UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: Informatyka

Piotr Jasina
nr albumu: 279183

Identyfikacja inteligentnych kontraktów w sieci Ethereum

Ethereum smart contracts identification

Praca licencjacka
napisana w Zakładzie Cyberbezpieczeństwa
pod kierunkiem dr. Damiana Rusinka

Lublin rok 2019

Spis treści

Wstęp

...

Rozdział 1

Ethereum

Ethereum jest to otwarta platforma oparta o technologie blockchain, która umożliwia użytkownikom tworzenie i uruchamianie zdecentralizowanych aplikacji. Programy tworzone i uruchamiane na blockchain nazywane są też inteligentnymi kontraktami.

Do uruchamiania aplikacji została stworzona maszyna wirtualna o nazwie **Ethereum Virtual Machine**, która może wykonywać kod inteligentnego kontraktu o dowolnej złożoność algorytmicznej. Kod aplikacji stworzonej na EVM jest przechowywany na blockchainie utrzymywanym przez jego użytkowników. Programiści mogą tworzyć aplikacje na EVM za pomocą przyjaznych języków programowania wzorowanych na Pythonie czy JavaScript [1].

Na potrzeby platformy Ethereum stworzono dedykowany język **Solidity**, który został stworzony z myślą o tworzeniu inteligentnych kontraktów.

1.1 Historia

Początki Ethereum zostały opisane przez programistę Vitalika Buterina w 2013 roku, kiedy jeszcze Ethereum nie nazywało się Ethereum. Programista Vi-

talik w październiku 2013 roku pracował z zespołem Mastercoin. Zaproponował im stworzenie bardziej uogólnionego protokołu, który wspierałby rożne rodzaje umów bez dodawania kolejnych funkcjonalności. Mastercoin było pod wrażeniem jego pomysłu, natomiast nie byli zainteresowania zmianami w tym kierunku. Vitalik czuł, że jego koncepcja jest słuszna i postanowił iść w tym kierunku. Około drugiego grudnia Vitalik uświadomił sobie, że inteligentne kontrakty, mogą być w pełni uogólnione, a do opisywania ich warunków można zastosować język skryptowy [2].

W grudniu 2013 do zespołu Valika dołączył między innymi Gavin Wood oraz programista klienta w języku Go Jeffrey Wilcke. Latem 2014 roku była już pierwsza stabilna wersja protokołu Ethereum oraz pół formalna specyfikacja w postaci żółtego papieru stworzonego przez Gavina [2].

Na początku lipca 2014 Ethereum rozdysponowało początkowy przydział krypto waluty Ether będącej częścią platformy Ethereum. Rozdysponowana wartość wynosiła około 18 milionów dolarów w zamian za ponad 50 milionów eterów. Wyniki sprzedażny zostały początkowo wykorzystane na opłacenie prac programistów oraz na finansowanie ciągłego rozwoju Ethereum [3]. Po wyprzedaży eteru rozwojem Ethereum zajmowała się organizacja non-profit o nazwie ETH DEV, której dyrektorami stali się: Vitalik Buterin, Gavin Wood oraz Jeffrey Wilcke.

W listopadzie 2014 roku ETH DEV zorganizowało w Berlinie wydarzenie **DEVCON-0**, które przyciągnęło programistów z całego świata interesujących się Ethereum [4].

Na początku 2015 roku odbyły się audyty bezpieczeństwa przed uruchomieniem, zorganizowane przez między innymi Jutta Steinera. Audyty dotyczyły przede wszystkim implementacji w Go i C++. Przeprowadzony został też mniejszy audyt dotyczący implementacji Vilika nazwanej **pyethereum**. Kontrola bezpieczeństwa wprowadziła do protokołu kilka małych zmian. Jedną zmianą było wprowadzenie

funkcji haszującej **SHA3** dla klucza i adresu drzewa Trie, która miała zapobiec atakowi DOS [5].

Siec **Ethereum** została uruchomiona 30 lipca 2015 roku. Był to moment, w którym użytkownicy przystąpili do sieci **Ethereum**, aby uzyskać eter z bloków górniczych. Natomiast programiści zaczęli pisać inteligentne umowy oraz zdecentralizowane aplikacje gotowe do wdrożenia w sieci **Ethereum**. Była to wersja testowa, ale jak się okazało, była ona bardziej udana, niż ktokolwiek by się tego spodziewał [8].

Idąc za ciosem, zorganizowano druga konferencje dla programistów nazwaną **DEVCON-1**, odbyła się ona w Londynie na początku listopada 2015 roku. Konferencja trwała pięć dni, a przedstawiano na niej ponad 100 prezentacji, paneli dyskusyjnych oraz krótkich rozmów. W konferencji wzięło udział ponad 400 uczestników, była to mieszanka przedsiębiorców, myślicieli, programistów oraz przedstawicieli biznesowych. W konferencji brały udział duże firmy jak IBM czy Microsoft, co wyraźnie wskazywało na duże zainteresowanie ta technologia [7].

1.2 Do czego służą Inteligentne kontrakty?

TODO

1.3 Ethereum Virtual Machine

EVM jest to środowisko uruchomieniowe dla inteligentnych kontraktów opartych o Ethereum. Początkowo wirtualna maszyna została opisana w żółtym dokumencie opracowanym przez dr. Gavina Wooda. Maszyna wirtualna jest całkowicie odizolowana od reszty głównej sieci blockchain, co pomaga w zapewnienia bezpieczeństwa wykonywania niezaufanego kodu przez komputery z całego świata.

Każdy węzeł w sieci Ethereum uruchamia u siebie własną implementację EVM oraz jest w stanie wykonywać na niej te same instrukcje co pozostałe węzły.

1.3.1 Kody operacji

Inteligentne kontrakty napisane w takich językach jak Solidity nie mogą być bezpośrednio wykonane na EVM. W celu wykonania kontraktu należy jego kod skompilować do nisko poziomowych instrukcji.

EVM wykorzystuje zbiór instrukcji do wykonywania określonych zadań. Operacje te umożliwiają stworzenie programu zupełnego w sensie Turinga. Operacje wykonywane na EVM można podzielić na siedem kategorii:

1. **Operacje wykorzystujące stos** (POP, PUSH, DUP, SWAP)

2. **Operacje udostępniające działania arytmetyczne** (ADD, SUB, GT, LT, AND, OR)

3. **Operacje środowiskowe** (CALLER, CALLVALUE, NUMBER)

4. **Operacje modyfikujące pamięć ulotna - memory** (MLOAD, MSTORE, MSTORE8, MSIZE) - jest to przestrzen w ktorej przechowywane sa tymczasowe dane takie jak argumenty funkcji, czy zmienne lokalne. Dane nie sa przechowywane na blockchanie

5. **Operacje modyfikujące pamięć nieulotna - storage** (SLOAD, SSTORE) - jest to miejsce w którym przechowywane są dane przechowywane na blockchainie. Każdy kontrakt posiada swój oddzielny obszar na blockchain.

6. **Operacje skoków oraz licznika programu** (JUMP, JUMPI, PC, JUMPDEST)

7. **Operacje zatrzymujące** (STOP, RETURN, REVERT, INVALID, SELFDESTRUCT)

Powyżej przedstawiono tylko przykłady operacji z danej kategorii, natomiast pozostałe operacje zostały przedstawione w dokumencie dr. Gavina Wooda [6].

1.3.2 Kod bajtowy

W celu efektywnego przechowywania operacji są one kodowane do kodu bajtowego. Każda operacja ma przydzielony jeden bajt. Przykładowo operacja PUSH1 wrzuca na stos jeden bajt, jest ona reprezentowana przez wartość 0x60. Dla kodu bajtowego 0x6080604001, pierwszy bajt to operacja PUSH1. Zgodnie ze specyfikacją opisaną przed Gavina operacja PUSH1 odczytuje kolejny bajt z kodu bajtowego i wrzuca go na stos, w tym przypadku na stos zostanie wrzucona wartość 0x80. Następną operacją jest ponownie PUSH1(0x60), tylko tym razem na stos została wrzucona wartość 0x40 [6].

Po wykonaniu dwóch pierwszych operacji na stosie znajdują się dwie wartości: 0x80 oraz 0x40. Kolejnym bajtem jest 0x01, który oznacza operacje ADD. Operacja ADD pobiera ze stosu dwie wartości, wykonuje operacje dodawania, po czym wynik wrzuca z powrotem na stos. W rezultacie górze stosu znajduje się wartość 0xC0. Podczas wykonywania kodu bajtowego jest on dzielony na pojedyncze bajty. Każdy bajt są to dwa znaki reprezentujące liczbę w systemie szesnastkowym [6].

