



**UNIWERSYTET OPOLSKI**  
**WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI**  
**Instytut Informatyki**

**PRACA INŻYNIERSKA**

Technologia inteligentnych kontraktów i związane z nią problemy bezpieczeństwa

Smart Contract Technology and security problems connected with Smart Contracts

**Autor:**

Robert Kurcwald

Praca wykonana pod kierunkiem  
**dr Sebastian Bala**

**Opole, 2020r.**



## **Streszczenie**

Inteligentne kontrakty można uznać za najbardziej popularne aplikacje bazujące na technologii Blockchain. W tej pracy przedstawiłem na przykładzie Ethereum, podstawowe pojęcia związane z tą technologią, przykładowe kontrakty, znane ataki i ich symulacje, a także przykłady poprawnego pisania umów i unikania błędów.

Słowa kluczowe: Ethereum, Blockchain, Inteligentne kontrakty, ataki na kontrakty, semantyka Solidity, weryfikacja kontraktów

## **Abstract**

Smart Contracts can be regarded as one of most popular blockchain-based applications. In this work, I presented on Ethereum Blockchain's example, basic concepts related with blockchain technology, known attacks with simulations and example codes which can help write accurate contracts.

Keywords: Ethereum, Blockchain, Smart Contracts, Smart Contract's attacks, Solidity semantics, Smart Contract's Verification

# Spis treści

Streszczenie .....	3
Abstract .....	3
Wprowadzenie.....	6
1. Architektura Ethereum i jej cele .....	7
1.1 Kryptografia.....	7
1.2 Ether .....	8
1.3 Gas .....	9
1.4 Węzły .....	9
1.5 Konta.....	10
1.6 Transakcje w Ethereum .....	10
1.7 Bloki .....	13
1.8 Inteligentne kontrakty .....	14
1.9 Przypisywanie transakcji do łańcucha bloków .....	15
2. Konfiguracja prywatnego środowiska Ethereum Virtual Machine i narzędzia używane.....	16
2.1 Go Ethereum – Geth.....	16
2.2 Praktyczna konfiguracja konsolowego środowiska tekstowego .....	16
2.3 Kompilator języka Solidity .....	20
2.4 Truffle .....	20
2.5 Ganache – graficzne środowisko testowe.....	21
3. Semantyka Inteligentnych Kontraktów .....	32
3.1 Struktura kontraktów.....	32
3.2 Zmienne i ich typy.....	34
3.3 Struktury kontrolne .....	35
3.4 Funkcje i modyfikatory funkcji.....	37
3.5 Eventy .....	41
4. Ataki na platformę Ethereum .....	42
4.1 Atak typu: Reentrancy.....	43
4.2 Teoretyczny atak 51% .....	47
4.3 Atak typu: Front-Running .....	48
4.4 Atak typu: Timestamp Dependence .....	48
4.5 Atak typu: Integer Overflow and Underflow .....	49
4.6 Atak typu: Denial-of-Service with revert .....	54
5. Poradnik jak pisać Inteligentne Kontrakty i unikać błędów .....	57

5.1 Dziedziczenie .....	57
5.2 Interfejsy .....	59
5.3 Kontrakty abstrakcyjne .....	59
5.4 Wyjątki.....	59
5.5 Dobre nawyki podczas programowania kontraktów.....	60
6. Sposoby weryfikacji inteligentnych kontraktów .....	63
6.1 Weryfikacja formalna.....	63
6.2 Zagrożenia związane z weryfikacją formalną .....	64
6.3 Przyszłościowe sposoby weryfikacji .....	65
Podsumowanie .....	66
Bibliografia.....	67
Spis rysunków .....	68
Spis listingów .....	69
Spis załączników.....	70

## **Wprowadzenie**

Głównym tematem pisanie tej pracy jest zapoznanie się z technologią Inteligentnych Kontraktów (z ang. Smart Contracts) opartą na Blockchainie o nazwie Ethereum w języku Solidity, a także przegląd oraz przeprowadzenie znanych ataków na nie.

Od wprowadzenia w 2009 roku pierwszej implementacji technologii blockchain, przez osobę (lub grupę osób) o pseudonimie Satoshi Nakamoto, popularność zdecentralizowanych sieci ciągle rośnie. Bitcoin swoją popularność zyskał na braku zaufania opartym względem centralnych instytucji (zwanych emitentami). Dzięki zdecentralizowaniu oraz implementacji protokołu peer-to-peer, platforma nie pozwala na ingerencję czy manipulację od wewnątrz, na ilość wybitych monet. Od 2009 roku powstało wiele platform implementujących technologię blockchain w tym Ethereum. Jest to udoskonalona wersja Bitcoina, oferuje użycie tej technologii w celu wprowadzenia inteligentnych kontraktów. Inteligentne umowy możemy porównać do klas w standardowych językach obiektowych. Główną zaletą Ethereum w porównaniu z Bitcoinem jest średni czas weryfikacji transakcji, a także zatwierdzania bloku.

W poniższych rozdziałach przedstawię architekturę Ethereum, narzędzia używane do tworzenia prywatnych sieci, strukturę oraz przykłady inteligentnych kontraktów, przykładowe ataki na platformę Ethereum, a także sposoby weryfikacji umów.

## 1. Architektura Ethereum i jej cele

Ethereum to zdecentralizowana sieć połączeń klient-klient (ang. Peer-to-peer), która została pokazana w 2015 roku, jako rozłam (ang. Hard fork) Bitcoina. Jest to druga platforma stworzona przez rosyjsko-kanadyjskiego programistę Vitalika Buterina, który pracował wcześniej przy już wspomnianej platformie. Nie był jednak zadowolony z mechanizmów tej technologii, przez co stworzył swoją ulepszoną wersję łańcucha bloków. Głównym jego celem było umożliwienie wykonywania kontraktów oraz aplikacji zdecentralizowanych (ang. DApps).

Podstawowymi pojęciami używanymi w łańcuchu bloków o nazwie Ethereum są:

- Ether;
- Gas;
- Węzły (ang. Nodes);
- Konta (ang. Accounts);
- Transakcje (ang. Transaction);
- Bloki (ang. Blocks);
- Inteligentne kontrakty (ang. Smart Contracts).

Porównując platformy Bitcoin oraz Ethereum, możemy zauważyć znaczą różnicę w liczbie zweryfikowanych transakcji na sekundę. W przypadku pierwszej z wymienionych technologii, wartość potwierdzeń na sekundę jest niska i waha się w przedziale od 1 do 2, a zatwierdzenie bloku trwa ok. 10 minut. Ethereum natomiast dzięki zaimplementowaniu protokołu GHOST (Greedy Heaviest Observed Subtree) [1], który odpowiada za odcięcie gałęzi (ang. Branch) łatwiejszej (gdzie suma parametru totalDifficulty jest mniejsza od innej gałęzi) lub promowanie dłuższego łańcucha bloków, pozwala na zweryfikowanie do 15 transakcji na sekundę, a czas zatwierdzenia bloku waha się w przedziale od 10 do 19 sekund.

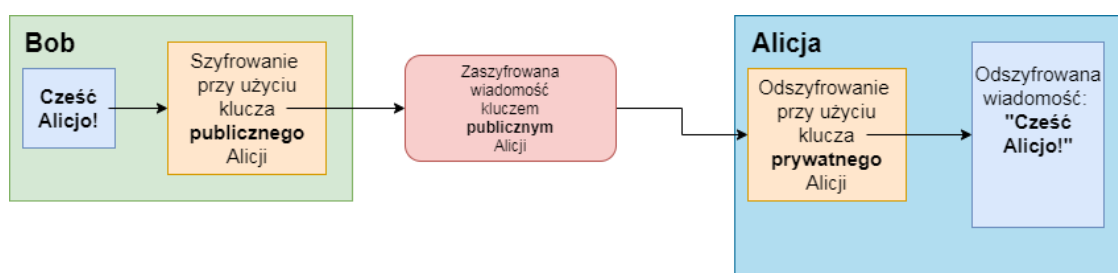
### 1.1 Kryptografia

Kryptografia jest to nauka zajmująca się szyfrowaniem, czyli zamienianiem zrozumiałych informacji w sekretne wiadomości, które nie są możliwe do odczytania bez znajomości odpowiedniego klucza. W informatyce posiadamy dwa główne typy szyfrowania:

- Symetryczne;
- Asymetryczne.

Kryptografia symetryczna może być używana przy różnych operacjach. Zarówno stosuje się ją do szyfrowania, jak i do uwierzytelniania danych. W obu przypadkach strony algorytmu używają tego samego klucza sekretu. W przypadku szyfrowania, ten sam klucz jest używany do szyfrowania i odszyfrowywania. Natomiast w przypadku uwierzytelniania danych, ten sam klucz sekret, jest używany do uwierzytelniania pochodzenia danych.

Kryptografia asymetryczna charakteryzuje się użyciem dwóch różnych kluczy – sekretów. W algorytmie Diffie-Hellmana, służącym do dzielenia sekretów, obie strony protokołu posiadają swoje własne klucze prywatne i wymieniają się swoimi kluczami publicznymi, w celu wygenerowania wspólnego sekretu. Algorytm RSA służy zarazem do wymiany kluczy, jak i podpisów cyfrowych. Algorytm ten oparty jest na dwóch kluczach publicznym i prywatnym, z których jeden z nich jest kluczem szyfrującym, a drugi jest kluczem odszyfrowującym lub publicznym i podpisującym w zależności od zastosowania. Istnieją również schematy klucza publicznego takie, jak DSA dedykowane podpisowi cyfrowemu. W przeciwieństwie do RSA, DSA używa się w schemacie podpisz-zweryfikuj podpis.



Rysunek 1. 1 Schemat szyfrowania i odszyfrowania przy użyciu kluczy prywatnych oraz publicznych

Źródło opracowania: własne

## 1.2 Ether

Podobnie, jak każda publiczna platforma oparta o technologie Blockchain, Ethereum posiada aspekt ekonomiczny. Kryptowalutą w tym łańcuchu bloków jest Ether. Każda aktywność związana z modyfikowaniem (lub dodawaniem) transakcji, kontraktów czy zdecentralizowanych aplikacji ma swoją opłatę, naliczaną właśnie w tej walucie. Ethereum posiada swój system metryczny jednostek Etheru, a jego wartości są opisane w deklaracji platformy [2]. Najmniejszą jednostką jest Wei, a tabela poszczególnych wartości przedstawia się w następujący sposób (Rysunek 1.2):

Nazwa	Wartość
Wei	$10^0$
Szabo	$10^{12}$
Finney	$10^{15}$
Ether	$10^{18}$

Rysunek 1. 2 Wartości systemu metrycznego Ethereum

Źródło opracowania: własne na podstawie „Yellow Paper” [2]

Skąd wziąć Ether? Walutę można kupić na różnego typu giełdach kryptowalut, lub zdobyć w formie nagrody za zatwierdzenie bloku do łańcucha. Drugi z wymienionych procesów nazywa się kopaniem kryptowalut (ang. Mining) i jest on prowadzony przez węzły kopiące (ang. Mining nodes).



### 1.3 Gas

Ether, jako zewnętrzny środek płatniczy, sprawdza się bardzo dobrze, jednakże gdyby płacić jego stałą wartość za transakcję, cena w różnych dniach znacznie wahała by się w zależności od kursu waluty na giełdzie. Właśnie dlatego twórcy stworzyli wewnętrzną walutę zwaną gazem (ang. Gas), która jest mierzalną wartością mocy obliczeniowej potrzebnej do wykonania transakcji. Przeliczona cena również jest wartością dynamiczną zależną od ceny gazu, wyrażonej w jednostce Gwei (1Ether to  $10^9$ Gwei), a także potrzebnej wartości mocy obliczeniowej. Wdrożenie oraz wykonanie transakcji ma jednakże przypisane stałe wartości opłat wyrażone właśnie w ilości gazu.

### 1.4 Węzły

Komputery tworzące sieć Ethereum nazywane są węzłami (ang. Nodes). Używają one protokołu połączeniowego peer-to-peer, który jest modelem zapewniającym wszystkim użytkownikom platformy te same uprawnienia. Teoretycznie w tym łańcuchu bloków wymieniamy dwa typy węzłów [1]:

- Ethereum Virtual Machine (EVM);
- Ethereum mining nodes.

W praktyce, zwykle nie ma dedykowanych komputerów, które byłyby węzłami EVM, a każdy użytkownik jest zarazem węzłem kopiącym oraz środowiskiem uruchomieniowym dla inteligentnych kontraktów [4].

Ethereum Virtual Machine to środowisko głównie odpowiedzialne za wykonywanie kodu bajtowego (ang. Bytecode), który jest tworzony przez kompilator języka wysokiego poziomu podczas procesu kompilacji, oraz wdrożenie instrukcji do sieci Ethereum. Jednakże, kiedy transakcja jest zatwierdzona, nie jest automatycznie dodana do łańcucha bloków, tylko trafia do puli (ang. Transaction pool), a dopiero po weryfikacji przez węzły kopiące zostanie tam wdrożona. Obecnie EVM używa również algorytmów przeciwdziałających atakom typu denial-of-service.

Ethereum mining nodes jest odpowiedzialny za dopisanie transakcji (lub zbioru transakcji) do łańcucha bloków. Potocznie jest nazywany górnikiem (ang. Miner), a jego praca jest podobna do księgowego, który weryfikuje, a następnie zapisuje transakcje z puli do rejestru. Po wykonaniu zadania oraz rozwiązania zagadki kryptograficznej, górnik dostaje wynagrodzenie. Proces ten nazywany jest jako „dowód wykonanej pracy” od nazwy protokołu Proof of Work (PoW) [21]. W zapowiadzianym już Ethereum 2.0, protokół PoW zostanie zastąpiony protokołem Proof of Stake (PoS) [22].

## 1.5 Konta

W platformie Ethereum rozróżniamy dwa rodzaje kont. Oznaczone są jako:

- Externally owned accounts (EOA);
- Contract accounts.

EOA to konto powiązane z kluczem prywatnym, które nie posiada kodu wykonywalnego, ale może przechowywać, jako walutę Ether, wykonywać transakcje klasyczne z innymi EOA oraz wywoływać funkcje kontraktów.

Konta kontraktów mają podobną formę, jak konta EOA z tą różnicą, że nie są powiązane z kluczem prywatnym, a dodatkowo posiadają kod kontraktu. Kontrakty przechowywane w tych kontach mogą zostać uruchomione przez transakcje lub wiadomość (ang. Message) otrzymaną z konta EOA, zwanego zewnętrznym. O koncie EOA mówimy, że jest zewnętrzne, jeśli nie z tego konta kontrakt został wdrożony do łańcucha bloków.

```
0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED
0xc28A3317410A3b5593bA403B5c869eF1c0B337cd
0x39aD3f34BC4854bD969Fcb4da9517511A8515961
0x847d5B2D768795c10CdAa21CDC21ACADB28928E9
0xCaf5323127115c9d4C60B3721Bd570538DBB77F6
```

Rysunek 1. 3 Przykłady kont w Ethereum

Źródło opracowania: własne

## 1.6 Transakcje w Ethereum

Termin „transakcja” jest używany w Ethereum by odnieść się do podpisanego pakietu danych, który zawiera wiadomość wysłaną z jakiegokolwiek EOA [1]. Jest to umowa pomiędzy przynajmniej dwiema stronami, która obejmuje jakąś wymianę dóbr. Istnieją trzy typy transakcji:

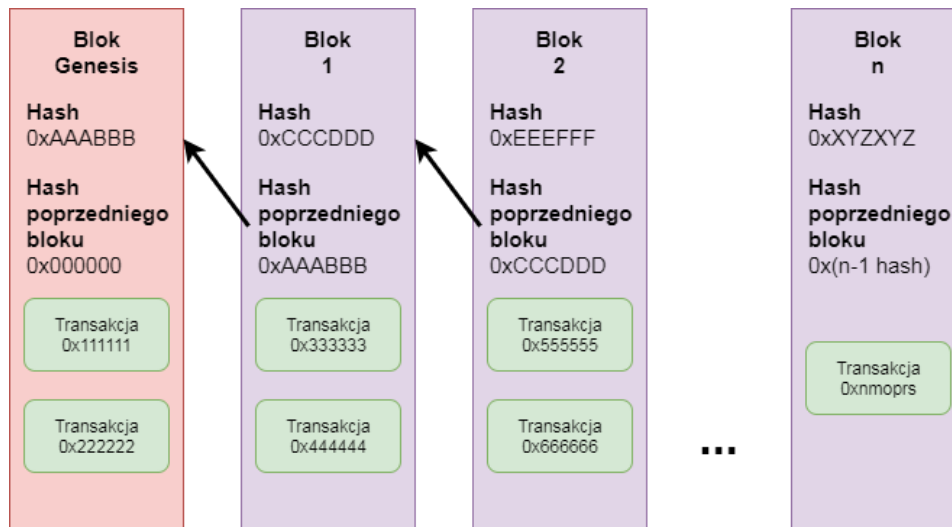
- Transfer Etheru z jednego konta na drugie – oba typy kont mogą zostać użyte
- Wdrożenie inteligentnego kontraktu – EOA może zainicjować wdrożenie kontraktu używając transakcji w EVM
- Użycie lub odwołanie się do funkcji w kontrakcie – transakcja jest używana tylko wtedy, gdy funkcja zmieni stan. W przeciwnym wypadku nie potrzeba odwoływać się do kontraktu poprzez transakcje.





## 1.7 Bloki

Struktura relacji między blokami zwana łańcuchem bloków, to połączenie n bloków w następujący sposób (Rys. 1.6):



Rysunek 1. 6 Uproszczony schemat połączenia bloków

Źródło opracowania: własne

Po przeanalizowaniu schematu, widzimy, że wszystkie bloki, oprócz bloku Genesis, posiadają hash poprzedniego bloku, przez co łatwo możemy określić, kto jest czym rodzicem. Blok Genesis jest swego rodzaju wyjątkiem, gdyż jako pierwszy blok, nie może posiadać hasha poprzedniego bloku, a jego formalna wartość, bazując na Ethereum Yellow Paper [2], wynosi  $H_i=0x000000$ . W prywatnych sieciach Ethereum właściwości bloku definiuje się w pliku genesis.json. Nie są to jedyne dane przechowywane przez blok. Główne właściwości struktury zatwierdzonego bloku przedstawiają się w następujący sposób (Rysunek 1.7) [18]:

- difficulty – złożoność zagadki kryptograficznej algorytmu konsensusu Ethash [19] potrzebna do zatwierdzenia bloku
- extraData – tablica bajtowa, zawierająca istotne informacje dotyczące bloku
- gasLimit – definicja możliwej maksymalnej ilości gazu do użycia, przez co możemy określić maksymalną ilość transakcji dostępnych w bloku. Obecnie może być to ok 70 transakcji, a oblicza się to ze wzoru:  $\frac{1,5 \cdot 10^6 Gas}{2,1 \cdot 10^4}$ , gdzie mianownik oznacza średnią złożoność potrzebną do wykonania transakcji z jednego konta na drugie
- gasUsed – wartość faktyczna gazu użyta do wykonania wszystkich transakcji w bloku
- hash – wynik funkcji skrótów Keccak-256 nagłówek bloku
- miner – hash osoby, która wydobyla blok

- **mixHash** – 256 bitowy hash, który połączony z nonce dowodzi, że wystarczająca ilość mocy obliczeniowej została poświęcona na blok
- **nonce** – 64 bitowy hash, który połączony z mixHash dowodzi, że wystarczająca ilość mocy obliczeniowej została poświęcona na blok
- **parentHash** – hash rodzica przedstawionego bloku
- **receiptsRoot, stateRoot, transactionsRoot** – właściwości odwołujące się do drzewa Merklego [Rys. 1.8]
- **timestamp** – znacznik czasu podany w sekundach, liczony od momentu zaczęcia nowej epoki (ang. Epoch), która trwa 30,000 bloków.
- **totalDifficulty** – złożoność łańcucha liczona od początku do momentu tego bloku
- **transactions** – tablica transakcji tego bloku

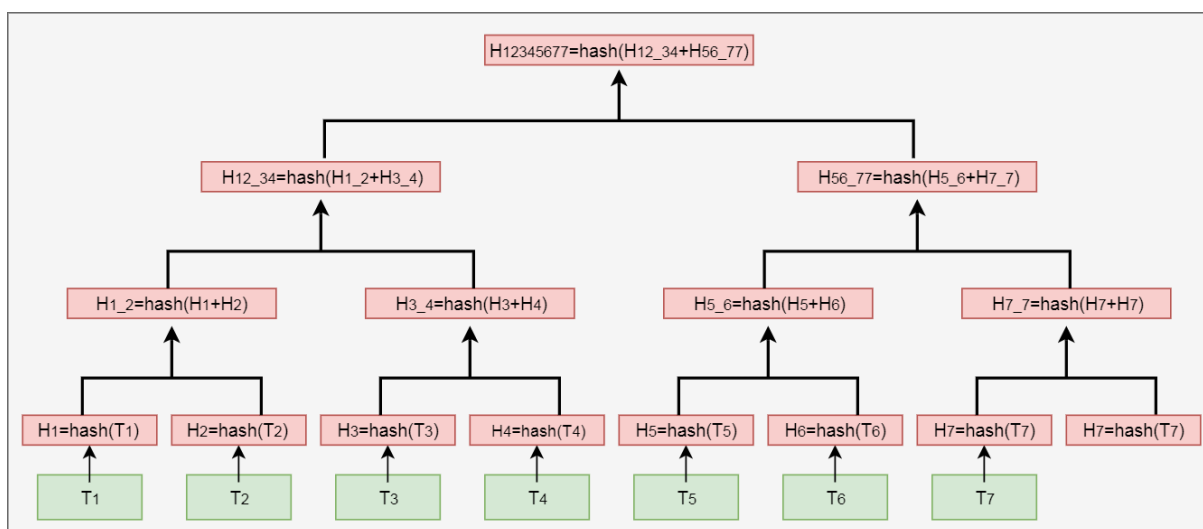
### Rysunek 1. 7 Struktura zatwierdzonego bloku

## 1.8 Inteligentne kontrakty

struktur władzy czy zewnętrznych mechanizmów, jednakże próbując wpływać na świat realny, muszą one odwoływać się do wyroczni (ang. Oracle), która jest zaufanym połączeniem między blockchainem, a zewnętrznymi danymi na świecie.

