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

Tematem pracy dyplomowej jest opracowanie narzędzia umożliwiającego identyfikacje inteligentnych kontraktów w sieci Ethereum na podstawie kodu bajtowego. Niniejsza praca została podzielona na trzy części - wstęp teoretyczny z wiedzy dotyczącej platformy Ethereum, informacje dotyczące języka Solidity oraz cześć praktyczną omawiająca implementacje rozwiązania.

Pierwszy rozdział zawiera informacje dotyczące powstania sieci Ethereum, zalet wykorzystania inteligentnych kontraktów oraz opisuje wykorzystanie maszyny wirtualnej jako cześć sieci Ethereum.

Drugi rozdział omawia specyfikę języka Solidity, sposób generowania akcesorów do zmiennych podczas kompilacji oraz metody interakcji z kontraktami. Podczas omawiania interakcji z inteligentnymi kontraktami została przybliżona tematyka generowania selektorów funkcji wykorzystywanych podczas implementacji rozwiązania.

Ostatni rozdział pracy przedstawia funkcjonalności utworzonej aplikacji, sposób wykrywania sygnatur funkcji w kodzie źródłowym oraz wyszukiwania selektorów funkcji w kodzie bajtowym. Po zaimplementowaniu aplikacji zostały przeprowadzone testy metod wyznaczania współczynnika dopasowania implementacji oraz skuteczności aplikacji pod kątem dopasowania implementacji do kodu bajtowego.

Rozdział 1

Ethereum

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

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

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

1.1 Historia

Początki Ethereum zostały opisane przez programistę Vitalika Buterina w 2013 roku. Vitalik w październiku 2013 roku pracował nad nową kryptowalu-

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

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

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

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

Na początku 2015 roku odbyły się audyty bezpieczeństwa przed uruchomieniem, zorganizowane przez między innymi Jutta Steinera. Audyty dotyczyły przede wszystkim implementacji w Go i C++. Przeprowadzony został też mniejszy audyt dotyczący implementacji Vilika nazwanej pyethereum. Kontrola bezpieczeństwa wprowadziła do protokołu kilka małych zmian. Jedną zmianą było wprowadzenie funkcji haszującej SHA3 dla klucza i adresu drzewa Trie, która miała zapobiec

atakowi DOS.[5]

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

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

1.2 Czym są Inteligentne kontrakty?

Określenie inteligentne kontrakty to zostało zaproponowane już w 1994 roku przez Nicka Szabo, który cztery lata później zajmował się projektowaniem kryptowaluty Bitgold, która ostatecznie nie została zaimplementowana.

Ogólnym celem projektowania inteligentnych umów według Nicka Szabo było spełnienie wspólnych warunków umowny (taki jak warunki płatności czy poufności) oraz zminimalizowanie potrzeby zaufanych pośredników. Inteligentne kontrakty miały za zadanie zapewnić większe bezpieczeństwo niż tradycyjne umowy oraz obniżenie kosztów transakcji związanych z tworzeniem umów.[6]

Dzięki inteligentnym kontraktom działa ICO(Initial Coin Offering), czyli nowoczesnej metodzie crowdfundingu, która umożliwia zbieranie kapitału za pomocą kryptowalut. W ICO umowy inwestycyjne są zastępowane cyfrowymi kontrak-

tami, a płatności są rozliczane w kryptowalutach.[9]

Pierwszą popularną kryptowalutą, która wykorzystuje inteligentne kontrakty jest *Bitcoin*. Wadą tej kryptowaluty jest język skryptowy, który umożliwia wykorzystywanie jedynie operacji udostępnionych przez autora. Tworzenie kontraktów jest bardzo ograniczone, ze względu na brak cechy kompletności Turinga w języku.

Inteligentne kontrakty zyskały nowe znaczenie oraz dużą popularność wraz z rozwojem platformy Ethereum. Platforma ta posiada język, który umożliwia tworzenie inteligentnych kontraktów, które mają cechę kompletności Turinga. Dzięki temu Ethereum jest platformą, która właściwie posiada programowalny blockchain, na którym są przechowywane zdecentralizowane aplikacje, które są uruchamiane przez użytkowników sieci za pomocą EVM. Współcześnie każdy użytkownik korzystający z platformy Ethereum może stworzyć aplikacje, która nie posiada jednego serwera, tylko jest rozproszona po całej sieci Ethereum.[9]

Niestety inteligentne kontrakty, jak większość rozwiązań informatycznych, posiadają swoje wady. Wszystkie kontrakty umieszczone na blockchainie są widoczne dla każdego użytkownika. W przypadku luki bezpieczeństwa nie ma możliwości szybkiej naprawy kontraktu umieszczonego w sieci, dodatkowo w praktyce jest trudno napisać dobry inteligenty kontrakt, w którym wszystko zostało przewidziane.

Każdy kontrakt na blockchainie jest przechowywany w postaci jawnego kodu bajtowego, do którego mają wgląd wszyscy użytkownicy sieci, dlatego bardzo ważne jest analizowanie kontraktów pod kątem bezpieczeństwa przed umieszczeniem ich w sieci oraz przed korzystaniem z udostępnionych już kontraktów.

W dalszej części pracy zostanie przedstawione, w jaki sposób stworzyć aplikację, która ułatwi odnalezienie implementacji pasującej do wybranego kodu bajtowego. Posiadanie implementacji kontraktu w znaczącym stopniu ułatwia analizę go pod kątem bezpieczeństwa oraz w celu zrozumienia działania kontraktu.

1.3 Ethereum Virtual Machine

EVM jest to środowisko uruchomieniowe dla inteligentnych kontraktów opartych o Ethereum. Początkowo wirtualna maszyna została opisana w żółtym dokumencie opracowanym przez dr. Gavina Wooda. Maszyna wirtualna jest całkowicie odizolowana od reszty głównej sieci blockchain, co pomaga w zapewnienia bezpieczeństwa wykonywania niezaufanego kodu przez komputery z całego świata. Każdy węzeł w sieci uruchamia u siebie własną implementację EVM oraz jest w stanie wykonywać na niej te same instrukcje co pozostałe węzły.

1.3.1 Kody operacji

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

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

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

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

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

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

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

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

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

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

1.3.2 Kod bajtowy

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

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

1.3.3 Analiza kodu bajtowego

Kontrakty, które zostały umieszczane na blockchainie są przechowywane w postaci kodu bajtowego. Chcąc przeanalizować działanie kontraktu umieszczonego w sieci Ethereum, może okazać się, że kod bajtowy nie jest wystarczająco czytelny. Z tego powodu powstały narzędzia umożliwiające analizę kodu bajtowego. Jednym z takich narzędzi jest eveem.org lub ethervm.io, są to aplikacje internetowe umożliwiające zdekompilowanie do asemblera.

Niestety podczas deasemblacji nazwy funkcji, atrybutów oraz wydarzeń są tracone z powodu optymalizacji. Istnieje jednak sposób na odzyskanie poprzez wykorzystanie bazy danych zawierającej popularne nazwy. Przykładem takiej bazy danych jest 4byte.directory, czyli aplikacja internetowa przechowującą sygnatury funkcji wraz z ich reprezentacjami bajtowymi.

