



**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

<b>Wstęp</b>	<b>5</b>
<b>1 Ethereum</b>	<b>7</b>
1.1 Historia . . . . .	7
1.2 Opis platformy . . . . .	7
1.3 Ethereum Virtual Machine . . . . .	7
1.4 Inteligentne kontrakty . . . . .	7
<b>2 Solidity</b>	<b>9</b>
2.1 Sygnatura funkcji . . . . .	9
2.2 Selektor funkcji . . . . .	9
2.3 Generowanie akcesorów podczas kompilacji . . . . .	9
<b>3 Projekt Aplikacji</b>	<b>11</b>
3.1 Opis funkcjonalności . . . . .	11
3.1.1 Identyfikacja inteligentnych kontraktów . . . . .	12
3.1.2 Dodanie kodu źródłowego kontraktu do aplikacji . . . . .	15
3.1.3 Interfejs programistyczny aplikacji . . . . .	15
3.2 Przedstawienie architektury . . . . .	20
3.3 Połączenie z bazą danych . . . . .	21
3.4 Identyfikacja sygnatur funkcji w kodzie źródłowym . . . . .	26

3.4.1 Kontroler interfejsu programistycznego . . . . .	27
3.4.2 Kontroler strony internetowej . . . . .	28
3.4.3 Przetwarzanie kodu źródłowego . . . . .	29
3.5 Wyszukiwanie selektorów funkcji w kodzie bajtowym . . . . .	38
3.6 Dopasowywanie implementacji na podstawie kodu bajtowego . . .	38
3.7 Wykorzystane technologie . . . . .	38
<b>Bibliografia</b>	<b>39</b>
<b>Spis tabel</b>	<b>41</b>
<b>Spis rysunków</b>	<b>43</b>
<b>Spis listingów</b>	<b>45</b>

# 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 implementację kontraktu napisaną 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 technologii.

### 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 intelligentnych kontraktów

Pierwszą opcją dostępną w aplikacji jest identyfikacja intelligentnych 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 wyświetlane wszystkie dopasowania.

Po naciśnięciu w jedną z wyświetlanych implementacji użytkownik zostanie przeniesiony na stronę 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 kopowane z nim liczby identyfikujące konkretną linię w kodzie. Istnieje też możliwość pobrania kodu źródłowego ze strony z rozszerzeniem **.sol**

The screenshot shows the Ethereum Smart Contract Identifier interface. At the top, there's a header bar with the title "Ethereum Smart Contract Identifier" on the left and "Identify bytecode" and "Upload solidity" on the right. Below the header, the main title "Identify you bytecoude" is displayed. A text input field contains the bytecode: "0x9d0b3a19f91ffff...". Below the input field is a blue "Identify" button. Underneath the input field, a section titled "Top ten the most matching files" displays a table of 10 rows. Each row contains an ID number, a file hash, and a percentage value. At the bottom of the table is a blue "Get all" button. The footer of the page includes credits: "Created by: Piotr Jasina in" with a LinkedIn icon, "Idea by: Damian Rusinek in" with a LinkedIn icon, and "© 2019 Copyright: JMCS".

ID	File Hash	%
1	0x06c61b8e505d7a407af9a91bdff8085560e90a133c77ab32bde32e686f6a8d52	66,67%
2	0x0c46280ef88919ba4b3d6331b155539511f5a1deec2ae9244073c1b365d1b3f7	66,67%
3	0x431d88558b35a9da20439e21d8d9603ae6080ff4b3be5eac13f073a6e0049a1a	66,67%
4	0x2f2fc43f304aebd17f4241d59a0eaef3c35aee60a9026beab05a9f2abc710b54	66,67%
5	0xb625a734b6b296b3bcfabe0d248a3cbae88b28a791483571a1bcfd3e2116c398	42,86%
6	0x720388d3126d49859644d84d87581b6ece88dfd52c62f24e344a22089857bf18	35,29%
7	0x268944d5bffe69a06a2cdceef20d7f1fc14403cce080fcad378006566bc2a849	33,33%
8	0xfd7745e0092c6a03c21c410f24e871dfc2d131115e014675284c588c782dbb04	30,00%
9	0x6922463e7e17e961f6b72ac685524a855ead2150f2e1338f3428fe4112336fa1	28,57%
10	0xfb83bfff67aef81c79d74d9f76d368cdcec904e0fc36787f8a6d4ecbf3ab01ed1	27,27%