1.3.3 Analiza kodu bajtowego

XXXXXXX Duza czesc inteligetnych kontraktow udostepnionych w sieci Ethereum jest przechowywana bez kodu zrodlowego. Ze wzgledow bezpieczenstwa istnieje potrzeba analizy tych kontraktow, w celu wykrycia potencjalnych podatnosci

na ataki lub w celu zrozumienia samego dzialania kontraktow. Do analizwy kodu bajtowego umieszczanego w sieciu Ethereum W sieci ethereum istnieje duza liczba kontraktow ref:xxxliczbakontraktow. Ze wzgledu na potrzebe analizy bezpieczenstwa udostepnianych kontraktow powstal szereg narzedzi umozliwiajacych analize kodu bajtowego. Jednymi z takich narzedzi jest binary ninja ktory pomaga podczas inzynieri wstecznej kontraktu. Innym narzedziem jest bytecode.dictianory - jest to baza danych przechowujaca sygnatury funkcji napisane w jezyku solidity, wraz z ich reprezentacja bajtową, ktora umozliwia wyszukanie sygnatur funkcji dla przekazanego kodu bajtowego.

Rozdział 2

Solidity

2.1 Sygnatura funkcji

TODO

2.2 Selektor funkcji

TODO

2.3 Generowanie akcesorów podczas kompilacji

TODO

Rozdział 3

Projekt Aplikacji

Celem mojej pracy licencjackiej było stworzenie aplikacji internetowej umożliwiającej identyfikację inteligentnych kontraktów w sieci **Ethereum**. Dzięki aplikacji użytkownik po wprowadzeniu na stronie kodu bajtowego kontraktu jest w stanie otrzymać najbardziej prawdopodobną implementacje kontraktu napisana w języku **Solidity**.

Aplikacja została stworzona przy wykorzystaniu frameworka **Spring Boot**, modułu **Spring Data MongoDB** oraz **Spring MVC**. Natomiast w celu przechowywania danych wykorzystano nierelacyjną bazę danych **MongoDB**.

W tym rozdziale znajduje się opis funkcjonalności, architektury, implementacji oraz wykorzystanych technologi.

3.1 Opis funkcjonalności

Na stronie głównej aplikacji znajduje się opis wraz z aktualna liczba kodów źródłowych znajdujących się w bazie danych oraz podstawowe definicje związane z aplikacją. Cała aplikacji udostępnia trzy główne funkcjonalności: identyfikację inteligentnych kontraktu, wprowadzanie plików źródłowych kontraktów do apli-

kacji oraz interfejs programistyczny aplikacji. Wszystkie funkcjonalności zostały opisane poniżej.

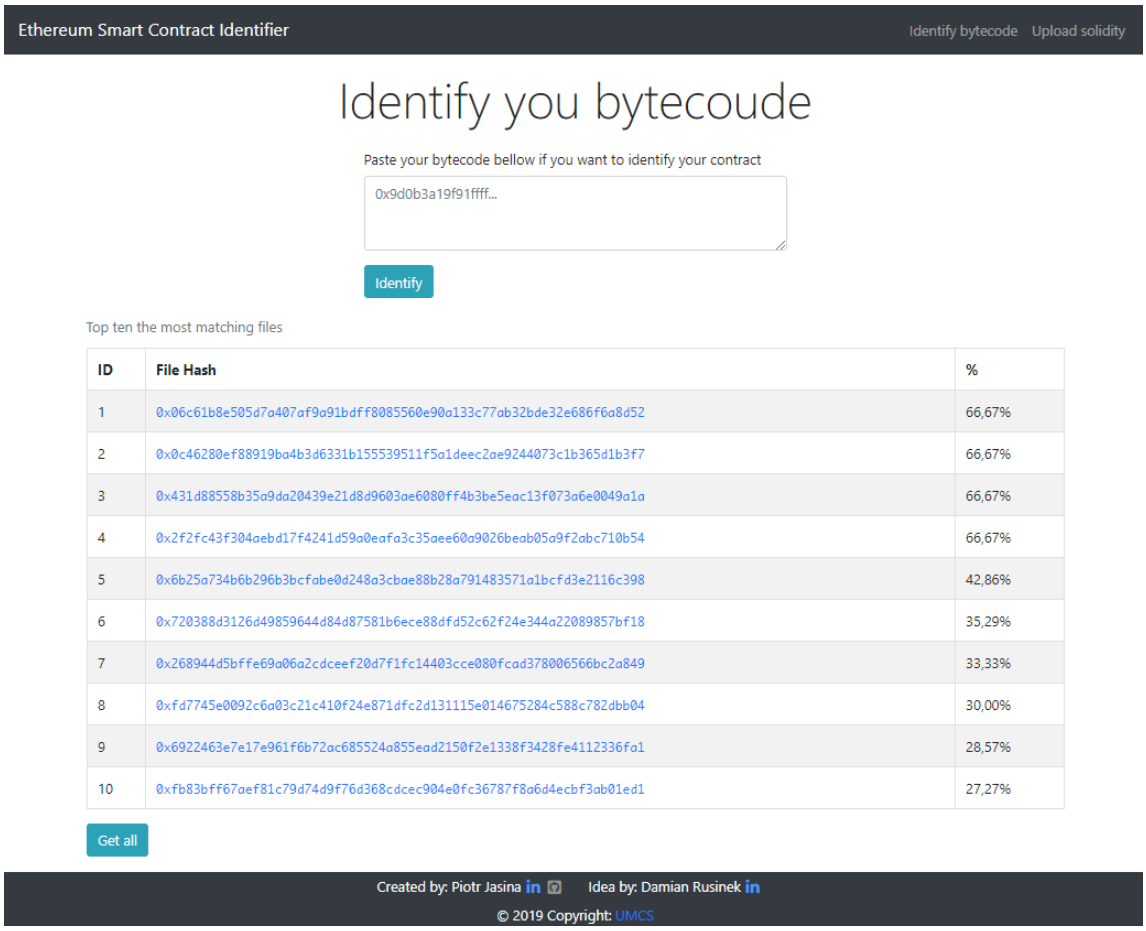
3.1.1 Identyfikacja inteligentnych kontraktów

Pierwszą opcją dostępną w aplikacji jest identyfikacja inteligentnych kontraktów. Zarówno na stronie głównej, jak i podstronie znajduje się pole, w którym można wprowadzić kod bajtowy. Po wprowadzeniu danych użytkownik zatwierdza je, w obu przypadkach klikając przycisk **Identify**. Przycisk **Identify bytecode**, służy do przejścia na podstronę związaną z identyfikacją kontraktu.

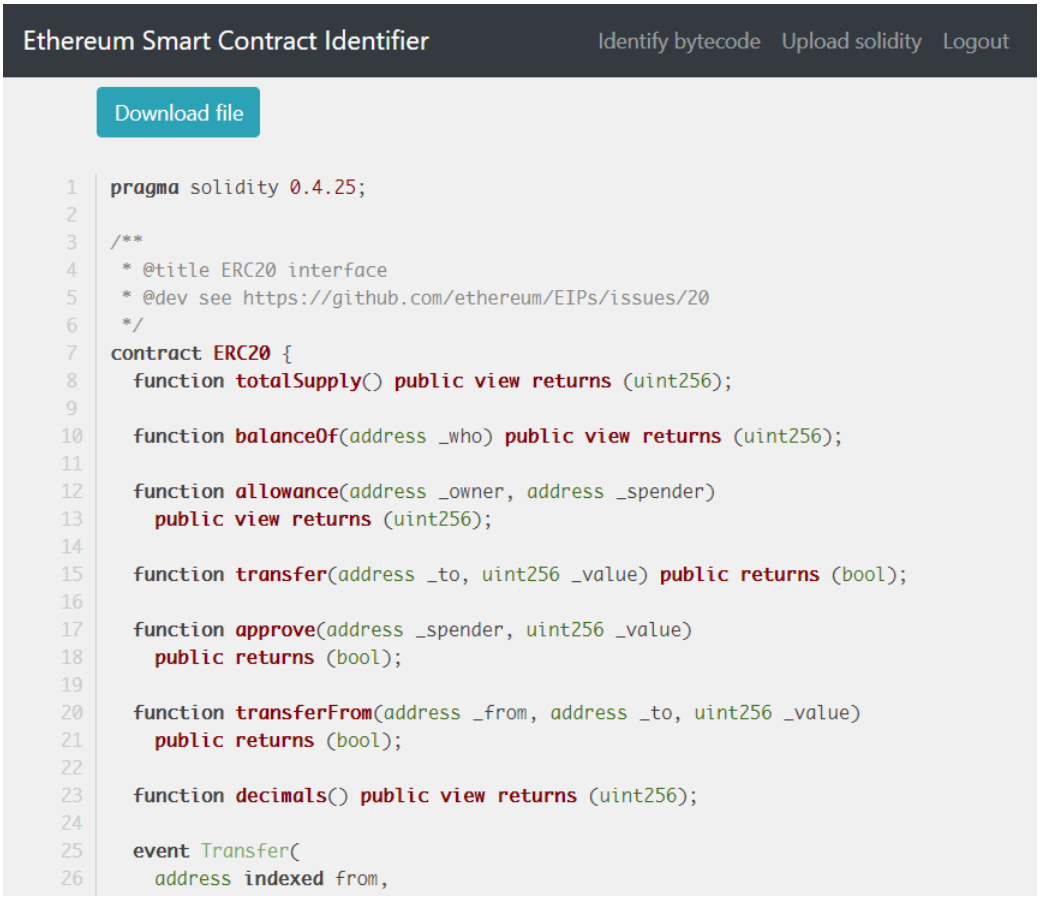
Po wprowadzeniu danych i zatwierdzeniu ich przyciskiem **Identify**, aplikacja rozpoczyna proces analizy wprowadzonego kodu bajtowego oraz wyszukiwane są najbardziej prawdopodobne implementacje kontraktu, posortowane malejąco według współczynnika dopasowania. Na rysunku 3.1 został przedstawiony przykładowy wynik identyfikacji.