## 1.9 Przypisywanie transakcji do łańcucha bloków

Proces ten polega na pobraniu przez wszystkich górników, dostępnych transakcji z puli. Górnicy tworzą nowy blok poprzez dodanie wszystkich transakcji do niego sprawdzając przy tym, czy wszystkie transakcje są poprawne, a także czy jakaś z nich nie została wcześniej dodana do łańcucha bloków. Na podstawie drzewa Merklego, ilość hashy transakcji większa niż jeden jest zamieniona w jeden hash zwany korzeniem drzewa (ang. Tree root). Tę operację można zaobserwować poniżej (Rys.1.8):



Rysunek 1. 8 Schemat drzewa Merklego

Źródło opracowania: własne

Górnicy biorą dwa wyniki funkcji hashującej i tworzą z nich jeden nowy hash. Przy nieparzystej wartości funkcji skrótów, proces zachodzi podobnie z tą różnicą, że ostatni hash jest duplikowany i traktowany, jako para (Rysunek 1.8 Hash  $H_7$  tworzy parę  $H_{7\_7}$ ). Wartość ostatniego hashu jest zapisywana do nagłówka bloku. W nowo utworzonym ogniwie, górnik znajduje, a następnie zapisuje hash poprzedniego bloku, który staje się jego rodzicem (ang. Parent). Kolejnym krokiem górników jest wyliczenie stanów i potwierdzeń transakcji, a następnie dodanie tych wartości, wraz z polami nonce oraz timestamp, do nagłówka bloku, po czym zostaje wykonana funkcja skrótu, która łączy nagłówek oraz ciało bloku w jeden wspólny hash. Proces właściwego kopania zaczyna się, gdy górnik zmienia wartość pola nonce i próbuje znaleźć rozwiązanie zagadki kryptograficznej algorytmu Ethash. Trzeba również pamiętać o tym, że każdy górnik dostępny w sieci próbuje zatwierdzić blok w jak najkrótszym czasie. Po znalezieniu rozwiązania górnik dopisuje do swojej

lokalnej kopii rejestru łańcucha nowy blok, a następnie rozsyła całej sieci (do wszystkich Ethereum mining nodes) rozwiązanie zagadki. Jeśli większość (powyżej 51%) zgodzi się z tym rozwiązaniem, blok zostaje przypisany do globalnego rejestru, każdy węzeł zostaje zaktualizowany, a górnik, który znalazł rozwiązanie, dostaje nagrodę (ang. Reward) w formie Etheru. Nie jest to jedyny przypadek dopisania bloku. Nie zawsze jeden górnik zatwierdza blok. Czasami zdarza się, że kilku zatwierdza ten sam blok do łańcucha. W takim wypadku tworzą się tzw. gałęzie (ang. Branches), a ten sam blok jest umieszczony w dwóch miejscach. Nagroda zostaje wtedy podzielona przez ilość górników i każdy z nich dostaje odpowiednią jej wartość. Jednakże w Blockchainie nie może być dwóch bloków mających jednego rodzica, więc jedna gałąź, która miała mniej dopisanych bloków lub mniejszą trudność całkowitą, zostaje odrzucona. Ta operacja jest przeprowadzona po zatwierdzeniu kilku (w przybliżeniu 5) bloków.

## **2. Konfiguracja prywatnego środowiska Ethereum Virtual Machine i narzędzia używane**

W tym rozdziale zostaną przedstawione narzędzia używane do tworzenia i zarządzania prywatnymi sieciami Ethereum, a także sposoby konfiguracji węzła EVM wraz z wykonanymi transakcjami oraz kontraktami na nim.

### **2.1 Go Ethereum – Geth**

Go Ethereum to implementacja protokołu Ethereum napisana w języku Go działająca na licencji GNU LGPL v3 [12]. Rozbudowany projekt, który m.in. umożliwia:

- stworzenie nowej sieci Ethereum używając narzędzia „puppeth”;
- połączenie się do niej poprzez spersonalizowanie parametrów polecenia „geth”;
- dostęp do narzędzi developerskich poprzez polecenie „evm”

### **2.2 Praktyczna konfiguracja konsolowego środowiska tekstowego**

Używając narzędzia „puppeth” dostępnego w pakiecie Geth, generujemy plik konfiguracyjny naszej prywatnej sieci Ethereum (EVM). W pierwszym kroku podajemy nazwę naszej EVM pamiętając o braku dużych liter, spacji oraz myślników (te wymagania zostały dodane w wersjach Geth wyższych niż v1.8.23). Po wybraniu opcji skonfigurowania improwizowanego (z ang. Scratch) nowego bloku Genesis, wybieramy algorytm konsensusu. W naszym przypadku będzie to Ethash, jako



przedstawiciel protokołu PoW. Następnie wybieramy czy (ewentualnie które) konta powinny zostać zasilone przy tworzeniu EVM. Ostatnim krokiem jest skonfigurowanie numeru identyfikacyjnego (ID) sieci. Ten etap można pominąć, a „puppeth” nada sieci losowy numer. Po wykonaniu wszystkich poprzednich kroków, narzędzie generuje nam plik o nazwie sieci z rozszerzeniem .json w bieżącej lokalizacji.

```
PS C:\Users\Robert\EthereumSkills\private> & 'C:\Program Files\geth\puppeth.exe'
Welcome to puppeth, your Ethereum private network manager

This tool lets you create a new Ethereum network down to
the genesis block, bootnodes, miners and ethstats servers
without the hassle that it would normally entail.

Puppeth uses SSH to dial in to remote servers, and builds
its network components out of Docker containers using the
docker-compose toolset.

Please specify a network name to administer (no spaces, hyphens or capital letters please)
> EthereumSkills
[31mERROR[0m[11-08|05:40:27.422] I also like to live dangerously, still no spaces, hyphens or capital letters
> ethereum_skills
Sweet, you can set this via --network=ethereum_skills next time!
[32mINFO [0m[11-08|05:41:23.359] Administering Ethereum network [32mname[0m=ethereum_skills
[33mWARN [0m[11-08|05:41:23.360] No previous configurations found [33mpath[0m=.puppeth\ethereum_skills
What would you like to do? (default = stats)
1. Show network stats
2. Configure new genesis
3. Track new remote server
4. Deploy network components
> 2
What would you like to do? (default = create)
1. Create new genesis from scratch
2. Import already existing genesis
> 1
Which consensus engine to use? (default = clique)
1. Ethash - proof-of-work
2. Clique - proof-of-authority
> 1
Which accounts should be pre-funded? (advisable at least one)
> 0x
Should the precompile-addresses (0x1 .. 0xff) be pre-funded with 1 wei? (advisable yes)
> yes
Specify your chain/network ID if you want an explicit one (default = random)
> 4224
[32mINFO [0m[11-08|05:46:19.792] Configured new genesis block
```

Rysunek 2. 1 Deklaracja i konfiguracja bloku Genesis

Źródło opracowania: własne

Przy pomocy geth następuje wykonanie kodu wygenerowanego w poprzednim kroku oraz stworzenie nowego konta (konto Genesis-główne). Po podaniu hasła, zostanie wygenerowany klucz publiczny oraz klucz prywatny (z ang. secret key). Po tym etapie nasza sieć jest gotowa do pierwszego uruchomienia.

```
PS C:\Users\Robert\EthereumSkills\private> & "C:\Program Files\geth\geth" --datadir . init ethereum_skills.json
[11-08|05:55:11.056] Maximum peer count: 50 |>=0 |<=50
[11-08|05:55:11.245] Allocated cache and file handles |>=50 |<=50
[11-08|05:55:11.277] Writing custom genesis block |>=50 |<=50
[11-08|05:55:11.293] Persisted trie from memory database |>=50 |<=50
[11-08|05:55:11.310] Successfully wrote genesis state |>=50 |<=50
[11-08|05:55:11.316] Allocated cache and file handles |>=50 |<=50
[11-08|05:55:11.340] Writing custom genesis block |>=50 |<=50
[11-08|05:55:11.352] Persisted trie from memory database |>=50 |<=50
[11-08|05:55:11.371] Successfully wrote genesis state |>=50 |<=50
PS C:\Users\Robert\EthereumSkills\private> ls

Directory: C:\Users\Robert\EthereumSkills\private

Mode                LastWriteTime         Length Name
----                -
d-----          08.11.2019    05:46           puppeth
d-----          08.11.2019    05:55           geth
d-----          08.11.2019    05:55           keystore
-a-----          08.11.2019    05:47       22738 ethereum_skills-aleth.json
-a-----          08.11.2019    05:47       21293 ethereum_skills-harmony.json
-a-----          08.11.2019    05:47       24353 ethereum_skills-parity.json
-a-----          08.11.2019    05:49       21294 ethereum_skills.json

PS C:\Users\Robert\EthereumSkills\private> cd .\keystore\
PS C:\Users\Robert\EthereumSkills\private\keystore> ls
PS C:\Users\Robert\EthereumSkills\private\keystore> cd ..
PS C:\Users\Robert\EthereumSkills\private> & "C:\Program Files\geth\geth" --datadir . account new
[11-08|05:59:45.317] Maximum peer count: 50 |>=0 |<=50
Your new account is locked with a password. Please give a password. Do not forget this password.
Password:
Repeat password:
Your new key was generated
Public address of the key: 0x9f9e7aa385fad937258b0048d27657878bf6d7dd
Path of the secret key file: keystore\UTC--2019-11-08T05:00:05.689850500Z--9f9e7aa385fad937258b0048d27657878bf6d7dd
- You can share your public address with anyone. Others need it to interact with you.
- You must NEVER share the secret key with anyone! The key controls access to your funds!
- You must BACKUP your key file! Without the key, it's impossible to access account funds!
- You must REMEMBER your password! Without the password, it's impossible to decrypt the key!
```

Rysunek 2. 2 Inicjacja bloku Genesis

Źródło opracowania: własne

Przy ponownym uruchomieniu skryptu, widzimy, że Ethereum Virtual Machine najpierw alokuje pamięć na cache, potem otwiera bazę danych z odpowiednimi parametrami, a następnie ładuje:

- ostatni nagłówek;
- pełny blok;
- listę transakcji (listę funkcji skrótów transakcji);

Kolejnym etapem jest uruchomienie prywatnej sieci Ethereum na lokalnym porcie RCP (Remote Procedure Calls) 8545, który jest opisywany jako punkt końcowy, oraz zewnętrznym RCP 30303. Główną różnicą między tymi dwoma portami jest widoczność. Pierwszy z wyżej wymienionych, jest dostępny jedynie na lokalnej sieci komputerowej, natomiast drugi, przy założeniu, że router przekierowuje ruch z tego portu na lokalny komputer, jest widoczny z zewnątrz. Na rysunku 2.3, możemy również zauważyć konto przypisane do bloku Genesis, ilość wątków na których pracuje maszyna, a także rozpoczęcie kopania kolejnego bloku.

```

PS C:\Users\Robert\EthereumSkills\private> .\startnode.cmd
C:\Users\Robert\EthereumSkills\private> "C:\Program Files\Geth\geth.exe" --networkid 4224 --mine --minerthreads 2 --datadir "." --nodiscover --rpc --rpcport "8
545" --port "30303" --rpcorsdomain "" --nat "any" --rpcapi eth,web3,personal,net --unlock 0 --password ./password.sec --allow-insecure-unlock
INFO [11-08 08:57:29.136] Maximum peer count
INFO [11-08 08:57:29.376] Starting peer-to-peer node
INFO [11-08 08:57:29.386] Allocated trie memory caches
INFO [11-08 08:57:29.393] Allocated cache and file handles
INFO [11-08 08:57:29.463] Opened ancient database
INFO [11-08 08:57:29.474] Initialised chain configuration
INFO [11-08 08:57:29.498] Disk storage enabled for ethash caches
INFO [11-08 08:57:29.518] Disk storage enabled for ethash DAGs
INFO [11-08 08:57:29.529] Initialising Ethereum protocol
INFO [11-08 08:57:29.656] Loaded most recent local header
INFO [11-08 08:57:29.644] Loaded most recent local full block
INFO [11-08 08:57:29.652] Loaded most recent local fast block
INFO [11-08 08:57:29.669] Loaded local transaction journal
INFO [11-08 08:57:29.683] Regenerated local transaction journal
WARN [11-08 08:57:29.694] Switch sync mode from fast sync to full sync
INFO [11-08 08:57:29.769] New local node record
INFO [11-08 08:57:29.770] IPC endpoint opened
INFO [11-08 08:57:29.780] Started P2P networking
INFO [11-08 08:57:29.787] HTTP endpoint opened
WARN [11-08 08:57:29.832]
WARN [11-08 08:57:29.841] Referring to accounts by order in the keystore folder is dangerous!
WARN [11-08 08:57:29.856] This functionality is deprecated and will be removed in the future!
WARN [11-08 08:57:29.870] Please use explicit addresses! (can search via 'geth account list')
INFO [11-08 08:57:30.844] Unlocked account
INFO [11-08 08:57:30.853] Transaction pool price threshold updated
INFO [11-08 08:57:30.862] Updated mining threads
INFO [11-08 08:57:30.875] Transaction pool price threshold updated
INFO [11-08 08:57:30.889] Etherbase automatically configured
INFO [11-08 08:57:30.905] Commit new mining work

```

Rysunek 2. 3 Uruchomienie prywatnej sieci Ethereum

Źródło opracowania: własne

Transakcje klasyczne wykonujemy poprzez uruchamianie konsoli Geth JavaScript z zastosowaniem modułów uruchomionych na naszej wirtualnej maszynie. Polecenie „eth.accounts” zwraca listę kont dostępnych w prywatnej sieci Ethereum. Przelanie środków z konta coinbase (konto Genesis) na konto o podanym indeksie następuje za pomocą funkcji „eth.sendTransaction” (Rysunek 2.5). W rezultacie otrzymujemy hash transakcji, który potem możemy zobaczyć w transakcjach jako fullhash. Kolejnym krokiem jest rozwiązanie zagadki kryptograficznej protokołu Ethash, zatwierdzenie transakcji i zapisanie jej do kolejnego bloku sieci (Rysunek 2.6).

```

PS C:\Users\Robert\EthereumSkills\private> & C:\Program Files\Geth\geth attach http://127.0.0.1:8545
Welcome to the Geth JavaScript console!
instance: Geth/v1.9.7-stable-a718daa6/windows-amd64/go1.13.4
coinbase: 0x9f9e7aa385fad937258b0048d27657878bf6d7dd
at block: 61 (Fri, 08 Nov 2019 06:37:05 CET)
modules: eth:1.0 net:1.0 personal:1.0 rpc:1.0 web3:1.0
> eth.accounts

```

Rysunek 2. 4 Uruchomienie konsoli Geth JavaScript

Źródło opracowania: własne

```

> eth.sendTransaction({from: eth.coinbase, to:eth.accounts[2],value:web3.toWei
(10,"ether")})
"0xb4eaa87d4f0de5cf75089cc217db0a7fee3c88efb14b6c1ba3076540b6ad95e4"
> eth.sendTransaction({from: eth.coinbase, to:eth.accounts[1],value:web3.toWei
(10,"ether")})
"0x2420cd493bede9aa622241fed8bec5becf3284a313f580118b3e6a61ba963a90"
>

```

Rysunek 2. 5 Transfer środków w transakcjach klasycznych

Źródło opracowania: własne

```

INFO [11-08 10:13:51.617] Commit new mining work      number=572 sealhash=30b4fc_062774 uncles=0 txs=0 gas=0 fees=0 elapsed=19.981ms
INFO [11-08 10:13:53.810] Setting new local account    address=0x9F9E7AA385Fad937258b0048d27657878bf6d7dd
INFO [11-08 10:13:53.817] Submitted transaction        fullhash=0xb4eaa87d4f0de5cf75089cc217db0a7fee3c88efb14b6c1ba3076540b6ad95e4 recipient=0x83c
7ad681eaaac4558da7fa75f6f04c8d6794ba4d
INFO [11-08 10:13:54.598] Commit new mining work      number=572 sealhash=8de735_7140f6 uncles=0 txs=1 gas=21000 fees=2.1e-05 elapsed=0s
INFO [11-08 10:13:55.607] Successfully sealed new block number=572 sealhash=8de735_7140f6 hash=023f4a_80d346 elapsed=1.009s
INFO [11-08 10:13:55.617]   [] block reached canonical chain number=565 hash=9e7cbb_bb1bc8
INFO [11-08 10:13:55.632]   [] mined potential block        number=572 hash=023f4a_80d346
INFO [11-08 10:13:55.649] Commit new mining work      number=573 sealhash=f7ab7f_31f4a4 uncles=0 txs=0 gas=0 fees=0 elapsed=30.993ms
INFO [11-08 10:13:59.733] Submitted transaction        fullhash=0x2420cd493bede9aa622241fed8bec5becf3284a313f580118b3e6a61ba963a90 recipient=0x40f
81337cd974458b0174e854489e5d37ae4d48
INFO [11-08 10:14:01.623] Commit new mining work      number=573 sealhash=60256f_76db06 uncles=0 txs=1 gas=21000 fees=2.1e-05 elapsed=5.001ms
INFO [11-08 10:14:02.228] Successfully sealed new block number=573 sealhash=60256f_76db06 hash=727d90_f38ead elapsed=605.006ms
INFO [11-08 10:14:02.235]   [] block reached canonical chain number=566 hash=832817_44711f
INFO [11-08 10:14:02.249]   [] mined potential block        number=573 hash=727d90_f38ead
INFO [11-08 10:14:02.268] Commit new mining work      number=574 sealhash=95983a_2cefd5 uncles=0 txs=0 gas=0 fees=0 elapsed=33.000ms
INFO [11-08 10:14:10.775] Successfully sealed new block number=574 sealhash=95983a_2cefd5 hash=e23d09_4a8d5c elapsed=8.539s
INFO [11-08 10:14:10.782]   [] block reached canonical chain number=567 hash=71158b_9a0622
INFO [11-08 10:14:10.794]   [] mined potential block        number=574 hash=e23d09_4a8d5c
INFO [11-08 10:14:10.810] Commit new mining work      number=575 sealhash=b9969f_ba8d2f uncles=0 txs=0 gas=0 fees=0 elapsed=28.002ms
INFO [11-08 10:14:12.015] Successfully sealed new block number=575 sealhash=b9969f_ba8d2f hash=ce4be5_05a048 elapsed=1.233s
INFO [11-08 10:14:12.028]   [] block reached canonical chain number=568 hash=1c5063_02aa15
INFO [11-08 10:14:12.037]   [] mined potential block        number=575 hash=ce4be5_05a048

```

Rysunek 2. 6 Zatwierdzenie transakcji i dołączenie jej do bloku

Źródło opracowania: własne

## 2.3 Kompilator języka Solidity

Solidity jest skryptowym wysokopoziomowym językiem programowania o podobnej składni semantycznej do JavaScript, C++ oraz Pythona. Ten język został stworzony do rozszerzania możliwości EVM. Jak każdy język wysokopoziomowy, posiada kilka kompilatorów, które weryfikują składnię języka, jednakże każdy z nich jest oparty o Solc.