Rysunek 3.1: Wynik identyfikacji inteligentnego kontraktu

The screenshot shows a web-based tool for identifying Ethereum smart contracts. At the top, there's a navigation bar with the title "Ethereum Smart Contract Identifier" and links for "Identify bytecode", "Upload solidity", and "Logout". Below the navigation bar is a button labeled "Download file". The main area contains the Solidity code for the ERC20 interface, with line numbers on the left:

```
1 pragma solidity 0.4.25;
2
3 /**
4  * @title ERC20 interface
5  * @dev see https://github.com/ethereum/EIPs/issues/20
6  */
7 contract ERC20 {
8     function totalSupply() public view returns (uint256);
9
10    function balanceOf(address _who) public view returns (uint256);
11
12    function allowance(address _owner, address _spender)
13        public view returns (uint256);
14
15    function transfer(address _to, uint256 _value) public returns (bool);
16
17    function approve(address _spender, uint256 _value)
18        public returns (bool);
19
20    function transferFrom(address _from, address _to, uint256 _value)
21        public returns (bool);
22
23    function decimals() public view returns (uint256);
24
25    event Transfer(
26        address indexed from,
```

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 przesyłany 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.

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. Odpowiedź zawierająca status 500 oznacza, że wystąpił błąd na serwerze i nie udało się zwrócić kodu źródłowego.

The screenshot shows the interface of the Ethereum Smart Contract Identifier. At the top, there is a navigation bar with links: "Identify bytecode", "Upload solidity", and "Logout". Below the navigation bar, the main title is "Upload Solidity source code". There are two input methods: a file upload section ("Select file" button, "Browse" button, "Upload" button) and a text input section ("Paste source code here" text area, "Upload" button). Below these sections, a message indicates a file was uploaded: "Uploaded file:0x06c61b8e505d7a407af9a91bdff8085560e90a133c77ab32bde32e686f6a8d52". Underneath, a table titled "Found functions in file" lists the following data:

ID	Signature	Selector
1	<code>totalSupply()</code>	<code>18160ddd</code>
2	<code>renounceOwnership()</code>	<code>715018a6</code>
3	<code>getAuthorizedAddresses()</code>	<code>d39de6e9</code>
4	<code>transferFrom(address,address,uint256)</code>	<code>23b872dd</code>
5	<code>addAuthorizedAddress(address)</code>	<code>42f1181e</code>
...	...	...

Rysunek 3.3: Rezultat przesłania inteligentnego kontraktu do aplikacji

---

Listing 3.1: Żądanie wysyłane w celu pobrania kodu źródłowego

---

```
1 GET /api/sourceCode/0
    x06c61b8e505d7a407af9a91bdff8085560e90a133c77ab32bde32e686f6a8d52
    .sol HTTP/1.1
2 Host: localhost:8080
3 Accept: text/plain; charset=UTF-8
4 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/bytocode. 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 wysyłane w celu identyfikacji kontraktu za pomocą API

---

```
1 POST /api/bytocode HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 13
5
6 bytecode=60803350200fe56abcde00229&allFiles=false
```

---

Na listingu 3.2 do atrybutu **bytecode** został wprowadzony kod bajtowy kontraktu. Do zmiennej **allFiles** została wprowadzona wartość **false**, więc 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:hasło**. 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 przesyłany 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
3     x8dea780e1286d12a957d40597b9171a5187f87f6e3f8303505bc53a4453ad5b6
4     ",
5   "sourceCode": "pragma solidity ^0.4.21;\\r\\ncontract Hello {\\r\\n
6     string public message;\\r\\n     function setMessage(string
7     newMessage) public {\\r\\n       message = newMessage;\\r\\n     }\\r\\n   }",
8   "solidityFunctions": [
9     {
10       "selector": "e21f37ce",
11       "signature": "message()"
12     },
13     {
14       "selector": "368b8772",
15       "signature": "setMessage(string)"
16     }
17   ]
18 }
```