Domyślnie jest wyświetlanych dziesięć najbardziej prawdopodobnych implementacji, po naciśnięciu przycisku **Get all**, znajdującego się pod listą kontraktów, zostaną wyświetlone wszystkie dopasowania.

Po naciśnięciu w jedną z wyświetlanych implementacji użytkownik zostanie przeniesiony na podstronę umożliwiającą podgląd implementacji. Na rysunku 3.2 znajduje się przykład przeglądania kodu źródłowego na stronie. Rozwiązanie z numerowaniem linii zostało zaimplementowane w taki sposób, aby podczas kopiowania kodu źródłowego ze strony, nie były kopiowane z nim liczby identyfikujące konkretną linię w kodzie. Istnieje też możliwość pobrania kodu źródłowego ze strony z rozszerzeniem **.sol**



Rysunek 3.1: Wynik identyfikacji inteligentnego kontraktu



Rysunek 3.2: Podgląd implementacji

3.1.2 Dodanie kodu źródłowego kontraktu do aplikacji

Strona umożliwia dodanie własnego kodu źródłowego kontraktu napisanego w języku **Solidity**. W tym celu należy zalogować się za pomocą panelu logowania, który zostaje wyświetlony automatycznie przy próbie korzystania z autoryzowanych funkcjonalności aplikacji. Po kliknięciu w **Identify Solidity** oraz zalogowaniu się na stronie, pojawiają się dwie możliwości wprowadzania kodów źródłowych.

Pierwszym sposobem jest przesłanie do aplikacji pliku zawierającego implementacje kontraktu. W tym celu użytkownik powinien nacisnąć przycisk **Browse** i wybrać konkretny plik, a następnie zatwierdzić go przyciskiem **Upload** widocznym na rysunku 3.3.

Innym sposobem na przesłania kodu źródłowego do aplikacji jest wklejenie kodu źródłowego bezpośrednio do formularza znajdującego się po prawej części strony internetowej.

Po prawidłowym dodaniu kodu źródłowego do aplikacji użytkownik powinien zobaczyć podobny rezultat do tego na rysunku 3.3. W momencie dodania nowej implementacji, na stronie pojawia się hasz dodanego pliku oraz lista sygnatur funkcji wraz z ich selektorami. Po naciśnięciu na wyświetlany na rysunku 3.3 hasz pliku, użytkownikowi wyświetli się przesłany kod źródłowy.

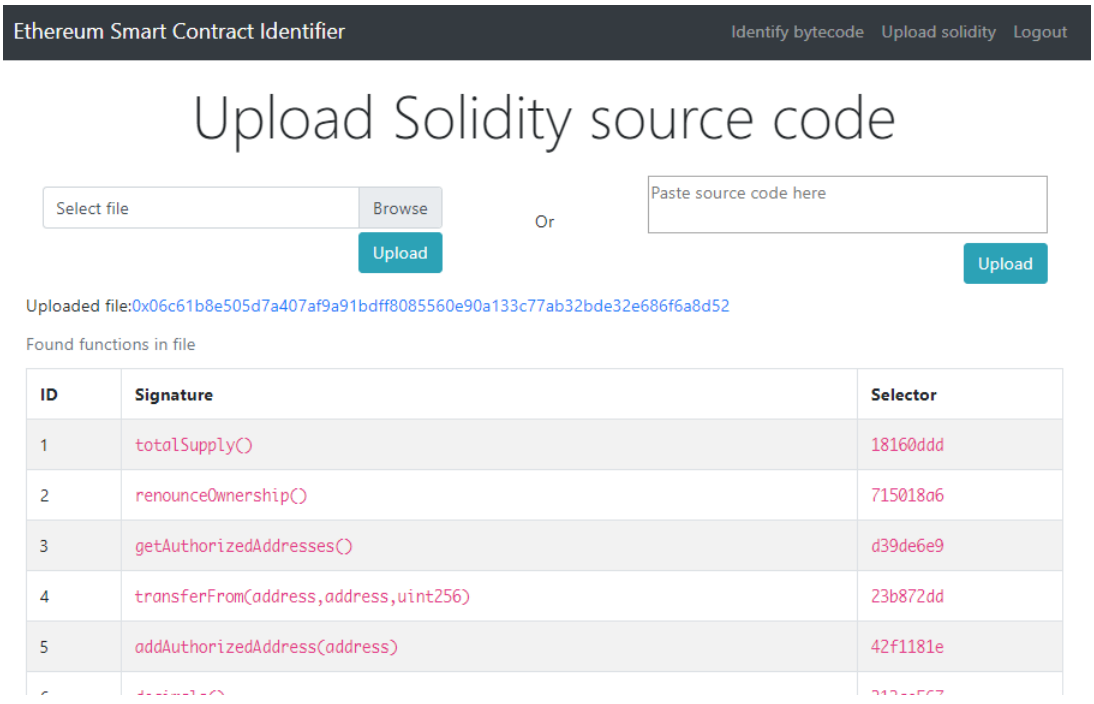
3.1.3 Interfejs programistyczny aplikacji

Trzecią funkcjonalnością aplikacji jest interfejs programistyczny. Dzięki niemu można wykorzystać mechanizmy zaimplementowane w aplikacji w innej aplikacji. Przykładowym zastosowaniem API jest utworzenie skryptu umożliwiającego zautomatyzowane wysyłanie kodów źródłowych do aplikacji, bez konieczności korzystania z interfejsu graficznego aplikacji [13].

Użytkownik za pomocą API ma możliwość pobrania informacji o kodzie źródłowym, identyfikacji kontraktu oraz przesłania nowego kontraktu do aplikacji.

Pobieranie informacji o kodzie źródłowym z API

Podczas pobierania informacji o kodzie źródłowym, użytkownik musi posiadać identyfikator pliku, który chce pobrać. Żądanie pobierające plik z API można zobaczyć na listingu 3.1. W odpowiedzi użytkownik dostaje zwykły tekst zawierający implementację kontraktu oraz status HTTP 200, 404 lub 500. Status 200 oznacza, że wszystko poszło pomyślnie. W sytuacji, gdy użytkownik otrzyma status 404, oznacza to, że nie udało się znaleźć implementacji o podanym haszu. Odpowiedz zawierająca status 500 oznacza, że wystąpił błąd na serwerze i nie udało się zwrócić kodu źródłowego [14].



Rysunek 3.3: Rezultat przesłania inteligentnego kontraktu do aplikacji

Listing 3.1: Żądanie wysyłane w celu pobrania kodu źródłowego

```
GET /api/sourceCode/0
    x06c61b8e505d7a407af9a91bdff8085560e90a133c77ab32bde32e686f6a8d52
    .sol HTTP/1.1
Host: localhost:8080
Accept: text/plain;charset=UTF-8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
    /537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
```

Identyfikacja inteligentnego kontraktu za pomocą API

W celu identyfikacji kontraktu należy wysłać żądanie pod adres /api/bytecode. W ciele żądania jest wymagane od użytkownika podanie dwóch atrybutów o nazwach: **bytecode** oraz **allFiles**. Atrybuty przesyłane do API powinny być zakodowane według schematu **nazwa_atrybutu=wartosc**, a wszystkie tak przygotowane atrybuty należy połączyć ze sobą pomocą ampersandu. Poprawny przykład żądania można zobaczyć na listingu 3.2 [16].

Listing 3.2: Żądanie wysylane w celu identyfikacji kontraktu za pomocą API

```
POST /api/bytecode HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

bytecode=60803350200fe56abcede00229&allFiles=false
```

Na listingu 3.2 do atrybutu **bytecode** został wprowadzony kod bajtowy kontraktu. Do zmiennej **allFiles** została wprowadzona wartość **false**, wiec w rezultacie zostanie zwrócone przez aplikację dziesięć najbardziej prawdopodobnych implementacji. Jeśli użytkownik chce pobrać wszystkie możliwe dopasowania, należy ustawić tą zmienną na **true**. W żądaniu został wprowadzony nagłówek **Content-Length** określający długość przesyłanych danych oraz **Content-Type** oznaczający rodzaj przesyłanych danych.

Jeśli wszystko poszło pomyślnie, użytkownik otrzyma status HTTP 200 wraz z listą składającą się z haszu pliku i współczynnika dopasowania danego pliku w formacie **JSON**. W przypadku gdy nie zostanie dopasowana żadna implementacja, to aplikacja zwróci status 404, natomiast jeśli w aplikacji wystąpi błąd, to zostanie zwrócony status 500.

Przesyłanie nowego kodu źródłowego za pomocą API

Gdy użytkownik chce przesłać nowy kontrakt do aplikacji, musi przejść proces uwierzytelniania. W tym celu należy do żądania dodać nagłówek **Authorization**. W nagłówku należy podać typ autoryzacji oraz zakodowane dane logowania za pomocą **Base64** według schematu **login:haslo**. Na przykładzie z listingu 3.3 został przesłany kod źródłowy, natomiast do autoryzacji wykorzystano login: 123 oraz hasło: 123.

