.py – pro ověření funkčnosti sběru pokrytí při nasazování kontraktů. Využívá kontrakty v souboru deploy_contract_coverage.sol.

Jako příklad systémového testu bylo zvoleno testování sběru pokrytí aplikovaných modifikátorů. Spuštění a výstup fuzz testu test_modifiers.py sloužícího pro testování pokrytí aplikovaných modifikátorů lze vidět ve výpisu 8.4.

**Výpis 8.4** Spuštění testu sběru pokrytí aplikovaných modifikátorů

```
1  $ woke fuzz tests/test_modifiers.py -n 2 -c 2 -v
2  Found 'test_modifiers_fuzz' function in 'tests.test_modifiers' file.
3
4
5
6  Fuzzing 'test_modifiers_fuzz' in 'tests.test_modifiers'.
7  Using seed 'a8ac691cd58e26da' for process #0
8  Using seed 'cdfd91654c48ceb7' for process #1
9    Finished, 0 processes remaining
10   Modifiers:modifier test1: 260
11   Modifiers:function func1(uint256 a) public test1 returns (uint): 160
12   Modifiers:function func2(uint256 a) public test1 test2 returns (uint): 100
13   Modifiers:function set_n(uint a) public: 100
14   Modifiers:modifier test2: 90
15   Modifiers:constructor() public: 10
```

Při testování je voláno 5 sekvencí po 10 flow, tedy dohromady 50 zavolání flow. V testu je pouze jedno flow, a to flow_call, které volá funkci func2 a set_n s argumentem vnitřního čítače testu začínajícího na 1 a zvyšujícího se o 1 po každém flow. Tento vnitřní čítač se resetuje pro každou sekvenci. Funkce set_n nastavuje interní čítač kontraktu a funkce func2 volá 2x funkci func1. Modifikátory test1 a test2 funkce func2 zajišťují, že tělo funkce nebude prováděno pro argumenty s hodnotou 3 a 4. Funkce func2 má aplikovaný modifikátor test1 zajišťující, že tělo funkce nebude prováděno pro argument s hodnotou 3. Test je spuštěn na 2 procesech a z obou je sbíráno pokrytí, bude tedy spuštěno dohromady 100 flow. Funkce func2 je spuštěna 100x, ale díky modifikátorům je tělo funkce provedeno pouze 80x. Jelikož tato funkce volá volá funkci func1 2x, je její celkový počet volání 160. Celkový počet volání modifikátoru test1 je 260, protože je aplikován na funkci func1, která byla volána 160x a func2, která byla volána 100x. Ve výstupním souboru woke-coverage.cov lze pak ověřit, tělo funkce func2 bylo skutečně vykonáno 80x a tělo funkce func1 bylo vykonáno 160x. Modifikátor test1 neměl na počet vykonání těla funkce func1 vliv, protože stejný modifikátor je aplikován na funkci func2 a ověřuje vnitřní stav čítače kontraktu. Pokud tedy modifikátor nezamezil vykonávání těla funkce func2, nemohl zamezit ani vykonávání těla funkce func1.

Všechny fuzz testy nebudou podrobně popisovány, protože princip testování zůstává stejný. V případě spouštění fuzz testů je nutné se nejprve seznámit s použitými kontrakty a poté se samotným fuzz testem, jeho testovacími flow a poté určit správné počty volání, které by měly

nastat. Fuzz test `test_deploy.py` je nutné spouštět s přepínačem `--network ganache`, protože vývojový blockchain Anvil, který Woke jinak používá, nevrací kvůli chybě v něm v API metodě `debug_traceTransaction` instrukce vykonané při nasazování kontraktu z jiného kontraktu.