Rozdział 2

Solidity

Solidity jest to język umożliwiający tworzenie inteligentnych kontraktów w sieci Ethereum. Kod napisany w Solidity jest bardzo podobny do C++, C#, JavaScript lub Pythona. Jedną z wad języka Solidity jest mała ilość literatury do nauki języka, ponieważ powstał on w 2014 roku i na moment pisania pracy jest dosyć nowy względem innych języków programowania.

2.1 Modyfikatory widoczności funkcji oraz zmiennych

W Solidity występują dwa rodzaje wywołań funkcji: wewnętrzne (funkcja jest wywoływana wewnątrz tej samej umowy) oraz zewnętrzne (funkcji jest wywoływana z innego kontraktu).[13]

Dodatkowo zostały utworzone cztery modyfikatory dla funkcji i zmiennych:

1. **external** - dzięki temu modyfikatorowi funkcja jest możliwa tylko do wywołania na zewnątrz kontraktu, wywołanie jej wewnątrz tego samego kontraktu spowoduje błąd kompilacji. Co ciekawe wywołanie takiej funkcji **f** za

pomocą **f()** nie zadziała, natomiast **this.f()** już zadziała.

2. **public** - funkcja określona w ten sposób jest częścią interfejsu kontraktu. Funkcja ta może zostać wywołana zarówno wewnętrznie jak i zewnętrznie. W przypadku publiczny zmiennych zostanie automatyczna wygenerowana funkcja zwana akcesorem. Widocznosc public posiadaja domyslnei funkcje.

3. **internal** - funkcje i zmienne z tym specyfikatorem są dostępne tylko wewnątrz umowy, w której się znajdują, ponad to są dostępne dla kontraktów pochodnych. Ten modyfikator jest domyślnie ustawiony dla utworzonych zmiennych.

4. **private** - funkcje i zmienne, która są prywatne są widoczne tylko wewnątrz kontraktu i nie mogą zostać odziedziczone.

Specyfikator widoczności znajduje się po typie zmiennej oraz w przypadku funkcji pomiędzy listą parametrów, a wartością zwracana.

Listing 2.1: Przykłady różnych specyfikatorów w Solidity

```
1 pragma solidity ^0.4.0;
2
3 contract cont1 {
4     uint private data;
5
6     function func(uint x) private returns(uint y) { return x + 1; }
7     function dataSet(uint x) { data = x; }
8     function dataGet() public returns(uint) { return data; }
9     function compute(uint x, uint y) internal returns (uint) {
          return x+y; }
10 }
11
12 contract cont2 {
13     function dataRead() {
```

```
14        cont1 z = new cont1();
15        uint local = z.func(7); // error: member "func" is not
              visible
16        z.dataSet(3);
17        local = z.dataGet();
18        local = z.compute(3, 5); // error: member "compute" is not
              visible
19    }
20 }
21
22 contract cont3 is cont1 {
23    function g() {
24        cont1 z = new cont1();
25        uint val = compute(3, 5); // access to internal member (
              from derived to parent contract)
26    }
27 }
```

Na listingu 2.1 widać przykład trzech kontraktów posiadających funkcje z różnego rodzaju specyfikatorami. Kontrakt **cont1** posiada funkcję prywatną **func**, oznacza to, że może ona zostać wywołana tylko wewnątrz kontraktu. W przypadku gdy w kontrakcie **cont2** zostanie utworzony obiekt kontraktu **cont1**, to podczas odwołania się do funkcji **func** z kontraktu **cont1** programista otrzyma błąd kompilacji. [13]

W kontrakcie **cont1** znajduje się wewnętrzna funkcja **compute**, mimo tego, że nie można jej wywołać z innego kontrakt, to można wykorzystać dziedziczenie w celu wywołania tej funkcji. Kontrakt **cont3** jest przykładem takiej rozwiązania. Kontrakt **cont3** dziedziczy po **cont1**, a następnie prawidłowo wywołuje wewnętrzną funkcję kontraktu **cont1**.[13]

2.2 Generowanie akcesorów podczas kompilacji

Kompilator podczas kompilacji tworzy zewnętrze akcesory dla wszystkich zmiennych z modyfikatorem public.

Listing 2.2: Przykłady zmiennej publicznej w Solidity

```
pragma solidity ^0.4.0;

contract C {
    uint public data;
    function x() public {
        data = 3; // internal access
        uint val = this.data(); // external access
    }
}
```

Na listingu 2.2 widać zadeklarowaną zmienną **data** z modyfikatorem public. Dla takich zmiennych tworzony jest akcesor zewnętrzny, oznacza to, że można go wywołać wewnątrz kontaktu tylko wykorzystując słowo kluczowe **this**. Natomiast można oczywiście odwołać się też do zmiennej wewnątrz kontraktu bez wykorzystywania akcesora.[13]

Listing 2.3: Przykłady bardziej skomplikowanej zmiennej publicznej w Solidity

```
pragma solidity ^0.4.0;

contract RealyComplex {
    struct Data {
        uint foo;
        bytes3 bar;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

Na listingu 2.3 widać trochę bardziej skomplikowany przykład tworzenia funkcji na podstawie publicznej zmiennej. W tym przypadku funkcji, która zostanie wygenerowana, będzie wyglądać następująco:

```
function data(uint x1, bool x2, uint x3)
returns (uint foo, bytes3 bar) {
foo = data[x1][x2][x3].foo;   bar = data[x1][x2][x3].bar; }
```

Warto zauważyć, że w przypadku struktury mapa jest pomijana, ponieważ nie ma dobrego sposobu na przekazanie klucza do struktury.[13]

2.3 Interakcja z inteligentnymi kontraktami

Inteligentne kontrakty po umieszczeniu na blockchanie są przechowywane w postaci kodu bajtowego, a wszystkie sygnatury funkcji kontraktu są zahaszowane podczas kompilacji. W celu ułatwienia innym korzystania z tworzonych kontraktów utworzono specyfikacje ABI, dzięki której użytkownik kontraktu może uzyskać informacje o metodach interakcji.

2.3.1 Czym jest ABI?

ABI(Application Binary Interface) jest to ustandaryzowany sposób interakcji z kontraktem w sieci Ethereum, ułatwia on wykorzystywanie funkcji udostępnionych przez kontrakty. ABI zawiera specyfikacje udostępnionych przez kontrakt funkcji wraz z nazwami i typami danych wejściowych i wyjściowych. Na podstawie ABI można wygenerować selektor funkcji, który umożliwia wywołanie konkretnej funkcji na kodzie bajtowym kontraktu.[14]

Podczas upubliczniania kontraktu w sieci Ethereum, zalecane jest upublicznienie kodu źródłowego kontraktu z danymi umieszczonych w konstruktorze, wersją kompilatora oraz flagami użytymi podczas kompilacji kontraktu. Dzięki temu

użytkownicy mogą zweryfikować czy kontrakt umieszczony w sieci pasuje do upublicznionego kodu źródłowego, a upubliczniony kod źródłowy sprawi, że kontrakt umieszczony w sieci będzie bardziej zaufany wśród innych użytkowników.

Jeśli twórca kontraktu nie chce udostępniać swojej implementacji, a chciałby, żeby inni użytkownicy mogli wygodnie korzystać z jego kontraktu powinien upublicznić ABI.

Istnieją też w sieci Ethereum kontrakty, których kod źródłowy, czy ABI nie zostało nigdzie upublicznione, w związku z tym ciężko jest ustalić, w jaki sposób prowadzić interakcję z takimi kontraktami. W takich sytuacjach konieczne jest wykorzystanie metod inżynierii wstecznej, baz danych sygnatur funkcji oraz innych aplikacji umożliwiających identyfikację takiego kodu bajtowego.

2.3.2 Przykład interakcji z innym kontraktem

Listing 2.4: Przykład wywołania metody z innego kontraktu

