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

Literatura: [3, ?]. TODO

1.2 Opis platformy

Literatura: [3, ?]. TODO

1.3 Ethereum Virtual Machine

Literatura: [3, ?]. TODO

1.4 Inteligentne kontrakty

Literatura: [3, ?]. TODO

Rozdział 2

Solidity

2.1 Sygnatura funkcji

Literatura: [3, ?]. TODO

2.2 Selektor funkcji

Literatura: [3, ?]. TODO

2.3 Generowanie akcesorów podczas kompilacji

Literatura: [3, ?]. TODO

Rozdział 3

Projekt Aplikacji

Celem mojej pracy licencjackiej było stworzenie umożliwiającej identyfikację inteligentnych kontraktów wykorzystywanych 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 bazując na bazie danych aplikacji.

Aplikacja realizująca cel pracy została stworzona przy wykorzystaniu frameworka Spring Boot, jest to aplikacja internetowa, dzięki czemu można ja wykorzystać wielu urządzeniach bez konieczności instalacji. W celu przechowywania danych wykorzystano nierelacyjną bazę danych MongoDB.

Poniżej opisałem działanie aplikacji wraz ze szczegółowym opisem funkcjonalności, architektury oraz wykorzystanych technologi.

3.1 Opis funkcjonalności

Na stronie głównej znajduje się opis aplikacji wraz z aktualna liczba kodów źródłowych znajdujących się w bazie danych oraz podstawowe definicje związane z aplikacją. Cała aplikacji udostępnia trzy główne funkcjonalności: identyfikację

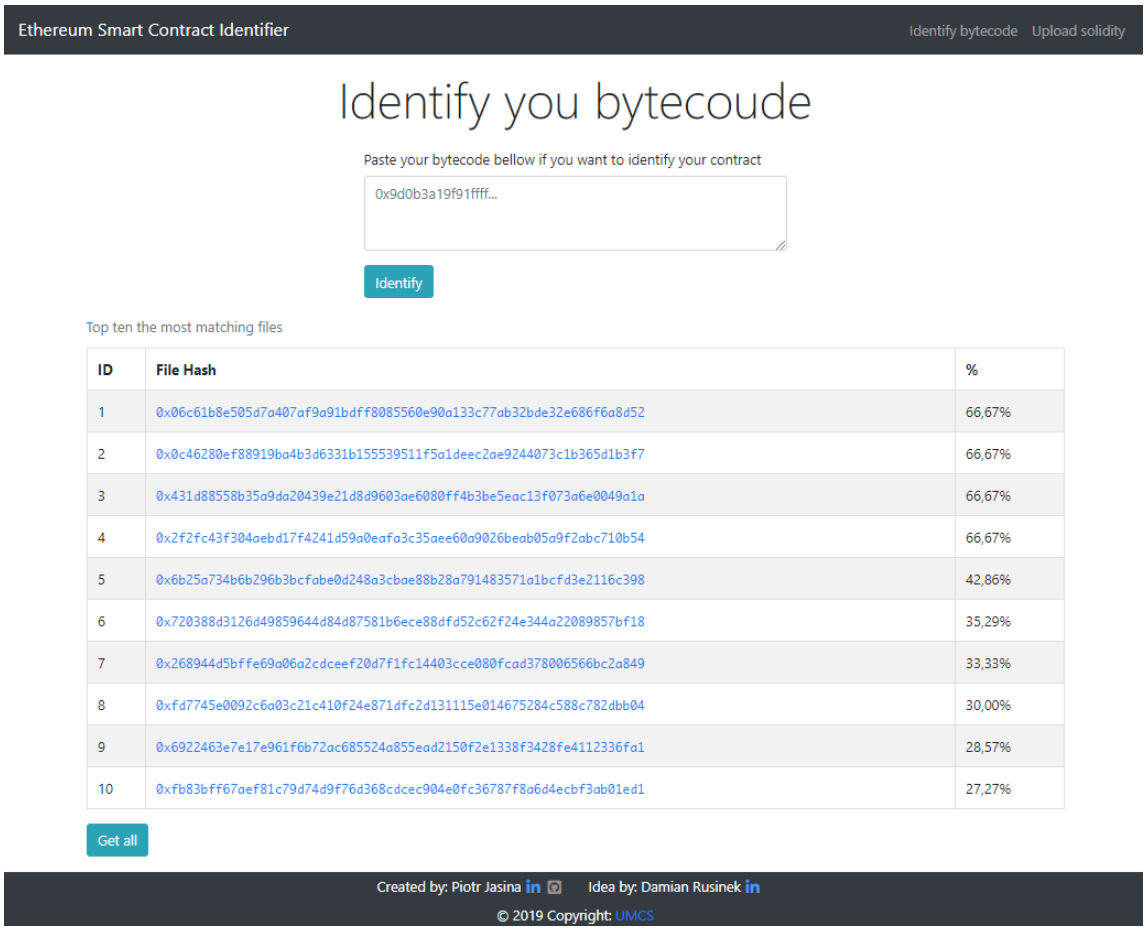
inteligentnych kontraktu, wprowadzanie plików źródłowych kontraktów do aplikacji oraz interfejs programistyczny aplikacji. Wszystkie funkcjonalności zostały opisane poniżej

