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

1.1 Historia

TODO

1.2 Opis platformy

TODO

1.3 Ethereum Virtual Machine

TODO

1.4 Inteligentne kontrakty

TODO

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.

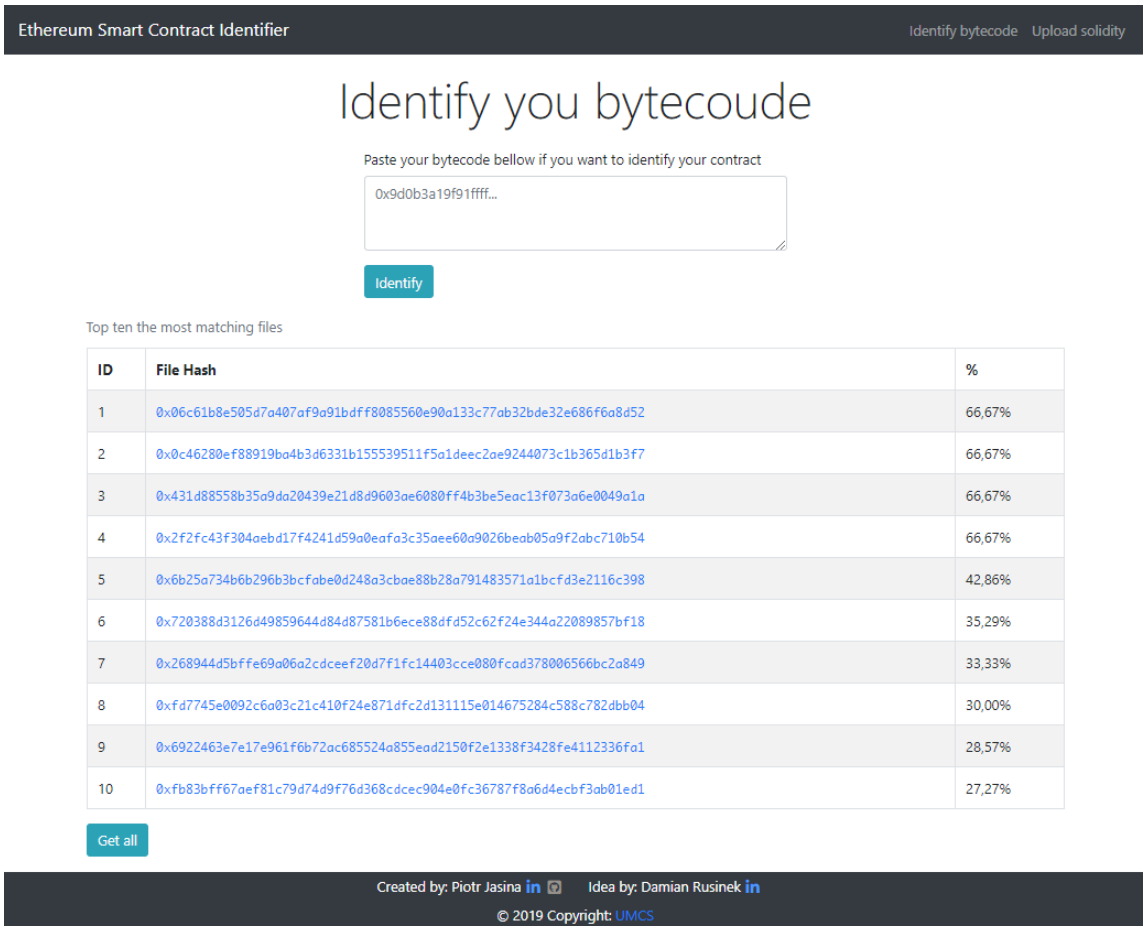
3.1.1 Identyfikacja inteligentnych kontraktów

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.