Listing 3.3: Przesyłanie kodu źródłowego za pomocą API

```
POST /api/solidityFiles HTTP/1.1
Host: localhost:8080
Content-Type: text/plain
Accept: application/json
Authorization: Basic MTIzOjEyMw==
Content-Length: 221
Accept: application/json

pragma solidity ^0.4.21;
contract Hello {
    string public message;
    function setMessage(string newMessage) public {
        message = newMessage;
    }
}
```

Po pomyślnym przesłaniu kontraktu w odpowiedzi od serwera użytkownik otrzymuje status HTTP 200. W odpowiedzi zostaje również przesłany kod źródło-

wy, hasz stworzony na podstawie kodu źródłowego oraz listę znalezionych sygnatur funkcji wraz z ich selektorami. W przypadku wystąpienia błędu na serwerze zostaje zwrócony status 500. Na listingu 3.4 można zaobserwować przykładowe dane zawarte w odpowiedzi od serwera.

Listing 3.4: Przykładowa odpowiedz w formacie JSON

```
{
    "sourceCodeHash": "0
       x8dea780e1286d12a957d40597b9171a5187f87f6e3f8303505bc53a4453ad5b6
       ",
    "sourceCode": "pragma solidity ^0.4.21;\r\ncontract Hello {\r\n
        string public message;\r\n function setMessage(string
       newMessage) public {\r\n message = newMessage;\r\n }\r\n}",
    "solidityFunctions": [
        {
            "selector": "e21f37ce",
            "signature": "message()"
        },
        {
            "selector": "368b8772",
            "signature": "setMessage(string)"
        }
    ]
}
```

3.2 Przedstawienie architektury

W tym podrozdziale opiszę architekturę aplikacji, która realizuje funkcjonalności opisane w sekcji 3.1. Poniżej zostały krótko opisane główne klasy będące częścią aplikacji, widoczne na rysunku 3.4.

LoginController - jest to klasa odpowiedzialna za wyświetlenie ekranu logowania

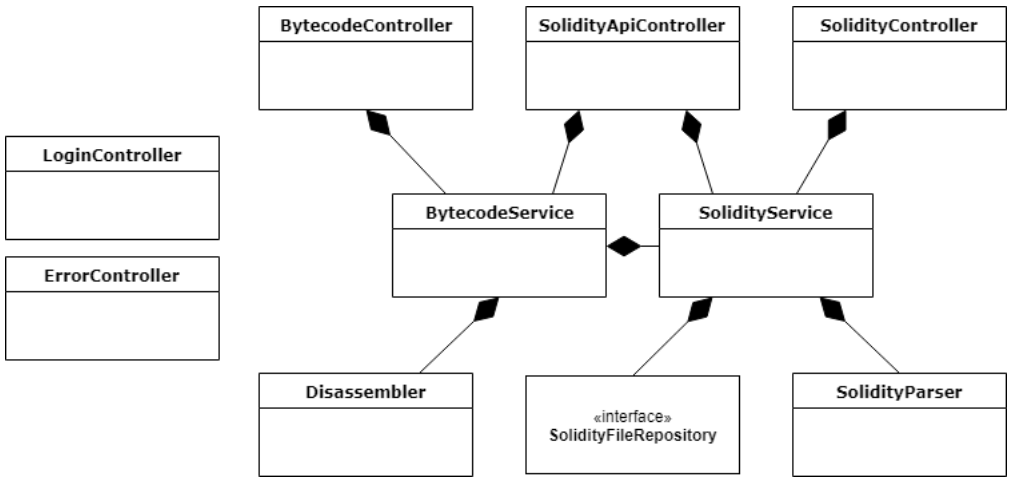
ErrorController - jej zadaniem jest przechwytywane wszystkich błędów w aplikacji. Po złapaniu błędu, zostaje wyświetlona użytkownikowi strona informująca, że pojawił się błąd w aplikacji, który jest zapisywany w logach aplikacji.

BytecodeController - służy do wyświetlania użytkownikowi strony związanej z identyfikacją kodu bajtowego oraz do mapowania żądań HTTP służących do identyfikacji.

SolidityApiController - zadaniem tej klasy jest nasłuchiwanie adresów związanych z API, zwracanie danych do użytkownika w formacie **JSON** lub zwykłego tekstu oraz komunikowanie się z obiektem klasy **SolidityService** oraz **BytecodeService**.

SolidityController - mapuje żądania HTTP związane z przetwarzaniem plików **Solidity** oraz umożliwia wyświetlenie użytkownikowi kodu źródłowego kontraktu.

Disassembler - odpowiada za analizę przekazanego kodu bajtowego. W rezul-



Rysunek 3.4: Architektura aplikacji

tacie zwraca listę instrukcji zawartych w kodzie. Szczegółowe działanie tej klasy zostało opisane w dalszej części pracy, w sekcji 3.5 dotyczącej wyszukiwania selektorów funkcji w kodzie bajtowym.

SolidityParser - wyciąga listę informacji o funkcjach z kodu źródłowego kontraktu. Lista zawiera takie informacje jak sygnatura oraz selektor funkcji. Sposób tworzenia selektorów funkcji oraz wyciągania z kodu źródłowego sygnatur funkcji został przedstawiony w sekcji 3.4

SolidityService - jest to klasa odpowiedzialna za odczytywanie danych z bazy danych oraz za przygotowanie przesłanych danych do zapisu w bazie danych.

BytecodeService - klasa odpowiada za dopasowywanie kodu bajtowego do kontraktu. W tym celu wykorzystywane są opisane powyżej klasy **Disassembler** oraz **SolidityService**, które w połączeniu umożliwiają wyznaczenie współczynnika dopasowania pomiędzy konkretnym plikiem a kodem bajtowym.

SolidityFileRepository - jest to część aplikacji odpowiedzialna za komunikację z bazą danych oraz mapowanie danych przechowywanych w bazie danych na obiekty zdefiniowane w kodzie aplikacji. Repozytorium jest interfejsem, który wykorzystuje moduł **Spring Data MongoDB**. Implementacja tego interfejsu spoczywa na frameworku Spring. Szczegóły łączenia z bazą danych zostały opisane w sekcji 3.3

3.3 Połączenie z bazą danych

W celu integracji aplikacji z bazą danych **MongoDB** został wykorzystany framework **Spring** oraz moduł **Spring Data MongoDB**. W związku z tym, że

projekt aplikacji jest budowany za pomocą narzędzia **Apache Maven**, należy dodać do pliku **pom.xml** wykorzystywane moduły [21].

Na listingu 3.5 został przedstawiony fragment pliku **pom.xml** odpowiedzialny za dodawanie modułu **spring-boot-start-data-mongodb** do projektu. Dodawanie innych modułów jest analogiczne do przykładu z listingu.

Listing 3.5: Przyklad dodania zależności w pliku pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

W pliku konfiguracyjnym **application.properties** zostały skonfigurowane dane do połączenia z bazą danych. Przykładowa zawartość pliku konfiguracyjnego została przedstawiona na listingu 3.6 [21].

Listing 3.6: Kofiguracja bazy danych

```
spring.data.mongodb.uri=mongodb://${ADMIN_DB_LOGIN}:${
    ADMIN_DB_PASSWORD}@ds129904.mlab.com:29904/${
    DATABASE_NAME_CONTRACT}
admin.login=${ADMIN_LOGIN}
admin.password=${ADMIN_PASSWORD}
```

Po skonfigurowaniu pliku **pom.xml** oraz **application.properties**, należało utworzyć interfejs **SolidityFileRepository**, który umożliwia serwisom aplikacji wykonywanie operacji na bazie danych oraz ustala mapowanie obiektów z bazy danych na obiekty klasy **SolidityFile**. Utworzone w aplikacji repozytorium można zobaczyć na listingu 3.7.

Listing 3.7: Stworzenie repozytorium za pomocą Spring Data MongoDB

```
1 @Repository
2 interface SolidityFileRepository extends MongoRepository<
    SolidityFile, String> {
3
4     @Query("{\"solidityFunctions\": {$elemMatch: {\"selector\": {
        $in: ?0}}}}")
5     List<SolidityFile> findSolidityFilesBySelectorContainsAll(List<
        String> functionSelector);
6
7     Optional<SolidityFile> findBySourceCodeHash(String
        sourceCodeHash);
8 }
```

W pierwszej linii listingu 3.7 znajduje się adnotacja **@Repository** pełniącą rolę stereotypu informującego framework, że ten interfejs jest wykorzystywany, w celu wykonywania operacji z bazą danych.

Kolejną adnotacją jest **@Query**. Parametrem tej adnotacji jest zapytanie do bazy danych **MongoDB**, wykonujące zapytanie o listę plików, które posiadają w sobie część przekazanych przez użytkownika selektorów funkcji. Za pomocą tej adnotacji można przypisać konkretnej metodzie z **SolidityFileRepository** konkretne zapytanie, które aplikacja ma wykonać.