Solc to podstawowy kompilator Solidity. Początkowo znajdował się w pakiecie Geth, ale obecnie wymaga własnej instalacji która może odbyć się poprzez polecenie **npm install -g solc**, jeśli posiadamy zainstalowane oprogramowanie Node.js. Kompilator zamienia język Solidity na Ethereum Bytecode, który jest zrozumiały przez EVM.

## 2.4 Truffle

Truffle to najbardziej popularne środowisko developerskie dla platformy Ethereum, które:

- ułatwia zarządzanie kontraktami;
- umożliwia tworzenie zautomatyzowanych testów dla kontraktów;
- ułatwia wdrażanie kontraktów do sieci;
- ułatwia zarządzanie siecią;
- posiada konsolę developerską, która ma dostęp do wszystkich napisanych kontraktów oraz komend środowiska truffle

Najbardziej lubianą opcją przez developerów jest złożoność środowiska, które nie tylko ma funkcjonalności do zarządzania kontraktami, ale również pozwala wdrożyć te umowy do aplikacji webowych, czy stworzyć front-end dla zdecentralizowanej aplikacji wraz z planem testów. Ponadto posiada moduł testowej platformy Ethereum Virtual Machine, do którego możemy się odwołać poprzez wpisanie polecenia **truffle develop**. W wyniku wywołania tej komendy, otrzymujemy

wygenerowane dziesięć kont wraz z kluczami prywatnymi oraz uruchomioną konsolę developerską (Rysunek 2.7).

```
PS C:\Users\Robert\inz\truffle> truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x4726d9c0295b1d3c827c78bec37542e3907a6af1
(1) 0xa2296b87295e859f85e8dae920366aae5fb88e80
(2) 0xde39139f200cdc90c9916efa66c9e9b168c23d40
(3) 0x3c9c328d1629c8c45b794a1b77b4a3909c7d8444
(4) 0x54f17bee34f2d71952858f5c2f6b78a8c8fa38a1
(5) 0x5d988425c416877b25e19d84dfd67459ec45914
(6) 0xa345a8dda006fc8dde8cfac01cd0aae817cf3f49
(7) 0x63a9904da9ed5d259c7c592d8bf4db5a7b9551f8
(8) 0x483a653d46ba10302b9e59bea3e3d38f7de9e926
(9) 0x1dc063c6eae4360500a2546fa0f7cb73bce9c329

Private Keys:
(0) 91cb0cd3582dc30050e1fa7a5c75379c1d12d861c1166f234ac96f155cc185f0
(1) 6bfddf493ee5fd8689c6a895c1b121719ecf872d4229aa73dcedde8e036cf823
(2) c615540a96647ab099585f3726cb1f4e8bcc0f494c081e54c1c72481014cb744
(3) 8fc45a7a51de694bd2ec8cc28577e8a5cc4b2bd0bd0129cdd9644224b1cf8e2b
(4) ab37615094d08594a0285ff7b3df96bc2bcf3c9f753e27783a6993d67108d3b5
(5) 5822e524faf9f0b18f1a4b8009445994f24535f3c74b75810906e9ce0538f01d
(6) 014494a4829bfaccb4260569fd349801a11fd2ccc111598b02d36184c82ecbe3
(7) 267d80dc5252623a96d6bb26b81b5992c5ed2345a05e2719040844a325fea9af
(8) da4c0c73d232d764bef00b6210e9a97d24e59af27d768998498a13e7308504df
(9) 7bb00633dd85aeea922c4ff066c23434c2d87713307a6473625acd014f115852

Mnemonic: vanish crisp lumber canoe brass aim zebra job seek sustain innocent voice

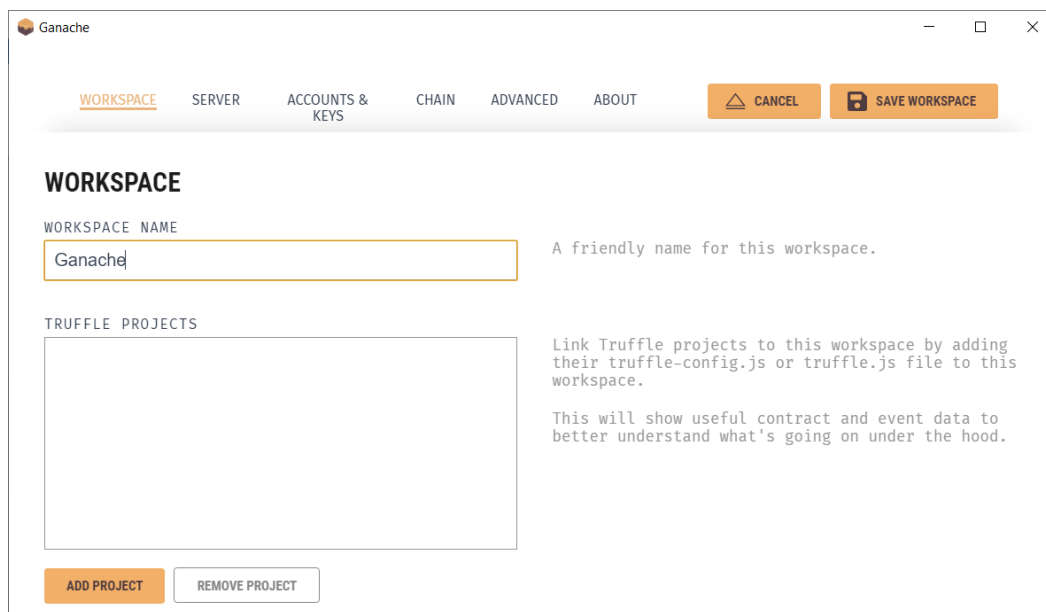
⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.
```

Rysunek 2. 7 Wygenerowanie kont przez polecenie „truffle develop”

Źródło opracowania: własne

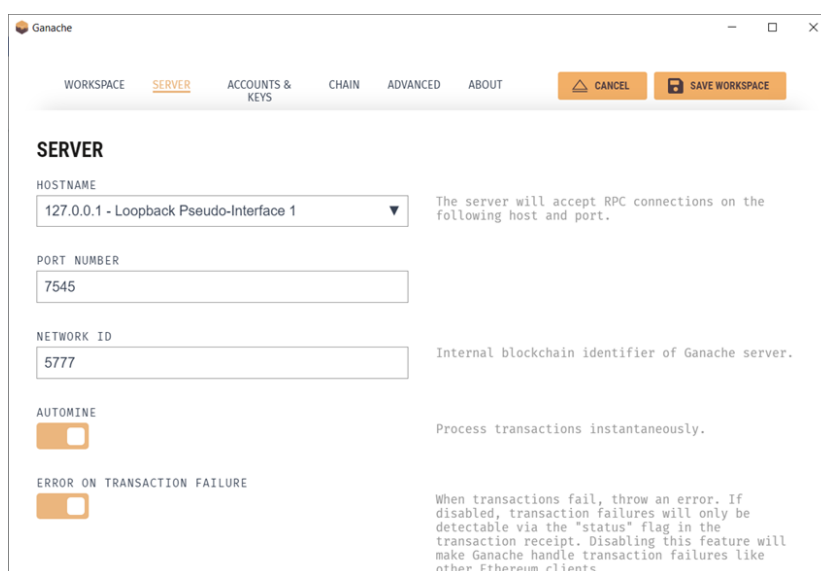
## 2.5 Ganache – graficzne środowisko testowe

Jest to darmowe narzędzie z otwartym źródłem (z ang. Open Source), wchodzące w skład pakietu „truffle”, umożliwiające uruchomienie prywatnego środowiska Ethereum w technologii Blockchain do wgrywania Inteligentnych Kontraktów, rozwijania aplikacji, a także przeprowadzania testów. Poprzez graficzny interfejs, możemy stworzyć prywatną maszynę ustawiając parametry łańcucha, kont, czy serwera (Rysunek 2.8-2.11).



**Rysunek 2. 8 Okno konfiguracji Ganache**

Źródło opracowania: własne



**Rysunek 2. 9 Lokalne ustawienia prywatnej maszyny**

Źródło opracowania: własne

**ACCOUNTS & KEYS**

ACCOUNT DEFAULT BALANCE  
100  
The starting balance for accounts, in Ether.

TOTAL ACCOUNTS TO GENERATE  
10  
Total number of Accounts to create and pre-fund.

AUTOGENERATE HD MNEMONIC  
☐  
Turn on to automatically generate a new mnemonic and account addresses on each run.

wood best truth code token clip call risk crater various dist  
Enter the Mnemonic you wish to use.

LOCK ACCOUNTS  
☐  
If enabled, accounts will be locked on startup.

Rysunek 2. 10 Deklaracja i ustawienia kont

Źródło opracowania: własne

**GAS**

GAS LIMIT  
6721975  
Maximum amount of gas available to each block and transaction. Leave blank for default.

GAS PRICE  
20000000000  
The price of each unit of gas, in WEI. Leave blank for default.

**HARDFORK**

HARDFORK  
Petersburg  
The hardfork to use. Default is Petersburg.

Rysunek 2. 11 Podstawowe ustawienia łańcucha

Źródło opracowania: własne

Rysunek 2.8 przedstawia możliwość wyboru nazwy prywatnej maszyny oraz dołączenia zewnętrznych projektów. Podstawowe ustawienia serwera w tym:




- Adres;
- Port;
- Identyfikator sieci;
- Ustawienia natychmiastowego kopania znajdują się w zakładce „Server” (Rysunek 2.9).



Kolejny rysunek przedstawia funkcje związane z zarządzaniem kontami. Na tej zakładce definiuje się ich ilość do wygenerowania wraz z saldem początkowym (Rys. 2.10). Następny rysunek (Rys. 2.11) pokazuje zakładkę „Chain”, która uwzględnia maksymalną ilość dostępnego gazu na każdy blok oraz jego cenę wyrażoną w jednostce Wei. Lista rozwijana „Hardfork” odnosi się do wersji systemu, na jakiej prywatna maszyna będzie pracować.

Po uruchomieniu skonfigurowanego środowiska, mamy możliwość uzyskania informacji o koncie, skonfigurowania w łatwy sposób zaawansowanych opcji kopania, czy sprawdzenia każdej transakcji w bloku wraz ze wszystkimi szczegółami (Rysunki 2.12-2.19).

Zakładka „Accounts” (Rys. 2.12) zawiera adresy, salda, liczniki przeprowadzonych transakcji (TX count), a także klucze prywatne dopisane do każdego konta. Domyślnie przy uruchomieniu nowego projektu, narzędzie tworzy ustawioną w procesie konfiguracji ilość kont, a następnie zasila je przypisaną kwotą. W przypadku przedstawionym poniżej, zostały ustawione wartości domyślne, przez co Ganache utworzył 10 kont, a następnie przypisał im kwotę 100ETH.

ADDRESS <b>0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED</b>	BALANCE <b>99.98</b> ETH	TX COUNT 4	INDEX 0	
ADDRESS <b>0xc28A3317410A3b5593bA403B5c869eF1c0B337cd</b>	BALANCE <b>100.00</b> ETH	TX COUNT 0	INDEX 1	
ADDRESS <b>0x39aD3f34BC4854bD969Fcb4da9517511A8515961</b>	BALANCE <b>100.00</b> ETH	TX COUNT 0	INDEX 2	

**Rysunek 2. 12 Wygenerowane konta i ich położenie**

Źródło opracowania: własne

Po kliknięciu obrazka po prawej stronie okna, pojawi się klucz prywatny przypisany do konta. Ten proces został przedstawiony na Rysunku 2.13.



0X2658ED570AE2B7A4D9003EA9A1DC62EE3FE36EED

 PRIVATE KEY

4918c1a9ec0c66db40920c4ce50dd969f94db869292a69e98eb3078f4b833e

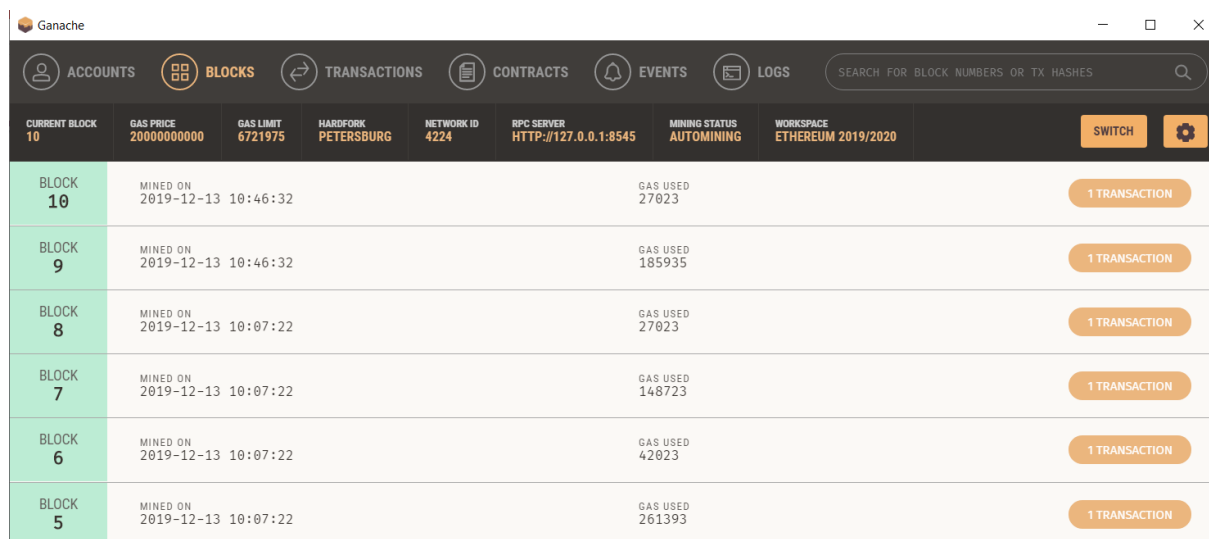
DONE


Rysunek 2. 13 Klucz prywatny jednego z kont

Źródło opracowania: własne

Z architektury Blockchain wiemy, że każda transakcja (lub ich zbiór przedstawiony w sposób drzewa Merklego) jest zapisywana w blokach. Ten proces można zaobserwować w zakładce „Blocks”, która została przedstawiona na Rysunku 2.14. Informacje widoczne na ekranie pozwalają zobaczyć podstawowe dane zbiorcze o bloku takie, jak:

- Numer bloku;
- Data wydobycia;
- Użyty Gas;
- Ilość transakcji w bloku.

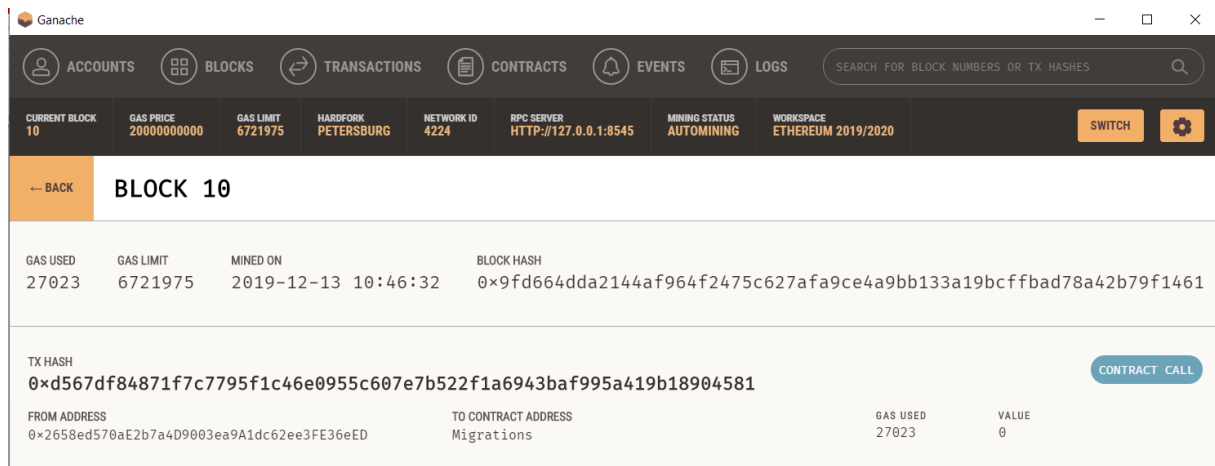


CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDWARE	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE	
10	20000000000	6721975	PETERSBURG	4224	HTTP://127.0.0.1:8545	AUTOMINING	ETHEREUM 2019/2020	SWITCH 
BLOCK 10	MINED ON 2019-12-13 10:46:32					GAS USED 27023		1 TRANSACTION
BLOCK 9	MINED ON 2019-12-13 10:46:32					GAS USED 185935		1 TRANSACTION
BLOCK 8	MINED ON 2019-12-13 10:07:22					GAS USED 27023		1 TRANSACTION
BLOCK 7	MINED ON 2019-12-13 10:07:22					GAS USED 148723		1 TRANSACTION
BLOCK 6	MINED ON 2019-12-13 10:07:22					GAS USED 42023		1 TRANSACTION
BLOCK 5	MINED ON 2019-12-13 10:07:22					GAS USED 261393		1 TRANSACTION

Rysunek 2. 14 Wykopane bloki w EVM przy użyciu Ganache'a

Źródło opracowania: własne

Podstawowe dane jednakże nie są zbyt obszerne. Większą szczegółowość otrzymamy klikając na interesujący nas blok, co możemy zauważyć na rysunku 2.15.



Rysunek 2. 15 Dane szczegółowe bloku

Źródło opracowania: własne

Oprócz informacji przedstawionych na rysunku 2.14, widzimy:

- Gas limit;
- Hash bloku;
- Hash transakcji (TX Hash);
- Adres konta, z którego została wywołana transakcja;
- Nazwę kontraktu wykonanego w transakcji;
- Odwołanie do kontraktu (Contract call), który jeszcze bardziej uszczegóławia informacje na temat transakcji, pokazując m.in. nazwę kontraktu, wartość użytego Etheru czy funkcje występujące w nim (Rysunek 2.15).