```
1  pragma solidity ^0.4.18;
2  contract ExistingWithoutABI {
3
4      address dc;
5
6      function ExistingWithoutABI(address adr) public {
7          dc = adr;
8      }
9
10     function setA_Signature(uint value) public returns(bool success
           ){
11         require(dc.call(bytes4(keccak256("setA(uint256)")), value))
               ;
12         return true;
13     }
14 }
```

Na listingu 2.4 widać przykład interakcji z innym udostępnionym w sieci kontraktem. W konstruktorze podawany jest adres kontraktu, na którym następnie będą wywoływane operacje. Jeśli osoba korzystająca z kontraktu ma informacje o jego sygnaturach funkcji, to łatwo może taką funkcję wywołać.

W celu wywołania funkcji na innym kontrakcie została użyta metoda **call** na zmiennej **dc**. Zmienna **dc** jest adresem innego kontraktu. Metoda **call** przyjmuje selektor funkcji którą zostanie wywołana oraz przekazywaną wartość. Selektor funkcji jest tworzony z sygnatury **setA(uint256)**, poprzez utworzenie skrótu za pomocą SHA3 oraz pobranie czterech pierwszych bajtów z utworzonego skrótu.

Sygnatura funkcji

Sygnatura funkcji, która jest wykorzystywana do tworzenia selektora funkcji, musi zostać odpowiednio zakodowana według specyfikacji ABI.[14] Warto zwrócić uwagę, że sygnatura funkcji może zostać utworzona również dla automatycznie generowanych akcesorów opisanych w poprzednim podrozdziale.

Sygnatura funkcji składa się z nazwy funkcji oraz otoczonej nawiasami listy typów parametrów oddzielonych przecinkami. Wszystkie typy parametrów należy sprowadzić do postaci kanonicznej.

Listing 2.5: Listing przedstawiający rożne sygnatury funkcji