3.1.1 Identyfikacja inteligentnych kontraktów

Pierwsza opcją dostępną w aplikacji jest identyfikacja inteligentnych kontraktów. Identyfikacje kontraktu można rozpocząć będąc na stronie główniej lub na podstronie dedykowanej specjalnie identyfikacji kontraktów. Zarówno na stronie głównej, jak i na podstronie znajduje się pole w którym można wprowadzić kod bajtowy. Po wprowadzeniu danych użytkownik zatwierdza je w obu przypadkach klikając przycisk **Identify**. Natomiast, jeśli użytkownik chce udać się na podstronę, należy wybrać przycisk w menu o nazwie **Identify bytecode**, następnie użytkownik zostanie przekierowany na podstronę dedykowaną identyfikacji kontraktów. W przypadku wprowadzenia kodu bajtowego na stronie głównej zostaniemy również przekierowani na podstronę z taka różnica, ze pojawią się od razu wyniki identyfikacji kontraktu. Zarówno na stronie głównej, jak i podstronie dedykowanej identyfikacji, użytkownik jest zobowiązany wprowadzać kod bajtowy w szesnastkowym systemie liczbowym, w innym wypadku identyfikacja nie przejdzie prawidłowo.

Po wprowadzeniu danych i zatwierdzeniu ich przyciskiem **Identify**, aplikacja rozpoczyna proces analizy wprowadzonego kodu bajtowego oraz wyszukiwane są najbardziej prawdopodobne implementacje posortowane malejąco według współczynnika dopasowania. W rezultacie jak możemy zobaczyć na rysunku 3.1 użytkownik otrzymuje listę dziesięciu najbardziej prawdopodobnych implementacji.

Mimo że domyślnie jest wyświetlanych tylko dziesięć najbardziej prawdopodobnych implementacji, istnieje też możliwość pobrania wszystkich wyników identyfikacji kontraktu używając przycisku **Get all**. Jak widać na rysunku 3.1,



Rysunek 3.1: Wynik identyfikacji inteligentnego kontraktu

przycisk **Get all** znajduje się pod pierwszą dziesiątka wyników. Po naciśnięciu przycisku strona zostanie załadowana ponownie wraz z pełną listą haszy plików i ich współczynnikami dopasowania.