Ganache

ACCOUNTS

BLOCKS

TRANSACTIONS

CONTRACTS

EVENTS

LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK  
10

GAS PRICE  
2000000000

GAS LIMIT  
6721975

HARDFORK  
PETERSBURG

NETWORK ID  
4224

RPC SERVER  
HTTP://127.0.0.1:8545

MINING STATUS  
AUTOMINING

WORKSPACE  
ETHEREUM 2019/2020

SWITCH

TX HASH

0xd567df84871f7c7795f1c46e0955c607e7b522f1a6943baf995a419b18904581

CONTRACT CALL

FROM ADDRESS

0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED

TO CONTRACT ADDRESS

Migrations

GAS USED

27023

VALUE

0

TX HASH

0x36318d4f7def9cf3516c374a960940f32738e19543994cc4e2f5782096b72cea

CONTRACT CREATION

FROM ADDRESS

0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED

CREATED CONTRACT ADDRESS

0x337D0D96C028Dbd3b06E498E2e4371d80EBC58ec

GAS USED

185935

VALUE

0

TX HASH

0x1393b7dff2e4e80b4a807b8d6e77ca00b453cb48f1987fe56c636923c1418631

CONTRACT CALL

FROM ADDRESS

0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED

TO CONTRACT ADDRESS

Migrations

GAS USED

27023

VALUE

0

TX HASH

0x6f6cac620ae1e92556e76435cb312a7581a149da28cfb8a0011f3bebfad781f

CONTRACT CREATION

FROM ADDRESS

0x2658ed570aE2b7a4D9003ea9A1dc62ee3FE36eED

CREATED CONTRACT ADDRESS

0xB95f086f579a745928b4227929A023187827567D

GAS USED

148723

VALUE

0

TX HASH

0x24604382e0b2db8702a4accab99177eb97c4f6e2b0b6e988dda6866eccee731

CONTRACT CALL

Rysunek 2. 17 Spis przeprowadzonych transakcji

Źródło opracowania: własne

← BACK TX 0xd28f949e84ad8be7f1ad54f54c4a3019a1d58b2d0c1c7198170d970945515dde				
SENDER ADDRESS 0xFc270144E833C364753cFC5F204bc242edEEEE43	CREATED CONTRACT ADDRESS 0x65d1Db9399Bb21AdccBe4b8f410E7aF99132A02	CONTRACT CREATION		
VALUE 0.00 ETH	GAS USED 141314	GAS PRICE 20000000000	GAS LIMIT 6721975	MINED IN BLOCK 33
TX DATA 0x60806040526008060006101000a81548160ff02191690831515021790555034801561002a57600080fd5b5061012c8061003a6000396000f3fe608060405260043610601c5760003560e01c806314df9e43146021575b600080fd5b606060048036036020811015603557600080fd5b81019080803573ff1690602001909291905050506062565b005b600080809054906101000a900460ff1660f4578073ff166002604051806000190506006040518083038185875af1925050503d806000811460d1576040519150601f19603f3d011682016040523d82523d6000602084013e60d6565b606091505b50505060016000806101000a81548160ff0219169083151502179055505b5056fea265627a7a7231582028aa533aa6bdd7f36f9a3bf12f70eee35269077388b0e344b0b80294ece0954e64736f6c634300050c0032				

Rysunek 2. 18 Dane szczegółowe stworzonej inicjalizacji kontraktu do puli

Źródło opracowania: własne

Szczegółowe dane kontraktu (obiektu) wraz z jego kodem operacyjnym, który wykonywany jest na EVM oraz mapą źródła (ang. SourceMap), odpowiadającą za przypisanie każdej instrukcji kontraktu do sekcji kodu źródłowego, można wygenerować używając polecenia: **solcjs -bin nazwa\_pliku.sol > nazwa\_pliku.bin**, a jego fragment, dla kontraktu Bob1.sol, przedstawiono na Listingu 2.1, natomiast cała zawartość pliku jest dołączona do pracy i oznaczona jako Załącznik 1.

```

1. {
2.     "linkReferences": {},
3.     "object": "608060405260008060006101000a81548160ff020217905...",
4.     "opcodes": "aaPUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x0 DUP1 ... ",
5.     "sourceMap": "28:148:0:-;;;55:5;45:15;;;;;;;;;;;;;28:148;8:9:- ... "
6. }

```

**Listing 2. 1 Fragment kodu BIN (Bytecode) dla kontraktu Bob1.sol**

Źródło opracowania: własne

Do wdrożenia kontraktu do puli transakcji, nie wystarczy tylko bytecode. Potrzeba również stworzyć plik zawierający potrzebne dane do uruchomienia konkretnych metod w kontrakcie. Ten dokument nosi nazwę ABI i można go wygenerować w podobny sposób co wcześniej wspomniany i przedstawiony na listingu 2.1 – bytecode, a mianowicie poprzez polecenie **solcjs -abi nazwa\_pliku.sol > nazwa\_pliku.abi**, a jego zawartość dla wyżej wymienionego kontraktu przedstawia listing 2.2.

```

1. [
2.     {
3.         "constant": false,
4.         "inputs": [
5.             {
6.                 "internalType": "address",
7.                 "name": "c",
8.                 "type": "address"
9.             }
10.        ],
11.        "name": "ping",
12.        "outputs": [],
13.        "payable": false,
14.        "stateMutability": "nonpayable",
15.        "type": "function"
16.    }
17. ]

```

**Listing 2. 2 Kod ABI dla kontraktu Bob1.sol**

Źródło opracowania: własne

Ganache posiada również zakładkę „contracts” (Rysunek 2.19), w której możemy znaleźć:

- lokalizację kontraktów;
- nazwy;
- adresy, a zarazem status kontraktu (wdrożony posiada adres w łańcuchu);
- ilość wykonanych transakcji do danego kontraktu.

NAME	ADDRESS	TX COUNT
Hello	Not Deployed	0
Migrations	0xBBCEb4676eba38fA773F24A4Ca43ad5635e1EF2d	2
Reentrancy_attack_insecure	0xB95f086f579a745928b4227929A023187827567D	0
Vulnerable	0x337D0D96C028Dbd3b06E498E2e4371d80EBC58ec	0

Rysunek 2. 19 Wdrożone kontrakty do łańcucha bloków

Źródło opracowania: własne

Chcąc sprawdzić szczegółowo konkretny blok po wdrożeniu używamy konsoli dołączonej do pakietu truffle, lub Geth JavaScript Console. W obu przypadkach, używając interfejsu web3, po wpisaniu polecenia **web3.eth.getBlock(numer)**, otrzymamy następujące dane (Rys.2.20):

```
truffle(development)> web3.eth.getBlock(33)
{ number: 33,
  hash:
    '0x4b260ab7e62a782f5c114c1b972c14546ed19187ec1f6eb4bde9d6bbcc4b7b10',
  parentHash:
    '0xee28e8f80981ee8ad45953e941361041e9b81203ed1aa46dcc9a39ffec58f16b',
  mixHash:
    '0x0000000000000000000000000000000000000000000000000000000000000000',
  nonce: '0x0000000000000000',
  sha3Uncles:
    '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
  logsBloom:
    '0x0000000000000000000000000000000000000000000000000000000000000000',
  transactionsRoot:
    '0x648823682a5f3c96837de863b6912269f1831ba466ae4878ec724166d82f193',
  stateRoot:
    '0x35dc54701a1ed6cd9e33610357436044fb5e9b139bbee56a8035ea746014bf75',
  receiptsRoot:
    '0xe80ccf09946a93a730ba1047dd6af374398eb8c8dd102f993b281c4422f72992',
  miner: '0x0000000000000000000000000000000000000000000000000000000000000000',
  difficulty: '0',
  totalDifficulty: '0',
  extraData: '0x',
  size: 1000,
  gasLimit: '0x6691b7',
  gasUsed: '0x22802',
  timestamp: 1580683402,
  transactions:
    [ '0xd28f949e84ad8be7f1ad54f54c4a3019a1d58b2d0c1c7198170d970945515dde' ],
  uncles: [] }
```

Rysunek 2. 20 Informacje o wdrożonym bloku

Źródło opracowania: własne

[illegible]

Źródło opracowania: własne

### 3. Semantyka Inteligentnych Kontraktów

Kontrakty to programy, które są nieodłączną częścią Ethereum, a ich struktura jest podobna do klas w znanych językach obiektowych. Jednak by poprawnie zostały przetłumaczone na bytecode zrozumiały dla EVM, potrzebna jest odpowiednia składnia języka wysokopoziomowego. Istnieje duża ilość języków do pisania inteligentnych umów. Vyper, Idris, Formality czy Solidity, to tylko kilka z nich. W tym rozdziale zostanie opisana struktura i składnia ostatniego z wymienionych języków na kilku przykładach.

#### 3.1 Struktura kontraktów

Podobnie jak klasy w obiektowych językach programowania (np. C#, Java), kontrakty napisane w Solidity mogą posiadać następujące deklaracje [16]:

- Zmiennych;
- Funkcji (w tym konstruktora, który domyślnie jest pusty, jeśli jego nie zdeklarujemy. W wersjach powyżej 0.5.0, deklaracja konstruktora, jako nazwa kontraktu, została zmieniona na słowo kluczowe „constructor”);
- Modyfikatorów;
- Struktur kontrolnych
- Eventów;
- Struktur danych;
- Interfejsów;
- Wyjątków;

Nie mogło również zabraknąć dyrektyw kompilatora, jednakże w tym przypadku, podajemy wersję kompilatora *solc* poprzez napisanie w pierwszej linii następującej sekwencji: **pragma solidity <wersja>**, gdzie w parametrze <wersja>, możemy podać:

- konkretną wersję (0.x.y, gdzie x jest główną wersją kompilatora, a y jest aktualizacją naprawiającą błędy (ang. Bugfix))
- przedział ( $\geq 0.4.2$   $< 0.7.0$ ),
- konkretną wersję z najnowszymi aktualizacjami ( $\wedge 0.x.0$ , gdzie  $\wedge$  pobiera najnowszą wersję bugfixa np. jeśli wpiszemy  $\wedge 0.5.0$ , a kompilator w wersji 5 posiada jakąś aktualizację, to nieświadomie użyjemy tej wersji)

Ogólną strukturę kontraktu, możemy zauważyć w listingu 3.1



```

1. pragma solidity ^0.5.12;
2.
3. contract exampleOfStructure{
4.     int public integerValue;
5.     uint uintVariable=10;
6.     string stringVariable;
7.     address addressVariable;
8.     companyStruct objOfCompanyStruct;
9.     bool internal internalBoolVariable;
10.    bool constant constantBoolVariable=true;
11.
12.    //Structure declaration
13.    struct companyStruct{
14.        string companyName;
15.        uint amountOfEmployee;
16.        bool listedCompany;
17.        string CEO;
18.    }
19.
20.    //constructor
21.    constructor(){
22.        objOfCompanyStruct= companyStruct("Test Company",12,false,"Robert Kurcwald");
23.        integerValue = 25;
24.        uint returnedVal=getUintValue();
25.    }
26.
27.    //functions
28.    function getUintValue() returns (uint){
29.
30.        //condition definition
31.        if(uint(integerVariable-10) >= uintVariable)
32.        {
33.            uintVariable = 5;
34.        } else{
35.            uintVariable += 10;
36.        }
37.        return uintVariable+uint(integerVariable);
38.    }
39. }

```

**Listing 3. 1 Przykładowy kontrakt**

Źródło opracowania: własne

## 3.2 Zmienne i ich typy

Analizując przykładowy kontrakt, przedstawiony jako listing 3.1, możemy zauważyć kilka typów danych [16]:

- int/uint – zmienne liczbowe typu całkowitego. Wykorzystują następujące operatory:
  - porównania („<=”, „==”, „>=”, „<”, „>”, „!=“)
  - bitowe („&”, „|”, „^”, „~“)
  - arytmetyczne („+”, „-”, „/”, „%”, „<”, „>”, „\*\*“)
- string – zmienne łańcucha znaków
- address – zmienne przechowujące konkretne adresy kont w blockchainie. Zwykle używane w celu przechowania adresu konta, na którym kontrakt wykonuje polecenia (np. przelew ETH z jednego adresu na drugi). Wykorzystują podstawowe operatory porównania
- bool – zmienne mogące przechowywać tylko dwie wartości: prawda lub fałsz. Wykorzystują podstawowe operatory logiczne:
  - „!” – negacja (wyrażenie „NEG”);
  - „&&” – spójnik (wyrażenie „AND”);
  - „||” – alternatywa (wyrażenie „OR”);
  - „==” – porównanie
  - „!=” – negacja porównania
- arrays – co prawda nie istnieje taki typ danych, ale każdy wyżej wymieniony typ, możemy przedstawić w sposób tablicowy np. uint[] tworzy dynamiczną tablicę wartości liczbowych, które nie posiadają znaku.
- struct – definicja własnej struktury danych.

Dodatkowe typy zmiennych występujących w kontraktach to:

- enum – spersonalizowany typ danych, służący do przechowywania ilości wystąpień
- bytes – 8bitowe oznaczone zmienne liczbowe typu całkowitego. Wszystko przechowywane w pamięci używa właśnie tej formy
- mapping – swojego rodzaju tablice hashujące, które zostają wirtualnie zainicjowane, by każdy możliwy klucz był dostępny i odpowiadał wartości domyślnej danego typu. Deklaracja tego typu przedstawia się w następujący sposób: **mapping(TypPoczątkowy => TypKońcowy)**, np.: **mapping(address => uint)**

Każdy z wyżej wymienionych typów danych, domyślnie jest traktowany, jakby miał dopisek „storage”, czyli zostaje przechowany w łańcuchu bloków. Wartości tych zmiennych możemy zmieniać podczas działania kontraktu, ale po jego wykonaniu, nie wrócą do wartości nominalnych. Takie zmienne nazywamy zmiennymi stanu (ang. State variables). Jeśli potrzebujemy zmiennych tymczasowych, to będzie trzeba użyć parametru „memory”, który zachowuje wartości w pamięci maszyny wirtualnej (EVM).

Niektóre zmienne przedstawione w listingu 3.1 są powiązane z kwalifikatorami, które przedstawiają się w następujący sposób:

- `internal` – domyślnie ustawiony parametr zmiennej, który pozwala na późniejszą zmianę wartości, tylko w obrębie kontraktu, w którym została ona zadeklarowana, lub w kontraktach dziedziczących
- `private` – parametr przypominający *internal*, jednakże pozwala na zmianę wartości tylko w kontrakcie, w którym została zadeklarowana
- `public` – parametr pozwalający odwołać się do bezpośrednio do zmiennej z każdego kontraktu. Kompilator automatycznie generuje do takich zmiennych funkcję `get()`, która zwraca wartość
- `constant` – parametr, który poprzez przypisanie wartości przy deklaracji, zmienia zmienną na stałą

### 3.3 Struktury kontrolne

Głównymi typami struktur kontrolnych są warunki oraz pętle, które deklaruje i wykonuje się podobnie, jak w języku C++ czy Java.

Warunki (ang. conditions) pozwalają na wykonywanie różnych instrukcji w zależności od prawdziwości zadeklarowanych wartości logicznych. Przykład tego typu struktury został przedstawiony w listingu 3.1 w linii 31, gdzie zostaje sprawdzone czy przerzutowana (przekonwertowana) na zmienną typu całkowitego dodatniego (`uint`), wartość wyrażenia jest większa lub równa zadeklarowanej wartości `uintVariable`.

Pętle umożliwiają wykonywanie instrukcji określoną ilość razy. W Solidity mamy trzy rodzaje pętli, a każda z nich ma określony warunek graniczny (kończący):

- `For` - pętla wykonująca się konkretną ilość razy, która po wykonaniu danego ciągu instrukcji wykonuje inkrementację lub dekrementację zadeklarowaną jako trzeci parametr wywołania. Przykładowa struktura przedstawia się w następujący sposób:

for(deklaracja\_zmiennej\_sterującej\_oraz\_wartości; porównanie\_zmiennej\_do\_wartości; inkrementacja\_lub\_dekrementacja\_zmiennej\_sterującej) np. for(uint i=0;i<=10;i++)

- While – pętla wykonująca się do momentu spełnienia zadeklarowanego warunku, który sprawdzany jest przed każdym wykonaniem danego ciągu instrukcji. Przykładowa struktura przedstawia się w następujący sposób: while(warunek\_kończący) np. while(i<10);
- Do...while – pętla, której schemat działania jest podobny do while, z tą różnicą, że pętla wykonuje się przynajmniej raz, gdyż warunek kończący jest sprawdzany dopiero po przejściu wszystkich instrukcji w tej strukturze

Struktury kontrole posiadają również dwa słowa kluczowe w odniesieniu do pętli połączonych z warunkami:

- Continue;
- Break;

Przykładowa funkcja implementująca słowa kluczowe w strukturach kontrolnych została przedstawiona poniżej. Po sprawdzeniu warunku „variable == 5”, słowo **continue** omija wszystkie poniższe instrukcję i wymusza na pętli przejście do następnego kroku. Natomiast **break** jest przeciwieństwem poprzedniego wyrazu kluczowego i oznacza przerwanie działania pętli w momencie wykonania warunku.

```
1. uint a=0;
2. function ControlStructure(uint variable) {
3.     while(variable >= 0){
4.         if(variable == 5) continue;
5.         variable = variable - 1;
6.         a++;
7.     }
8.     for(uint i=0; i<=50; i++)
9.     {
10.        a++;
11.        if(a == 4) break;
12.    }
13. }
```

Listing 3. 2 Struktury kontrolne – przykład

Źródło opracowania: własne

### 3.4 Funkcje i modyfikatory funkcji

Funkcje to wykonywalne w kontraktach jednostki, które pozwalają na odczyt i modyfikację zmiennych, a także wywoływanie innych kontraktów, czy odczytywanie danych z łańcucha bloków.

Dzieli się je na cztery typy względem widoczności:

- `private` – funkcje prywatne, podobnie jak zmienne, mogą zostać wykonane tylko w kontrakcie, w którym zostały zadeklarowane
- `public` – funkcje publiczne, podobnie jak publiczne zmienne, mogą zostać wywołane w kontrakcie deklaracji lub poprzez wywołanie zapytania (ang. `message`) z innych umów czy zapytań (np. konsola Truffle)
- `internal` – funkcje wewnętrzne, podobnie jak zmienne, mogą zostać wywołane w kontrakcie, w którym zostały zadeklarowane (bez potrzeby używania słowa kluczowego `this`, jak to ma miejsce w `external`) lub w kontraktach które po nim dziedziczą
- `external` – funkcje zewnętrzne mogą zostać wywołane z innych kontraktów czy zewnętrznych transakcji. Funkcje te nie mogą zostać wywołane wewnętrznie bez słowa kluczowego `this` np. `this.function()` wykona się w podanym kontrakcie, lecz `function()` zwróci błąd na poziomie kompilacji

Funkcje posiadają również dodatkowe typy kwalifikatorów, które zmieniają zachowanie względem możliwości modyfikowania zmiennych stanu, a przedstawiają się one w sposób następujący:

- `constant` – funkcje używające tego słowa kluczowego, nie mogą modyfikować blockchaina. Służą jedynie do odczytania wartości zmiennych i zwrócenia ich do osoby wywołującej (ang. `caller`), a co za tym idzie, nie mogą zmodyfikować zmiennych, stworzyć kolejnego kontraktu, odwołać się do eventu itp.
- `view` – w nowszych wersjach Solidity (>0.4.0), to słowo kluczowe zastąpiło `constant`, a jego schemat działania pozostał niezmienny
- `pure` – funkcje używające tego słowa kluczowego, nie mogą odczytywać wartości zmiennych stanu, ani ich zapisywać. Obecnie Solidity daje możliwość odwołania modyfikacji przy błędzie. Służą do tego metody `revert()` oraz `require()`
- `payable` – funkcje używające tego słowa kluczowego, mają możliwość akceptowania Etheru od wywołującego.