Jeśli metoda w interfejsie nie posiada wspomnianej adnotacji, wtedy framework wygeneruje zapytanie do bazy danych, bazując na nazwie metody oraz przyjmowanych i zwracanych przez metodę typach danych.

Na listingu 3.8 została przedstawiona klasa **SolidityFile**, która reprezentuje obiekt przechowywany w bazie danych. Składa się ona z trzech atrybutów: haszu kodu źródłowego, kodu źródłowego, oraz listy funkcji znalezionych w tym kodzie źródłowym.

Atrybut **sourceCodeHash** został utworzony, ponieważ baza danych nie umożliwia utworzenia unikalnego atrybutu w bazie danych z taką duża ilością znaków

jak kod źródłowy. Tworzeniem haszu odbywa się w klasie **SolidityService** widocznej się na rysunku 3.4. Hasz kodu źródłowego jest identyfikatorem, wiec posiada adnotację **@Id**.

Listing 3.8: Przyklad klasy wykorzystywanej przez Spring Data MongoDb

```
1  public class SolidityFile {
2      @Id
3      private final String sourceCodeHash;
4      private final String sourceCode;
5      private final Set<SolidityFunction> solidityFunctions;
6
7      SolidityFile(String sourceCodeHash, String sourceCode, Set<
           SolidityFunction> solidityFunctions) {
8          requireNonNull(sourceCodeHash, "Expected not-null
               sourceCodeHash");
9          requireNonNull(sourceCode, "Expected not-null sourceCode");
10         requireNonNull(solidityFunctions, "Expected not-null
               solidityFunctions");
11         this.sourceCodeHash = sourceCodeHash;
12         this.sourceCode = sourceCode;
13         this.solidityFunctions = solidityFunctions;
14     }
15
16     public String getSourceCodeHash() { return sourceCodeHash; }
17     public String getSourceCode() { return sourceCode; }
18     public Set<SolidityFunction> getSolidityFunctions() { return
           solidityFunctions; }
19
20     @Override
21     public String toString() {
22         return "SolidityFile{" + "sourceCodeHash='"
23                 + sourceCodeHash
24                 + '\'' + ", sourceCode='" + sourceCode + '\''
25                 + ", solidityFunctions=" + solidityFunctions + '}'
                       ;}
26
27     @Override
28     public boolean equals(Object o) {
29         if (this == o) return true;
```

```
30          if (!(o instanceof SolidityFile)) return false;
31          SolidityFile that = (SolidityFile) o;
32          return Objects.equals(sourceCodeHash, that.sourceCodeHash)
               &&
33                  Objects.equals(sourceCode, that.sourceCode) &&
34                  Objects.equals(solidityFunctions, that.
                       solidityFunctions);}
35
36      @Override
37      public int hashCode() {
38          return Objects.hash(sourceCodeHash, sourceCode,
               solidityFunctions);}
39  }
```

W momencie tworzenia obiektu klasy **SolidityFile**, konstruktor wywołuje metody sprawdzające, czy użytkownik nie wprowadził wartości **null**, ponieważ każdy obiekt plik musi posiadać hasz, kod źródłowy oraz listę funkcji.

Klasa **SolidityFunction**, która jest częścią klasy **SolidityFile**, posiada dwa atrybuty **selector** oraz **signature** typu **String**. Klasa ta nie wy wymagała tworzenia identyfikatora, wiec nie została użyta adnotacja **@Id** nad żadnym atrybutem.

3.4 Identyfikacja sygnatur funkcji w kodzie źródłowym

Wyszukiwanie sygnatur funkcji jest rozpoczynane podczas przesłania nowego kodu źródłowego do aplikacji. W momencie wyszukiwania sygnatur funkcji są generowane selektory funkcji, które są finalnie używane podczas identyfikacji kodu bajtowego.

Problemem podczas wyszukiwania sygnatur w kodzie źródłowym jest to, że część sygnatur jest niejawna, ponieważ są one dodatkowo generowane przez kom-

pilator **Ethereum Virtual Machine** dla wybranych atrybutów kontraktu [12].

3.4.1 Kontroler interfejsu programistycznego

Na listingu 3.9 została przedstawiona metoda kontrolera **SolidityApiController** umożliwiająca przesyłanie kodu źródłowego. Adnotacje, które wykorzystuje ta metoda, są częścią modułu **Spring MVC**. Pierwszą adnotacją wykorzystaną w metodzie jest **@PostMapping**, która zajmuje się mapowaniem żądań HTTP przesyłanych do API. W parametrze tej adnotacji podany został adres, pod którym aplikacja oczekuje żądania [19].

Listing 3.9: Metoda kontrolera mapująca żądania POST

```
1 @PostMapping("/api/solidityFiles")
2 public ResponseEntity<SolidityFile> uploadFile(@RequestBody String
      sourceCode) throws IOException {
3     requireNonNull(sourceCode, "Expected not-null sourceCode");
4     return ResponseEntity.ok(solidityService.save(sourceCode));
5 }
```

Kolejną adnotacją jest **@RequestBody**. Adnotacja ta informuje framework o tym, żeby ciało żądania HTTP było umieszczone pod wskazaną zmienną.

Podczas zwracania danych w API wykorzystana została klasa **ResponseEntity**. Jest to wrapper umożliwiający zwrócenie statusu HTTP wraz z danymi. Informacja z tej klasy jest wykorzystywana przez framework podczas tworzenia odpowiedzi HTTP.

Głównym celem tej metody jest zapisanie nowego kodu źródłowego do aplikacji, w związku z tym na listingu 3.9 widać wywołanie metody **save** na atrybucie **solidityService**.

3.4.2 Kontroler strony internetowej

SolidityControler jest to klasa odpowiedzialna, za tworzenie strony internetowej korzystając z szablonów HTML oraz modułu **Thymeleaf**. Listing 3.10 przedstawia metodę przyjmująca w żądaniu HTTP kod źródłowy. Metoda ta działa podobnie jak w przypadku API, tylko w tym przypadku zwracana zostaje zwrócona nazwa szablonu wykorzystywanego do renderowania strony. Istnieje możliwość przekazania danych do szablonu, w tym celu wykorzystywany jest parametr **model**, na którym wywoływana jest metoda **addAttribute** [20].

Listing 3.10: Przechwytywanie żądania o dodanie nowego kodu źródłowego

```
1  @PostMapping("/solidity/text")
2  public String handleSourceCodeUpload(@RequestParam("sourceCode")
       String sourceCode, Model model) throws Exception {
3      requireNonNull(sourceCode, "Expected not-null sourceCode");
4      requireNonNull(model, "Expected not-null model");
5
6      SolidityFile savedSolidityFile = solidityService.save(
           sourceCode);
7
8      model.addAttribute("solidityFileFunctions", savedSolidityFile.
           getSolidityFunctions());
9      model.addAttribute("solidityFileHash", savedSolidityFile.
           getSourceCodeHash());
10     return "solidity-page";
11 }
```

3.4.3 Przetwarzanie kodu źródłowego

SolidityService po otrzymaniu kodu źródłowego od kontrolerów przekazuje po jednej linii do **SolidityParser**, który definiuje czy w danej linii jest sygnatura funkcji. Jeśli podczas parsowania linii znaleziono sygnaturę funkcji, to zostaje ona dodana wraz z selektorem do listy funkcji. Po przeanalizowaniu wszystkich linii

tworzony jest obiekt **SolidityFile**, który następnie za pomocą **SolidityFileRepository** jest zapisywany do bazy danych.

Listing 3.11: Metoda wyszukujaca sygnatury funkcji

```
Optional<SolidityFunction> findFunctionInLine(String line) {
    List<Optional<SolidityFunction>> functions =
            Stream.of(
                    findFunctionSignature(line),
                    findMappingGetter(line),
                    findArrayGetter(line),
                    findNormalVariableGetter(line)
            ).filter(Optional::isPresent).collect(toList());

    if (functions.size() > 1) {
        throw new IllegalStateException("Expected only one function
            , but found :" + functions.size());
    } else if (functions.size() == 1) {
        return functions.listIterator().next();
    }
    return Optional.empty();
}
```

Na listingu 3.11 widać metodę klasy **SolidityParser** wyszukującą funkcję w implementacji kontraktu. Metoda po przyjęciu linii w rezultacie zwraca obiekt klasy **Optional<SolidityFunction>** [18]. Metoda sprawdza cztery możliwe przypadki, w których istnieje możliwość wykrycia funkcji w kodzie źródłowym kontraktu.

Do wykrywania błędów podczas wyszukiwania funkcji sprawdzane są wszystkie cztery przypadki, jeśli okaże się, że więcej niż jedna metoda wykryła funkcję, oznacza to, że jedna z metod działa niepoprawnie i fałszywie wykrywa funkcje. Wszystkie cztery przypadki zostały opisane poniżej.