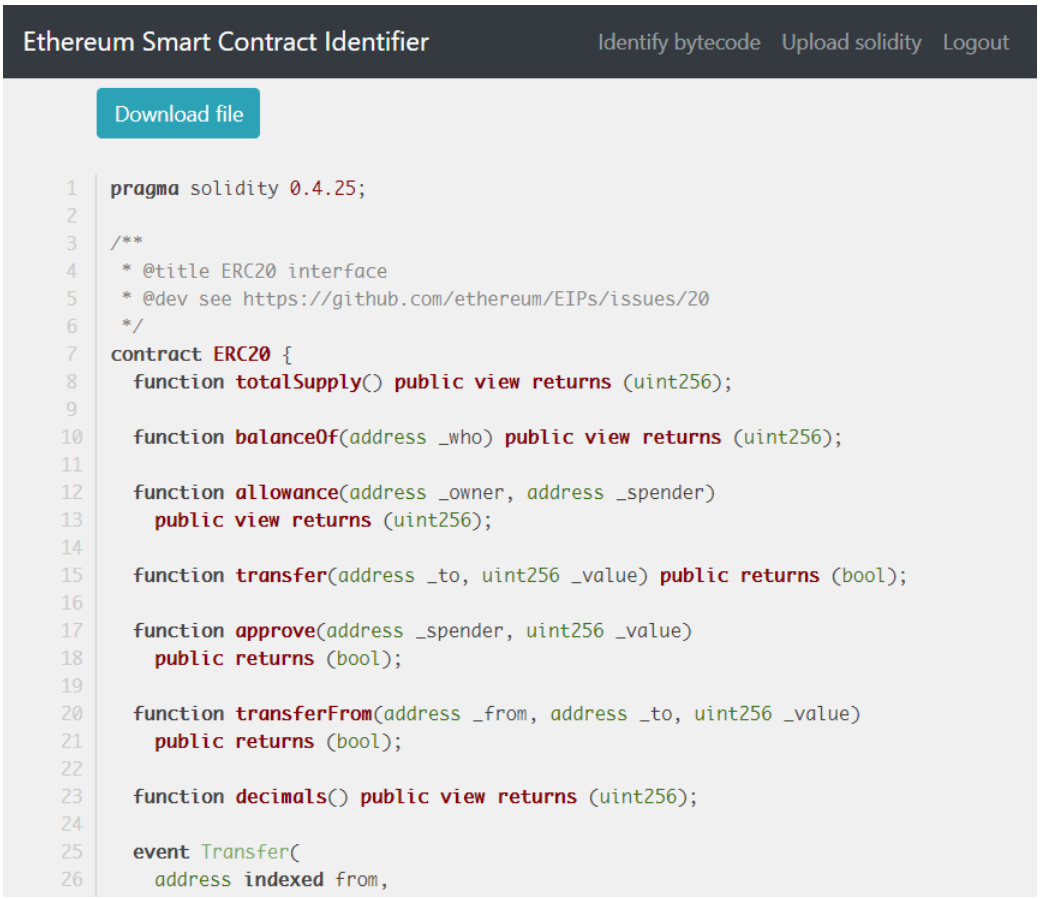
Po naciśnięciu w jedną z wyświetlanych implementacji, użytkownikowi pojawi się w nowej karcie przeglądarki podstrona umożliwiająca podgląd implementacji.

Jak widać na rysunku 3.2 kod źródłowy kontraktu jest wyświetlany ze specjalnie przygotowanym dla języka Solidity podświetleniem składni przygotowanym, natomiast po lewej stronie można zobaczyć przygotowaną numeracje wierszy, która ma za zadanie ułatwić nawigację po kodzie źródłowym na stronie internetowej. 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.

3.1.2 Dodanie kodu źródłowego kontraktu do aplikacji

Kolejna funkcjonalnością dostępną dla użytkownika jest możliwość dodania własnego kodu źródłowego kontraktu napisanego w języku Solidity. Opcja ta umożliwia użytkownikowi uzupełnienie aktualnej bazy danych o kolejne kody źródłowe inteligentnych kontraktów. W rezultacie zgromadzenia dużej ilości implementacji w bazie danych, wszyscy pozostali użytkownicy maja większą szansę na precyzyjną identyfikacje kontraktu. W celu wykorzystania tej funkcjonalności użytkownik musi zalogować się za pomocą panelu logowania. Zakładając, że osoba korzystająca z tej części aplikacji posiada już odpowiednie uprawnienia to po naciśnięciu przycisku **Upload solidity**, zostanie przekierowana na podstronę, na której ma możliwość wprowadzenia kodu źródłowego. Zostały utworzone dwie możliwości wprowadzania kodów źródłowych, które opiszę poniżej.