---

## 3.2 Przedstawienie architektury

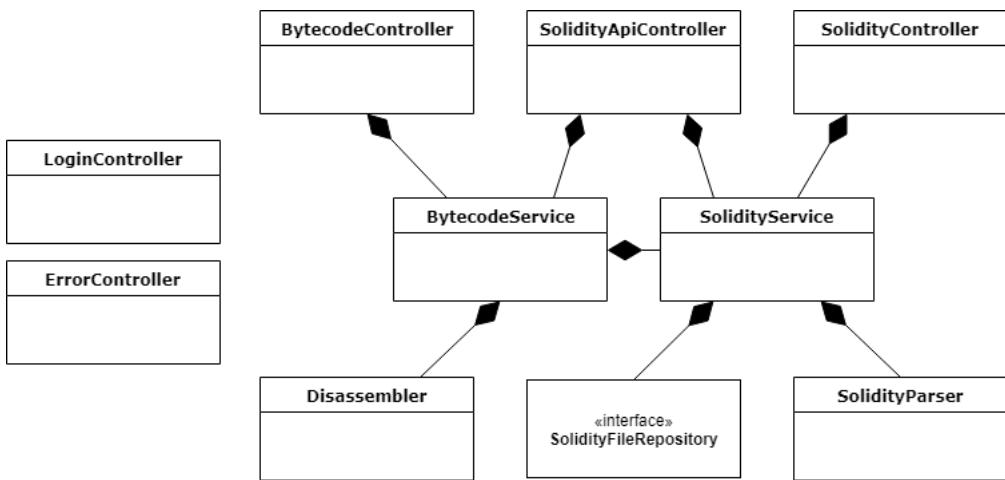
W tym podrozdziale omówię architekturę aplikacji, która realizuje funkcjonalności opisane w sekcji 3.1. Poniżej zostały krótko opisane główne klasy będące częścią aplikacji, widoczne na rysunku 3.4.

**LoginController** jest to klasa wyświetlająca ekran logowania

**ErrorController** jej zadaniem jest przechwytywanie wszystkich błędów 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 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.

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: Przykład dodania zależności w pliku pom.xml

---

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-mongodb</artifactId>
4 </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: Konfiguracja bazy danych

---

```
1 spring.data.mongodb.uri=mongodb://${ADMIN_DB_LOGIN}:${${ADMIN_DB_PASSWORD}}@ds129904.mlab.com:29904/${${DATABASE_NAME_CONTRACT}}
2 admin.login=${ADMIN_LOGIN}
3 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**. Utworzona 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<
3   SolidityFile, String> {
4
5   @Query("{\"solidityFunctions\": {$elemMatch: {\"selector\": {$
6     in: ?0}}}}}")
7   List<SolidityFile> findSolidityFilesBySelectorContainsAll(List<
8     String> functionSelector);
9
10  Optional<SolidityFile> findBySourceCodeHash(String
11    sourceCodeHash);
12}
```

---

W pierwszej linii listingu 3.7 znajduje się adnotacja **@Repository** pełniąca rolę stereotypu oraz informującą 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 adnotację atrybutu.

---

Listing 3.8: Przykład 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<
8         SolidityFunction> solidityFunctions) {
9             requireNonNull(sourceCodeHash, "Expected not-null
10                 sourceCodeHash");
11             requireNonNull(sourceCode, "Expected not-null sourceCode");
12             requireNonNull(solidityFunctions, "Expected not-null
13                 solidityFunctions");
14             this.sourceCodeHash = sourceCodeHash;
15             this.sourceCode = sourceCode;
16             this.solidityFunctions = solidityFunctions;
17         }
18
19
20     @Override
21     public String toString() {
22         return "SolidityFile{" + "sourceCodeHash='"
23             + sourceCodeHash
24             + '\'' + ", sourceCode='"
25             + sourceCode + '\''
26             + ", solidityFunctions='"
27             + solidityFunctions +
28             '\'';}
```