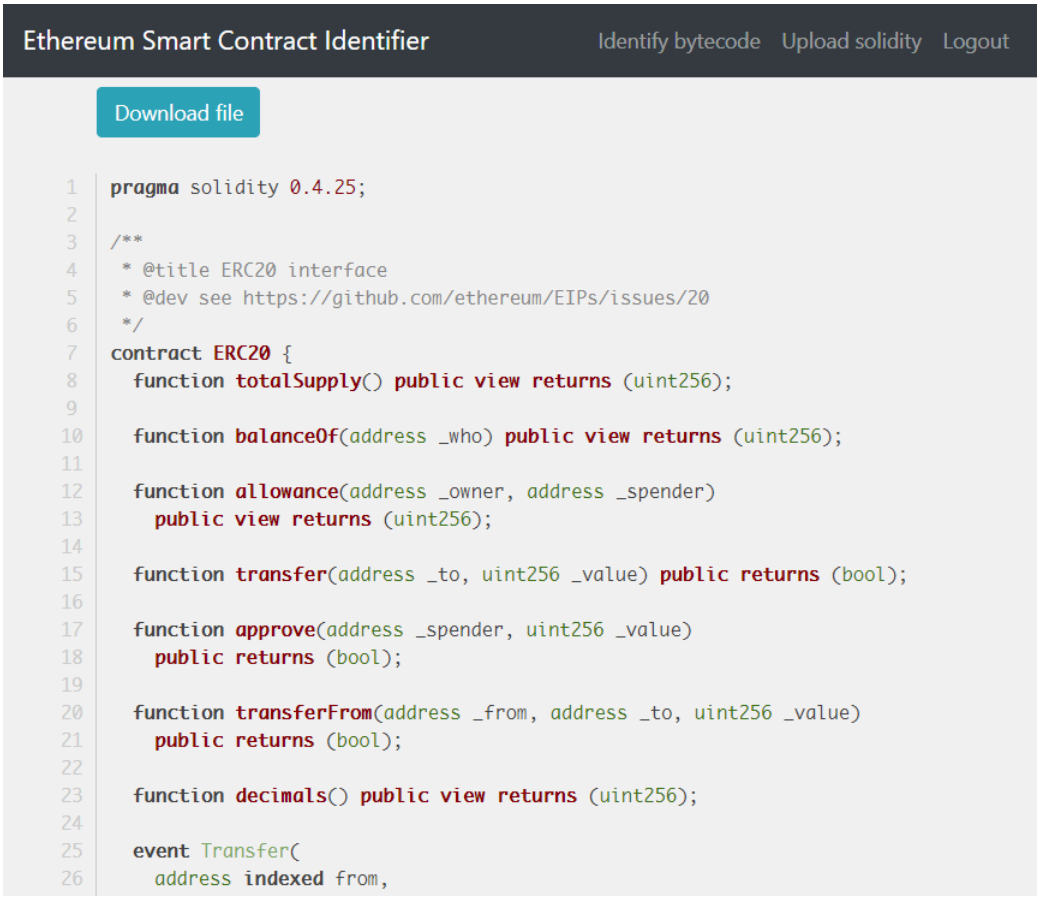
Po wprowadzeniu danych i zatwierdzeniu ich przyciskiem **Identify**, aplikacja rozpoczyna proces analizy wprowadzonego kodu bajtowego oraz wyszukiwane są najbardziej prawdopodobne implementacje kontraktu, posortowane malejąco według współczynnika dopasowania. Na rysunku 3.1 został przedstawiony przykładowy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

Domyślnie jest wyświetlanych dziesięć najbardziej prawdopodobnych implementacji, po naciśnięciu przycisku **Get all**, znajdującego się pod lista 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 linie w kodzie. Istnieje tez 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 probie korzystania z autoryzowanych funkcjonalności aplikacji. Po kliknięciu w **Identify Solidity** oraz zalogowaniu się na stronie, pojawiają się dwie możliwości wprowadzania kodów źródłowych.

Pierwszą 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.