Pierwszą sposobem jest przesłanie do aplikacji pliku zawierającego implementacje kontraktu napisana w języku Solidity. W tym przypadku użytkownik po-



Rysunek 3.2: Podgląd implementacji

winien kliknąć przycisk **Browse**, który umożliwi mu wybranie za pomocą przeglądarki internetowej konkretnego pliku znajdującego się na dysku lokalnym, a następnie zatwierdzić go przyciskiem **Upload** znajdującym się obok wcześniej wspomnianego przycisku.

Innym sposobem na przesłanie kodu źródłowego do aplikacji jest wklejenie kodu źródłowego bezpośrednio do formularza znajdującego się po prawej części strony internetowej. Ta opcja została utworzona w celu zapewnienia użytkownikowi większej elastyczności i komfortu w korzystaniu z aplikacji. Przykładowo podczas korzystania z aplikacji, użytkownik może bezpośrednio skopiować kod źródłowy, który jest w dowolnym innym źródle tekstowym np. innej stronie internetowej i wkleić go bezpośrednio do aplikacji bez konieczności tworzenia pliku tymczasowego.

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.

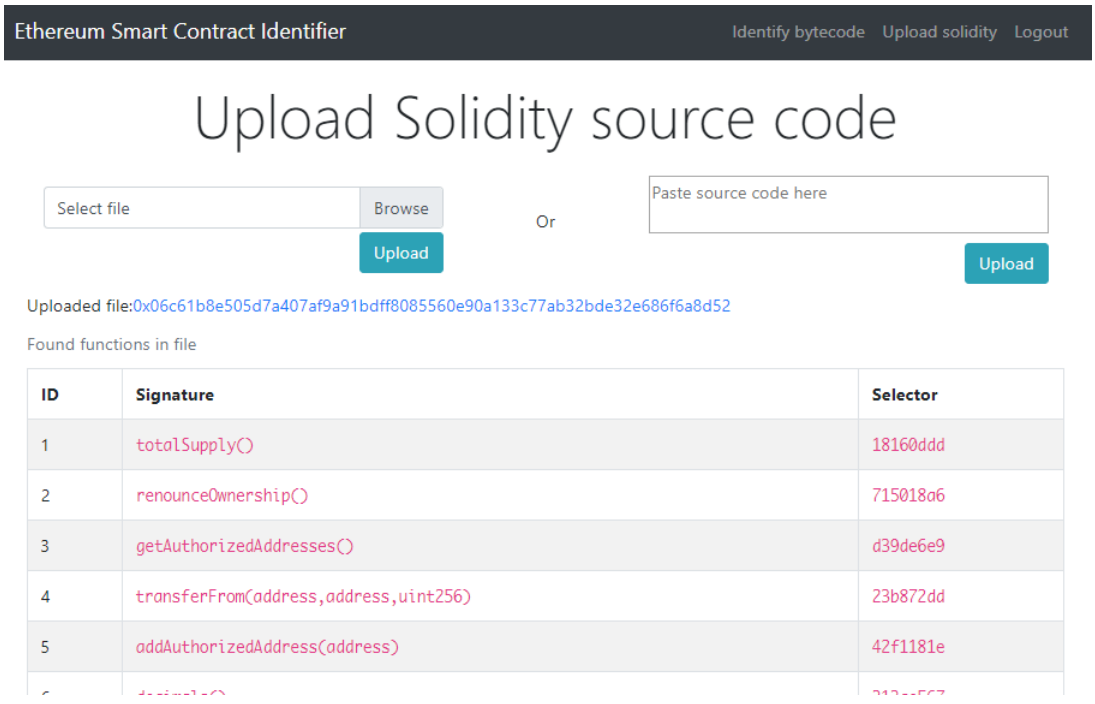
Użytkownik ma możliwość przeglądania kodu źródłowego, który wysłał na serwer. W tym celu należy nacisnąć na hasz pliku wyświetlany poniżej formularza dodawania, następnie użytkownik zostanie przeniesiony na stronę na której może zobaczyć dodany przez siebie kod źródłowy z numeracją linii oraz podświetleniem składni.

Na dolnej części rysunku 3.3 znajduje się tabela z funkcjami wyszukanymi w implementacji. W kolumnie **Signature** znajdują się wszystkie sygnatury funkcji, natomiast w kolumnie **Selector** odpowiadajace im selektory funkcji.