Wykrywanie sygnatury zadeklarowanej funkcji

Pierwszym przypadkiem są funkcje jawnie zadeklarowane w kodzie źródłowym nie posiadające modyfikatora **internal** lub **private**. Wykrywaniem takiej funkcji zajmuje się metoda **findFunctionSignature** widoczna na listingu 3.11. Do wyszukiwania sygnatury zadeklarowanej funkcji zostało wykorzystane wyrażenie regularne zaprezentowane poniżej:

```
^\s*function\s*([a-zA-Z_][a-zA-Z0-9_]*)\s*\(\s*([^(){}]*)\s*\)\s
    *(?!.*(internal|private)).*$
```

Pierwsza grupa w wyrażeniu wyciąga z linii kodu źródłowego nazwę funkcji, natomiast druga parametry funkcji. Wyrażenie wyszukuje w pojedynczej linii kodu źródłowego frazy **function**, po której następuje nazwa funkcji oraz lista parametrów w nawiasach **??**.

Na listingu 3.12 została przedstawiona metoda wyszukującą sygnaturę funkcji za pomocą przedstawionego wyrażenia regularnego.

Listing 3.12: Metoda odbierająca żądanie o zapisanie kodu źródłowego na serwerze

```
1  private Optional<SolidityFunction> findFunctionSignature(String
       line) {
2      Matcher matcher = FUNCTION_PATTERN.matcher(line);
3      if (matcher.find()) {
4          String functionName = matcher.group(FUNCTION_NAME_GROUP_ID)
               ;
5          String functionArguments = matcher.group(
               FUNCTION_ARGUMENTS_GROUP_ID);
6          String functionSignature = normalizeFunctionSignature(
               functionName, functionArguments);
7          String functionSelector = getFunctionSelector(
               functionSignature);
8          return Optional.of(new SolidityFunction(functionSelector,
               functionSignature));
9      }
10     return Optional.empty();
11 }
```

Po wykryciu funkcji za pomocą wyrażenia regularnego wyciągana jest informacja o parametrach i nazwie funkcji z przekazanej linii. Za pomocą wyciągniętych informacji metoda **normalizeFunctionSignature** tworzy sygnaturę funkcji. Sygnatura składa się z nazwy oraz typów parametrów funkcji podanych w nawiasie. Niektóre typy parametrów zostają sprowadzone do postaci kanonicznej, natomiast pozostałe typy pozostają bez zmian. Poniżej zostały przedstawione typy, które zostają sprowadzane do postaci kanonicznej:

$$\text{uint} => \text{uint256}$$
$$\text{int} => \text{int256}$$
$$\text{byte} => \text{bytes1}$$

Po utworzeniu sygnatury funkcji zostaje wygenerowany selektor. Na listingu 3.13 została przedstawiona metoda tworząca selektor. Sygnatura funkcji zostaje haszowana za pomocą funkcji **sha3String**, ktora pochodzi z biblioteki **web3j**, następnie z hasza pobierane są cztery pierwsze bajty, które są selektorem funkcji. Metoda **sha3String** zwraca hasz w systemie szesnastkowym w postaci napisu, dlatego wyłuskiwane są znaki od dwa do dziesięć.

Listing 3.13: Metoda generująca selektor funkcji

```
1 private String getFunctionSelector(String
    normalizedFunctionSignature) {
2     return sha3String(normalizedFunctionSignature).substring(2, 10)
        ;
3 }
```

Po pomyślnej identyfikacji zadeklarowanej funkcji zwracany jest obiekt **Optional<SolidityFunction>** przez metodę **findFunctionSignature**. Analizując w ten sposób każdą linię kodu źródłowego kontraktu, **SolidityService** uzyskuje zbiór obiektów **SolidityFunction**.

Generowanie sygnatury funkcji dla publicznych atrybutów typu mapa

Drugim rodzajem jest atrybut publiczny typu **mapping**. W tym przypadku nie zostało jawnie pokazane, że istnieje w kodzie źródłowym sygnatura funkcji, ponieważ jest ona generowana przez kompilator [12]. W tej sekcji przedstawię proces generowania sygnatury funkcji na podstawie wspomnianego atrybutu.

W celu wykrycia deklaracji mapy w kodzie źródłowym, zostało wykorzystane wyrażenie regularne przedstawione poniżej:

```
^\\s*mapping\\s*\\(\\s*([a-zA-Z][a-zA-Z]*)\\s*=>\\s*(.*)\\s*\\)\\s
    *public\\s*([a-zA-Z_$][a-zA-Z_$0-9]*)\\s*(=.*)?\\s*;+\\s*(//.*)
    ?$
```

Pierwsza grupa w wyrażeniu regularnym oznacza typ klucza mapy, natomiast druga typ zwracanej wartości przez mapę. Trzecia grupa oznacza nazwę atrybutu, która również jest nazwa sygnatury funkcji. Typ klucza mapowania jest umieszczany w parametrze tworzonej sygnatury funkcji. Jeśli typ zwracany przez mapę jest typem tablicowym lub kolejną mapą, wtedy należy dodać kolejny parametr do sygnatury funkcji.

Na listingu 3.14 została przedstawiona pętla, która jest częścią metody **findMappingGetter**. Pętla działa dopóki występuje zagnieżdżanie map.

Listing 3.14: Pętla wyszukująca tablicę lub mapę w typie zwracanym przez mapę

```
1 while (true) {
2     Matcher mappingMatcher = MAPPING_PATTERN.matcher(mappingValue);
3     Matcher arrayMatcher = ARRAY_PATTERN.matcher(mappingValue);
4
5     if (mappingMatcher.find()) {
6         String canonicalArgument = toCanonicalType(mappingMatcher.
              group(MAPPING_KEY_GROUP_ID));
7         canonicalMappingKeys.add(canonicalArgument);
8         mappingValue = mappingMatcher.group(MAPPING_VALUE_GROUP_ID)
              ;
9         continue;
```

```
10      }
11      if (arrayMatcher.find()) {
12          String arrayValue = arrayMatcher.group(ARRAY_VALUE_GROUP_ID
                );
13          int dimensionCount = getArrayDimensionCount(arrayValue);
14          for (int i = 0; i < dimensionCount; i++) {
15             canonicalMappingKeys.add(CANONICAL_ARRAY_KEY_TYPE);
16          }
17      }
18      break;
19  }
```

Do wyszukiwania map w typie zwracanym przez poprzednią mapę zostało wykorzystane następujące wyrażenie regularne:

```
^\s*mapping\s*\(\s*([a-zA-Z0-9][a-zA-Z0-9]*)\s*=>\s*(.*)\s*\)\s*
```

Pierwsza grupa wyrażenia wyciąga z fragmentu deklaracji mapy informacje o typie klucza, natomiast druga o typie zwracanej przez nią wartości. Typ klucza zagnieżdżonej mapy jest dodawany do listy typów parametrów generowanej sygnatury funkcji, natomiast typ zwracany przez mapę jest wykorzystany z tym samym wyrażeniem regularnym w kolejnej iteracji pętli.

W przypadku, gdy nie wykryto mapy, jest sprawdzane, czy zwracanym typem jest tablica. W tym celu wykorzystane zostało następujące wyrażenie regularne:

```
^\s*[a-zA-Z0-9][a-zA-Z0-9]*((\s*\[\s*[a-zA-Z0-9]*\s*]\s*)+)\s*
```

W wyrażeniu została określona grupa, która służy do wyznaczenia ilości wymiarów tablicy. Dla każdego wymiaru zostaje dodany parametr typu **uint256** do listy parametrów sygnatury. Wspomniany parametr reprezentuje indeks tablicy. Wykrycie tablicy w typie zwracanym przez mapę jest równoznaczne z ostatnią iteracja pętli.

Ostatnim krokiem jest wygenerowanie sygnatury funkcji za pomocą nazwy funkcji oraz listy zgromadzonych typów parametrów. Wszystkie typy przed do-

daniem do listy zostają najpierw sprowadzone do postaci kanoniczej.

Na listingu 3.15 widać w jaki sposób jest formułowana sygnatura funkcji, następnie jest generowany z jej selektor oraz tworzony obiekt typu **Optional<SolidityFunction>**.

Listing 3.15: Tworzenie SolidityFunction na podstawie nazwy funkcji i listy typów

```
1 String functionSignature = mappingName + "(" + join(",",
    canonicalMappingKeys) + ")";
2 String functionSelector = getFunctionSelector(functionSignature);
3 return Optional.of(new SolidityFunction(functionSelector,
    functionSignature));
```

Wykrywanie sygnatury funkcji dla publicznych atrybutów typu tablicowego

Trzecią metodą widoczną na listingu 3.11 jest **findArrayGetter**. Metoda służy do wykrywania publicznego atrybutu, który jest tablicą i w tym celu wykorzystuje wyrażenie regularne przedstawione poniżej:

```
^\s*[a-zA-Z0-9][a-zA-Z0-9]*((\s*\[\s*[a-zA-Z0-9]*\s*]\s*)+)\s*
    public\s*([a-zA-Z_$][a-zA-Z_$0-9]*)\s*(=.*)?\s*;+\s*(//.*)?$
```

Wyrażenie to wyszukuje w linii kodu źródłowego publiczny atrybut tablicowy. Pierwsza grupa w wyrażeniu wyodrębnia nawiasy definiujące ilość wymiarów tablicy. Kolejna grupa, znajdująca się po wyrazie **public**, reprezentuje nazwę atrybutu. Wyrażenie regularne bierze pod uwagę możliwość inicjalizacji wartości podczas deklaracji atrybutu.