```
1 pragma solidity >=0.4.16 <0.7.0;
2
3 contract Foo {
4   function bar(bytes3[2] memory) public pure {}
5   function sam(bytes memory, bool, uint[] memory) public pure {}
6   function baz(uint32 x, bool y) public pure returns (bool z) { z =
        x > 12 || y; }
7 }
```

Na listingu 2.5 widać przykład trzech metod. Sygnatury funkcji dla tych metod będą wyglądały następująco:

1. **bar(bytes3[2])**
2. **sam(bytes,bool,uint256[])**
3. **baz(uint32,bool)**

W przypadku funkcji **sam** parametr **uint** został sprowadzony do postaci kanonicznej, czyli do typu **uint256**. Ze wszystkich trzech funkcji widocznych na listingu zostały wyciągnięte tylko typy ich parametrów oraz nazwa funkcji.

Język Solidity wspiera wszystkie typy wykorzystywane w ABI, natomiast niektóre typy z Solidity nie są obsługiwane przez ABI.

Tabela 2.1: Mapowanie typów Solidity do ABI

Solidity	ABI
address payable	**address**
contract	**address**
enum	najmniejszy typ **uint**, który przechowa wszystkie wartości
uint	**uint256**
int	**int256**
byte	**bytes1**
fixed	**fixed128x18**
ufixed	**ufixed128x18**

Na tabeli 2.1 po lewej stronie widać typy, które nie są wspierane przez ABI, a po prawej typy, które są częścią ABI. W przypadku typów uint, int, byte, fixed, ufixed są to aliasy, dlatego należy je również sprowadzić do postaci kanonicznej. Typy fixed i ufixed nie są obecnie zaimplementowane w Solidity.[15]

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

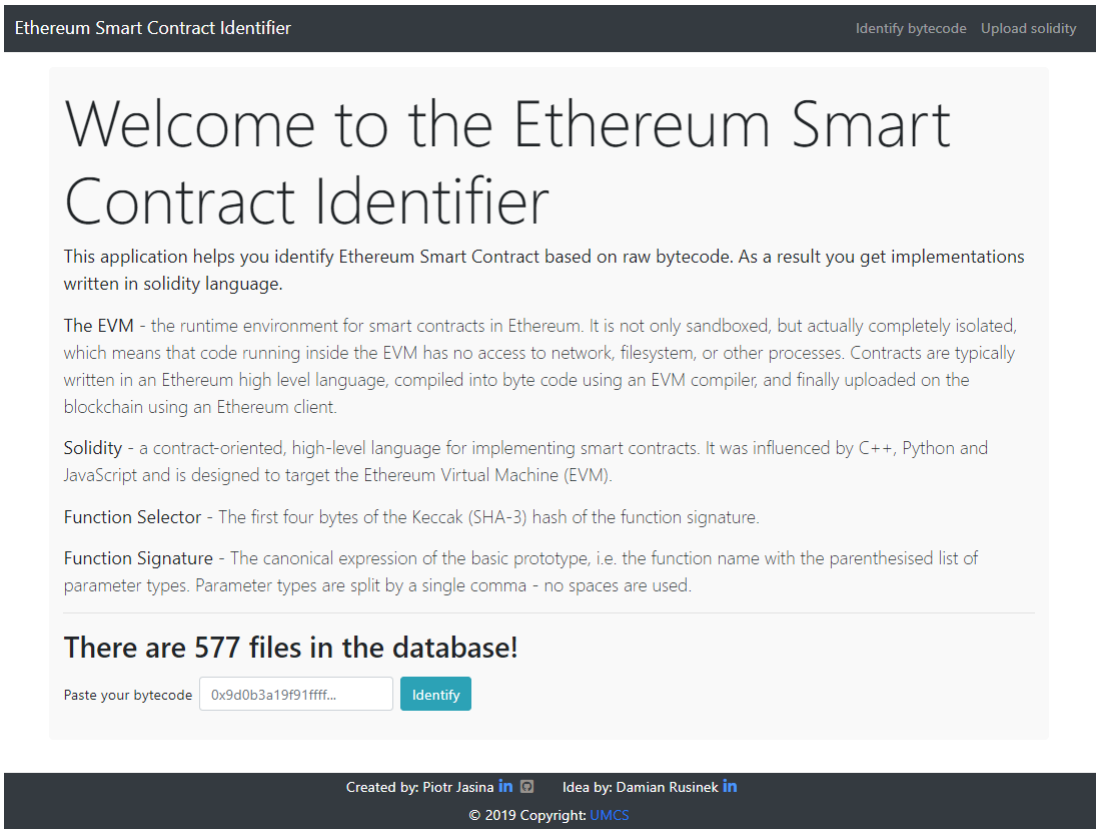
Na stronie głównej aplikacji znajduje się opis wraz z aktualna liczba kodów źródłowych znajdujących się w bazie danych oraz podstawowe definicje związane z aplikacją. Cała aplikacji udostępnia trzy główne funkcjonalności: identyfikację inteligentnych kontraktu, wprowadzanie plików źródłowych kontraktów 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 widocznej na rysunku 3.1, jak i podstronie znajduje się pole, w którym można wprowadzić kod bajtowy. Po wprowadzeniu danych użytkownik zatwierdza je, w obu przypadkach klikając przycisk **Identify**.

Po wprowadzeniu danych i zatwierdzeniu ich przyciskiem, aplikacja rozpoczyna proces analizy wprowadzonego kodu bajtowego oraz wyszukiwane są naj-

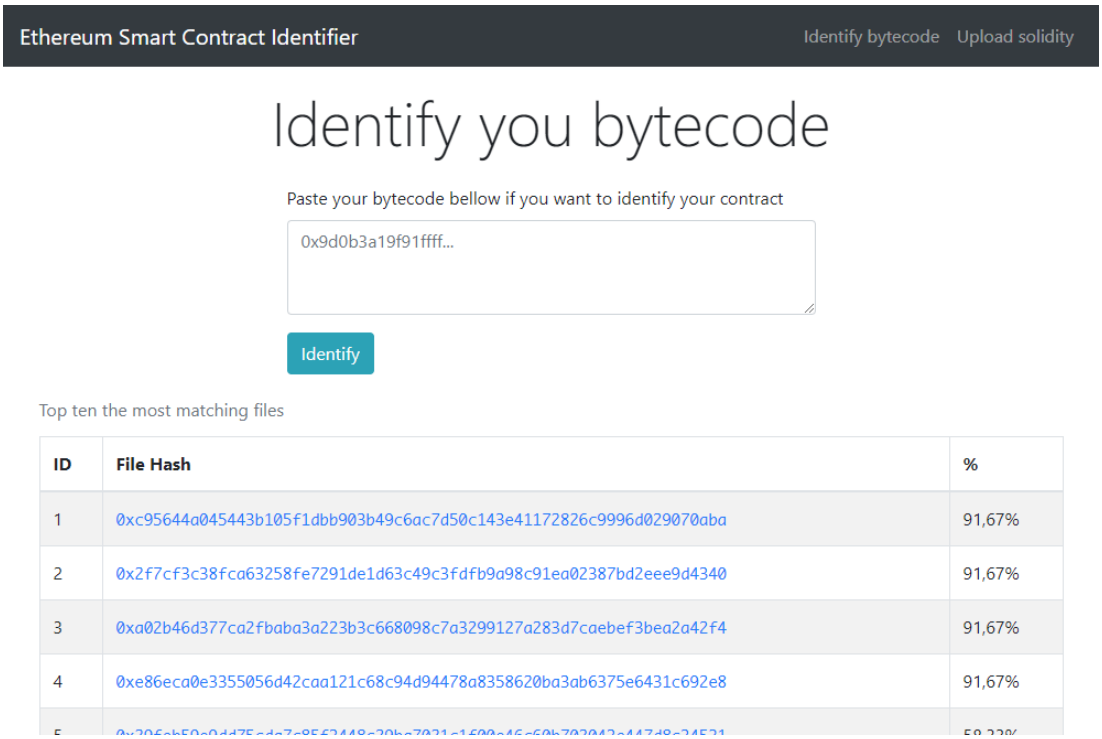


Rysunek 3.1: Strona główna

bardziej prawdopodobne implementacje kontraktu, posortowane malejąco według współczynnika dopasowania. Na rysunku 3.2 został przedstawiony przykładowy wynik identyfikacji.

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

Po naciśnięciu w jedną z wyświetlanych implementacji użytkownik zostanie przeniesiony na podstronę umożliwiającą podgląd implementacji. Na rysunku 3.3 znajduje się przykład przeglądania kodu źródłowego na stronie. Rozwiązanie z numerowaniem linii zostało zaimplementowane w taki sposób, aby podczas kopiowania kodu źródłowego ze strony, nie były kopiowane z nim liczby identyfikujące konkretną linię w kodzie. Istnieje też możliwość pobrania kodu źródłowego

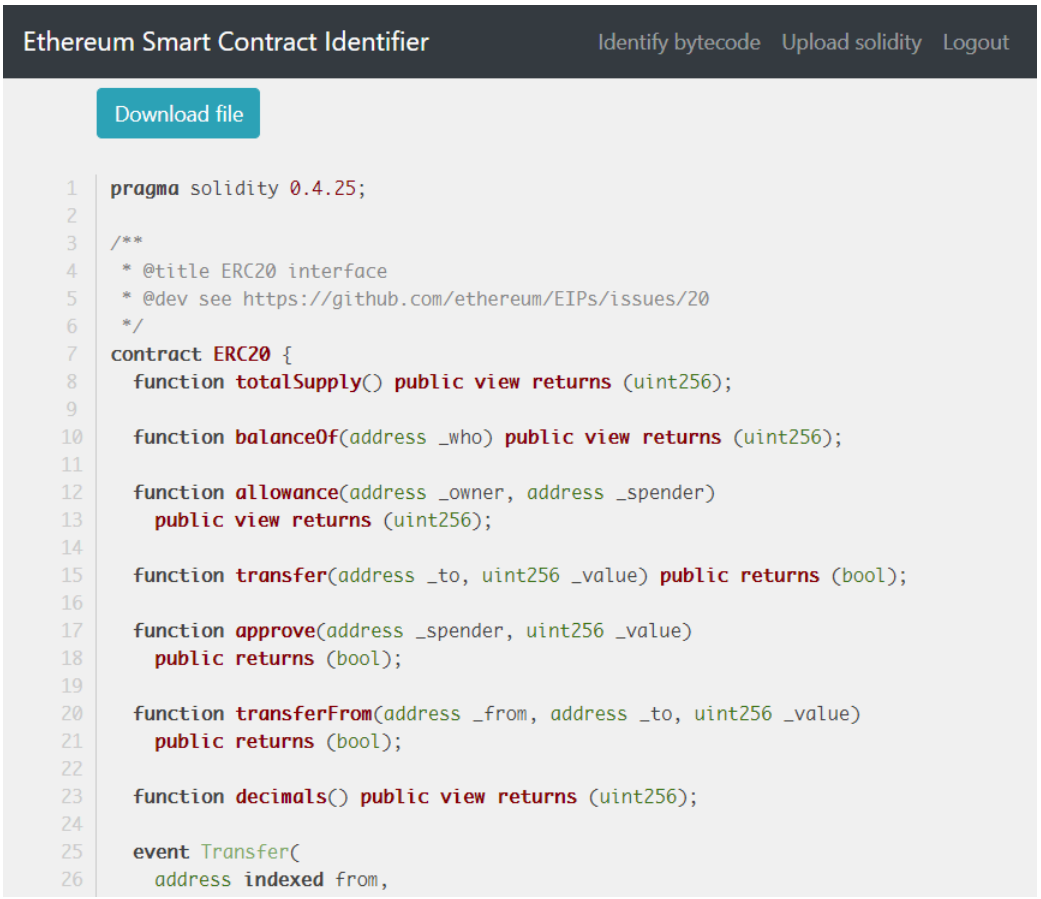


Rysunek 3.2: Wynik identyfikacji inteligentnego kontraktu

ze strony z rozszerzeniem **.sol**

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.

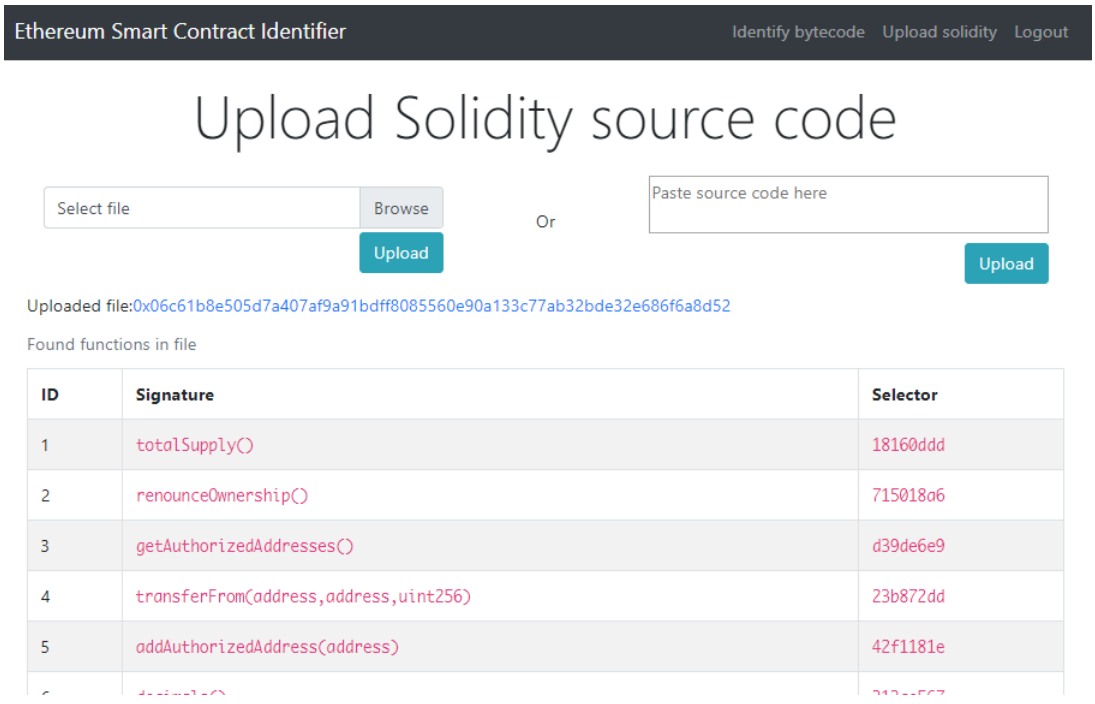


Rysunek 3.3: Podgląd implementacji