3.1.3 Interfejs programistyczny aplikacji

Trzecia funkcjonalnością aplikacji jest interfejs programistyczny. Umożliwia on tworzenie przez innych użytkowników własnego oprogramowania bazując na istniejącej aplikacji. Dzięki temu można wykorzystać mechanizmy zaimplementowane w aplikacji w celu rozszerzenia ich w innej aplikacji lub w celu zautomatyzowania niektórych procesów bez wykorzystania GUI (ang. graphical user interface) aplikacji.

W celu skorzystania z interfejsu programistycznego należy utworzyć żądanie HTTP. Za pomocą żądania użytkownik aplikacji ma możliwość dostarczenia na serwer nowego kodu źródłowego, pobrania istniejącego kodu źródłowego z bazy danych aplikacji oraz identyfikację kontraktu. Zarówno zadania jak i odpowiedzi są charakterystyczne dla protokołu HTTP.



Rysunek 3.3: Rezultat przesłania inteligentnego kontraktu do 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. W tym przypadku użytkownik, który chce przesłać nowy plik do aplikacji musi najpierw przejść proces autoryzacji, a następnie wykonać wybrane żądanie HTTP.

Pobieranie informacji o kodzie źródłowym z API

Aplikacja umożliwia pobranie kodu źródłowego wykorzystując interfejs programistyczny. Użytkownik API musi posiadać hasz pliku, który jest umieszczany w adresie żądania HTTP. Hasz pliku może otrzymać w odpowiedzi na identyfikacje kodu bajtowego lub po przesłaniu nowej implementacji na serwer. Przykładowe 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 zawierajaca status 500 oznacza ze wystąpił błąd na serwerze i nie udało się zwrócić kodu źródłowego.

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 wyslać żadanie pod adres /api/bytecode. W ciele żądanie jest wymagane od użytkownika podanie dwóch atrybutów o nazwach: **bytecode** oraz **allFiles**. Wartości są zakodowane poprzez przypisanie do nazwy atrybutu wartość oraz połączenie wszystkich atrybutów za pomocą ampersandu. Poprawny przykład żądania można zobaczyć na listingu 3.2.

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

```
1 POST /api/bytecode HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 13
5
6 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 dziesięć najbardziej prawdopodobnych implementacji, natomiast jeśli użytkownik chce zobaczyć jak wyglądają pozostałe dopasowania należy ustawić tą zmienną na **true**. W żądaniu został wprowadzony nagłówek **Content-Length**, określający długość przesyłanych danych, natomiast **Content-Type** oznacza rodzaj przesyłanych danych.

Jeśli wszystko pójdzie pomyślnie użytkownik otrzyma status HTTP 200 oraz listę składająca 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 nalezy do żądania dodać nagłówek **Authorization**. Składnia nagłówka wygląda następująco: **Authorization: <type> <credentials>**. W miejscu **type** należy podać Basic, a w miejscu **credentials** zakodowane za pomocą Base64 według schematu **login:haslo** dane do autoryzacji. Na przykładzie z listingu 3.3 został przesłany kod źródłowy wykorzystując 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 powinien otrzymac status HTTP 200. Poza statusem, zostaje przesłany przez niego 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

```
{
    "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 omówię architekturę aplikacji która realizuje opisane w sekcji 3.1 funkcjonalności. Na wstępie pokrótce opisze elementy systemu, a te bardziej złożone w swoim działaniu opiszę w podrozdziałach.