```
29
30     @Override
31     public boolean equals(Object o) {
32         if (this == o) return true;
33         if (!(o instanceof SolidityFile)) return false;
34         SolidityFile that = (SolidityFile) o;
35         return Objects.equals(sourceCodeHash, that.sourceCodeHash)
36             &&
37             Objects.equals(sourceCode, that.sourceCode) &&
38             Objects.equals(solidityFunctions, that.
39                 solidityFunctions);}
```

```
40
41     @Override
42     public int hashCode() {
43         return Objects.hash(sourceCodeHash, sourceCode,
44             solidityFunctions);}
```

<sup>39</sup> }

---

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 **nulli** 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 generowane selektory funkcji, które są wykorzystywane podczas identyfikacji kodu bajtowego. Problemem podczas wyszukiwania sygnatur

w kodzie źródłowym jest to, że część sygnatur jest niejawną, ponieważ są one dodatkowo generowane przez kompilator **Ethereum Virtual Machine** dla publicznych atrybutów kontraktów [4].

W przypadku API wykorzystywana jest klasa **SolidityApiController**, która widać na rysunku 3.4. 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**.

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

### 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 żądań.

Listing 3.9: Metoda kontrolera mapująca żądania POST

---

```

1 @PostMapping("/api/solidityFiles")
2 public ResponseEntity<SolidityFile> uploadFile(@RequestBody String
3     sourceCode) throws IOException {
4     requireNonNull(sourceCode, "Expected not-null sourceCode");
5     return ResponseEntity.ok(solidityService.save(sourceCode));
6 }
```

---

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

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

### 3.4.2 Kontroler strony internetowej

**SolidityController**, jest to kontroler odpowiedzialny, za tworzenie strony internetowej w postaci kodu HTML jest oznaczony adnotacją **@Controller**. Działa on podobnie jak wcześniej opisany kontroler. Główną różnicą jest to, że w odpowiedzi zwraca kod HTML, który jest wyświetlany jako strona internetowa u użytkownika. Do generowania kodu HTML został wykorzystany moduł **Thymeleaf**. W jaki sposób są dodawane nowe moduły do projektu, zostało przedstawione na przykład na listingu 3.5

Listing 3.10 przedstawia metodę przyjmującą 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, że 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 **addAttribute** 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")
3                                     String sourceCode, Model model) throws Exception {
4     requireNonNull(sourceCode, "Expected not-null sourceCode");
5     requireNonNull(model, "Expected not-null model");
6
7     SolidityFile savedSolidityFile = solidityService.save(
8         sourceCode);
9
10    model.addAttribute("solidityFileFunctions", savedSolidityFile.
11                      getSolidityFunctions());
12    model.addAttribute("solidityFileHash", savedSolidityFile.
13                      getSourceCodeHash());
14
15    return "solidity-page";
16 }

```

---

### 3.4.3 Przetwarzanie kodu źródłowego

Oba kontrolery korzystają z obiektu klasy **SolidityService**. Jak widać na rysunku 3.4, 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. Jeśli podczas parsowania linii znaleziono sygnaturę funkcji, to zostaje ona dodana wraz z selektorem do listy funkcji analizowanego pliku. Po przeanalizowaniu wszystkich linii tworzony jest obiekt **SolidityFile**, który następnie jest zapisywany do bazy danych.

Na listingu 3.11 można zaobserwować główną metodę klasy **SolidityParser** wyszukującą funkcję. Metoda przyjmuje pojedynczą linię kodu źródłowego, a w rezultacie zwraca obiekt klasy **Optional<SolidityFunction>**. Metoda sprawdza cztery możliwe przypadki, w których istnieje możliwość wykrycia funkcji 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, że wykryto więcej niż jedna funkcje, oznacza to, że jedna z tych metod fałszywie wykrywa. Takie rozwiązanie obniża to wydajność aplikacji, ale w przypadku fałszywych wyników zastanie zwrócony wyjątek. Wszystkie cztery przypadki zostały opisane w dalszej części pracy.