Pro otestování detektorů bylo vytvořeno 5 souborů s kontrakty obsahující chyby a zranitelnosti, které mají být detekovány. Tyto testy jsou umístěny v adresáři `exe/detectors_tests`, kde jsou připraveny kontrakty pojmenované podle detektorů a tedy i toho, co je v kontraktech detekováno. Jako příklad testu detektorů byl vybrán kontrakt `no_return.sol`, který obsahuje funkci nevracející všechny návratové hodnoty. Detekce nástrojem Woke byla spuštěna příkazem `woke detect no_return.sol` a detekované nevrácení všech hodnot lze vidět na obrázku 8.1

**Obrázek 8.1** Ukázka detekce funkce která nevrací všechny návratové hodnoty

```
┌ Not all paths have return or revert statement and the return values are not set either ┐
│   1 contract NoReturn {
│   2     bytes data;
│ ❱ 3     function func1(uint a) public returns (bytes memory x, bytes memory z) {
│   4         data.push("a");
│   5         bytes memory y;
│   6
└ no_return.sol ──────────────────────────────────────────────────────────────────────────┘
└──  ┌ Block where return is missing or return values are not set ──────────────────────┐
     │   3     function func1(uint a) public returns (bytes memory x, bytes memory z) {
     │   4         data.push("a");
     │   5         bytes memory y;
     │ ❱ 6         if (a == 0) {
     │   7             data.push("c");
     │   8             (y, x, y, x) = (data, data, data, data);
     │   9
     └ no_return.sol ────────────────────────────────────────────────────────────────────┘
```

Kapitola 9

# Závěr

Cílem této diplomové práce bylo rozšířit open source nástroj Woke sloužícího pro statickou a dynamickou analýzu smart kontraktů naprogramovaných v Solidity a nasazených v síti Ethereum. Konkrétně měl být rozšířen o sběr pokrytí kódu fuzz testy a nové statické detektory. Sběr pokrytí kódu fuzz testy by měl umožnit určení počtu volání jednotlivých funkcí, jejich modifikátorů a větví ve funkcích. To by mělo ulehčit uživatelům nástroje orientaci v testovaném kódu. Statické detektory by měly pokrýt hned několik oblastí detekce, a to od detekce obecných chyb ve smart kontraktech přes detekci zranitelností, až po detekce informačního charakteru vedoucí k větší čistotě kódu.

V rámci seznámení se s použitými technologiemi bylo nejdříve popsáno Ethereum, na němž jsou nasazovány smart kontrakty. Následně byl popsán jazyk Solidity, jakožto hlavní programovací jazyk používaný k vytváření těchto smart kontraktů. Dále byla popsána statická a dynamická analýza programů se zaměřením na části statické a dynamické analýzy, které se týkají nástroje Woke a nově implementovaných rozšíření v této práci. Ten byl spolu s nástrojem Slither analyzován a byly popsány jejich schopnosti. Nástroj Woke oproti nástroji Slither přidává zejména schopnost dynamického testování ve formě fuzzeru. Poté byla provedena analýza navrhovaných rozšíření a požadavků na ně, a to jak funkčních, tak obecných a byly připraveny případy užití těchto rozšíření.

Na základě analýzy byla navržena implementace jednotlivých rozšíření a jejich integrace do nástroje Woke. Tento návrh byl zrealizován a důležité části této realizace byly popsány. Implementovaná vylepšení byla otestována na úrovni unit testů, které budou správnost implementace automatizovaně ověřovat i v případě budoucích změn v implementaci. Také bylo provedeno manuální testování na úrovni systémových testů, kde byla ověřena funkčnost celého nástroje i s rozšířeními.

Výsledkem této práce je přidání funkčního sběru pokrytí kódu fuzz testy a statických detektorů splňujících specifikované funkční i obecné požadavky. Jako takové lze tedy cíle této práce považovat za splněné.

# Příloha A
# Instalace nástroje Woke

V této příloze bude popsána instalace nástroje Woke z přiloženého instalačního média. Předpokládá se, že instalace bude prováděna z adresáře `exe/woke`. Instalace vyžaduje verzi Python 3.7 a vyšší, předpokládá pro něj příkaz `python3` a bude využívat virtuální prostředí `venv`. Doporučenou platformou na instalaci nástroje Woke jsou Linuxové operační systémy, ale nástroj podporuje i macOS a Windows. Prerekvizitou fuzz testování nástrojem Woke je také přítomnost vývojových blockchainů. Hlavním podporovaným vývojovým blockchainem je Anvil, jehož instalační manuál lze najít na adrese `https://github.com/foundry-rs/foundry#installation`. Pro sběr pokrytí při nasazování kontraktu z jiného kontraktu je nutné nainstalovat vývojový blockchain Ganache, jehož instalační manuál je na adrese `https://trufflesuite.com/docs/ganache/quickstart/`. Po nainstalování vývojových blockchainů by měly fungovat příkazy `anvil` a `ganache`.