- Fallback – specjalny typ funkcji występujący wyłącznie w Ethereum, nie posiadający żadnych argumentów czy nazwy. Nie zwraca również wartości, a jego widoczność jest ustawiona na external. Jeśli funkcja fallback nie posiada parametru payable, funkcja zwraca wyjątek podczas przesyłania Etheru. Główne zadania tego typu funkcji to:
  - przechwytywanie wywołań (ang. call to the contract), jeśli inne funkcje nie odpowiadają identyfikatorowi odwołania.
  - odwołania do kontraktu poprzez wysłanie Etheru bez danych (np. klasyczna transakcja)
  - przy wystarczającej ilości gazu, funkcja może zostać wywołana, jak każda inna funkcja z parametrem widoczności external

Istnieje również pojęcie przepełnienia funkcji, które działa podobnie, jak w C++ czy Java i zostało przedstawione na listingu 3.3. Przepełnienie polega na użyciu tej samej nazwy funkcji w innym celu. W podanym przykładzie, funkcje overloadFunc zwracają dwie wartości (w tym wypadku zwracają wartości zmiennych, jednakże, gdyby nie było nazwy np. returns(uint,string), funkcja zwraca wartości według kolejności deklaracji. Należy wtedy pamiętać, że oba parametry muszą mieć wartość np. return 1; return „Tak”);, które będą miały inne parametry wejściowe, względem wybranej funkcji.

Przy odwołaniu do:

- linii 7, uzyskane wyniki są predefiniowane, a wartości zmiennych wynoszą: add=14, sub=6;
- linii 12, uzyskane wyniki są zależne od parametru firstValue
- linii 18, uzyskane wyniki są zależne od parametrów: firstValue oraz secondValue
- Przy ponownym odwołaniu do linii 7 po wykonaniu linii 12 lub 18 wyniki predefiniowane zmieniają się względem przyjętych parametrów

```

1. pragma solidity ^0.5.12;
2.
3. contract A{
4.     uint uintVariable=10;
5.     uint uintVariable2=4;
6.
7.     function overloadFunc() public pure returns (uint add, uint sub){
8.         add=uintVariable+uintVariable2;
9.         sub=uintVariable-uintVariable2;

```

```

10. }
11.
12. function overloadFunc(uint firstValue) public pure returns (uint add, uint sub){
13.     uintVariable=firstValue;
14.
15.     add=uintVariable+uintVariable2;
16.     sub=uintVariable-uintVariable2;
17. }
18.
19. function overloadFunc(uint firstValue, uint secondValue) public pure returns (uint add, uint
    sub){
20.     uintVariable=firstValue;
21.     uintVariable2=secondValue;
22.
23.     add=uintVariable+uintVariable2;
24.     sub=uintVariable-uintVariable2;
25. }
26. }

```

**Listing 3. 3 Kontrakt ilustrujący przepełnienie funkcji**

Źródło opracowania: własne

Wcześniej wspomniano o możliwościach modyfikowania zmiennych stanu oraz funkcji i w tym celu stworzono modyfikatory, które w łatwy sposób mogą zmienić właściwości funkcji. Mogą automatycznie sprawdzić, podczas wykonywania funkcji, czy warunek zadeklarowany, jest spełniony np. czy użytkownik posiada wystarczające środki do transakcji lub czy jest właścicielem danego konta (Listing 3.4). Przy poprawnym wykonaniu modyfikatora, do zmiennej stanu zostaje dodany na początku znak „\_”, który oznacza, że nie został uruchomiony żaden wyjątek. Listing 3.4 przedstawia przykładowe użycie modyfikatorów w celu sprawdzenia, czy transakcja może zostać wykonana przez wywołującego (msg.sender), poprzez sprawdzenie czy posiada on wystarczające środki. Drugim użyciem tych specjalnych funkcji jest sprawdzenie przed zmianą ceny, czy dany użytkownik jest na pewno właścicielem tego adresu, pod którym znajduje się poprzednia wartość.

```

1. pragma solidity ^0.5.12;
2.
3. contract isOwned{
4.     address payable owner;
5.     constructor() public { owner = msg.sender; }
6.
7.     //modifier declaration
8.     modifier isOwner{
9.         require(
10.             msg.sender == owner,
11.             "By wywołać tę funkcję, trzeba być jej właścicielem"
12.         );
13.         _;
14.     }
15. }
16.
17. contract isPriced{
18.     modifier cost(uint price){
19.         if(msg.value >= price){
20.             _;
21.         }
22.     }
23. }
24.
25. contract Transaction is isPriced,isOwned{
26.     mapping(address => bool) regAddress;
27.     uint price;
28.
29.     constructor(uint initPrice) public { price = initPrice;}
30.
31.     function reg() public payable cost(price){
32.         regAddress[msg.sender] = true;
33.     }
34.     function changePrice(uint _price) public isOwner{
35.         price = _price;
36.     }
37. }

```

**Listing 3. 4 Przykład używania modyfikatorów**

Źródło opracowania: własne na podstawie dokumentacji Solidity 0.5.12



### 3.5 Eventy

Eventy to swego rodzaju interfejsy (rozdział 5.2) i również mogą zostać odziedziczone przez kontrakt. Zdarzenia mogą zostać wyemitowane przez każdą funkcję przy użyciu ich nazwy wraz z niezbędnymi parametrami. Po tym etapie, argumenty zostają zapisane do logów transakcji (parametr `LogsBloom`). Te wartości są przechowywane w łańcuchu bloków do momentu, gdy kontrakt nie zostanie nadpisany lub zaktualizowany. Dostęp do logów, w których zapisane są wyemitowane eventy, jest niemożliwy z poziomu kontraktów (nawet z tych umów, które wyemitowały daną właściwość), a jedynym sposobem odczytu tych wartości jest odwołanie się poprzez adres wdrożonego kontraktu w łańcuchu bloku.

Przykładowy kod zdarzenia znajduje się na listingu 3.5, gdzie przy użyciu eventu, dopisujemy zadeklarowaną wartość przez użytkownika do funkcji `doSomething()`, w której możemy odwołać się do poszczególnych parametrów.

```
1. pragma solidity ^0.5.12;
2.
3. contract eventExample{
4.     event someEvent(address owner, uint amount);
5.
6.     function doSomething() public payable{
7.         emit someEvent(msg.sender,msg.value);
8.     }
9. }
```

**Listing 3. 5 Przykład eventu**

Źródło opracowania: własne

## 4. Ataki na platformę Ethereum

Założenie, że na świecie nie występują próby oszukiwania, kradzieży i innych przestępstw, jest naiwne, gdyż nigdy nie mamy całkowitej pewności, czy dany system jest bezpieczny. Platforma Ethereum, choć zabezpieczona zaawansowaną kryptografią i algorytmiką, również podatna jest na ataki. W tym rozdziale zostaną przedstawione i przeanalizowane znane podatności występujące w Inteligentnych Kontraktach.

Ważnym krokiem w poszukiwaniu podatności i błędów jest sprawdzenie kodu źródłowego danego kontraktu. Używając konsoli zawartej w pakiecie truffle, można uzyskać szczegółowe dane odnośnie danego kontraktu, wpisując nazwę wdrożonego kontraktu. Fragment danych powiązanych z wdrożoną umową, został przedstawiony na rysunku 4.1, a całość została dołączona do pracy i oznaczona jako Załącznik 2.

```
_property_values: {},
_json:
  { contractName: 'Bob',
    abi: [ [Object] ],
    metadata:
      { "compiler": {"version": "0.5.12+commit.7709ece9"}, "language": "Solidity", "output": { "abi": [ { "constant": false, "inputs":
        [ { "internalType": "address", "name": "c", "type": "address" } ], "name": "ping", "outputs": [ ], "payable": false, "stateMutability":
        "nonpayable", "type": "function" } ], "devdoc": { "methods": { }, "userdoc": { "methods": { }, "settings": { "compilationTarget": { "/C:/
        Users/Robert/inz/contracts/Bob.sol": "Bob", "evmVersion": "petersburg", "libraries": { }, "optimizer": { "enabled": false, "runs":
        200 }, "remappings": [ ] }, "sources": { "/C:/Users/Robert/inz/contracts/Bob.sol": { "keccak256": "0x3f4fa0c478ac6ff0e2f6186fa71d716
        ef5cfb2e0d59b8f45a1561c56f9f1e102", "urls": [ "bzz-raw://2884364329ea84399d13735cf167203fca5c375670064c5da6b99e55e163a56c",
        "dweb:/ipfs/Qmetau2CBWpxcetrAJdTUWYf8HBPZnHcJd7oE7MFSzYYTV" ] }, "version": "1" },
        bytecode:
          '0x608060405260008060006101000a81548160ff02191690831515021790555034801561002a57600080fd5b506101418061003a600039600
          0f3fe608060405234801561001057600080fd5b5060004361061002b5760003560e01c806314d9e4314610030575b600080fd5b61007260048036036
          02081101561004657600080fd5b81019080803573fffffffffffffffffffffffffffffffffffffffff1690602001909291905050610074565b6005b6
          00809054906101000a900460ff16610109578073fffffffffffffffffffffffffffffffffffffffff166014604051806000019050600060405180830
          b8185875af1925050503d80600081146100e6576040519150601f19603f3d011682016040523d82523d6000602084013e6100eb565b606091505b505
          05060016000806101000a81548160ff0219169083151502179055505b5056fea265627a7a723158203ae54cc535ac19a942c9433c9e9f8d00637eb97
          4cdb762ed27c74c35acb51b5664736f6c634300050c0032',
          deployedBytecode:
            '0x608060405234801561001057600080fd5b5060004361061002b5760003560e01c806314d9e4314610030575b600080fd5b6100726004803
            603602081101561004657600080fd5b81019080803573fffffffffffffffffffffffffffffffffffffffff1690602001909291905050610074565b0
            05b6000809054906101000a900460ff16610109578073fffffffffffffffffffffffffffffffffffffffff16601460405180600001905060006040518
            08303b8185875af1925050503d80600081146100e6576040519150601f19603f3d011682016040523d82523d6000602084013e6100eb565b606091505
            b505060016000806101000a81548160ff0219169083151502179055505b5056fea265627a7a723158203ae54cc535ac19a942c9433c9e9f8d00637
            eb974cdb762ed27c74c35acb51b5664736f6c634300050c0032',
            sourceMap:
              '28:148:0:-;55:5;45:15;,,,,,,,,,,,,,,,,,,,,,,,,28:148;8:9:-1;5:2;30:1;27;20:12;5:2;28:148:0;,,,,,,',
            deployedSourceMap:
              '28:148:0:-;8:9:-1;5:2;30:1;27;20:12;5:2;28:148:0;,,,,,,,,,,,,,,,,65:108;13:2:-1;8:3;5:11;2:2;29:1;2
              6;19:12;2;2;65:108:0;,,,,,,,,,,,,,,,,1:1:-;108:4;,,,,,,,,104:64;122:1;6;135:2;122:20;,,,,,,,,,,,,,,,,
              14:1:-1;21;16;31;75:4;69:11;64:16;144:4;140:9;133:4;115:16;111:27;107:43;104:1;100:51;94:4;87:65;169:16;166:1;159:27
              ;225:16;222:1;215:4;212:1;208:12;193:49;7:242;16:31;36:4;31:9;7:242;122:20:0;156:4;151;9;,,,,,,,,104:64;6
              5:108;:::o',
            source:
              'pragma solidity ^0.5.12;\n\ncontract Bob{\n\n  bool sent=false;\n\n  function ping(address c) public {\n\n    if(!sent){\n\n      c.call.value(20)("");\n\n      sent=true;\n\n    }\n\n  }\n\n}\n\n',
            sourcePath: 'C:/Users/Robert/inz/contracts/Bob.sol',
            ast:
              { absolutePath: '/C:/Users/Robert/inz/contracts/Bob.sol',
                exportedSymbols: [Object],
                id: 30,
                nodeType: 'SourceUnit',
                nodes: [Array],
                src: '0:178:0' },
            LegacyAST:
              { absolutePath: '/C:/Users/Robert/inz/contracts/Bob.sol',
                exportedSymbols: [Object],
                id: 30,
                nodeType: 'SourceUnit',
                nodes: [Array],
                src: '0:178:0' },
            compiler:
```

Rysunek 4. 1 Fragment wdrożonego kontraktu przedstawiający kod źródłowy

Źródło opracowania: własne

## 4.1 Atak typu: Reentrancy

Jednym z głównych zagrożeń w odwoływaniu się do zewnętrznych kontraktów jest przejęcie kontroli nad przebiegiem działania funkcji i zmuszenie jej do działań niezamierzonych. Pierwszym wykorzystaniem takiej podatności jest atak z 2016 roku, nazwany „The DAO” [23], a klasa tego typu ataków nosi nazwę „reentrancy”.

Pierwsza wersja tej podatności została zauważona, gdy funkcja ciągle się wykonywała przez złe użycie zewnętrznej metody „call”. Niskopoziomowa funkcja „call” jest interfejsem przekazującym wiadomości do kontraktu, które mogą wywoływać funkcje o podanej nazwie. Struktura tej metody pozwala również na transfer środków w formie Etheru poprzez odwołanie się do konkretnego kontraktu. Ta ostatnia właściwość funkcji „call”, pozwala na odwołanie rekurencyjne, przez co w miejscu, gdzie nie powinna znajdować się pętla, tworzy się luka bezpieczeństwa. Ustalona wartość Etheru zostaje przelewana do momentu przesłania wszystkich możliwych środków z konta ofiary (lub ofiar) na konto atakującego lub do skończenia się gazu w bloku. Przykładowy kod podatności został przedstawiony na listingu 4.1, na którym głównym błędem staje się niewyzerowanie salda konta użytkownika, przed wykonaniem funkcji z właściwością „call”. Poprawiona wersja tego przykładu znajduje się na listingu 4.2, gdzie tablica *userBalances*, przechowująca bilans kont użytkowników, zostaje wyzerowana przed zainicjowaniem metody, przez co podatność zostanie ograniczona do minimum.

```
1. pragma solidity ^0.5.12;
2.
3. contract Reentrancy_attack_insecure {
4.     mapping (address => uint) private userBalances;
5.
6.     function withdrawBalance() public {
7.         uint amountToWithdraw = userBalances[msg.sender];
8.         (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
9.         //W tym punkcie, kod zostaje wykonany, ale funkcję
10.        withdrawBalance można wywołać ponownie
11.        require(success);
12.        userBalances[msg.sender] = 0;
13.    }
14. }
```

Listing 4. 1 Reentrancy attack – przykładowa implementacja

Źródło opracowania: własne na podstawie rejestru podatności [17]

```

1. pragma solidity ^0.5.12;
2.
3. contract Reentrancy_attack_solve {
4.     mapping (address => uint) private userBalances;
5.
6.     function withdrawBalance() public {
7.         uint amountToWithdraw = userBalances[msg.sender];
8.         userBalances[msg.sender] = 0;
9.         (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
           // Saldo użytkownika jest 0, więc przyszłe odwołania nie wypłacą
           żadnej kwoty
10.        require(success);
11.    }
12. }

```

**Listing 4. 2 Reentrancy attack - rozwiązanie problemu funkcji "call"**

Źródło opracowania: własne na podstawie rejestru podatności [17]

W praktyce przeprowadzenie ataku jest bardziej skomplikowane niż przedstawiona teoria. Atakujący w pierwszej kolejności musi znaleźć adres transakcji w bloku (zwykle używając polecenia `web3.eth.getTransactionFromBlock` w konsoli truffle lub Geth JS Console), lub nazwę kontraktu. Oba te kroki prowadzą do możliwości odwołania się do wdrożonej w łańcuch bloków umowy, poprzez jej nazwę (Rys. 4.1). W naszym przypadku nazwa kontraktu jest znana (Bob.sol), dlatego możemy pominąć pierwszy krok znajdowania podatności. Następnym krokiem jest utworzenie spreparowanego kontraktu (Listing 4.4), odwołującego się do umowy z luką bezpieczeństwa, po czym dodanie go do puli transakcji. W tym celu w konsoli truffle wykonujemy polecenie **migrate**, które dodatkowo wdraża kontrakt znajdujący się w katalogu kontraktów (contracts) w lokalizacji truffle, używając skryptu wdrażającego (ang. Deployer) o nazwie `5_initial_reenter_mallory.js` (Listing 4.3) znajdującego się w folderze **migrations**. Po weryfikacji (w środowisku testowym ten proces jest pomijany) kontrakt trafia do łańcucha (Rys. 4.3), gdzie tworzona jest transakcja przelewu Etheru na konto atakującego poprzez zapętlenie funkcji `call`. Składnia skryptu wdrażającego przedstawia się w następujący sposób:

```

1. const Nazwa_zmiennej = artifacts.require("Nazwa_kontraktu");
2. module.exports = function(deployer) {
3.     deployer.deploy(Nazwa_zmiennej);
4. };

```

**Listing 4. 3 Skrypt wdrażający 5\_initial\_reenter\_mallory.js**

Źródło opracowania: własne

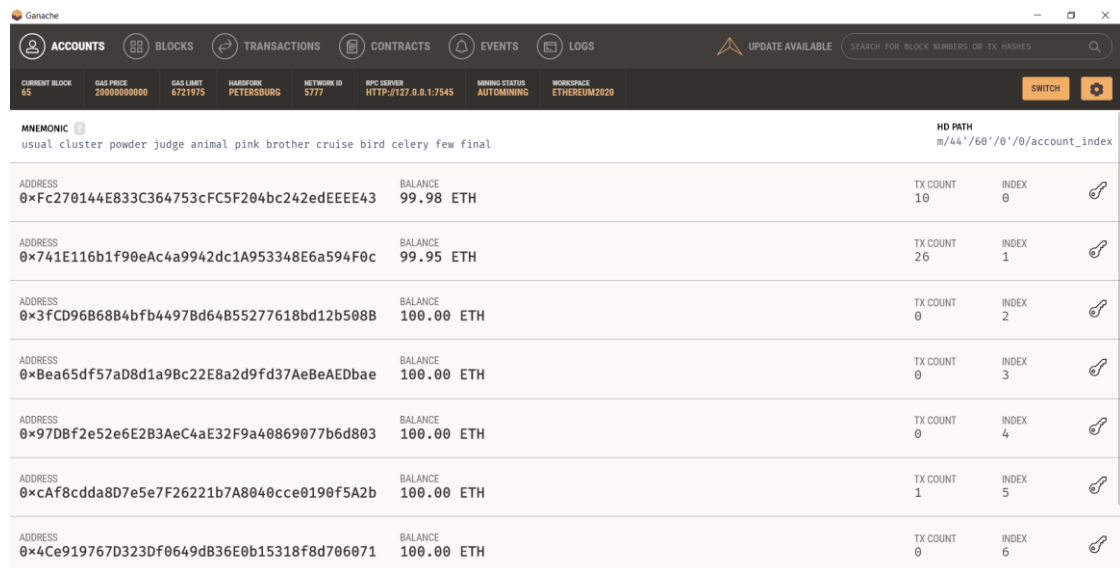
```

1. pragma solidity ^0.5.12;
2.
3. contract Bob{
4.
5.     function ping(address a) public{
6.     }
7.
8. contract Reenter_mallory{
9.     function() external payable{
10.         Bob(msg.sender).ping(address(this));
11.     }
12. }

```

Listing 4. 4 Kod spreparowanego kontraktu, odwołującego się do Bob.sol

Źródło opracowania: własne



ADDRESS	BALANCE	TX COUNT	INDEX
0xFc270144E833C364753cFC5F204bc242edEEEE43	99.98 ETH	10	0
0x741E116b1f90eAc4a9942dc1A953348E6a594F0c	99.95 ETH	26	1
0x3fCD96B68B4bf4497Bd64B5277618bd12b508B	100.00 ETH	0	2
0xBea65df57aD8d1a9Bc22E8a2d9fd37AeBeAEDbae	100.00 ETH	0	3
0x97DBf2e52e6E2B3AeC4aE32F9a40869077b6d803	100.00 ETH	0	4
0xcAf8cdda8D7e5e7F26221b7A8040cce0190f5A2b	100.00 ETH	1	5
0x4Ce919767D323Df0649d8B36E0b15318f8d706071	100.00 ETH	0	6