Liczba wymiarów tablicy zostaje określona na podstawie liczby podanych nawiasów podczas deklaracji tablicy. Dla każdego wymiaru zostaje dodany typ **uint256** do listy typów parametrów, ponieważ tablica posiada numeryczny indeks [15].

Bazując na nazwie oraz liście typów tworzony jest obiekt **Optional<SolidityFunction>** w analogiczny sposób, jaki został przedstawiony na listingu 3.15.

Wykrywanie sygnatury funkcji dla pozostałych publicznych atrybutów

Ostatnim przypadkiem, który należało rozpatrzyć podczas wykrywania sygnatury funkcji, są wszystkie atrybuty zadeklarowane jako publiczne niebędące tablicami lub mapami. Ten przypadek odzwierciedla metoda **findNormalVariableGetter** widoczna na listingu 3.11 jako ostatnia z czterech metod wykorzystywany do analizy linii kodu źródłowego. Zostało zastosowane tutaj wyrażenie regularne przedstawione poniżej:

```
^\s*[a-zA-Z0-9][a-zA-Z0-9]*\s*(\bconstant)*\s*public\s*(\bconstant
    )*\s*([a-zA-Z_$][a-zA-Z_$0-9]*)\s*(=.*)?\s*;+\s*(//.*)?$
```

Wyrażenie regularne wyszukuje linię, w której znajduje się deklaracja atrybutu publicznego, który może opcjonalnie posiadać modyfikator **constant**. Wyrażenie deklaracje atrybutow bez typu **mapping** oraz tablic.

Sygnatura funkcji składa się z nazwy atrybutu wyodrębnianej za pomocą grupy zdefiniowanej w wyrażeniu regularnym. W tym przypadku sygnatura funkcji nie posiada żadnych parametrów, wiec wystarczy po nazwie dodać pusty nawias.

Selektor funkcji jest generowany analogicznie jak w pozostałych przypadkach. Na listingu 3.16 widać metodę **findNormalVariableGetter** generującą sygnaturę na podstawie tego rodzaju atrybutu.

Listing 3.16: Metoda wyszukujaca sygnature funkcji dla atrybutów niebędących mapą ani tablica

```
1 private Optional<SolidityFunction> findNormalVariableGetter(String
      line) {
2     Matcher matcher = NORMAL_VARIABLE_PATTERN.matcher(line);
```

```
3     if (matcher.find()) {
4         LOGGER.info("Found public normal variable: {}", line);
5         String variableName = matcher.group(
              NORMAL_VARIABLE_NAME_GROUP_ID);
6
7         String functionSignature = variableName + "()";
8         String functionSelector = getFunctionSelector(
              functionSignature);
9
10        return Optional.of(new SolidityFunction(functionSelector,
              functionSignature));
11    }
12    return Optional.empty();
13 }
```

Wszystkie wykryte funkcje w poszczególnych liniach implementacji, zostają zapisane w bazie danych wraz z analizowanym kodem źródłowym oraz jego haszem.

3.5 Wyszukiwanie selektorów funkcji w kodzie bajtowym

Podczas identyfikacji inteligentnego kontraktu podawany jest kod bajtowy, który posiada w sobie informacje selektory funkcji. Następnie na podstawie selektorów funkcji zawartych w kodzie bajtowym, zostają dopasowane implementacje inteligentnych kontraktów przechowywane w bazie danych.

W tym podrozdziale przedstawię, w jaki sposób na podstawie kodu bajtowego są wyszukiwane selektory funkcji wykorzystywane podczas dopasowywania implementacji.

W celu wykrycia selektorów funkcji należy najpierw uzyskać listę instrukcji wykonywanych w kodzie bajtowym wraz z parametrami tych instrukcji. Do tego posłuży klasa **Dissassembler**, do której jest przekazywany kod bajtowy za

pomocą metody **dissassembly** . W rezultacie wspomnianej metody zwracana jest lista instrukcji reprezentowanych przez klase **Instruction**. Klasa **Instruction** składa się z dwóch atrybutów: **opcode** typu **Opcode** oraz **hexParameters** typu **String**.

3.5.1 Reprezentacja operacji EVM wewnątrz aplikacji

Opcode jest to typ wyliczeniowy, który definiuje wszystkie kody operacji, jakie można wykonać na **Ethereum Virtual Machine**. Wszystkie te operacje zostały opisane w dokumencie definiującym **EVM** [6].

Typ wyliczeniowy posiada w sobie trzy atrybuty: **hexValue**, **operandSize** oraz **description**, które są przekazywane w konstruktorze. Na listingu 3.17 widać fragment deklaracji poszczególnych operacji. Pierwszym parametrem konstruktora jest **hexValue**, czyli kod operacji w postaci szesnastkowej. Następnie podawany jest **operandSize**, który określa rozmiar parametru danej operacji za pomocą liczby bajtów. Ostatnią przekazywaną informacją jest opis operacji.

Listing 3.17: Fragment kodu źródłowego definiującego kody operacji w typie wyliczeniowym

```
1 ...
2 MSIZE(0x59, 0, "Get the size of active memory in bytes."),
3 GAS(0x5A, 0, "Get the amount of available gas, including the
    corresponding reduction the amount of available gas."),
4 JUMPDEST(0x5B, 0, "Mark a valid destination for jumps."),
5
6 PUSH1(0x60, 1, "Place 1 byte item on stack."),
7 PUSH2(0x61, 2, "Place 2-byte item on stack."),
8 PUSH3(0x62, 3, "Place 3-byte item on stack."),
9 ...
```

Ponieważ kod bajtowy przechowuje kody operacji oraz ich parametry w postaci szesnastkowej, w tym celu została utworzona klasa **OpcodeTable**, która

umożliwia mapowanie kodu operacji na konkretny obiekt zdefiniowany w typie wyliczeniowym **Opcode**. Klasa **OpcodeTable** została zaprezentowana na listingu 3.18. W przypadku, gdy podany kod nie posiada swojego odpowiednika w typie wyliczeniowym, wtedy zostaje zwrócony kod **UNKNOWNCODE**.

Listing 3.18: Klasa mapujaca identyfikator operacji na reprezentacje operacji typu wyliczeniowego

```
class OpcodeTable {
    private OpcodeTable() {
        throw new UnsupportedOperationException();
    }

    private static final Map<Integer, Opcode> opcodes =
            unmodifiableMap(new HashMap<Integer, Opcode>() {{
                for (Opcode opcode : Opcode.values()) {
                    put(opcode.getHexValue(), opcode);
                }
            }});

    static Opcode getOpcodeByHex(String stringHex) {
        if(stringHex.length() != 2){
            throw new IllegalArgumentException("Expected length=2
                stringHex");
        }
        return getOpcodeByHex(Integer.parseInt(stringHex, 16));
    }

    static Opcode getOpcodeByHex(int hex) {
        Opcode opcode = opcodes.get(hex);
        if (isNull(opcode)) {
            return Opcode.UNKNOWNCODE;
        }
        return opcode;
    }
}
```

3.5.2 Odczytywanie instrukcji z kodu bajtowego

Mając zdefiniowane w aplikacji wszystkie operacje maszyny wirtualnej **Ethereum**, można przystąpić do wyszukiwania wszystkich instrukcji z kodu bajtowego. Na listingu 3.19 widać metodę przestawiającą ten proces.

Listing 3.19: Metoda wyszukująca instrukcje instrukcje w kodzie bajtowym

```
1  private List<Instruction> getInstructions(String bytecode) {
2      String validBytecode = getValidBytecode(bytecode);
3      HexStringIterator hexStringIterator = new HexStringIterator(
           validBytecode);
4
5      List<Instruction> instructions = new ArrayList<>();
6      while (hexStringIterator.hasNext()) {
7          Opcode opcode = getOpcodeByHex(hexStringIterator.next());
8          String instructionParameter = getInstructionOperand(opcode.
               getOperandSize(), hexStringIterator);
9          instructions.add(new Instruction(opcode,
               instructionParameter.toLowerCase()));
10     }
11     return instructions;
12 }
```

Do iteracji po kodzie bajtowym został wykorzystany iterator typu **HexStringIterator**, który odczytuje po jednym bajcie, dopóki istnieją kolejne bajty. W każdej iteracji pętli jest definiowany konkretny obiekt typu **Opcode** za pomocą metody **getOpcodeByHex** przedstawionej na listingu 3.18. Następnie w zależności od kodu operacji zostają pobierane kolejne bajty, które są parametrem instrukcji [6]. Na podstawie tych informacji tworzony jest obiekt typu **Instruction**, który jest dodawany do listy **instructions**, zwracanej na końcu wykonywania metody.

3.5.3 Wyszukiwanie selektorów funkcji z listy instrukcji

Po wykryciu wszystkich operacji wraz z ich parametrami przez klasę **Disassembly**, klasa **BytecodeService** może przystąpić do wyszukiwania ostatecznych selektorów funkcji.