Následující příkazy vytvoří nové virtuální prostředí, provedou jeho aktivaci a nainstalují nástroj Woke z lokálního adresáře:

```
1  $ python3 -m venv venv
2  $ source venv/bin/activate
3  $ pip install -e ".[tests,dev]"
```

Díky těmto příkazům bude nástroj Woke a jeho závislosti nainstalovány lokálně ve virtuálním prostředí. Následně je možné ověřit instalaci pomocí následujícího příkazu a jeho výstupu:

```
1  $ woke --version
2  1.3.1
```

Pro spuštění detektorů na Solidity soubor pak lze využít příkaz `woke detect soubor.sol` a pro spuštění fuzz testů se 2 procesy se sběrem pokrytí na obou z nich lze použít příkaz `woke fuzz -n 2 -c 2 -v tests/test_fuzz.py`. Nápovědu pro příkazy lze vypsat pomocí příkazu `woke fuzz --help` a `woke detect --help`. Pro spuštění nástroje Woke s fuzz testy ze systémového testování stačí přejít se zapnutým virtuálním prostředím do adresáře s nimi a spustit dané příkazy. V případě změny kódu testovaných kontraktů ve fuzz testech pro sběr pokrytí je nutné nejprve spustit příkaz `woke init pytypes` v adresáři projektu a případně adekvátně upravit fuzz test. Virtuální prostředí lze vypnout příkazem `deactivate`.

# Bibliografie


1. WOOD, Gavin et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2014, roč. 151, č. 2014, s. 1–32.
2. CHAINALYSIS TEAM. *DeFi Hacks Are Steling More Crypto Than Ever Before* [online]. [cit. 2022-11-27]. Dostupné z: `https://blog.chainalysis.com/reports/2022-defi-hacks/`.
3. YANG, Rebecca; WAKEFIELD, Ron; LYU, Sainan; JAYASURIYA, Sajani; HAN, Fengling; YI, Xun; YANG, Xuechao; AMARASINGHE, Gayashan; CHEN, Shiping. Public and private blockchain in construction business process and information integration. *Automation in construction*. 2020, roč. 118, s. 103276.
4. ETHEREUM FOUNDATION. *Ethereum Virtual Machine (EVM)* [online]. [cit. 2022-11-28]. Dostupné z: `https://ethereum.org/en/developers/docs/evm/`.
5. ETHEREUM FOUNDATION. *Ethereum Accounts* [online]. [cit. 2022-12-17]. Dostupné z: `https://ethereum.org/en/developers/docs/accounts/`.
6. ETHEREUM FOUNDATION. *Consensus mechanisms* [online]. [cit. 2022-11-25]. Dostupné z: `https://ethereum.org/en/developers/docs/consensus-mechanisms/`.
7. ETHEREUM FOUNDATION. *Proof-of-stake (PoS)* [online]. [cit. 2022-12-06]. Dostupné z: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`.
8. HILDENBRANDT, Everett; SAXENA, Manasvi; RODRIGUES, Nishant; ZHU, Xiaoran; DAIAN, Philip; GUTH, Dwight; MOORE, Brandon; PARK, Daejun; ZHANG, Yi; STEFANESCU, Andrei et al. Kevm: A complete formal semantics of the ethereum virtual machine. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, s. 204–217.
9. SAINI, Vaibhav. *Getting Deep Into EVM: How Ethereum Works Backstage* [online]. [cit. 2022-11-25]. Dostupné z: `https://medium.com/swlh/getting-deep-into-evm-how-ethereum-works-backstage-ab6ad9c0d0bf`.
10. SZABO, Nick. *Smart Contracts* [online]. [cit. 2022-12-10]. Dostupné z: `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`.
11. WANG, Shuai; YUAN, Yong; WANG, Xiao; LI, Juanjuan; QIN, Rui; WANG, Fei-Yue. An Overview of Smart Contract: Architecture, Applications, and Future Trends. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018, s. 108–113. Dostupné z DOI: `10.1109/IVS.2018.8500488`.