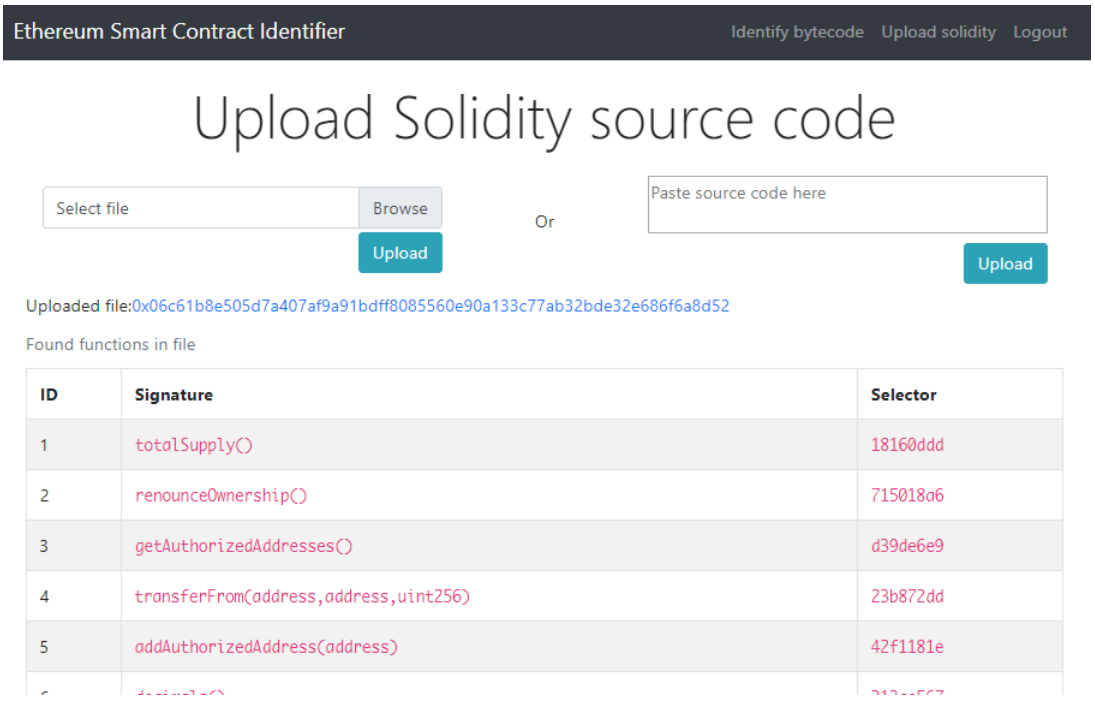
3.1.3 Interfejs programistyczny aplikacji

Trzecia funkcjonalnością aplikacji jest interfejs programistyczny. Dzięki niemu można wykorzystać mechanizmy zaimplementowane w aplikacji w innej aplikacji. Przykładowym zastosowaniem API (ang. Application programming interface) jest utworzenie skryptu umożliwiającego zautomatyzowane wysyłanie kodów źródłowych do aplikacji, bez konieczności korzystania z interfejsu graficznego aplikacji.

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.



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.

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 oraz listę składającą się z haszu pliku oraz 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, a do autoryzacji wykorzystano login: 123 oraz hasło: 123.

Listing 3.3: Przesyłanie kodu źródłowego za pomocą API

```
1 POST /api/solidityFiles HTTP/1.1
2 Host: localhost:8080
3 Content-Type: text/plain
4 Accept: application/json
5 Authorization: Basic MTIzOjEyMw==
6 Content-Length: 221
7 Accept: application/json
8
9 pragma solidity ^0.4.21;
10 contract Hello {
11     string public message;
12     function setMessage(string newMessage) public {
```

```
13         message = newMessage;
14     }
15 }
```

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

```
1 {
2     "sourceCodeHash": "0
        x8dea780e1286d12a957d40597b9171a5187f87f6e3f8303505bc53a4453ad5b6
        ",
3     "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}",
4     "solidityFunctions": [
5         {
6             "selector": "e21f37ce",
7             "signature": "message()"
8         },
9         {
10            "selector": "368b8772",
11            "signature": "setMessage(string)"
12        }
13    ]
14 }
```