**Optional** jest to generyczna klasa, która zabezpiecza przed przypadkowym odwołaniem do referencji o wartości **null** oraz pozwala zabezpieczyć przed nieoczekiwany przekazywaniem wartości **null** wewnątrz aplikacji.

---

Listing 3.11: Metoda wyszukujaca sygnatury funkcji

---

```
1 Optional<SolidityFunction> findFunctionInLine(String line) {
2     List<Optional<SolidityFunction>> functions =
3         Stream.of(
4             findFunctionSignature(line),
5             findMappingGetter(line),
6             findArrayGetter(line),
7             findNormalVariableGetter(line)
8         ).filter(Optional::isPresent).collect(toList());
9
10    if (functions.size() > 1) {
11        throw new IllegalStateException("Expected only one function
12            , but found :" + functions.size());
13    } else if (functions.size() == 1) {
14        return functions.listIterator().next();
15    }
16    return Optional.empty();
```

<sup>16</sup> }

### 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 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
2   line) {
3   Matcher matcher = FUNCTION_PATTERN.matcher(line);
4   if (matcher.find()) {
5     String functionName = matcher.group(FUNCTION_NAME_GROUP_ID)
6       ;
7     String functionArguments = matcher.group(
8       FUNCTION_ARGUMENTS_GROUP_ID);
9     String functionSignature = normalizeFunctionSignature(
10       functionName, functionArguments);
```

```

7     String functionSelector = getFunctionSelector(
8         functionSignature);
9     return Optional.of(new SolidityFunction(functionSelector,
10            functionSignature));
11    }

```

---

Po wykryciu funkcji za pomocą wyrażenia regularnego wyciągana jest informacja o argumentach oraz nazwie funkcji z przekazanej linii. Metoda **normalizeFunctionSignature** tworzy sygnaturę funkcji za pomocą listy parametrów i nazwy. Sygnatura składa się z nazwy, nawiasów oraz typów parametrów 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. Na listingu 3.13 została przedstawiona metoda tworząca selektor. Sygnatura funkcji zostaje haszowana za pomocą **SHA3**, następnie z hasza pobierane są cztery pierwsze bajty, które są selektorem funkcji. Metoda **sha3String** zwraca hasz w systemie szesnastkowym w postaci napisu. Do tworzenia hasza została wykorzystana biblioteka **web3j**, która udostępnia metodę **sha3String**.

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

### 3.4. IDENTYFIKACJA SYGNATUR FUNKCJI W KODZIE ŹRÓDŁOWYM33

Listing 3.13: Metoda generująca selektor funkcji

---

```
1 private String getFunctionSelector(String
2     normalizedFunctionSignature) {
3     return sha3String(normalizedFunctionSignature).substring(2, 10)
4     ;
5 }
```

---

#### Wykrywanie sygnatury funkcji dla publicznych atrybutów typu mapa

Drugim przypadkiem jest atrybut publiczny typu mapa. Teraz nie jest jawnie pokazane w kodzie źródłowym, że istnieje tutaj sygnatura funkcji, ponieważ jest ona generowana przez kompilator. 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_]\\s*[a-zA-Z_\\$0-9]*)\\s*(=.*)?\\s*;+\\s*(//.*)
?$_
```