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 3.4.

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 3.4. 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 3.4 hasz pliku, użytkownikowi wyświetli się przesłany kod źródłowy.



Rysunek 3.4: Rezultat przesłania inteligentnego kontraktu do aplikacji

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.[16]

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.[17]

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/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.[19]

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 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 kodowania Base64 według schematu **login:haslo**. Na przykładzie z listingu 3.3 został przesłany kod źródłowy, natomiast do autoryzacji wykorzystano login 123 oraz hasło 123.

Listing 3.3: Przesyłanie kodu źródłowego za pomocą API

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

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

Po pomyślnym przesłaniu kontraktu w odpowiedzi od serwera użytkownik otrzymuje status HTTP 200. W odpowiedzi zostaje również przesłany kod 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 opiszę architekturę aplikacji, która realizuje funkcjonalności opisane w sekcji 3.1. Poniżej zostały krótko opisane główne klasy będące częścią aplikacji, widoczne na rysunku 3.5.

LoginController - jest to klasa odpowiedzialna za wyświetlenie ekranu logowania

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

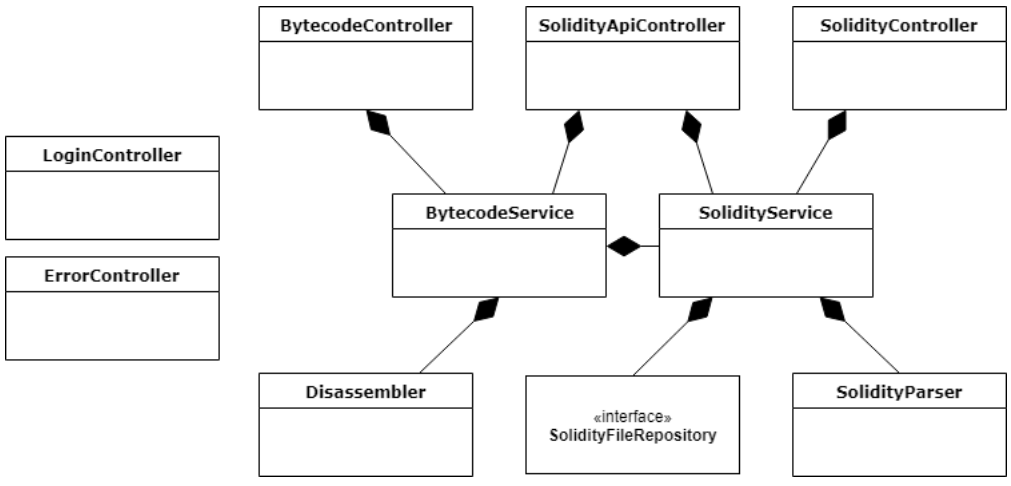
BytecodeController - służy do wyświetlania użytkownikowi strony związanej z identyfikacją kodu bajtowego oraz do mapowania żądań HTTP służących do identyfikacji.

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

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

Disassembler - odpowiada za analizę przekazanego kodu bajtowego. W 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 listę informacji o funkcjach z kodu źródłowego kon-



Rysunek 3.5: Architektura aplikacji

traktu. Lista zawiera takie informacje jak sygnatura oraz selektor funkcji. Sposób tworzenia selektorów funkcji oraz wyciągania z kodu źródłowego sygnatur funkcji został przedstawiony w sekcji 3.4

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

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

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

3.3 Połączenie z bazą danych

W celu integracji aplikacji z bazą danych MongoDB został wykorzystany framework Spring oraz moduł Spring Data MongoDB. W związku z tym, że projekt aplikacji jest budowany za pomocą narzędzia Apache Maven, należy dodać do pliku **pom.xml** wykorzystywane moduły.[24]

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

Listing 3.5: Przyklad dodania zależności w pliku pom.xml

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

W pliku konfiguracyjnym **application.properties** zostały skonfigurowane dane do połączenia z bazą danych. Przykładowa zawartość pliku konfiguracyjnego została przedstawiona na listingu 3.6.[24]

Listing 3.6: Kofiguracja bazy danych

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

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

Listing 3.7: Stworzenie repozytorium za pomocą Spring Data MongoDB