3.2 Przedstawienie architektury

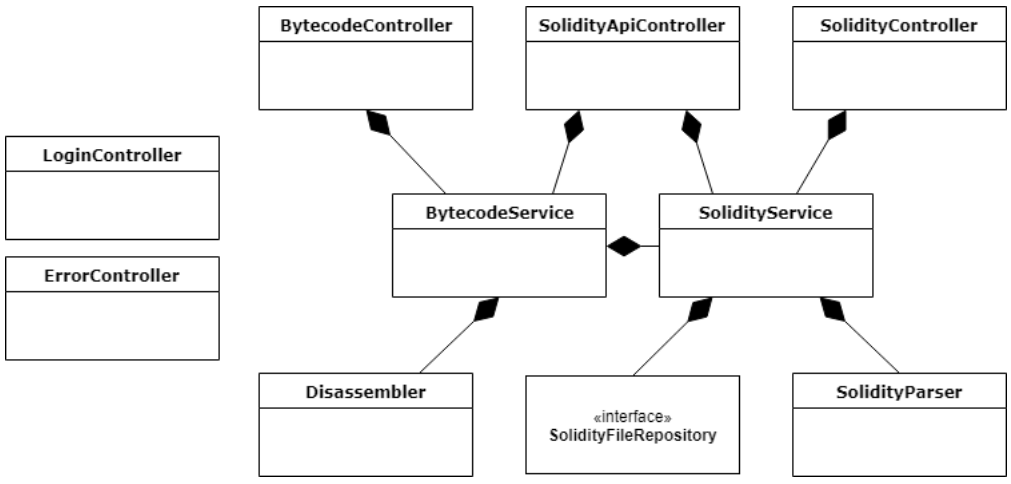
W tym podrozdziale omówię architekturę aplikacji, która realizuje funkcjonalności opisane w sekcji 3.1. Poniżej zostały krótko opisane główne klasy będące częścią aplikacji, widoczne na rysunku 3.4.

LoginController jest to klasa wyświetlająca ekran logowania

ErrorController jej zadaniem jest przechwytywane wszystkich błędy w aplikacji. Po złapaniu błędu, użytkownikowi zostaje wyświetlona strona informująca, że pojawił się błąd w aplikacji, dodatkowo wszystkie błędy zostają logowane.

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

SolidityController mapuje żądania HTTP związane z przetwarzaniem plików Solidity.



Rysunek 3.4: Architektura aplikacji

Disassembler odpowiada za analizę przekazanego kodu bajtowego. W rezultacie 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 z kodu źródłowego listy funkcji, składających się z sygnatury oraz selektora. Sposób tworzenie 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 przed zapisem w bazie danych.

BytecodeService odpowiada za dopasowywanie kodu bajtowego do kontraktu. W tym celu serwis wykorzystuje opisane wyżej klasy **Disassembler** oraz **Solidity**, które w połączeniu umożliwiają wyznaczenie współczynnika dopasowania pomiędzy konkretnym plikiem, a kodem bajtowym. Szczegółowo zostało to opisane w sekcji 3.5, która znajduje się w dalszej części pracy.

SolidityFileRepository jest to część aplikacji odpowiedzialna za komunikację z bazą danych oraz mapowanie danych przechowywanych w bazie danych na obiekty. 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 baza 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.

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**, który jest wykorzystywany przez framework, należy skonfigurować dane do połączenia z bazą danych. Przykładowa zawartość pliku konfiguracyjnego została przedstawiona na listingu 3.6.

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}
```

Spring umożliwia wykorzystanie zmiennych środowiskowych w pliku konfiguracyjnym. Przykładem takiego zastosowania jest linia przedstawiona poniżej:

```
admin.login=${ADMIN_LOGIN}
```

Zmienna środowiskowa została podana w nawiasach klamrowych. Dzięki temu można zabezpieczyć aplikację przed wyciekiem wrażliwych danych, takich jak loginy i hasła do bazy danych, podczas upubliczniania kodu źródłowego na przykład w repozytorium na **GitHub**.

Po skonfigurowaniu pliku **pom.xml** oraz **application.properties**, został utworzony 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** pełniąca role stereotypu oraz informująca Springa, że ten interfejs jest wykorzystywany, w celu wykonywania operacji z bazą danych.

Kolejna adnotacja jest **@Query**. Jej parametrem jest zapytanie do bazy danych **MongoDB**, pytające 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 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.

W celu wykorzystania tego interfejsu konieczne było utworzenie klasy reprezentującej encję bazodanową w aplikacji. Na listingu 3.8 została przedstawiona taka klasa **SolidityFile**. Składa się ona z trzech atrybutów: haszu kodu źródłowego, kodu źródłowego, oraz listy funkcji znalezionych w tym kodzie źródłowym. Atrybut **sourceCodeHash** został utworzony w celu zapewnienia unikalności samego źródłowego, którego rozmiar jest zbyt duży, by nadać mu indeks bazodanowy. Tworzeniem haszu zajmuje się klasa **SolidityService** znajdująca się na rysunku 3.4 w wyższej warstwie. Dodatkowo poza zapewnieniem unikalności, hasz jest identyfikatorem kodu źródłowego w bazie danych, w związku z tym posiada on adnotację **@Id**. Atrybut ten jest odpowiednikiem identyfikatora w bazie danych. W związku z tym, że hasz jest identyfikatorem, gwarantuje on unikalność kodu źródłowego w bazie danych. Jeśli wartość pod adnotacją nie została wcześniej przypisana to moduł **Spring Data MongoDB** automatycznie wygeneruje identyfikator dla nowego obiektu i przypisze go do wskazanego przez adnotacje atrybutu.

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 }
```