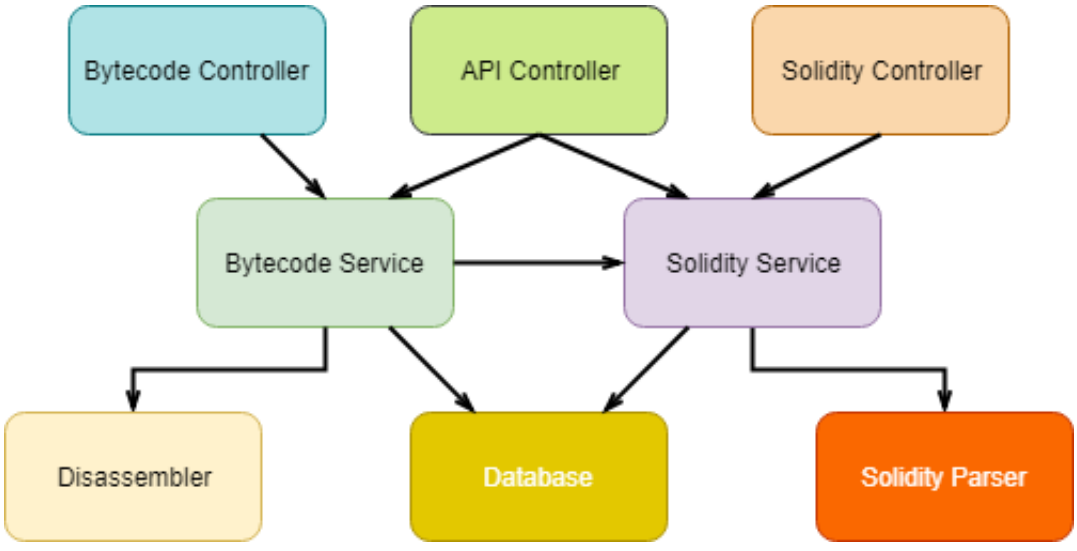
Jak widać na rysunku 3.4 pierwsza warstwa aplikacji składa się z kontrolerów, które obsługują połączenie, miedzy serwerem, a przeglądarką internetowa. Celem kontrolerów jest obsługa żądań HTTP i wywoływanie określonych akcji na pozostałej części systemu oraz wyświetlanie użytkownikowi wyników w postaci kodu HTML. Do tworzenia strony internetowej został wykorzystany moduł Spring Thymeleaf oraz Bootstrap

Disassembler - Jest to klasa odpowiedzialna analizę przekazanego kodu bajtowego. W rezultacie zwraca listę instrukcji zawartych w kodzie. Szczegółowe działanie tej klasy zostało opisane w dalszej części pracy, w sekcji 3.4 dotyczącej wyszukiwania selektorów funkcji w kodzie bajtowym.

Solidity Parser - Odpowiada wyciągniecie 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 3.5

Solidity Service - jest to klasa odpowiedzialna za odczytywanie danych z bazy danych oraz za przygotowanie przesłanych danych przez użytkownika przed zapisem w bazie danych.

Bytecode Service - odpowiada za dopasowywanie kodu bajtowego do kontraktu. W tym celu wykorzystuje opisane wyżej klasy Disassembler oraz Solidity, które w połączeniu umożliwiają wyznaczenie współczynnika dopasowania pomiędzy konkretnym plikiem, a kodem bajtowym. Sposób dopasowywania oraz wy-



Rysunek 3.4: Architektura aplikacji

znaczania współczynnika dopasowania został opisany w sekcji 3.5 która znajduje się w dalszej części pracy.

Database - jest to cześć aplikacji odpowiedzialna za komunikację z bazą danych oraz mapowanie danych przechowywanych w bazie danych na obiekty Javowe. W tym fragmencie aplikacji interfejs role pośrednika miedzy serwisem, a realną bazą danych, pełni interfejs SolidityFileRepository, który wykorzystuje moduł Spring Data MongoDB. Rola implementacji tego interfejsu spoczywa na frameworku Spring. Szczegoly łączenia z bazą danych zostały opisane w sekcji 3.2.1

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 pomocż narzędzia Apache Maven, należy dodać do pliku **pom.xml** wykorzystywany moduł w projekcie.

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

Listing 3.5: Przyklad 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 Spring, należy skonfigurować dane do połączenia z bazą

danych. Przykładowa zawartość pliku konfiguracyjnego została przedstawiona na listingu 3.6.