12. ETHEREUM FOUNDATION. *Smart contract languages* [online]. [cit. 2022-11-27]. Dostupné z: https://ethereum.org/en/developers/docs/smart-contracts/languages/.
13. ATZEI, Nicola; BARTOLETTI, Massimo; CIMOLI, Tiziana. A survey of attacks on ethereum smart contracts (sok). In: *International conference on principles of security and trust*. Springer, 2017, s. 164–186.
14. WÖHRER, Maximilian; ZDUN, Uwe. Design patterns for smart contracts in the ethereum ecosystem. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, s. 1513–1520.
15. ETHEREUM FOUNDATION. *Upgrading smart contracts* [online]. [cit. 2022-11-29]. Dostupné z: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/.
16. SOLIDITY TEAM. *Introduction to Smart Contracts* [online]. [cit. 2022-11-29]. Dostupné z: https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html.
17. ETHEREUM FOUNDATION. *EIP-1967: Proxy Storage Slots* [online]. [cit. 2022-12-17]. Dostupné z: https://eips.ethereum.org/EIPS/eip-1967.
18. OPENZEPPELIN. *Proxy Patterns* [online]. [cit. 2022-11-29]. Dostupné z: https://blog.openzeppelin.com/proxy-patterns/.
19. OPENZEPPELIN. *Proxy Upgrade Pattern* [online]. [cit. 2022-12-19]. Dostupné z: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.
20. SOLIDITY TEAM. *Units and Globally Available Variables* [online]. [cit. 2022-12-03]. Dostupné z: https://docs.soliditylang.org/en/v0.8.17/units-and-global-variables.html.
21. HETMAN, Adrian. *Unboxing tx.origin. Rune Token case* [online]. [cit. 2022-12-03]. Dostupné z: https://www.adrianhetman.com/unboxing-tx-origin/.
22. VICTOR, Friedhelm; LÜDERS, Bianca Katharina. Measuring ethereum-based erc20 token networks. In: *International Conference on Financial Cryptography and Data Security*. Springer, 2019, s. 113–129.
23. CONSENSYS. *tx.origin - Ethereum Security Contract Best Practices* [online]. [cit. 2022-12-03]. Dostupné z: https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/tx-origin/.
24. BUTERIN, Vitalik. *cibtract desugb - How do I make my DAPP "Serenity-Proof?"* [online]. [cit. 2022-12-03]. Dostupné z: https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof/200.
25. ETHEREUM FOUNDATION. *Ethereum Accounts* [online]. [cit. 2022-12-17]. Dostupné z: https://ethereum.org/en/developers/docs/development-networks/.
26. SOLIDITY TEAM. *Solidity* [online]. [cit. 2022-12-04]. Dostupné z: https://docs.soliditylang.org/en/v0.8.17/.
27. GRAMLICH, Benjamin. Smart Contract Languages: A Thorough Comparison. *ResearchGate Preprint*. 2020.
28. SOLIDITY TEAM. *Layout of a Solidity Source File* [online]. [cit. 2022-12-04]. Dostupné z: https://docs.soliditylang.org/en/v0.8.17/layout-of-source-files.html.
29. SOLIDITY TEAM. *Types* [online]. [cit. 2022-12-14]. Dostupné z: https://docs.soliditylang.org/en/v0.8.17/types.html.