Klasa ta została stworzona tak, aby uniknąć niechcianych modyfikacji pól obiektów tej klasy przez aplikację. Pola zostały ustawione na **final**, a wartości obiektu można przypisać tylko podczas jego inicjalizacji. W momencie tworzenia obiektu klasy **SolidityFile**, zostają wywoływane w konstruktorze metody sprawdzające, czy użytkownik nie wprowadził wartości **null**. Jeśli zostanie wprowadzona w konstruktorze taka wartość, wtedy aplikacja zgłasza wyjątek i informuje, że dzieje się coś niedobrego. Dzięki temu aplikacja została zabezpieczona przed niechcianym i niekontrolowanym przekazywaniem **null**i do innych części systemu.

Kolejnym elementem tej klasy są akcesory, nadpisana metoda **toString** w taki sposób, który będzie przedstawiać stan obiektu oraz nadpisane metody **equals** i **hashCode**, które są wykorzystywane przez kolekcje używane w Javie, oraz inne klasy z nimi związane.

Ponieważ klasa **SolidityFunction** jest wykorzystywana w klasie **SolidityFile**, została ona zaimplementowana w analogiczny sposób z taką różnicą, że posiada atrybuty: **selector** oraz **signature** typu **String**. W dodatku w tej klasie nie została użyta adnotacja **@Id**, ponieważ klasa nie jest wykorzystywana niezależnie w bazie danych.

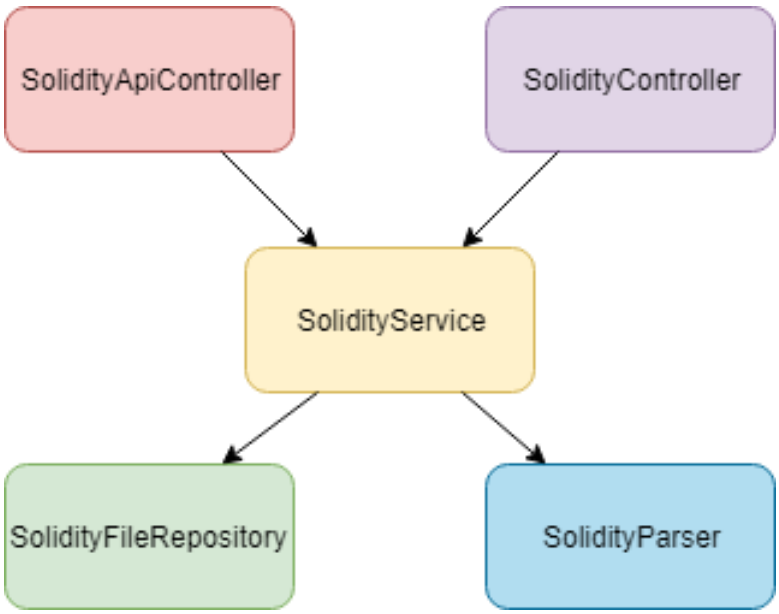
3.4 Identyfikacja sygnatur funkcji w kodzie źródłowym

Wyszukiwanie sygnatur funkcji jest rozpoczynane w momencie przesłania nowego kodu źródłowego przez użytkownika do aplikacji. Podczas wyszukiwania sygnatur funkcji są też generowane selektory funkcji, które są wykorzystywane podczas identyfikacji kodu bajtowego. Problemem podczas wyszukiwania sygnatur w

kodzie źródłowym jest to, że część sygnatur jest niejawna tzn. są one dodatkowo generowane przez kompilator **Ethereum Virtual Machine** dla publicznych atrybutów kontraktów.