Listing 3.6: Kofiguracja 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}
```

Dzięki temu można zabezpieczyć aplikacje 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 wykorzystywane zapytanie do bazy danych MongoDB, pytające o listę plików, w których znajdują się przekazane przez użytkownika selektory funkcji. W ten sposób 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 danych przez metodę.

W celu wykorzystania tego interfejsu konieczne było utworzenie klasy reprezentującej encję bazodanowa w aplikacji. Na listingu 3.8 została przedstawiona taka klasa **SolidityFile**. Składa się ona z trzech atrybutów, z haszu kodu źródłowego, kodu źródłowego, oraz listy funkcji znalezionych w tym kodzie źródłowym. Atrybut sourceCodeHash został utworzony ze względu na brak możliwości zapewnienia unikalności samego kodu źródłowego, którego rozmiar jest zbyt duży by przetwarzać w bazie danych. Tworzeniem haszu zajmuje się klasa **SolidityService** będąca 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 adnotacje **@Id**, która informuje że atry-

but ten jest identyfikatorem w bazie danych. W związku z tym ze hasz jest identyfikatorem, jest on też unikalny, a unikalny hasz daje unikalny kod źródłowy. Jeśli wartość pod adnotacja 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
3      @Id
4      private final String sourceCodeHash;
5      private final String sourceCode;
6      private final Set<SolidityFunction> solidityFunctions;
7
8      SolidityFile(String sourceCodeHash, String sourceCode, Set<
           SolidityFunction> solidityFunctions) {
9          requireNonNull(sourceCodeHash, "Expected not-null
               sourceCodeHash");
10         requireNonNull(sourceCode, "Expected not-null sourceCode");
11         requireNonNull(solidityFunctions, "Expected not-null
               solidityFunctions");
12         this.sourceCodeHash = sourceCodeHash;
13         this.sourceCode = sourceCode;
14         this.solidityFunctions = solidityFunctions;
15     }
16
17     public String getSourceCodeHash() { return sourceCodeHash; }
18     public String getSourceCode() { return sourceCode; }
19     public Set<SolidityFunction> getSolidityFunctions() { return
           solidityFunctions; }
20
21     @Override
22     public String toString() {
23         return "SolidityFile{" + "sourceCodeHash='"
24                 + sourceCodeHash
25                 + '\'' + ", sourceCode='" + sourceCode + '\''
```

```
26                 + ", solidityFunctions=" + solidityFunctions +
                       '}';}
27
28     @Override
29     public boolean equals(Object o) {
30         if (this == o) return true;
31         if (!(o instanceof SolidityFile)) return false;
32         SolidityFile that = (SolidityFile) o;
33         return Objects.equals(sourceCodeHash, that.sourceCodeHash)
               &&
34                 Objects.equals(sourceCode, that.sourceCode) &&
35                 Objects.equals(solidityFunctions, that.
                       solidityFunctions);}
36
37     @Override
38     public int hashCode() {
39         return Objects.hash(sourceCodeHash, sourceCode,
               solidityFunctions);}
40 }
```

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 związku z tym, że plik solidity zawsze ma hasz, kod źródłowy oraz listę funkcji w momencie zapisywania do bazy danych, zostały w konstruktorze wykorzystane metody sprawdzające, czy przypadkiem 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 w inne części systemu.

Kolejnym elementem tej klasy są akcesory, nadpisana metoda **toString** w taki sposób, który będzie zrozumiale przedstawiać stan obiektu oraz nadpisane metody **equals** i **hashCode**, które są wykorzystywane przez kolekcje, wykorzystywane 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 ona atrybuty: **selector** oraz **signature** typu **String**. W dodatku w tej klasie nie została użyta adnotacja **@Id**, ponieważ nie jest ona używana jako niezależny obiekt w bazie danych, tylko jest zawsze częścią obiektu klasy **SolidityFile**.

3.4 Identyfikacja sygnatur funkcji w kodzie źródłowym

Wyszukiwanie sygnatur funkcji jest rozpoczynane w momencie przesłania nowego kodu źródłowego przez uzytkownika do aplikacji. Podczas wyszukiwania sygnatur funkcji są też generowane selektory funkcji, które są wykorzystywane podczas identyfikacji kodu bajtowego, który również posiada w sobie selektory funkcji. Dzięki temu cała aplikacja jest w stanie dopasować kod bajtowy do konkretnej implementacji. 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**.