30. SOLIDITY TEAM. *Expressions and Control Structures* [online]. [cit. 2022-12-10]. Dostupné z: `https://docs.soliditylang.org/en/v0.8.17/control-structures.html`.
31. SOLIDITY TEAM. *Contracts* [online]. [cit. 2022-12-10]. Dostupné z: `https://docs.soliditylang.org/en/v0.8.17/contracts.html`.
32. SOLIDITY TEAM. *Inline Assembly* [online]. [cit. 2022-12-10]. Dostupné z: `https://docs.soliditylang.org/en/v0.8.16/assembly.html`.
33. SOLIDITY TEAM. *Using the Compiler* [online]. [cit. 2022-12-10]. Dostupné z: `https://docs.soliditylang.org/en/v0.8.16/using-the-compiler.html`.
34. SOLIDITY TEAM. *Source Mappings* [online]. [cit. 2022-11-28]. Dostupné z: `https://docs.soliditylang.org/en/latest/internals/source_mappings.html`.
35. SOLIDITY TEAM. *Solidity Empty Byte Array Copy Bug* [online]. [cit. 2022-11-29]. Dostupné z: `https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/`.
36. AGGARWAL, Ashish; JALOTE, Pankaj. Integrating static and dynamic analysis for detecting vulnerabilities. In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. IEEE, 2006, sv. 1, s. 343–350.
37. GOSEVA-POPSTOJANOVA, Katerina; PERHINSCHI, Andrei. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*. 2015, roč. 68, s. 18–33.
38. CHESS, Brian; MCGRAW, Gary. Static analysis for security. *IEEE security & privacy*. 2004, roč. 2, č. 6, s. 76–79.
39. POEPLAU, Sebastian; FRANCILLON, Aurélien. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, s. 163–176.
40. TOAL, Ray. *Intermediate Representations* [online]. [cit. 2022-12-13]. Dostupné z: `https://cs.lmu.edu/~ray/notes/ir/`.
41. GRUNE, Dick; VAN REEUWIJK, Kees; BAL, Henri E; JACOBS, Ceriel JH; LANGENDOEN, Koen. *Modern compiler design*. Springer Science & Business Media, 2012.
42. PRÄHOFER, Herbert; ANGERER, Florian; RAMLER, Rudolf; GRILLENBERGER, Friedrich. Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application. *IEEE Transactions on Industrial Informatics*. 2016, roč. 13, č. 1, s. 37–47.
43. OH, Nahmsuk; SHIRVANI, Philip P; MCCLUSKEY, Edward J. Control-flow checking by software signatures. *IEEE transactions on Reliability*. 2002, roč. 51, č. 1, s. 111–122.
44. LI, Jun; ZHAO, Bodong; ZHANG, Chao. Fuzzing: a survey. *Cybersecurity*. 2018, roč. 1, č. 1, s. 1–13.
45. IVANKOVIĆ, Marko; PETROVIĆ, Goran; JUST, René; FRASER, Gordon. Code coverage at Google. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, s. 955–963.
46. JORGENSEN, Paul C. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.
47. HEBERT, Fred. *Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do*. Pragmatic Bookshelf, 2019.
48. CHEN, Chen; CUI, Baojiang; MA, Jinxin; WU, Runpu; GUO, Jianchao; LIU, Wenqian. A systematic review of fuzzing techniques. *Computers & Security*. 2018, roč. 75, s. 118–137.

49. TAKANEN, Ari; DEMOTT, Jared D; MILLER, Charles; KETTUNEN, Atte. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.
50. MANÈS, Valentin JM; HAN, HyungSeok; HAN, Choongwoo; CHA, Sang Kil; EGELE, Manuel; SCHWARTZ, Edward J; WOO, Maverick. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*. 2019, roč. 47, č. 11, s. 2312–2331.
51. CRYTIC. *crytic/slither: Static Analyzer for Solidity* [online]. [cit. 2022-12-09]. Dostupné z: `https://github.com/crytic/slither`.
52. FEIST, Josselin; GRIECO, Gustavo; GROCE, Alex. 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, s. 8–15.
53. CRYTIC. *Adding a new detector* [online]. [cit. 2022-12-09]. Dostupné z: `https://github.com/crytic/slither/wiki/Adding-a-new-detector`.
54. CRYTIC. *Python API - crytic/slither Wiki* [online]. [cit. 2022-12-09]. Dostupné z: `https://github.com/crytic/slither/wiki/Python-API`.
55. CRYTIC. *Printer documentation - crytic/slither Wiki* [online]. [cit. 2022-12-09]. Dostupné z: `https://github.com/crytic/slither/wiki/Printer-documentation`.
56. THE GO-ETHEREUM AUTHORS. *debug Namespace* [online]. [cit. 2022-12-25]. Dostupné z: `https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug`.

# Obsah přiloženého média

```
readme.txt ........................ stručný popis obsahu média
exe ................................ adresář se spustitelnou formou implementace
  woke ............................. repozitář nástroje Woke
  coverage_tests ................... systémové testy pokrytí
  detectors_tests .................. systémové testy detektorů
src
  impl_blame ....................... zdrojové kódy implementace s příkazem git blame
  impl ............................. zdrojové kódy implementace
  thesis ........................... zdrojová forma práce ve formátu LaTeX
text ............................... text práce
  thesis.pdf ....................... text práce ve formátu PDF
```