W przypadku API wykorzystywana jest klasa **SolidityApiController**, którą widać na rysunku 3.5. Zadaniem tej klasy jest nasłuchiwanie adresów związanych z API, zwracanie danych w formacie **JSON** lub zwykłego tekstu oraz komunikowanie się z obiektem klasy **SolidityService**.

Jeśli użytkownik korzysta ze zwykłej strony internetowej, wtedy przetwarzaniem żądań HTTP zajmuje się **SolidityController**. Klasa ta działa podobnie do poprzedniej, z taka różnica, że ten kontroler nasłuchuje inne adresy oraz zwraca dane w postaci kodu **HTML**, zamiast w formacie **JSON** lub zwykłego tekstu.



Rysunek 3.5: Schemat zależności miedzy klasami podczas identyfikacji sygnatur funkcji

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.

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 zmienną, przy której jest adnotacja.

Ostatnia rzeczą, która pochodzi z modułu **Spring MVC** jest klasa **ResponseEntity**. Umożliwia ona zwrócenie statusu HTTP z danymi zwracanymi przez metodę. Dzięki temu framework wie jaki status ma zwrócić użytkownikowi.

Głównym celem tej metody jest zapisanie za pomocą serwisu nowego kodu źródłowego do aplikacji, w związku z tym na listingu 3.9 widać wywołanie metody **save** na atrybucie **solidityService**. Implementacja wywoływanej metody została opisana w dalszej części pracy.

Nad cała klasą **SolidityApiController** znajduje się adnotacja **@RestController**, która informuje Springa, że jest to kontroler wykorzystywany do tworzenia API Restowego.

3.4.2 Kontroler strony internetowej

SolidityControler, jest to kontroler odpowiedzialny za tworzenie strony internetowej w postaci kodu HTML jest oznaczony adnotacja **@Controller**. Działa on podobnie jak wcześniej opisany kontroler. Główną różnica jest to ze w odpowiedzi zwraca kod HTML, który jest wyświetlany jako strona internetowa u użytkownika. Do generowania kodu HTML zostal wykorzystany moduł **Thymeleaf**. To w jaki sposób dodaje się nowe moduły do projektu zostało przedstawione na przykład na listingu 3.5

Listing 3.10 przedstawia metodę przyjmująca w żądaniu HTTP kod źródłowy. Metoda ta działa podobnie jak w przypadki API, tylko w tym przypadku za pomocą **return** zwracana jest nazwa pliku zawierającego szablon HTML. Spring widząc, ze zwrócono taką nazwę przeszukuje katalog **resources**, znajdujący się w projekcie, w celu odnalezienia pasującego szablonu HTML. Istnieje możliwość przekazania danych do szablonu. W tym celu wykorzystywany jest parametr model, na którym wywoływana jest metoda **addAttribute**. Dane które są przekazywane za pomocą metody **addAtribute** pochodzą z obiektu klasy **SolidityFile**.

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

Oba kontrolery korzystają z obiektu klasy **SolidityService**. Jak widać na rysunku 3.5, klasa ta wykorzystuje: **SolidityParser** oraz opisaną już wcześniej, w sekcji 3.3, klasę **SolidityFileRepository**.

Serwis po otrzymaniu kodu źródłowego od kontrolerów, przekazuje po jednej linii do parsera, który definiuje czy w danej linii jest sygnatura funkcji. Następnie jeśli znaleziono sygnaturę to zostaje ona zwracana wraz z jej selektorem. Po sparsowaniu w serwisie wszystkich linii kodu źródłowego, jest do bazy danych zapisywany obiekt klasy **SolidityFile**, który składa się z hasza, kod źródłowego oraz listy funkcji znajdujących się w kodzie źródłowym kontraktu.