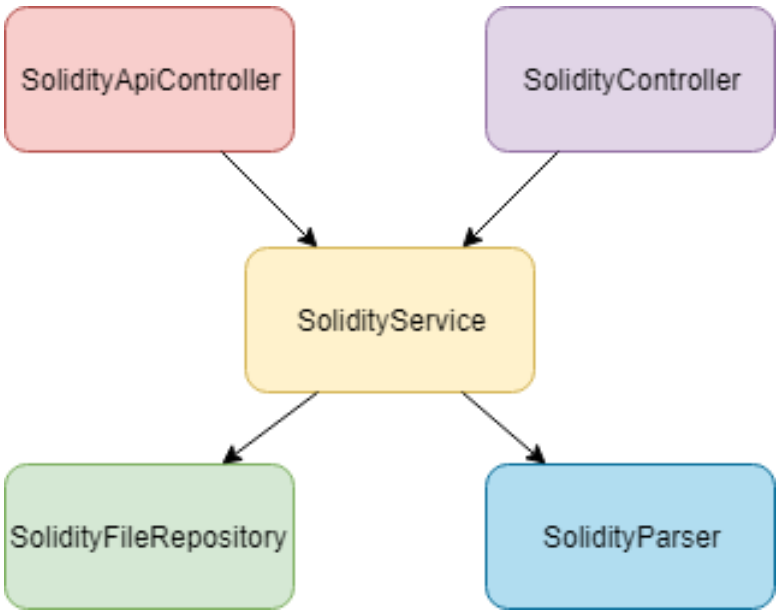
Jeśli użytkownik korzysta ze zwykłej strony internetowej, wtedy przetwarzaniem żądań HTTP zajmuje się **SolidityController**. Klasa ta działa podobnie do poprzedniej, z taka różnica, że ten kontroler nasłuchuje inne adresy oraz zamiast zwracać dane w formacie JSON lub zwykłego tekstu, zajmuje się generowaniem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

3.4.1 Kontroler interfejsu programistycznego

Na listingu 3.9 została przedstawiona metoda kontrolera **SolidityApiController**. 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 }
```



Rysunek 3.5: Schemat zależności miedzy klasami podczas identyfikacji sygnatur funkcji

Kolejną wykorzystaną adnotacją jest **@RequestBody**. Adnotacja ta informuje framework, o tym żeby ciało żądania HTTP było umieszczona 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 podrozdziału.

Nad cała klasą **SolidityApiController** znajduje się adnotacja **@RestController**, ktora informuje Springa, że jest to kontroler wykorzystywany do tworzenia API Restowego. Dzięki tej adnotacji domyślnie kontroler będzie na przykład starał się zwrócić dane w formacie JSON.

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 regularych 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 **NoSuchElementException**. Klasa **Optional** została wprowadzona w ósmej wersji Javy [2].

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, but found :" + functions.size());
12        } else if (functions.size() == 1) {
13            return functions.listIterator().next();
14        }
15        return Optional.empty();
16    }
```

Wykrywanie sygnatury zadeklarowanej funkcji

Pierwszym przypadkiem, są funkcje jawnie zadeklarowane w kodzie źródłowym nie posiadające modyfikatora **internal** lub **private**. W celu wykrycia takich funkcji wykorzystane zostało wyrażenie regularne zaprezentowane poniżej:

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

W wyrażeniu tym zostały wykorzystane dwie grupy. Pierwsza grupa wyciąga z linii kodu źródłowego nazwę funkcji, natomiast drugą sygnaturę funkcji. 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    }
```

Wykrywanie sygnatury funkcji dla publicznych atrybutów typu mapa

TODO

Wykrywanie sygnatury funkcji dla publicznych atrybutu tablicowych

TODO

Wykrywanie sygnatury funkcji dla pozostałych publicznych atrybutów

TODO

3.5 Wyszukiwanie selektorów funkcji w kodzie bajtowym

Literatura: [3, ?]. TODO

3.6 Dopasowywanie implementacji na podstawie kodu bajtowego

Literatura: [3, ?]. TODO

3.7 Wykorzystane technologie

Literatura: [3, ?]. TODO

Bibliografia

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

[2] Oracle `https://docs.oracle.com/javase/8/docs/`.

[3] Bibliografia 2. *Nazwa*.

Spis tabel

Spis rysunków

Spis listingów