---

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.
7             group(MAPPING_KEY_GROUP_ID));
8         canonicalMappingKeys.add(canonicalArgument);
9         mappingValue = mappingMatcher.group(MAPPING_VALUE_GROUP_ID)
10        ;
11        continue;
12    }
13    if (arrayMatcher.find()) {
14        String arrayValue = arrayMatcher.group(ARRAY_VALUE_GROUP_ID
15        );
16        int dimensionCount = getArrayDimensionCount(arrayValue);
17        for (int i = 0; i < dimensionCount; i++) {
18            canonicalMappingKeys.add(CANONICAL_ARRAY_KEY_TYPE);
19        }
20    }
21    break;
22}

```

---

Do wyszukiwania map w typie zwracanym przez mapę zostało wykorzystane kolejne wyrażenie regularne, które widać poniżej:

---

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

---

Po wykryciu mapy za pomocą wyrażenia, wykorzystane zostają dwie zdefiniowane w nim grupy. Pierwsza grupa wyciąga z fragmentu deklaracji mapy informacje o typie klucza, natomiast druga o typie zwracanej przez nią wartości.

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

### 3.4. IDENTYFIKACJA SYGNATUR FUNKCJI W KODZIE ŹRÓDŁOWYM35

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 wyznaczania 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 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, 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**

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 inicjalizację wartości podczas deklaracji.

Liczba wymiarów tablicy zostaje policzona na podstawie liczby podanych nawiasów podczas deklaracji tablicy. Dla każdego wymiaru zostaje dodany typ **uint256** do listy typów parametrów. 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 wykrywania sygnatury funkcji, są wszystkie atrybuty 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 oczywiście pomija deklarację publicznych tablic lub map.

### 3.4. IDENTYFIKACJA SYGNATUR FUNKCJI W KODZIE ŹRÓDŁOWYM37

Sygnatura funkcji składa się z nazwy atrybutu wyodrębnianej za pomocą grupy zdefiniowanej w wyrażeniu regularnym. W tym przypadku sygnatura funkcji nie ma żadnych parametrów, więc 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 wyszukująca sygnature funkcji dla atrybutów niebędących mapą ani tablicą

---

```
1 private Optional<SolidityFunction> findNormalVariableGetter(String
2     line) {
3     Matcher matcher = NORMAL_VARIABLE_PATTERN.matcher(line);
4     if (matcher.find()) {
5         LOGGER.info("Found public normal variable: {}", line);
6         String variableName = matcher.group(
7             NORMAL_VARIABLE_NAME_GROUP_ID);
8
9         String functionSignature = variableName + "()";
10        String functionSelector = getFunctionSelector(
11            functionSignature);
12
13        return Optional.of(new SolidityFunction(functionSelector,
14            functionSignature));
15    }
16    return Optional.empty();
17 }
```

---

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

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



## Spis tabel



# Spis rysunków

3.1	Wynik identyfikacji inteligentnego kontraktu . . . . .	13
3.2	Podgląd implementacji . . . . .	14
3.3	Rezultat przesłania inteligentnego kontraktu do aplikacji . . . . .	16
3.4	Architektura aplikacji . . . . .	20



# Spis listingów

3.1	Żądanie wysyłane w celu pobrania kodu źródłowego . . . . .	17
3.2	Żądanie wysyłane w celu identyfikacji kontraktu za pomocą API . . . . .	17
3.3	Przesyłanie kodu źródłowego za pomocą API . . . . .	18
3.4	Przykładowa odpowiedz w formacie JSON . . . . .	19
3.5	Przykład dodania zależności w pliku pom.xml . . . . .	22
3.6	Konfiguracja bazy danych . . . . .	22
3.7	Stworzenie repozytorium za pomocą Spring Data MongoDB . . . . .	23
3.8	Przykład klasy wykorzystywanej przez Spring Data MongoDB . . . . .	24
3.9	Metoda kontrolera mapująca żądania POST . . . . .	27
3.10	Przechwytywanie żądania o dodanie nowego kodu źródłowego . . . . .	29
3.11	Metoda wyszukujaca sygnatury funkcji . . . . .	30
3.12	Metoda odbierająca żądanie o zapisanie kodu źródłowego na serwerze . . . . .	31
3.13	Metoda generująca selektor funkcji . . . . .	33
3.14	Pętla wyszukująca tablicę lub mapę w typie zwracanym przez mapę . . . . .	33
3.15	Tworzenie SolidityFunction na podstawie nazwy funkcji i listy typów . . . . .	35
3.16	Metoda wyszukujaca sygnature funkcji dla atrybutów niebędących mapą ani tablicą . . . . .	37