Na listingu 3.11 można zaobserwować główną metodę wyszukującą funkcję. Metoda przyjmuje pojedyncza linie kodu źródłowego, a zwraca w rezultacie obiekt klasy generycznej **Optional<SolidityFunction>**. Metoda ta sprawdza cztery możliwe przypadku kiedy powinna wykryć funkcje w kodzie źródłowym.

W celu wykrywania błędów w wyrażeniach regularnych lub metodach wyszukujących funkcji, sprawdzane jest czy w danej linii została wykryta tylko jedna funkcja. Jeśli okaże się, ze wykryto więcej niż jedna funkcje, oznacza to że któraś z tych metod fałszywie wykrywa. Co prawda obniża to wydajność aplikacji, ale w przypadku fałszywych wyników zastanie zwrócony wyjątek. Wszystkie cztery przypadki zostały opisane w dalszej części pracy.

Optional jest to klasa, która zabezpieczenia przed przekazywaniem referencji na wartość **null**. Dzięki temu że wartość **null** nie będzie przekazywana dalej, łatwiej jest zlokalizować miejsce usterki, ponieważ podczas pobierania obiektu z klasy **Optional**, jeśli jest ona pusta, zostaje wyrzucony wyjątek **NoSuchE-**

lementException. Klasa **Optional** została wprowadzona w ósmej wersji Javy [?].

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();
}
```

Wykrywanie sygnatury zadeklarowanej funkcji

Pierwszym przypadkiem, są funkcje jawnie zadeklarowane w kodzie źródłowym nie posiadające modyfikatora **internal** lub **private**. Wykrywaniem takiej funkcji zajmuje się funkcja **findFunctionSignature** widoczna na listingu 3.11. W celu wykrycia linii w kodzie źródłowym zawierającej zadeklarowaną funkcje, wykorzystywane jest wyrażenie regularne zaprezentowane poniżej:

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

W wyrażeniu zostały wykorzystane dwie grupy. Pierwsza grupa wyciąga z linii kodu źródłowego nazwę funkcji, natomiast druga parametry funkcji. Wyrażenie wyszukuje w pojedynczej linii kodu zrodlowego frazy function, po której następuje nazwa funkcji oraz w nawiasach lista parametrów.

Na listingu 3.12 widać metodę wyszukującą sygnaturę funkcji za pomocą przygotowanego 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, wyciagana jest informacja o argumentach i nazwie funkcji z podanej linii. Metoda **normalizeFunctionSignature** tworzy sygnaturę funkcji wykorzystując jej parametry i nazwę. Sygnatura składa się z nazwy, nawiasów oraz typów parametrow funkcji. Niektóre typy parametrów zostają sprowadzone do postaci kanonicznej, natomiast pozostałe typy pozostają bez zmian. Poniżej zostały przedstawione typy, które zostają sprowadzane do postaci kanonicznej:

```
uint => uint256
int  => int256
byte => bytes1
```

Po utworzeniu sygnatury funkcji zostaje wygenerowany selektor funkcji na podstawie sygnatury.

Na listingu 3.13 została przedstawiona metoda tworząca selektor. Do tworzenia hasza sygnatury została wykorzystana biblioteka **web3j**, która udostępnia metodę **sha3String**, która zwraca hasz w systemie szesnastkowym. Z hasza zostają pobrane cztery pierwsze bajty, które są selektorem funkcji.

Po pomyślnej identyfikacji zadeklarowanej funkcji zwracany jest obiekt **Optional<SolidityFunction>**, który przechowuje informacje o sygnaturze oraz selektorze funkcji.

Listing 3.13: Metoda generująca selektor funkcji

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

Wykrywanie sygnatury funkcji dla publicznych atrybutów typu mapa

Drugim przypadkiem kiedy istnieje możliwość wykrycia sygnatury funkcji w kodzie źródłowym jest zmienna publiczna typu mapa. W tym przypadku nie jest jawnie w kodzie zrodlowym pokazane ze jest tutaj sygnatura funkcji. W tej sekcji przedstawię jak wygenerować sygnaturę funkcji na podstawie takiego atrybutu.

W celu sprawdzenia czy dana linia kodu spełnia ten przypadek, 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_$0-9]*)\\s*(=.*)?\\s*;+\\s*(//.*)
    ?$
```

Jeśli dana linia nie została dopasowana do powyższego wyrażenia regularnego, wtedy zwracany jest od razu pusty **Optional**.