Rysunek 4. 2 Stan kont w sieci przed przeprowadzeniem ataku reentrancy

Źródło opracowania: własne

```

truffle(development)> migrate

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Starting migrations...
=====
> Network name:    'development'
> Network id:      5777
> Block gas limit: 0x6691b7

5_initial_reenter_mallory.js
=====

Replacing 'Reenter_mallory'
-----
> transaction hash: 0x9cf9641c77631d31b1381f630e30f3e3885b1a6347727defdaa600b8321959b71
> Blocks: 0
> contract address: 0x2E46193763f22edC6b125ee047244Bc209F1F3A8
> block number: 66
> block timestamp: 1581061650
> account: 0x741E116b1f90eAc4a9942dc1A953348E6a594F0c
> balance: 99.9506802
> gas used: 110113
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00220226 ETH

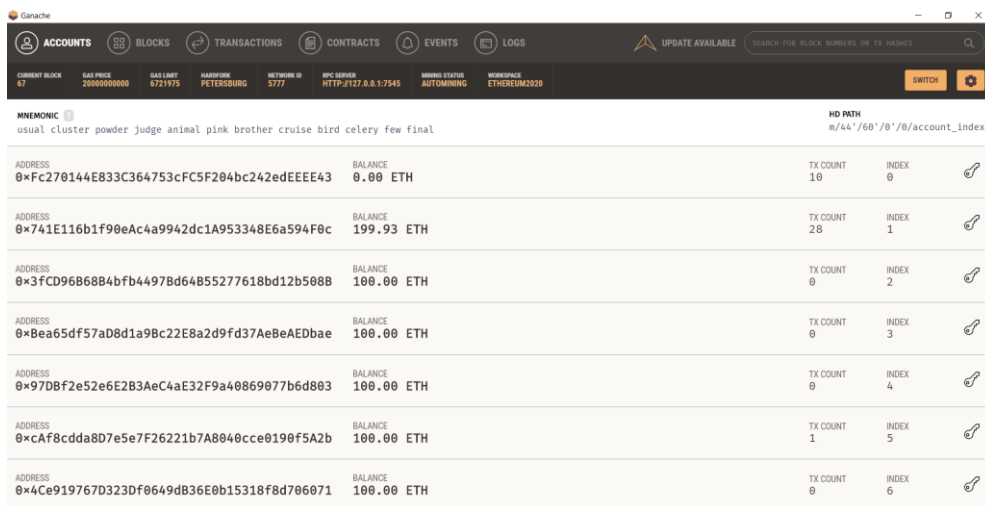
> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00220226 ETH

Summary
=====
> Total deployments: 1
> Final cost: 0.00220226 ETH

```

Rysunek 4. 3 Wdrożenie kontraktu reenter\_mallory.sol

Źródło opracowania: własne



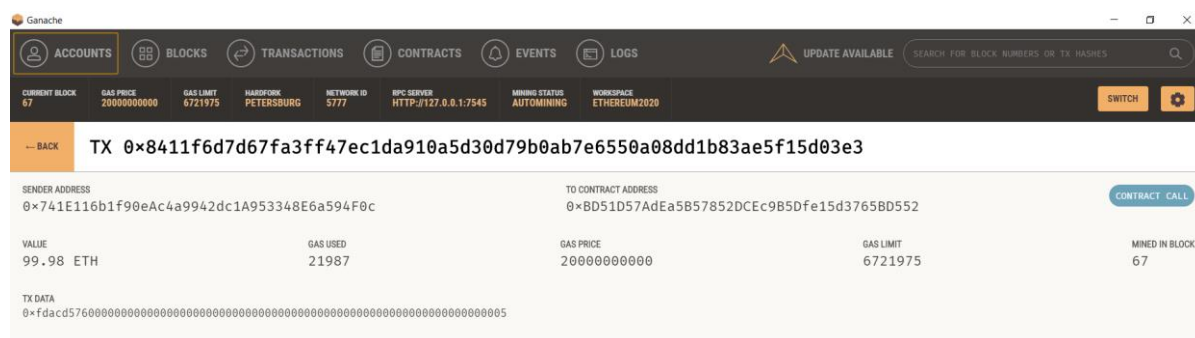
ADDRESS	BALANCE	TX COUNT	INDEX
0xFc270144E833C364753cFC5F204bc242edEEEE43	0.00 ETH	10	0
0x741E116b1f90eAc4a9942dc1A953348E6a594F0c	199.93 ETH	28	1
0x3fCD96B68B4bf4497Bd64855277618bd12b508B	100.00 ETH	0	2
0xBea65df57aD8d1a9Bc22E8a2d9fd37AeBeAEDbae	100.00 ETH	0	3
0x97DBf2e52e6E2B3AeC4aE32F9a40869077b6d803	100.00 ETH	0	4
0xcAf8cdda8D7e5e7F26221b7A8040cce0190f5A2b	100.00 ETH	1	5
0x4Ce919767D323Df0649dB36E0b15318f8d706071	100.00 ETH	0	6

Rysunek 4. 4 Stan kont w sieci po przeprowadzeniu ataku reentrancy

Źródło opracowania: własne

Rysunek 4.2 przedstawia adresy kont oraz ich salda przed wdrożeniem kontraktu Reenter\_mallory.sol, natomiast rysunek 4.4 prezentuje wyżej wymienione wartości, po przeprowadzeniu ataku Reentrancy. Porównując oba rysunki, możemy zauważyć wyzerowane saldo konta [0], a także zwiększoną wartość konta [1] o równowartość dawnego salda konta [0]. Ten efekt widoczny jest w zakładce Transactions (lub poprzez odwołanie się do hashu transakcji w konsoli

truffle) i został przedstawiony na rysunku 4.5. Cały proces został przedstawiony na filmie dołączonym do pracy i opisanym jako: „Załącznik 3 – atak typu Reentrancy.mp4”.



Rysunek 4. 5 Szczegóły transakcji wdrożenia kontraktu reenter\_mallory.sol wraz z przelewem kwoty etheru z konta [0]

Źródło opracowania: własne

Podatność w niskopoziomowej funkcji „call”, znana jest już kilka lat. Najlepszym działaniem prewencyjnym, według jednego z twórców Solidity oraz Truffle – **chriseth**, byłoby nieużywanie tej przestarzałej metody [24]. Podobnie nie powinno się używać metod „transfer” oraz „send”, jednakże jeśli już musimy przelewać Ether, najbardziej optymalną metodą jest właśnie funkcja „call” z wzorcami sprawdzania-efektów interakcji (ang. Checks-effects-interactions pattern). Proces używania tych szablonów przedstawia się w następujący sposób:

1. Sprawdzenie podstawowych parametrów funkcji (osoby wywołującej funkcję, zakres argumentów, saldo kont itp.) i zwrócenie stanu w formie zmiennej boolowskiej
2. Po otrzymaniu wartości prawdziwej z poprzedniego kroku, zostają tworzone efekty poprzedniego kroku, a następnie następuje dopisanie ich do lokalnych zmiennych stanu w podanym kontrakcie
3. Kolejnym krokiem jest interakcja funkcji niebezpiecznych z zewnętrznymi kontraktami
4. Ostatnim krokiem jest sprawdzenie wyniku interakcji, a także błędów i ostrzeżeń kompilatora.

## 4.2 Teoretyczny atak 51%

Teoretyczna podatność występująca w implementacjach technologii blockchain opartych o protokół konsensusu Proof-of-Work [21] Atak polega na kontrolowaniu 51% mocy obliczeniowej całej platformy, przez osobę (lub grupę osób) zwaną jako atakujący. Posiadając większość mocy obliczeniowej, atakujący przejmuje kontrolę nad całą platformą, przez co może wprowadzać fałszywe transakcje (np. przelew ETH z jednego konta na konto atakującego, bez zgody pierwszej strony), odrzucać poprawne, czy nawet ingerować w strukturę platformy, a to może przedłożyć się na wycofywanie, co teoretycznie jest niemożliwe (ingerencja programistów platformy podczas ataku The DAO i rozłam Ethereum na Ethereum oraz Ethereum Classic), już wdrożonych transakcji.

### 4.3 Atak typu: Front-Running

Podatności typu Front-running są ciągiem zdarzeń, gdzie ktoś zyskuje na wcześniejszym dostępie do informacji na temat niezatwierdzonych jeszcze transakcji. Zwykle luka bezpieczeństwa tworzona jest przez osobę uprzywilejowaną, która pracuje przy zatwierdzaniu transakcji do bloku. Atak polegający na obserwowaniu puli transakcji przez atakującego przed ich zatwierdzeniem, w celu wdrożenia do łańcucha bloków swojego zmodyfikowanego polecenia, zanim poprawna transakcja zostanie dopisana. Tego typu podatności są nielegalne za względu na łamanie regulacji bezpieczeństwa klienta (osoby, która zainicjowała transakcję). Atak może zostać przeprowadzony, gdyż każda transakcja próbująca być dopisana do bloku, trafia najpierw do puli pamięci (ang. Mempool), gdzie górnik ma możliwość podejrzenia przyszłych transakcji, a następnie wykorzystanie ich do własnych celów.

Potencjalnym rozwiązaniem tej podatności jest wywoływanie zatwierdzenia transakcji z właściwością odślonienia hasha. Metoda polega na żądaniu od górnika zatwierdzającego transakcję, dodanie do hasha wartości liczbowej wygenerowanej (tzw. Salt) przez inicjującego. Po dodaniu transakcji do bloku, górnik wysyła kontrakt do inicjującego, który zawiera zapisany w zmiennej hash, a w parametrach hashowania, dodaje salt, adres konta górnika oraz dane, które powinny zostać zapisane w bloku.

### 4.4 Atak typu: Timestamp Dependence

Decentralizacja platformy Ethereum pozwala na pobieranie czasu tylko w pewnym ograniczonym zakresie. Kontrakty natomiast potrzebują dokładnego momentu inicjacji czy wdrożenia do wywoływania eventów zależnych od czasu (ang. Time-dependent events) (Listing 4.5). Górnicy chcąc oszukać ten mechanizm, a przez to zyskać, zatwierdzając transakcję lub kontrakt, łączą timestamp bloku podatnego z transakcją (czy kontraktem), pamiętając o tym, że ten parametr nie może być niższy niż wartość ostatniego bloku. W przeciwnym wypadku transakcja zostanie automatycznie odrzucona przez większość kont kopiących. Zbyt niska wartość timestamp nie jest jedyną opcją niezaakceptowania bloku. Kolejnym przykładem może stać się zbyt odległa wartość tego parametru.

Podczas próby zasymulowania ataku na prywatnej sieci Ethereum, używając Ganache'a oraz Truffle'a, odkryto iż łańcuch bloków oraz czas pracy na nim, jest za krótki przez co nie było znaczących różnic w znacznikach definiowanych w kontraktach, a występujących rzeczywiście na maszynie.



```

1. pragma solidity ^0.5.12;
2.
3. contract TimestampDependenceSolve{
4.     event Finished();
5.     event notFinished();
6.
7.     function isOperationFinished() private returns
      (bool){
8.         return block.timestamp >=1580389200;
9.         /* Operacja powinna zakończyć się
10.            po godzinie 13:00 w dniu 30.01.2020
11.        */
12.     }
13.
14.     function run() public{
15.         if(isOperationFinished()){
16.             emit Finished();
17.         }
18.         else
19.         {
20.             emit notFinished();
21.         }
22.     }
23. }

```

**Listing 4. 5 Przykładowy kontrakt ze sprawdzaniem czasu ukończenia operacji**

Źródło opracowania: własne na podstawie rejestru podatności [17]

Ważnym krokiem w przeciwdziałaniu tej podatności jest pisanie umów, w których znajduje się wartość parametru timestamp dla ostatnio zaakceptowanego bloku, a także obecny czas inicjacji. Dzięki tym dwóm wartościom dopisanym do kontraktu, można ustawić czas, który przez prosty warunek będzie sprawdzał czy nie był modyfikowany znacznik czasu. Zwykle wartość różnicy porównuje się z czasem pół minuty.

## 4.5 Atak typu: Integer Overflow and Underflow

Podatności przepełniania i niedomiaru polegają na przekroczeniu wartości granicznej dla danego typu zmiennej. Przykładem może być zmienna typu uint. Przepełnienie występuje, gdy wartość tego typu zmiennej przekracza  $2^{256}$  (typ uint domyślnie jest ustawiony na uint256, a przy niższych wartościach, górna granica zmienia się analogicznie do wielkości typu np. przy uint8 wartość przepełnienia nastąpi przy  $2^8 + 1$ ). Następnie zmienna zostaje wyzerowana, po czym następuje dopisanie do niej pozostałej wartości. Na podobnej zasadzie działa niedomiar, który występuje po tym, jak wartość zmiennej uint, jest niższa niż wartość graniczna – 0. Proces jest odwrotnością przepełnienia, przez co, gdy wydarzy się ten przypadek, wartość zmiennej będzie wynosiła  $2^{256}$  minus pozostała wartość.

```

1. pragma solidity ^0.5.12;
2.
3. contract IntegerUnderflowOverflow{
4.     mapping(address => uint256) public balance;
5.
6.     function transfer(address _to, uint8 value) public payable{
7.         /* Sprawdzenie wartości salda */
8.         require(balance[msg.sender] >= value);
9.
10.        balance[msg.sender] -= value;
11.        balance[_to] += value;
12.    }
13.
14. }

```

**Listing 4. 6 Przykład kontraktu podatnego na atak Integer Overflow and Underflow**

Źródło opracowania: własne na podstawie rejestru podatności [17]

```

1. pragma solidity ^0.5.12;
2.
3. contract IntegerUnderflowOverflow{
4.     mapping(address => uint256) public balance;
5.
6.     function transfer(address _to, uint8 _value)
7.     public payable{
8.         /* Sprawdzenie przepiętnień i niedomiarów
9.         */
10.        require(balance[msg.sender] >= _value
11.        && balanceOf[_to] + _value >=
12.        balanceOf[_to]);
13.
14.        balance[msg.sender] -= value;
15.        balance[_to] += value;
16.    }
17.
18. }

```

**Listing 4. 7 Przykład kontraktu niwelującego atak Integer Overflow and Underflow**

Źródło opracowania: własne na podstawie rejestru podatności [17]

Analizując kontrakty przedstawione na listingach 4.6 oraz 4.7, jesteśmy w stanie zauważyć różnicę w składni. Na listingu 4.6 sprawdzamy tylko, czy użytkownik inicjujący funkcję transfer, posiada wystarczający bilans konta. Następnie następują operacje na saldach kont, przez co może dojść do wspomnianych już przepiętnień. Sytuacja zmienia się w sposób znaczący, poprzez dodanie do warunku sprawdzenia czy odbiorca (\_to) posiada wystarczającą wartość do przyjęcia konkretnego transferu. (Listing 4.7)

```

1. pragma solidity ^0.5.12;
2.
3. contract TokenSaleChallenge {
4.     mapping(address => uint8) public balanceOf;
5.     uint256 constant PRICE_PER_TOKEN = 1 ether;
6.
7.     constructor() public payable {
8.         require(msg.sender.balance >= 1 ether, "Użytkownik ma za mało etheru");
9.     }
10.
11.     function isComplete() public view returns (bool) {
12.         return msg.sender.balance < 1 ether;
13.     }
14.
15.     function buy(uint8 numTokens) public payable {
16.         require(msg.sender.balance >= numTokens * PRICE_PER_TOKEN);
17.
18.         balanceOf[msg.sender] += numTokens;
19.     }
20.
21.     function sell(uint8 numTokens) public payable {
22.         require(balanceOf[msg.sender] >= numTokens);
23.
24.         balanceOf[msg.sender] -= numTokens;
25.     }
26. }

```

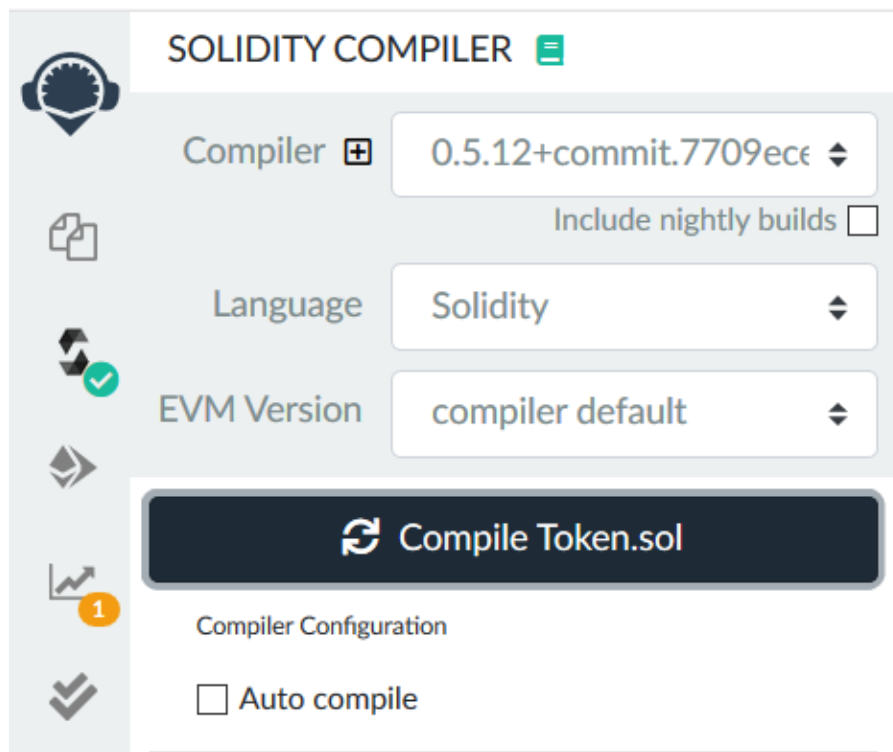
**Listing 4. 8 Przykład nr. 2 kontraktu podatnego na atak Integer Overflow**

Źródło opracowania: własne na podstawie rejestru podatności [17]

Podczas mapowania adresu i deklaracji zmiennych, użyto typu uint8 (Listing 4.8), którego wartość maksymalna 2<sup>8</sup>, przez co kontrakt jest bardzo podatny na przepełnienie. Używając internetowego narzędzia Remix (wspierane przez twórców, łatwe w obsłudze, ale zaawansowane – szczególnie przy obsłudze wyjątków - narzędzie do testowania umów na różnych konfiguracjach sieci) [26], kontrakt został skompilowany, a następnie wdrożony do sieci JavaScript Virtual Machine. Rysunek 4.6 przedstawia zakładkę kompilatora, w której możemy wybrać język używany (dostępne są dwie opcje: Solidity oraz Yul), wersję programu translacyjnego oraz wersję maszyny wirtualnej (HardFork). Po lewej stronie znajduje się pasek z funkcjonalnościami, które przedstawiają się w podany sposób (od góry):

1. Strona główna
2. Eksplorator plików
3. Kompilator języka
4. Sieć wdrożeniową – miejsce, w którym znajdują się wszystkie skompilowane kontrakty oraz funkcje pozwalające je wdrożyć i wykonać


5. Analitika powiązana z kompilacją ostatniego kontraktu
6. Wbudowany panel testów







Rysunek 4. 6 Konfiguracja kompilatora w środowisku Remix dla wdrożenia kontraktu Token.sol



Źródło opracowania: własne


Po skompilowaniu umowy, przejściu do zakładki wdrożenia i wybraniu podstawowych parametrów (środowiska, konta właściciela kontraktu, limitu gazu, wartości dołączonej kwoty do bloku, a także nazwy kontraktu) kontrakt wdrożono (Rys. 4.7), a przejrzysty interfejs pozwala na modyfikowanie wartości w funkcjach buy oraz sell (Rys 4.8). Tablica balanceOf[adres], przedstawia aktualny stan ilości żetonów. Na rysunku 4.15 wartość tablicy balanceOf[adres] wynosi 250, co oznacza, że wartość jest bliska limitu 8 bitów. Po wywołaniu funkcji buy z parametrem 7, w funkcji następuje przepełnienie pokazane na rysunku 4.9, a cała operacja, w formie filmu, została dołączona do pracy i oznaczona jako „Załącznik 4 – atak Integer Overflow.mp4”



DEPLOY & RUN TRANSACTIONS 

Environment JavaScript VM  

Account  0xCA3...a733c (99.9999%) 