```
@Repository
interface SolidityFileRepository extends MongoRepository<
    SolidityFile, String> {

```

```
4     @Query("{\"solidityFunctions\": {$elemMatch: {\"selector\": {
          $in: ?0}}}}")
5     List<SolidityFile> findSolidityFilesBySelectorContainsAll(List<
          String> functionSelector);
6
7     Optional<SolidityFile> findBySourceCodeHash(String
          sourceCodeHash);
8 }
```

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

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

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

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

Atrybut **sourceCodeHash** został utworzony, ponieważ baza danych nie umożliwia utworzenia unikalnego atrybutu w bazie danych z taką duża ilością znaków jak kod źródłowy. Tworzeniem haszu odbywa się w klasie **SolidityService** widocznej się na rysunku 3.5. Hasz kodu źródłowego jest identyfikatorem, wiec posiada adnotację **@Id**.

Listing 3.8: Przyklad klasy wykorzystywanej przez Spring Data MongoDb

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

```
34                Objects.equals(solidityFunctions, that.
                      solidityFunctions);}
35
36      @Override
37      public int hashCode() {
38          return Objects.hash(sourceCodeHash, sourceCode,
                solidityFunctions);}
39  }
```

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

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

3.4 Identyfikacja sygnatur funkcji w kodzie źródłowym

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

Problemem podczas wyszukiwania sygnatur w kodzie źródłowym jest to, że część sygnatur jest niejawna, ponieważ są one dodatkowo generowane podczas kompilacji dla wybranych atrybutów kontraktu.[13]

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.[22]

Listing 3.9: Metoda kontrolera mapująca żądania POST

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

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

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

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

3.4.2 Kontroler strony internetowej

SolidityControler jest to klasa odpowiedzialna, za tworzenie strony internetowej korzystając z szablonów HTML oraz modułu Thymeleaf. Listing 3.10 przedstawia metodę przyjmująca w żądaniu HTTP kod źródłowy. Metoda ta działa podobnie jak w przypadku API, tylko w tym przypadku zwracana zostaje zwrócona nazwa szablonu wykorzystywanego do renderowania strony. Istnieje możliwość przekazania danych do szablonu, w tym celu wykorzystywany jest parametr **model**, na którym wywoływana jest metoda **addAttribute**.[23]

Listing 3.10: Przechwytywanie żądania o dodanie nowego kodu źródłowego

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

3.4.3 Przetwarzanie kodu źródłowego

SolidityService po otrzymaniu kodu źródłowego od kontrolerów przekazuje po jednej linii do **SolidityParser**, który definiuje czy w danej linii jest sygnatura funkcji. Jeśli podczas parsowania linii znaleziono sygnaturę funkcji, to zostaje ona

dodana wraz z selektorem do listy funkcji. Po przeanalizowaniu wszystkich linii tworzony jest obiekt **SolidityFile**, który następnie za pomocą **SolidityFileRepository** jest zapisywany do bazy danych.

Listing 3.11: Metoda wyszukujaca sygnatury funkcji

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

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

Na listingu 3.11 widać metodę klasy **SolidityParser** wyszukującą funkcję w implementacji kontraktu. Metoda po przyjęciu linii w rezultacie zwraca obiekt klasy **Optional<SolidityFunction>**.[21] Metoda sprawdza cztery możliwe przypadki, w których istnieje możliwość wykrycia funkcji w kodzie źródłowym kontraktu.

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

Wykrywanie sygnatury zadeklarowanej funkcji

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

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

Pierwsza grupa w wyrażeniu wyciąga z linii kodu źródłowego nazwę funkcji, natomiast druga parametry funkcji. Wyrażenie wyszukuje w pojedynczej linii kodu źródłowego frazy **function**, po której następuje nazwa funkcji oraz lista parametrów w nawiasach.[10]

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

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

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

```
9     }
10    return Optional.empty();
11 }
```

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

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

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

Listing 3.13: Metoda generująca selektor funkcji

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

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

Generowanie sygnatury funkcji dla publicznych atrybutów typu mapa

Drugim rodzajem jest atrybut publiczny typu **mapping**. W tym przypadku nie zostało jawnie pokazane, że istnieje w kodzie źródłowym sygnatura funkcji, ponieważ jest ona generowana przez kompilator.[13] W tej sekcji przedstawię proces generowania sygnatury funkcji na podstawie wspomnianego atrybutu.

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

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

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

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

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

```
1 while (true) {
```

```
2     Matcher mappingMatcher = MAPPING_PATTERN.matcher(mappingValue);
3     Matcher arrayMatcher = ARRAY_PATTERN.matcher(mappingValue);
4
5     if (mappingMatcher.find()) {
6         String canonicalArgument = toCanonicalType(mappingMatcher.
              group(MAPPING_KEY_GROUP_ID));
7         canonicalMappingKeys.add(canonicalArgument);
8         mappingValue = mappingMatcher.group(MAPPING_VALUE_GROUP_ID)
              ;
9         continue;
10    }
11    if (arrayMatcher.find()) {
12        String arrayValue = arrayMatcher.group(ARRAY_VALUE_GROUP_ID
              );
13        int dimensionCount = getArrayDimensionCount(arrayValue);
14        for (int i = 0; i < dimensionCount; i++) {
15           canonicalMappingKeys.add(CANONICAL_ARRAY_KEY_TYPE);
16        }
17    }
18    break;
19 }
```

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

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

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

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

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

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

Ostatnim krokiem jest wygenerowanie sygnatury funkcji za pomocą nazwy funkcji oraz listy zgromadzonych typów parametrów. Wszystkie typy przed dodaniem do listy zostają najpierw sprowadzone do postaci kanoniczej.

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

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

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

Wykrywanie sygnatury funkcji dla publicznych atrybutów typu tablicowego

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

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

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

Liczba wymiarów tablicy zostaje określona na podstawie liczby podanych nawiasów podczas deklaracji tablicy. Dla każdego wymiaru zostaje dodany typ **uint256** do listy typów parametrów, ponieważ tablica posiada numeryczny indeks.[18]

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

Wykrywanie sygnatury funkcji dla pozostałych publicznych atrybutów

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

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

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

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

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

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

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

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

Listing 3.17: Zapisywanie kodu źródłowego ze zbiorem jego funkcji oraz haszu

```
SolidityFile save(byte[] sourceCodeBytes) throws IOException {
    String sourceCode = new String(sourceCodeBytes,
       StandardCharsets.UTF_8);
    final String preparedSourceCode = sourceCode
        .replaceAll("(?m)\\s+$", "").replaceAll("(?m) +", " ")
        .replaceAll("(?m)^\\s+","");

        Set<SolidityFunction> functionsFromFile =
           findSolidityFunctionsFromSourceFile(new
           ByteArrayInputStream(sourceCodeBytes));

        return solidityFileRepository.save(new SolidityFile(
           sha3String(preparedSourceCode), sourceCode,
           functionsFromFile));
}
```

Na listingu 3.17 widać metodę klasy **SolidityService**, która zapisuje do bazy danych kod źródłowy, jego hasz oraz zbiór funkcji zidentyfikowanych w kodzie. Przed utworzeniem hasza kodu źródłowego, usuwane są z niego wszystkie nadmierne spacje oraz puste linie, następnie jest tworzony hasz za pomocą funkcji haszującej SHA3.

3.5 Wyszukiwanie selektorów funkcji w kodzie bajtowym

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

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

implementacji.

W celu wykrycia selektorów funkcji należy najpierw uzyskać listę instrukcji wykonywanych w kodzie bajtowym wraz z parametrami tych instrukcji. Do tego posłuży klasa **Dissassembler**, do której jest przekazywany kod bajtowy za pomocą metody **dissassembly** . W rezultacie wspomnianej metody zwracana jest lista instrukcji reprezentowanych przez klase **Instruction**. Klasa **Instruction** składa się z dwóch atrybutów: **opcode** typu **Opcode** oraz **hexParameters** typu **String**.

3.5.1 Reprezentacja operacji EVM wewnątrz aplikacji

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

Listing 3.18: Reprezentacja kodów operacji w postaci typy wyliczeniowego

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

Typ wyliczeniowy posiada w sobie trzy atrybuty: **hexValue**, **operandSize** oraz **description**, które są przekazywane w konstruktorze. Na listingu 3.18 widać fragment deklaracji poszczególnych operacji. Pierwszym parametrem konstruktora jest **hexValue**, czyli kod operacji w postaci szesnastkowej. Następnie

podawany jest **operandSize**, który określa rozmiar parametru danej operacji za pomocą liczby bajtów. Ostatnią przekazywaną informacją jest opis operacji.

Listing 3.19: Klasa mapujaca identyfikator operacji na reprezentacje operacji typu wyliczeniowego