Pierwsza grupa w wyrażeniu regularnym oznacza typ klucza mapowania, natomiast druga typ zwracanej wartości przez mapę. Trzecia grupa jest to nazwa atrybutu. Typ klucza mapowania jest częścią sygnatury funkcji, która będzie w późniejszym etapie tworzona. Jeśli typ zwracany przez mapę jest typem tablicowym lub kolejna mapą wtedy należy dodać kolejny parametr do tworzonej sygnatury funkcji.

Na listingu 3.14 została przedstawiona pętla będąca częścią metody **findMappingGetter**. Pętla działa dopóki występuje zagnieżdżanie map lub tablica.

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 zostało wykorzystane kolejne wyrażenie regularne, które widać poniżej:

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

Po pomyślnym wyszukaniu za pomocą wyrażenia, wykorzystane zostają dwie zdefiniowane w nim grupy. Pierwsza grupa wyciąga z fragmentu napisu informacje o typie klucza, natomiast druga o typie zwracanej wartości przez mapę.

Typ klucza mapy jest dodawany do listy typów parametrów generowanej sygnatury funkcji, natomiast typ zwracany jest wykorzystany podczas sprawdzania w kolejnej iteracji pętli. Jeśli w danej iteracji pętli nie znaleziono mapy w typie zwracanym przez poprzednia mapę, wtedy będzie to ostatnia iteracja pętli.

Wszystkie typy przed dodaniem do listy parametrów sygnatury funkcji zostają najpierw sprowadzone do postaci kanoniczej.

W przypadku gdy nie wykryto mapy sprawdzane jest, 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 wykorzystana jedna grupa, która służy do wyznaczenia ilości wymiarów tablicy. Dla każdego wymiaru zostaje dodany parametr typu **uint256** do listy parametrów sygnatury. Wykrycie tablicy w typie zwracanym przez mapę jest tez równoznaczne z ostatnia iteracja pętli.

Ostatnim krokiem jest wygenerowanie sygnatury funkcji za pomocą nazwy funkcji oraz listy zgromadzonych parametrów. Na listingu 3.15 widać w jaki sposób jest formułowana sygnatura funkcji, z której następnie jest generowany 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 atrybutu tablicowych

Trzecia metoda widoczna na listingu 3.11 jest metoda **findArrayGetter**. Do wykrywania atrybutu będącego tablicą, zostało zastosowane 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. Następna grupa znajdująca się po wyrazie **public** reprezentuje nazwę atrybutu.

Wyrażenie regularne bierze pod uwagę opcjonalną deklaracje wartosci podczas deklaracji.

Jeśli przekazana do metody linia kodu nie zostanie dopasowana przy użyciu wyrażenia regularnego, wtedy metoda zwraca pusty obiekt **Optional**.

Liczba wymiaru tablicy zostaje policzona na podstawie liczby podanych nawiasów podczas deklaracji tablicy. Dla każdego wymiaru zostaje dodana do listy typów parametrów typ **uint256**.

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

Wykrywanie sygnatury funkcji dla pozostałych publicznych atrybutów

Ostatnim przypadkiem który należy rozpatrzyć podczas analizy kodu źródłowego są atrybuty, które nie są tablicami i 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 oczywiście pomija deklarację publicznych tablic lub map.

Sygnatura funkcji składa się z nazwy atrybutu wyodrębnianej za pomocą grupy zdefiniowanej w wyrażeniu regularnym. W tym przypadku sygnatura funkcji nie ma 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

Literatura: [?, ?]. TODO

3.6 Dopasowywanie implementacji na podstawie kodu bajtowego

Literatura: [?, ?]. TODO

3.7 Wykorzystane technologie

Literatura: [?, ?]. TODO

Bibliografia

[1] Network Working Group `https://www.ietf.org/rfc/rfc2616.txt`

[2] Solidity Grammar `https://github.com/ethereum/solidity/blob/develop/docs/grammar.txt`

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

Spis tabel

Spis rysunków

Spis listingów