Na listingu 3.20 została przedstawiona metoda zwracająca listę selektorów funkcji znajdujących się w kodzie bajtowym. Odczytując z listy po trzy kolejne instrukcje, zostaje wyszukiwany schemat operacji wykonywanych na **EVM**, na podstawie którego aplikacja wyszukuje moment wrzucenia selektora funkcji na stos [10].

Listing 3.20: Metoda wyszukująca selektory funkcji na podstawie listy instrukcji

```
1  private List<String> findFunctionSelectors(String bytecode) {
2      List<Instruction> instructions = disassembler.disassembly(
           bytecode);
3      List<String> functionSelectors = new ArrayList<>();
4      for (int i = 0; i < instructions.size() - 2; i++) {
5          Instruction first = instructions.get(i);
6          Instruction second = instructions.get(i + 1);
7          Instruction third = instructions.get(i + 2);
8          boolean isFunctionSchemeFound =
9                  first.hasMnemonic(PUSH_4_MNEMONIC) && second.
                       hasMnemonic(EQ_MNEMONIC) && third.hasMnemonic(
                       PUSH_2_MNEMONIC);
10         if (isFunctionSchemeFound) {
11             functionSelectors.add(first.getHexParameters());
12         }
13     }
14     return functionSelectors;
15 }
```

Gdy zmienna **isFunctionSchemeFound** przechowuje wartość **true**, wtedy wiadomo, że został wyszukany charakterystyczny dla selektorów funkcji schemat kodów operacji. W takim przypadku pobierany jest parametr ze zmiennej **first** oraz dodawany jest on do listy **functionSelectors**.

3.6 Dopasowywanie implementacji na podstawie kodu bajtowego

Po wyszukaniu wszystkich selektorów funkcji z kodu bajtowego zostają one wykorzystane do wyszukania odpowiedniej implementacji. Z bazy danych zostają wyszukane wszystkie pliki posiadające chociaż jeden selektor funkcji znajdujący się w kodzie bajtowym. Dla każdego pobranego w ten sposób pliku wyznaczany jest współczynnik dopasowania kodu bajtowego z plikiem. Do wyznaczania współczynnika dopasowania kodu bajtowego dla konkretnej implementacji można było zastosować kilka podejść. Poniżej opiszę trzy przetestowane przez mnie sposoby.

1. Pierwszym sposobem jest wyznaczenie ilości dopasowanych selektorów funkcji kodu bajtowego w danym pliku Solidity względem wszystkich selektorów funkcji w kodzie bajtowym. Rozwiązanie to można zaprezentować za pomocą wzoru:

$$W = M/B$$

gdzie:

W - współczynnik dopasowania implementacji

M - liczba selektorów funkcji wspólnych dla pliku oraz kodu bajtowego

B - liczba selektorów funkcji wystepujacych w kodzie bajtowym

Przy zastosowaniu takiego rozwiazania, jesli uzytkownik otrzyma wynik 100% wtedy jest pewny ze wszystkie selektory znalezione w kodzie bajtowym maja swoj odpowiednik w pliku. Natomiast problemem moga sie okazac implementacje w Solidity posiadajace bardzo duza ilosc selektorow

funkcji np. biblioteki matematyczne, wtedy wyniki dla takich bibliotek napisanych w solidity beda mialy rownie wysoki wspolczynnik dopasowani co mniejsza, bardziej zblizona do prawdziwej implementacja.

2. Drugim sposobem jest zwracanie ilosci dopasowanych selektorow funkcji z kodu bajtowego wzgledem ilosci selektorow w pliku solidity. Rozwiazanie to mozna zaprezentowac za pomoca wzoru:

$$W = M/S$$

W - wspolnczynnika dopasowania implementacji

M - liczba selektorow funkcji wspolnych dla pliku oraz kodu bajtowego

S - liczba selektorow funkcji wystepujacych w pliku solidity

Aplikacja przyjmuje caly kod zrodlowy w ktorym moze byc kilka implementacji kontraktu, natomiast w kodzie bajtowym znajduja sie selektory tylko jednego kontraktu. Z tego powodu moze wystapic sytuacji gdy dla prawdziwej implementacji uzytkownik nigdy nie uzyska 100% dopasowania bo w implementacji solidity sotanie wykryte wiecej selektorow funkcji niz jest w kodzie bajtowym.

3. Trzecic sposobem jest ktory moglby rozwiazac problemy twoch pierwszych jest obliczenie wspolczynnika dla dwoch pierwszych metod a nastepnie usrednienie wynikow.

Zostaly przeprowadzone testy trzech wspomianych sposobow. Jako grupe testową pobralem 474 implementacji kontraktow napisanych w jezyku solidity ze strony Etherscan wraz z wersjami w ktorych powinny zostac skompilowane. Kazdy plik zostal skompilowany za pomoca kompilatora solc oraz biblioteki py-solc ktora ulatwia kompilacje z poziomu Pythona. Wszystkie pobrane implementacje

zostaly skompilowane za pomoca kompilatora w wersji od 0.22.2 do 0.22.4. Informacja za pomoca jakiej wersji kompilatora nalezy skompilowac kontrakt zostala pobrana z Etherscan w zaleznosci od konkretnego pliku. Testy zostaly napisane w w jezyku Python z wykorzystaniem biblioteki py-solc ktora ulatwia kompilacje w Pythonie.

Na tabeli xx widac przedstawione wszystkie trzy metody dopasowywania

x

x

Istenieje sposob ktory moglby zagwarantowac niemal 100% dopasowanie, ale wymaga on przekazania do aplikacji kazdego kontraktu w odzielnym pliku lub sprawdzania w jednej duzej implementacji, ktora metoda nalezy do jakiego kontraktu. W przypadku gdy posiadalbym informacje o selektorach z konkretnego kontraktu, a nie z calej implementacji, wtedy mozna uzyskac bardzo dobre wyniki identyfikacji. Problemem na chwile pisania aplikacji bylo w jaki sposob poprawnie zidentyfikowac ktora sygnatury funkcji nalezy do ktorego kontraktu w calej implementacji pobranej ze strony Etherscan.

3.7 Wykorzystane technologie

Literatura: [?, ?]. TODO

Bibliografia

[1] http://ethdocs.org/en/latest/introduction/what-is-ethereum.html

[2] Vitalik Buterin *A Prehistory of the Ethereum Protocol* https://vitalik.ca/2017-09-15-prehistory.html

[3] Ethereum Blog *Ether Sale: A Statistical Overview* https://blog.ethereum.org/2014/08/08/ether-sale-a-statistical-overview/

[4] Ethereum *DEVCON-8 recap* https://blog.ethereum.org/2014/12/05/d%CE%BEvcon-0-recap/

[5] Seung Woo Kim *Secure Tree: Why State Tries Key is 256 Bits* https://medium.com/codechain/secure-tree-why-state-tries-key-is-256-bits-1276beb68485

[6] Dr. Gavin Wood *Ethereum: A secure decentralised generalised transaction Ledger. Byzantium version 4e05aa0 - 2019-03-04.*

[7] Ethereum Homestead *History of Ethereum* https://ethereum-homestead.readthedocs.io/en/latest/introduction/history-of-ethereum.html

[8] Ethereum *Ethereum Launch Process* https://ethereum.github.io/blog/2015/03/03/ethereum-launch-process/

[9] Ethereum *Solidity Grammar* https://github.com/ethereum/solidity/blob/develop/docs/grammar.txt

[10] Alejandro Santander *Deconstructing a Solidity Contract* https://blog.zeppelin.solutions/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737.

[11] Brandon Arvanaghi *Reversing Ethereum Smart Contracts* https://arvanaghi.com/blog/reversing-ethereum-smart-contracts/.

[12] Solidity Documentation *Getter Functions* https://solidity.readthedocs.io/en/v0.5.5/contracts.html#getter-functions.

[13] Elliot Bettilyon *What Is an API and Why Should I Use One?* https://medium.com/@TebbaVonMathenstien/what-is-an-api-and-why-should-i-use-one-863c3365726b.

[14] Internet Engineering Task Force (IETF) *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* https://tools.ietf.org/html/rfc7231.

[15] Solidity Documentation *Arrays* https://solidity.readthedocs.io/en/v0.5.3/types.html#arrays.

[16] Dinesh Balaji *Understanding HTML Form Encoding: URL Encoded and Multipart Forms* https://dev.to/sidthesloth92/understanding-html-form-encoding-url-encoded-and-multipart-forms-3lpa.

[17] Network Working Group *Hypertext Transfer Protocol - HTTP/1.1* https://www.ietf.org/rfc/rfc2616.txt.

[18] Yannick Majoros *Java 8 Optional - Avoid Null and NullPointerException Altogether - and Keep It Pretty* https://dzone.com/articles/java-8-optional-avoid-null-and.

[19] Baeldung *Spring Web Annotations* `https://www.baeldung.com/spring-mvc-annotations`.

[20] Rafał Borowiec *Spring MVC and Thymeleaf: how to access data from templates* `https://www.thymeleaf.org/doc/articles/springmvcaccessdata.html`.

[21] Spring *Accessing Data with MongoDB* `https://spring.io/guides/gs/accessing-data-mongodb/`.

Spis tabel

Spis rysunków

Spis listingów