```
1  class OpcodeTable {
2      private OpcodeTable() { throw new UnsupportedOperationException
           ();}
3
4      private static final Map<Integer, Opcode> opcodes =
5              unmodifiableMap(new HashMap<Integer, Opcode>() {{
6                  for (Opcode opcode : Opcode.values()) {
7                      put(opcode.getHexValue(), opcode);
8                  }
9              }});
10
11     static Opcode getOpcodeByHex(String stringHex) {
12         if(stringHex.length() != 2){
13             throw new IllegalArgumentException("Expected length=2
                   stringHex");
14         }
15         return getOpcodeByHex(Integer.parseInt(stringHex, 16));
16     }
17
18     static Opcode getOpcodeByHex(int hex) {
19         Opcode opcode = opcodes.get(hex);
20         if (isNull(opcode)) {
21             return Opcode.UNKNOWNCODE;
22         }
23         return opcode;
24     }
25 }
```

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

umożliwia mapowanie kodu operacji na konkretny obiekt zdefiniowany w typie wyliczeniowym **Opcode**. Klasa **OpcodeTable** została zaprezentowana na listingu 3.19. W przypadku, gdy podany kod nie posiada swojego odpowiednika w typie wyliczeniowym, wtedy zostaje zwrócony kod **UNKNOWNCODE**.

3.5.2 Odczytywanie instrukcji z kodu bajtowego

Mając zdefiniowane w aplikacji wszystkie operacje maszyny wirtualnej Ethereum, można przystąpić do wyszukiwania wszystkich instrukcji z kodu bajtowego. Na listingu 3.20 widać metodę przestawiającą ten proces.

Listing 3.20: Metoda wyszukująca instrukcje instrukcje w kodzie bajtowym

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

Do iteracji po kodzie bajtowym został wykorzystany iterator typu **HexStringIterator**, który odczytuje po jednym bajcie, dopóki istnieją kolejne bajty. W każdej iteracji pętli jest definiowany konkretny obiekt typu **Opcode** za pomocą

metody **getOpcodeByHex** przedstawionej na listingu 3.19. Następnie w zależności od kodu operacji zostają pobierane kolejne bajty, które są parametrem instrukcji.[6] Na podstawie tych informacji tworzony jest obiekt typu **Instruction**, który jest dodawany do listy **instructions**, zwracanej na końcu wykonywania metody.

3.5.3 Wyszukiwanie selektorów funkcji z listy instrukcji

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

Na listingu 3.21 została przedstawiona metoda zwracająca listę selektorów funkcji znajdujących się w kodzie bajtowym. Odczytując z listy po trzy kolejne instrukcje, zostaje wyszukiwany schemat operacji wykonywanych na EVM, na podstawie którego aplikacja wyszukuje moment wrzucenia selektora funkcji na stos.[11]

Listing 3.21: Metoda wyszukująca selektory funkcji na podstawie listy instrukcji

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