Gas limit 3000000  


Value 0 wei 


TokenSaleChallenge - browser/Tok  




Deploy


or


At Address Load contract from Address


Transactions recorded: 1 

Deployed Contracts 


 TokenSaleChallenge at 0x692...77b3A  

buy uint8 numTokens 

sell uint8 numTokens 

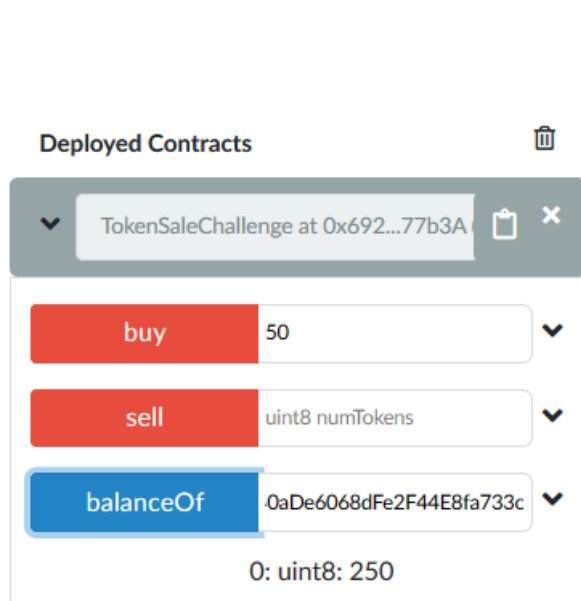
balanceOf address 

isComplete

Low level interactions 

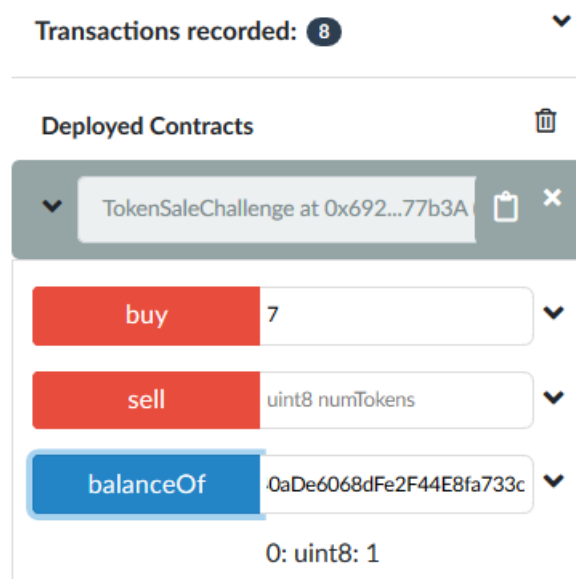
Rysunek 4. 7 Wdrożenie kontraktu TokenSaleChallenge do sieci JavaScript VM (REMIX)

Źródło opracowania: własne



Rysunek 4. 8 "Kupienie" 250 żetonów kontraktu  
TokenSaleChallenge

Źródło opracowania: własne



Rysunek 4. 9 "Kupienie" 7 żetonów, które powodują  
przepełnienie uint8

Źródło opracowania: własne

Dobrym sposobem zabezpieczania się przed tym atakiem jest pamiętanie, by użytkownik nie mógł wywoływać funkcji, które znacząco mogą modyfikować parametr wartości zmiennej. W tym celu warto dodać warunek sprawdzający przepełnienie, przez co funkcja zwróci wyjątek. Przeprowadzenie przykładowego ataku przepełnienia i niedomiaru zostało dołączone do pracy i oznaczone jako: „Załącznik 4 – atak Integer Overflow”.

## 4.6 Atak typu: Denial-of-Service with revert

W klasycznych sieciach komputerowych występuje atak DoS (ang. Denial-of-service attack). Polega on na działaniach zamierzonych w celu odłączenia maszyny lub całej sieci od reszty usług. Zwykle kojarzony jest z zbyt dużym przesyłaniem danych na urządzenie, przez co ono zaczyta nie odpowiadać (ang. Flood). W Ethereum działa to inaczej. Dobrym przykładem pokazującym przebieg tego ataku jest aukcja bazująca na inteligentnych kontraktach (Listing 4.9). Uczestnik, którego zamiary wygrania, przewyższają możliwości, może posunąć się do przeprowadzenia ataku polegającego na odwołaniu się do funkcji fallback, która usunęłaby wszystkie płatności, oprócz kwoty atakującego, przez co staje się on liderem aukcji. Ten proces będzie powtarzany do momentu skończenia aukcji.

```

1. pragma solidity ^0.5.12;
2.
3. contract Auction {
4.     address currentLeader;
5.     uint highestBid;
6.
7.     function bid() payable {
8.         require(msg.value > highestBid);
9.
10.        require(currentLeader.send(highestBid));
11.
12.        currentLeader = msg.sender;
13.        highestBid = msg.value;
14.    }
15. }

```

**Listing 4. 9 Przykład działania DoS with revert**

Źródło opracowania: rejestr podatności [17]

W celu uniknięcia ataku Denial-of-Service with revert, warto używać płatności opartych na przysługach pull/push (ang. Favor pull over push payments). Przykładowy kod takiego kontraktu przedstawiono na listingu 4.10. Analizując ten kontrakt, możemy zauważyć dodatkową zmienną (refunds), która odpowiada za zwrot wartości przetransferowanych do kontraktu. Po wykonaniu warunku, który sprawdza adres osoby z adresem poprzedniego lidera. Jeśli lider się zmienił, funkcja zarządza zwrotami.

```

1. pragma solidity ^0.5.12;
2.
3. contract Auction {
4.     address highestBidder;
5.     uint highestBid;
6.     mapping(address => uint) refunds;
7.
8.     function bid() payable external {
9.         require(msg.value >= highestBid);
10.
11.         if (highestBidder != address(0)) {
12.             //Zapisywanie zwrotu dla użytkownika
13.             //by potem mógł odzyskać tę wartość
14.             refunds[highestBidder] += highestBid;
15.         }
16.
17.         highestBidder = msg.sender;
18.         highestBid = msg.value;
19.     }
20.
21.     function withdrawRefund() external {
22.         uint refund = refunds[msg.sender];
23.         refunds[msg.sender] = 0;
24.         (bool success, ) = msg.sender.call.value(refund)("");
25.         require(success);
26.     }
27. }

```

**Listing 4. 10** Przykładowy kontrakt rozwiązujący podatność DoS with revert (metoda Favor pull over push)

Źródło opracowania: własne na podstawie rejestru podatności [17]



## 5. Poradnik jak pisać Inteligentne Kontrakty i unikać błędów

Istnieje internetowy rejestr [17], w którym podane są wszystkie znane słabości w Inteligentnych Kontraktach wraz z ich kodami oraz funkcjami testowymi. Takiego typu strony są dobrym źródłem nauki pisania kontraktów, gdyż jak to mówi przysłowie by nie popełniać błędów, trzeba się na nich uczyć i wyciągać wnioski. W tym rozdziale przedstawiono podstawowe techniki, które uczynią nasze kontrakty bardziej użyteczne i mniej podatne na błędy.

Zacznijmy od rozbudowania składni języka Solidity. W rozdziale 3 przedstawiono podstawową semantykę tego języka. Dobrymi praktykami w programowaniu inteligentnych umów jest używanie:

- Dziedziczenia;
- Interfejsów;
- Kontraktów abstrakcyjnych;
- Wyjątków;

Ponadto warto również używać:

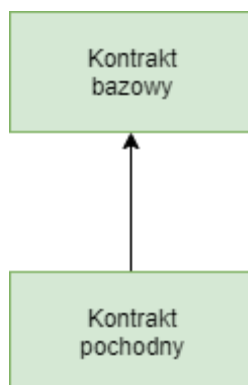
- Jednej konwencji nazywania;
- Odpowiednich i przemyślanych argumentów widoczności dla funkcji i zmiennych;
- Uproszczenia i modularności kontraktów;
- Frameworków do testowania zdecentralizowanych aplikacji;
- Kontraktu rejestrującego wersje lub *delegatecall* do przekazania danych i odwołań;
- „Przycisków bezpieczeństwa” lub „Przerywaczy”;
- Opóźnień w wykonywaniu akcji kontraktu;
- Ograniczeń transferu Etheru między kontami;

### 5.1 Dziedziczenie

Jednym z filarów programowania obiektowego jest dziedziczenie. Podobnie jak w znanych językach programowania, Solidity również je implementuje pomiędzy swoistymi klasami – kontraktami. Dziedziczenie polega na zdeklarowaniu przynajmniej dwóch kontraktów i połączeniu ich w relacji rodzic-dziecko. Umowy, po których dziedziczą inne klasy, nazywane są rodzicem (lub kontraktami bazowymi), natomiast mianem dziecka (kontrakt pochodny) określany jest kontrakt, który dziedziczy po rodzicach. Słowem kluczowym używanym w tym procesie jest *is*. Pomimo podobnej struktury dziedziczenia Solidity i C# czy Javy, nie jest to taka sama metoda. W przypadku tej operacji w technologii Ethereum, bytecode kontraktu bazowego, zostaje skopiowany do bytecode’u kontraktu pochodnego.

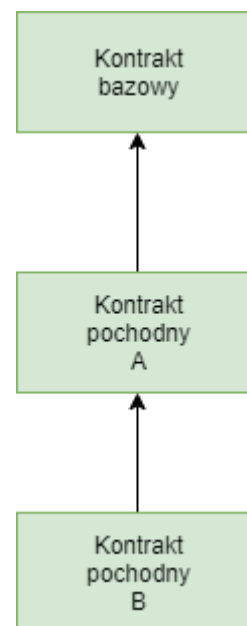
Wymieniamy następujący typy dziedziczenia w łańcuchu bloków:

- Dziedziczenie pojedyncze (ang. Single inheritance) – pozwala na dziedziczenie zmiennych, funkcji, modyfikatorów, eventów z kontraktu bazowego, do pochodnego (Rys.5.1)
- Dziedziczenie wielopoziomowe (ang. Multi-level inheritance) – podobne dziedziczenie do pojedynczego, z tą różnicą, że mamy kilka poziomów dziedziczenia (Rys. 5.2)
- Dziedziczenie hierarchiczne (ang. Hierarchical inheritance) – podobne dziedziczenie do pojedynczego, z tą różnicą, że jeden kontrakt bazowy, może być dziedziczony przez kilka pochodnych (Rys. 5.3)
- Wielodziedziczenie (ang. Multiple inheritance) – pozwala na dziedziczenie, polegające na połączeniu wyżej wymienionych typów. Będąc kontraktem pochodnym, pobiera wartości kontraktu bazowego z innego kontraktu pochodnego (Rys. 5.4)



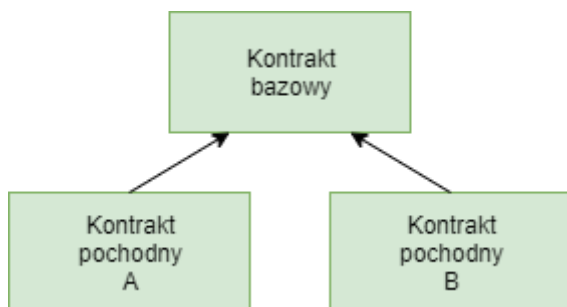
Rysunek 5. 1 Dziedziczenie pojedyncze

Źródło opracowania: własne

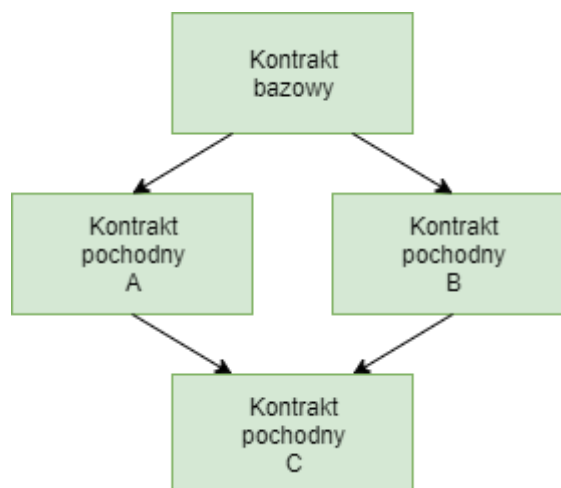


Rysunek 5. 2 Dziedziczenie wielopoziomowe

Źródło opracowania: własne



Rysunek 5. 3 Dziedziczenie hierarchiczne  
Źródło opracowania: własne



Rysunek 5. 4 Dziedziczenie wielopoziomowe  
Źródło opracowania: własne

## 5.2 Interfejsy

Interfejsy są to swego rodzaju kontrakty, które deklarowane są przez słowo kluczowe *interface*. Podstawową różnicą od zwykłej umowy jest deklaracja każdej funkcji bez jej implementacji. Interfejsy nie mogą dziedziczyć po innych typach danych, ale mogą zostać odziedziczone. Dodatkowo każda zadeklarowana funkcja, musi mieć ustawiony typ *external*. Oprócz wyżej wymienionych restrykcji, istnieją również inne. Interfejsy nie mogą definiować [16]:

- Konstruktorów;
- Zmiennych stanu;

## 5.3 Kontrakty abstrakcyjne

Kontrakty abstrakcyjne mają podobną składnię do interfejsów. Główną różnicą jest to, że w ich przypadku, wystarczy przynajmniej jedna funkcja bez implementacji, by nazwać kontrakt abstrakcyjnym.

## 5.4 Wyjątki

Solidity używa wyjątków przywrócenia stanu (ang. State-reverting exceptions), do obsługi błędów. Nie jest jednak możliwe, wychwycenie wyjątku w tym języku. Istnieją cztery podstawowe typy obsługi błędów [16]:

- Assert
- Require
- Revert

Dwa pierwsze typy obsługi błędów, mogą być użyte do sprawdzania warunków, a następnie zwrócić wyjątek, jeśli warunek nie został napotkany. Funkcja `assert` powinna być tylko używana do testowania błędów na prywatnych sieciach lub środowiskach testowych, ponieważ mogą one prowadzić do luk bezpieczeństwa. Należy pamiętać, by przed wdrożeniem kontraktu do publicznej sieci, usunąć warunki `assert` lub zastąpić je (w miejscach, w których się da) funkcją `require`. Funkcja `require` jest używana do zapewnienia poprawnych warunków, które nie mogą zostać wykryte do momentu wdrożenia kontraktu. Istnieje jeszcze jedna wyższość używania funkcji `require` zamiast `assert` i jest to konsumpcja gazu. Używając pierwszej z wymienionych, podczas wychwycenia wyjątku, gaz użyty w transakcji jest zwracany, natomiast w drugim przypadku, ten parametr przepada.

Kolejnym przykładem wychwycenia wyjątków jest funkcja `revert`. Działa podobnie, jak funkcja `require` i również zwraca gaz do wywołującego. Obie te funkcje posiadają parametr optional string `message`, przez który programista, może zwrócić spersonalizowaną wartość błędu.

## 5.5 Dobre nawyki podczas programowania kontraktów

Nie wystarczy używać wyjątków, czy interfejsów by kody naszych kontraktów spełniały oczekiwania. Warto wybrać jedną konwencję nazywania zmiennych, funkcji, interfejsów czy kontraktów. Lepsza czytelność kodu sprawia, że nie musimy się zastanawiać po jakimś okresie, do czego była potrzebna ta zmienna. Dobrym nawykiem jest również dodawanie komentarzy. Poniżej (Rys. 5.5) przedstawiono różnicę w czytelności 2 identycznych kontraktów. Na pierwszy rzut oka widać, który kontrakt jest bardziej przejrzysty i łatwiej się go czyta dla osoby postronnej, a to ważne przy pracy w grupach.

<pre>1  pragma solidity ^0.5.12; 2 3  contract exampleContract() 4  { 5      uint uintBalance; 6      address owner; 7 8      constructor(){ 9          uintBalance=0; 10     } 11 12     function setBalance(address paramOwner, uint paramBalance) public { 13         owner=paramOwner; 14         uintBalance=paramBalance; 15     } 16 }</pre>	<pre>1  pragma solidity ^0.5.12; 2 3  contract Abc() 4  { 5      uint a; 6      address o; 7 8      constructor(){ 9          a=0; 10     } 11 12     function f(address o, uint a) public { 13         this.o=o; 14         this.a=a; 15     } 16 }</pre>
---	--

Rysunek 5. 5 Porównanie czytelności kontraktu

Źródło opracowania: własne

Podobnie, jak w językach C++ czy Java, w Solidity istnieją wzorce projektowe, które zawierają metody poprawiające czytelność, skalowalność czy bezpieczeństwo kontraktów. Nie zawsze można

ufać takim wzorcom. Najlepiej korzystać tych elementów, które są przedstawione w dokumentacji, gdyż wersje eksperymentalne, zwykle posiadają błędy, tworzące nowe luki bezpieczeństwa.

Używanie odpowiednich kwalifikatorów dla funkcji i zmiennych, również odgrywa znaczącą rolę. Hermetyzacja zabezpiecza kod kontraktu przed zewnętrznymi atakami, a co z tym jest związane, warto jej używać, dla zwiększenia bezpieczeństwa.

Kolejnym dobrym nawykiem jest tworzenie modularnego kodu. Dzięki takim rozwiązaniom jesteśmy w stanie zlikwidować duplikacje w kodzie i zwiększyć używalność funkcji. Modularność kodu polega na zapisywaniu różnych funkcjonalności w innych kontraktach, przez co łatwiej znaleźć funkcje i modyfikować je, bez ingerencji w główny kod kontraktu. Ważną praktyką w modularności jest używanie dostępnych i popularnych narzędzi czy bibliotek, niż pisanie ich od nowa.

Każdy kontrakt powinien przejść testy pod względem bezpieczeństwa czy składni. Takie testy są ciężkie do opracowania przez młodych deweloperów, przez co nieodłączną częścią tego etapu są zewnętrzne frameworki. Niektóre z nich posiadają wbudowane moduły testujące czy całe środowisko testowe. Przykładem może być Truffle, który posiada swoją konsolę developerską, schematy testów, a nawet swoją prywatną sieć EVM, w której możemy przeprowadzać testy bez uprzedniego wdrażania EVM poprzez np. go-ethereum.

Warto również zaimplementować swojego rodzaju bazę wersji kontraktu, która pokazuje czy kontrakt dostał jakieś aktualizacje i będzie odpowiadała za przyszłe uaktualnienia umów. Głównymi metodami dającymi rozwiązanie konceptu bazy wersji są: kontrakty rejestrowe (ang. registry contracts) oraz delegat wywołania (ang. delegatecall). Jak każde rozwiązania, przedstawione metody mają zalety i wady. Głównymi wadami pierwszego rozwiązania są adresy, które użytkownik za każdym razem musi pobierać od nowa, a jeśli ktoś zapomni o tym szczególe, może narazić się na luki bezpieczeństwa, będące w poprzednich wersjach kontraktu. Ponadto problemem dla dewelopera może być również myślenie o danych. Co się wydarzy z wartościami, gdy podmienimy kontrakt na inny. W drugim rozwiązaniu unikamy poprzednich dwóch problemów, ale spotykamy kolejne. Programista musi być bardzo ostrożny deklarując, jak będzie przechowywał dane w tym kontrakcie. Przy aktualizacji kontraktu, jeśli wygląd przechowywania będzie się różnił, dane mogą zostać utracone. Dodatkowo to rozwiązanie nie może zwracać wartości z funkcji, tylko przekierowuje na nie, co ogranicza jego możliwości do tworzenia decentralizowanych aplikacji.

Kolejnym ważnym elementem w pisaniu kontraktów są różnego typu zabezpieczenia na wypadek ataku (np. reentrancy attack). Takie składniki umów, to przerywacze (ang. Circuit Breakers). Jak sama ich nazwa sugeruje, przerywają one wykonanie kontraktu, jeśli napotkano na konkretny

warunek. Dzięki zaimplementowaniu odpowiednich warunków sprawdzających np. eventy, można zautomatyzować wywoływanie się funkcji blokujących. Przykładem może być natrafienie na błąd w domu aukcyjnym. Dzięki przerywaczom, deweloper (lub skrypt) może zablokować wszystkie akcję oprócz wypłaty waluty.