```
12          }
13      }
14      return functionSelectors;
15  }
```

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

3.6 Dopasowywanie implementacji na podstawie kodu bajtowego

Po wyszukaniu wszystkich selektorów funkcji z kodu bajtowego zostają one wykorzystane do wyszukania odpowiedniej implementacji. Z bazy danych zostają wyszukane wszystkie pliki posiadające chociaż jeden selektor funkcji znajdujący się w kodzie bajtowym. Dla każdego pobranego w ten sposób pliku wyznaczany jest współczynnik dopasowania kodu bajtowego z plikiem.

3.6.1 Sposoby na wyznaczanie współczynnika dopasowania

Do wyznaczania współczynnika dopasowania kodu bajtowego dla konkretnej implementacji można było zastosować kilka podejść. Poniżej opiszę trzy przetestowane przez mnie sposoby:

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

$$W = M/B$$

gdzie:

W - współczynnik dopasowania implementacji

M - liczba selektorów funkcji wspólnych dla pliku oraz kodu bajtowego

B - liczba selektorów funkcji występujących w kodzie bajtowym

Po zastosowaniu takiego rozwiązania, gdy użytkownik otrzyma wynik 100%, wtedy może mieć pewność, że wszystkie selektory znalezione w kodzie bajtowym mają swój odpowiednik w implementacji. Problem mogą sprawić implementacje posiadające bardzo dużą ilość selektorów funkcji np. biblioteki matematyczne, wtedy wyniki dla takich bibliotek będą miały równie wysoki współczynnik dopasowania co implementacja, która jest mniejsza oraz bardziej zbliżona do prawdziwej, a to z kolei może fałszywie zakłamywać wyniki.

2. Drugim sposobem jest zwracanie ilości dopasowanych selektorów funkcji z kodu bajtowego względem ilości wszystkich selektorów zidentyfikowanych w kodzie źródłowym. Rozwiązanie to przedstawia wzór:

$$W = M/S$$

W - współczynnik dopasowania implementacji

M - liczba selektorów funkcji wspólnych dla pliku oraz kodu bajtowego

S - liczba selektorów funkcji występujących w kodzie źródłowym

Problemem w tym rozwiązaniu jest to, że podczas wyszukiwania sygnatur funkcji w implementacji wyszukiwane są wszystkie funkcje w pliku z różnych kontraktów zależnych od siebie. Podczas dziedziczenia jest też dziedziczona

sygnatura funkcji, która jest zapisywana w kodzie bajtowym ostatecznego kontraktu. Natomiast w przypadku innej relacji kontraktów w kodzie bajtowym nie występują selektory funkcji z innego kontraktu. Przykładowo podczas parsowania pliku Solidity możne być taka sytuacja, że zostaną wykryte wszystkie sygnatury funkcji, natomiast podczas analizy kodu bajtowego nie zostaną one wykryte. W takim wypadku zaprezentowany współczynnik będzie mógł zawyżać wynik dopasowania.

3. Trzecim sposobem, który może rozwiązać problemy dwóch poprzednich, jest obliczenie **współczynnika podobieństwa Jaccarda**. Współczynnik ten jest zdefiniowany jako iloraz mocy części wspólnej zbiorów i mocy sumy zbiorów. Biorąc pod uwagę, że zarówno wyszukiwanie funkcji w kodzie bajtowym, jak i w implementacji może zwrócić czasami niedokładny wynik, wtedy współczynnik ten będzie odpowiednio ustalać wynik, mając na uwadze problemy dwóch poprzednich sposobów. Po dostosowaniu indeksu Jaccarda w aplikacji wzór będzie wyglądać następująco:

$$W = M / (S + B - M)$$

W - współczynnik dopasowania implementacji

M - liczba selektorów funkcji wspólnych dla pliku oraz kodu bajtowego

S - liczba selektorów funkcji występujących w kodzie źródłowym «««< HEAD

B - liczba selektorów funkcji występujących w kodzie bajtowym =======

B - liczba selektorów funkcji występujących w kodzie bajtowym

Mianownik oznacza moc sumy zbiorów.

3.6.2 Testowanie aplikacji pod kątem dopasowywania implementacji

Zostały przeprowadzone testy trzech wspominanych sposobów. Jako grupę testową pobrałem 441 implementacji kontraktów ze strony etherscan.io skompilowanych dla wersji kompilatora od 0.4.22 do 0.4.25. Do kompilacji kontraktów wykorzystałem kompilator solc oraz bibliotekę py-solc, która ułatwia kompilacje z poziomu języka Python.

Każdy kod źródłowy skompilowałem wersją kompilatora sugerowaną przez etherscan.io dla danej implementacji. Dla każdego skompilowanego pliku wybrany został kod bajtowy kontraktu wskazanego na stronie etherscan.io.

Tabela 3.1 przedstawia wyniki testów przeprowadzonych na wszystkich trzech wspomnianych metodach wyznaczania współczynnika dopasowania. Wyniki testów przedstawiają następujące wartości:

1. **Liczba dopasowanych implementacji** oznacza sumę implementacji posiadających współczynnik dopasowania powyżej 80% dla wszystkich identyfikowanych kodów bajtowych. Wysoka liczba dopasowanych implementacji oznacza zawyżone wyniki dopasowania.

2. **Liczba skutecznie dopasowanych implementacji** - skutecznie dopasowana implementacja to taka dopasowana implementacja, z której rzeczywiście powstał identyfikowany kod bajtowy. Jeśli przesłana implementacja po dopasowaniu ma najwyższą wartość współczynnika dopasowania względem innych, oznacza to, że jest skutecznie dopasowana. Test ten pokazuje czy podczas dopasowania prawdziwa implementacja jest na pierwszej pozycji.

3. **Średnia wartość współczynnika dla skutecznie dopasowanych implementacji** - przedstawia średni wynik realnych implementacji wszystkich testowanych kodów bajtowych

Tabela 3.1: Wyniki testów metod dopasowywania

Metoda dopasowywania	1	2	3
Liczba identyfikowanych kodów bajtowych	441		
Liczba dopasowanych implementacji	18398	9485	2722
Liczba skutecznie dopasowanych implementacji	422	311	388
Średnia wartość współczynnika dla skutecznie dopasowanych implementacji	96.64%	82.58%	80.58%

Dla pierwszej metody uzyskano bardzo wysoką skuteczność dopasowania. Problemem w przypadku tej metody jest bardzo duża liczba dopasowanych implementacji, która świadczy o bardzo zawyżonych wynikach. Metoda ta sprawdzała, czy dla danego kodu bajtowego, zostały dopasowane w nim wszystkie selektory funkcji. W przypadku bardzo dużych implementacji ten wynik jest zawyżany, ponieważ jest szansa, że pliki z dużą ilością sygnatur funkcji będą miały często wysoki współczynnik dopasowania podczas identyfikacji małych kodów bajtowych. Mimo wysokiej skuteczności w testach jest bardzo dużo dopasowanych implementacji, co świadczy o niejednoznacznych wynikach dopasowania.

Drugie rozwiązanie mimo mniejszej liczby dopasowanych implementacji posiada nie największą liczbę skutecznie dopasowanych implementacji. Problemem są duże implementacje z wieloma kontraktami, które nie są dziedziczone przez główny kontrakt. Natomiast w kodzie bajtowym są wykrywane tylko te selektory funkcji, które są częścią kontraktu lub są dziedziczone po innym.

Okazuje się, że najlepszym sposobem na wyznaczanie współczynnika dopasowania jest metoda 3 (indeks Jaccarda), ze względu na wysoką liczbę poprawnie zidentyfikowanych implementacji (co świadczy wysokiej o skuteczności) oraz małą liczbę implementacji uznanych za dopasowane (co świadczy o mniejszej ilości

zawyżonych wyników).

3.6.3 Jak poprawić dopasowywanie implementacji?

Pomimo wyznaczenia najlepszego sposobu na wyliczanie współczynnika dopasowania, są jeszcze pewne problemy, przez które dla pewnych kodów bajtowych nie da się uzyskać 100% wartości dopasowania dla realnej implementacji kodu bajtowego, mimo tego, że implementacja ta znajduje się w bazie danych.

Na ten moment największym problemem jest poprawne parsowanie pliku Solidity. W celu polepszenia wyników dopasowywania można ulepszyć aktualny parser tak, aby podczas parsowania ignorował sygnatury funkcji, które nie są dziedziczone przez finalny kontrakt lub można skorzystać z innych gotowych rozwiązań do parsowania plików Solidity. Najprostszym sposobem będzie wykorzystanie kompilatora solc, który po kompilacji zwraca ABI dla danego kontraktu.

Wykorzystując kompilator solc do uzyskania sygnatur funkcji, zamiast stworzonego przez mnie parsera, średnia wartość dopasowania powinna być bliska 100%. Wtedy podczas przesyłania nowego kontraktu do aplikacji wystarczyłoby podać wersje kompilatora. Po przesłaniu aplikacja kompilowałaby implementacje, po udanej kompilacji zostanie zwrócone ABI, z którego można wyciągnąć wszystkie sygnatury funkcji.

Podsumowanie

Celem powyższej pracy było utworzenie aplikacji umożliwiającej identyfikacje kodu bajtowego kontraktu. Dokładniejsza analiza tych założeń spowodowała, że zdecydowano się zrealizować stronę internetową, która pozwala na korzystanie z aplikacji bez konieczności instalacji.

Efektem pracy jest aplikacja internetowa o nazwie Ethereum Smart Contract Identifier. Aplikacja udostępnia przyjazny interfejs użytkownika oraz API dla programistów. Identyfikacja kontraktów polega na wyszukaniu odpowiedniej implementacji ze zgromadzonej bazy danych, jeśli użytkownik chce dodać nowy kontrakt do bazy danych, wtedy musi przejść proces autoryzacji. Na podstawie powyższego opisu aplikacji, można stwierdzić, że cele pracy zostały w pełni zrealizowane.

W przyszłości aplikacja możne zostać ulepszona o lepszy sposób wyszukiwania sygnatur funkcji w implementacjach oraz o dokładniejszy sposób wyszukiwania selektorów funkcji w kodzie bajtowym. Można ulepszyć parser do Solidity, tak aby wspierał wszystkie wersje języka lub zastąpić parser przez inne gotowe narzędzie.

Dalszym kierunkiem rozwoju może być utworzenie zautomatyzowanego systemu do gromadzenia implementacji kontraktów z całego internetu oraz połączenie takiego systemu z utworzona przez mnie aplikacją. Dzięki dużej ilości implementacji kontraktów można uzyskać bardzo dokładne wyniki podczas identyfikacji kontraktów.

Bibliografia

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

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

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

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

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

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

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

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

[9] Ethos - *What are smart contracts?* https://www.ethos.io/smart-contracts

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

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

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

[13] Solidity Documentation *Getter Functions* https://solidity.readthedocs.io/en/v0.5.5/contracts.html.

[14] Solidity Documentation *Contract ABI Specification* https://solidity.readthedocs.io/en/develop/abi-spec.html.

[15] Solidity Documentation *Types* https://solidity.readthedocs.io/en/v0.5.7/types.html.

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

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

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

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

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

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

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

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

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

Spis tabel

Spis rysunków

Spis listingów