Przedostatnim sposobem na poprawienie bezpieczeństwa kontraktów jest opcja opóźnienia wykonywania umowy. Progi zwalniające (ang. Speed bumps), zwalniają akcję, więc jeśli zaczną dziać się podejrzane rzeczy, to będzie możliwość powrotu do poprzedniego stanu.

Ostatnim sposobem na poprawę jakości kontraktów jest ograniczenie transferu Etheru z konta czyli tzw. limit przepustowości (ang. Rate limiting). Ustanawia on pewien limit na transakcje, przez co zatrzymuje potencjalne zagrożenie przelewu wszystkich środków na inne konta. Przykładem może być funkcja, która pozwala właścicielowi konta wypłacić 10% wartości wpłaconych środków na dzień, przez co 90% jego środków jest bezpieczne.

Wszystkie powyżej przedstawione sposoby mają jakiś wpływ na poprawę pisania kontraktów, jednakże najwięcej zyskamy stosując nie jedną, a wszystkie (lub większość) technik, przez co nasze kontrakty stają się bezpieczne i bardziej użyteczne.

## 6. Sposoby weryfikacji inteligentnych kontraktów

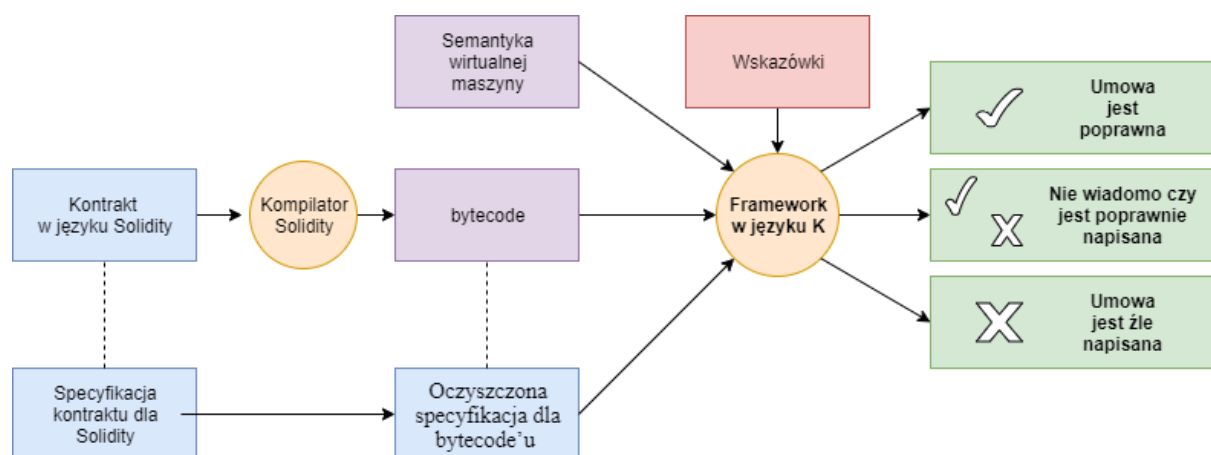
Sprawdzenie działania kontraktu w sieciach blockchain, odgrywa znaczącą rolę, gdyż naprawa błędów raz wdrożonej umowy jest bardzo trudna, a nawet niewykonalna, bez implementacji baz wersji. Na przykładzie ataku The DAO, można zauważyć niszczącą siłę i konsekwencje związane z pomyłkami programistów. Od tamtego momentu, przykładą się większą wagę do weryfikacji inteligentnych umów, a służy do tego weryfikacja formalna.

### 6.1 Weryfikacja formalna

Pisząc kontrakt, chcemy mieć pewność, że zostanie on wykonany zgodnie z założeniem. Nie jest to łatwe stwierdzenie, a nawet niemożliwe, jeśli nie jesteśmy w stanie tego udowodnić. Przykładem weryfikatora jest środowisko *Runtime Verification*.

Model formalnej weryfikacji jest przedstawiony na Rys. 6.1, a poszczególnymi jego składnikami są:

- Specyfikacja kontraktu;
- Oczyszczona specyfikacja dla bytecode'u (ang. Refined Specification)
- Semantyka wirtualnej maszyny (ang. Virtual machine semantics)
- Wskazówki (ang. Hints) wspomagające programy weryfikacyjne napisane np.. w języku K



Rysunek 6. 1 Model formalnej weryfikacji

Źródło opracowania: własne na podstawie Runtime Verification [20]

Sprecyzowane założenia kontraktu dotyczące jego działania, są określane jako specyfikacja kontraktu. Tworzenie tej specyfikacji, jest procesem, który potrzebuje dużego wkładu manualnego, a zadania wymagają fachowej ekspertyzy. Widać, że nie jest to tani proces, ale niezbędny do wykonania weryfikacji formalnej.

Weryfikacja formalna może porównać specyfikację kontraktu napisanego w Solidity ze specyfikacją Solidity, który jest oczyszczoną wersją bytecode'u przy użyciu detali maszyny wirtualnej.

Proces oczyszczania, przez technicznie zawite tłumaczenia, jest obecnie wykonywany manualnie. Semantyka wirtualnej maszyny jest potrzebna do sprawdzenia poprawności bytecode'u kontraktu. Do tego procesu musimy wiedzieć co robi każdy bytecode, a także znać strukturę bitową (ang. Bit-level structure) danych, na których on pracuje. Tworzenie takiej semantyki to ciężka praca dla programisty, ale wystarczy jej jedna wersja dla danej maszyny wirtualnej.

Po stworzeniu specyfikacji kontraktu oraz wirtualnej maszyny, a także oczyszczeniu jej dla bytecode'u, możemy wykorzystać zautomatyzowaną część framework'a K verification, w celu znalezienia dowodu, że skompilowany kontrakt implementuje specyfikację. W idealnym świecie otrzymalibyśmy dwie odpowiedzi:

- „Oto wyprowadzony dowód, że umowa jest poprawna”
- „Umowa jest źle napisana”

Jednakże nie żyjemy w idealnym świecie, a to oznacza, że weryfikator może nie zwrócić nam żadnej wartości, dlatego ważnym składnikiem stają się wskazówki, które są dyrektywami od programisty dla programu. Te wyznaczniki polegają na ominięciu pewnego etapu weryfikacji, na którym program próbuje wykonać pracę bardzo długo. Tworzenie wskazówki przebiega w następujący sposób:

1. Program weryfikujący zgłasza problem z weryfikacją danego etapu.
2. Programista otrzymuje identyfikator danego etapu, po czym ręcznie sprawdza jego składnię, semantykę oraz logikę.
3. Po tym etapie, dodaje on wskazówkę do bazy, czy dana struktura jest poprawna czy nie, przez co następnym razem, gdy weryfikator napotka na taką konstrukcję, będzie mógł przejść ten etap, bez tworzenia dowodu.

## 6.2 Zagrożenia związane z weryfikacją formalną

Pomimo utworzenia dowodu dotyczącego kontraktu i plików związanych z tym etapem, istnieje dodatkowe ryzyko, które może prowadzić do luk bezpieczeństwa. Przykładami takich możliwości są:

- Błąd w programie weryfikacyjnym
- Błędna semantyka maszyny wirtualnej
- Błędna specyfikacja języka Solidity
- Błędne tłumaczenie specyfikacji języka Solidity na bytecode



### 6.3 Przyszłościowe sposoby weryfikacji

Twórcy oprogramowań weryfikacyjnych ciągle rozwijają swoje projekty, przez co przyszłość wydaje się bardzo zautomatyzowana. Weryfikatory będą posiadały coraz większą moc obliczeniową, a wkład ludzki będzie ograniczony do minimum, gdyż w każdym etapie, za który jest odpowiedzialny człowiek, może pojawić się błąd. Twórcy *Runtime Verification* pracują obecnie nad udoskonaleniem swoich kontrolerów. Obecnie stworzyli oni w pełni zautomatyzowany program tworzący semantykę wirtualnej maszyny. Niestety jeszcze nie jest on dostępny dla wszystkich, gdyż twórcy szukają błędów, które nie wpłyną na przyszłe luki bezpieczeństwa i ataki.

## Podsumowanie

Celem mojej pracy było przedstawienie tematu Inteligentnych Kontraktów oraz problemów bezpieczeństwa związanych z nimi, a także przeprowadzenie symulacji ataków na platformę Ethereum.

Początek niniejszej pracy zawiera opis architektury platformy Ethereum, narzędzia używane do stworzenia swojej prywatnej sieci, a także semantykę jednego z języków do programowania kontraktów.

Główną częścią pracy, a zarazem częścią praktyczną, było przeprowadzenie symulacji ataków na wdrożone umowy. Używając truffle'a oraz Remix'a, udało się przeprowadzić dwa ataki na platformę Ethereum m.in. atak typu reentrancy na kontrakt, w którym użyto niskopoziomowej funkcji call, a także atak typu Integer Overflow, w którym nastąpiło przepełnienie zmiennej typu uint8. Rozdział 4 zawiera opracowanie tych ataków wraz z wynikami, a do pracy dołączono dwa filmy przedstawiające te działania.

Ostatecznie w pracy zostały przedstawione techniki pisania kontraktów w języku Solidity, a także sposoby weryfikacji umów. Oba końcowe rozdziały odpowiadają za zwiększenie poziomu bezpieczeństwa na platformie Ethereum.

Moim zdaniem technologia jest bardzo obiecująca, a sama decentralizacja prowadzi do zwiększenia bezpieczeństwa (choć nie w każdym przypadku) danych. W Polsce ten temat jest niestety zbyt mało ukształtowany, a przepisy nie ujednolicają umów, przez co minie jeszcze kilkanaście lat, zanim poznamy prawdziwe możliwości tej technologii, jak np. kupno samochodu bez wychodzenia z domu do urzędów czy ogólnoświatowa dokumentacja medyczna pacjenta. Przykładów wykorzystania tej technologii może być znacznie więcej, co prowadzi do znacznego zapotrzebowania na programistów czy prawników.

## Bibliografia

- [ 1 ] Vitalik Buterin, Gavin Wood, Joseph Lubin: *A Next-Generation Smart Contract and Decentralized Application Platform* (<https://github.com/ethereum/wiki/wiki/white-paper>) [wersja 1 – 11.2013]
- [ 2 ] Gavin Wood *Ethereum: A secure decentralised generalised transaction ledger* (<https://github.com/ethereum/yellowpaper>) [wersja 1 – 04.2014]
- [ 3 ] Sebastian Bala, Tomasz Kopyściański, Witold Srokosz: *Kryptowaluty jako elektroniczne instrumenty płatnicze bez emitenta* (Wydawnictwo Uniwersytetu Wrocławskiego) [2016]
- [ 4 ] Debajani Mohanty: *Ethereum for Architects and Developers - With Case Studies and Code Samples* (Wydawnictwo: Apress) [30.10.2018]
- [ 5 ] Ritesh Modi: *Solidity Programming Essentials* (Packt Publishing) [wydanie 1 – 20.04.2018]
- [ 6 ] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli: *A survey of attacks on Ethereum smart contracts* (<https://eprint.iacr.org/2016/1007.pdf>) [2016]
- [ 7 ] Preethi Kasireddy: *How does Ethereum work, anyway?* (<https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway>) [13.09.2017]
- [ 8 ] Jing Liu: *A Survey on Security Verification of Blockchain Smart Contracts* (<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8732934>) [20.05.2019]
- [ 9 ] Jiao Jiao: *Formal Specification and Verification of Smart Contracts* ([https://www.researchgate.net/publication/336848406\\_Formal\\_Specification\\_and\\_Verification\\_of\\_Smart\\_Contracts](https://www.researchgate.net/publication/336848406_Formal_Specification_and_Verification_of_Smart_Contracts)) [2019]
- [ 10 ] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer: *Formal Specification and Verification of Smart Contracts in Azure Blockchain* (<https://www.microsoft.com/en-us/research/uploads/prod/2019/05/1812.08829.pdf>) [04.2019]
- [ 11 ] [www.ethereum.org](http://www.ethereum.org) [02.02.2020]
- [ 12 ] [www.github.com/ethereum](http://www.github.com/ethereum) [02.02.2020]
- [ 13 ] [www.ethereum.stackexchange.com](http://www.ethereum.stackexchange.com) [07.02.2020]
- [ 14 ] [www.github.com/trufflesuite/truffle](http://www.github.com/trufflesuite/truffle) [01.02.2020]
- [ 15 ] [www.github.com/trufflesuite/ganache](http://www.github.com/trufflesuite/ganache) [01.02.2020]
- [ 16 ] [www.solidity.readthedocs.io](http://www.solidity.readthedocs.io) [22.01.2020]
- [ 17 ] <https://swcregistry.io> [22.01.2020]
- [ 18 ] <https://ethon.consensys.net> [26.01.2020]
- [ 19 ] [www.bitdegree.org](http://www.bitdegree.org) [27.01.2020]
- [ 20 ] [www.github.com/runtimeverification](http://www.github.com/runtimeverification) [27.01.2020]
- [ 21 ] [https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/#What\\_is\\_the\\_Proof\\_of\\_work](https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/#What_is_the_Proof_of_work) [30.01.2020]

- [ 22 ] <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ#what-is-proof-of-stake>  
[30.01.2020]
- [ 23 ] [www.coindesk.com/understanding-dao-hack-journalists](http://www.coindesk.com/understanding-dao-hack-journalists) [27.01.2020]
- [ 24 ] <https://github.com/ethereum/solidity/issues/2884#issuecomment-329169020> [30.01.2020]
- [ 25 ] <https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages> [07.02.2020]
- [ 26 ] <https://remix-ide.readthedocs.io/> [11.02.2020]

## Spis rysunków

### Rozdział 1

Rysunek 1. 1 Schemat szyfrowania i odszyfrowania przy użyciu kluczy prywatnych oraz publicznych .....	8
Rysunek 1. 2 Wartości systemu metrycznego Ethereum .....	8
Rysunek 1. 3 Przykłady kont w Ethereum .....	10
Rysunek 1. 4 Transakcja klasyczna .....	11
Rysunek 1. 5 Wdrożenie inteligentnego kontraktu .....	12
Rysunek 1. 6 Uproszczony schemat połączenia bloków .....	13
Rysunek 1. 7 Struktura zatwierdzonego bloku .....	14
Rysunek 1. 8 Schemat drzewa Merklego .....	15

### Rozdział 2

Rysunek 2. 1 Deklaracja i konfiguracja bloku Genesis .....	17
Rysunek 2. 2 Inicjalizacja bloku Genesis .....	18
Rysunek 2. 3 Uruchomienie prywatnej sieci Ethereum .....	19
Rysunek 2. 4 Uruchomienie konsoli Geth JavaScript .....	19
Rysunek 2. 5 Transfer środków w transakcjach klasycznych .....	19
Rysunek 2. 6 Zatwierdzenie transakcji i dołączenie jej do bloku .....	20
Rysunek 2. 7 Wygenerowanie kont przez polecenie „truffle develop” .....	21
Rysunek 2. 8 Okno konfiguracji Ganache .....	22
Rysunek 2. 9 Lokalne ustawienia prywatnej maszyny .....	22
Rysunek 2. 10 Deklaracja i ustawienia kont .....	23
Rysunek 2. 11 Podstawowe ustawienia łańcucha .....	23
Rysunek 2. 12 Wygenerowane konta i ich położenie .....	24
Rysunek 2. 13 Klucz prywatny jednego z kont .....	25
Rysunek 2. 14 Wykopane bloki w EVM przy użyciu Ganache'a .....	25
Rysunek 2. 15 Dane szczegółowe bloku .....	26
Rysunek 2. 16 Szczegółowe dane wdrożonej transakcji .....	27
Rysunek 2. 17 Spis przeprowadzonych transakcji .....	28
Rysunek 2. 18 Dane szczegółowe stworzonej inicjalizacji kontraktu do puli .....	28
Rysunek 2. 19 Wdrożone kontrakty do łańcucha bloków .....	30
Rysunek 2. 20 Informacje o wdrożonym bloku .....	30
Rysunek 2. 21 Szczegóły transakcji o podanym hash'u .....	31

## Rozdział 4

Rysunek 4. 1 Fragment wdrożonego kontraktu przedstawiający kod źródłowy .....	42
Rysunek 4. 2 Stan kont w sieci przed przeprowadzeniem ataku reentrancy .....	45
Rysunek 4. 3 Wdrożenie kontraktu reenter_mallory.sol .....	46
Rysunek 4. 4 Stan kont w sieci po przeprowadzeniu ataku reentrancy .....	46
Rysunek 4. 5 Szczegóły transakcji wdrożenia kontraktu reenter_mallory.sol wraz z przelewem kwoty etheru z konta [0] .....	47
Rysunek 4. 6 Konfiguracja kompilatora w środowisku Remix dla wdrożenia kontraktu Token.sol .	52
Rysunek 4. 7 Wdrożenie kontraktu TokenSaleChallenge do sieci JavaScript VM (REMIX) .....	53
Rysunek 4. 8 "Kupienie" 250 żetonów kontraktu TokenSaleChallenge.....	54
Rysunek 4. 9 "Kupienie" 7 żetonów, które powodują przepełnienie uint8 .....	54

## Rozdział 5

Rysunek 5. 1 Dziedziczenie pojedyncze .....	58
Rysunek 5. 2 Dziedziczenie wielopoziomowe.....	58
Rysunek 5. 3 Dziedziczenie hierarchiczne .....	59
Rysunek 5. 4 Dziedziczenie wielopoziomowe.....	59
Rysunek 5. 5 Porównanie czytelności kontraktu .....	60

## Rozdział 6

Rysunek 6. 1 Model formalnej weryfikacji .....	63
--	----

## Spis listingów

### Rozdział 2

Listing 2. 1 Fragment kodu BIN (Bytecode) dla kontraktu Bob1.sol .....	29
Listing 2. 2 Kod ABI dla kontraktu Bob1.sol .....	29

### Rozdział 3

Listing 3. 1 Przykładowy kontrakt.....	33
Listing 3. 2 Struktury kontrolne – przykład .....	36
Listing 3. 3 Kontrakt ilustrujący przepełnienie funkcji .....	39
Listing 3. 4 Przykład używania modyfikatorów .....	40
Listing 3. 5 Przykład eventu.....	41

### Rozdział 4

Listing 4. 1 Reentrancy attack – przykładowa implementacja .....	43
Listing 4. 2 Reentrancy attack - rozwiązanie problemu funkcji "call".....	44
Listing 4. 3 Skrypt wdrażający 5_initial_reenter_mallory.js .....	44
Listing 4. 4 Kod spreparowanego kontraktu, odwołującego się do Bob.sol .....	45

Listing 4. 5 Przykładowy kontrakt ze sprawdzaniem czasu ukończenia operacji .....	49
Listing 4. 6 Przykład kontraktu podatnego na atak Integer Overflow and Underflow .....	50
Listing 4. 7 Przykład kontraktu niwelującego atak Integer Overflow and Underflow .....	50
Listing 4. 8 Przykład nr. 2 kontraktu podatnego na atak Integer Overflow .....	51
Listing 4. 9 Przykład działania DoS with revert .....	55
Listing 4. 10 Przykładowy kontrakt rozwiązujący podatność DoS with revert (metoda Favor pull over push) .....	56

## Spis załączników

1. Bytecode kontraktu Bob1.sol wraz z kodami operacyjnymi oraz mapą źródła  
[[Bob1.bin](#)],[[Bob1.assembly](#)]
2. Szczegóły kontraktu Bob.sol wraz z kodem źródłowym [[Bob\\_details.docx](#)]
3. Film przedstawiający atak Reentrancy [[Załącznik 3 - atak typu Reentrancy.mp4](#)]
4. Film przedstawiający atak Overflow oraz Underflow [[Załącznik 4 - atak Integer Overflow.mp4](#)]