Cuprins

Introducere

1.1 Context

Un smart contract este un self-executing program prin intermediul căruia poţi transfera bani sau orice alte bunuri într-o manieră transparentă, evitând intervenţia unui intermediar.

Smart contractele au fost gândite pentru a rezolva diverse probleme, reprezentând fundamentul pentru multe ICO-uri (Initial Coin Offering). Un ICO reprezintă un mijloc nereglementat prin care sunt strânse fonduri pentru o nouă afacere în domeniul cryptocurrency. Un ICO este folosit de un startup pentru a ocoli procesul legal în ceea ce priveşte capitalul cerut, spre exemplu, de bănci. Smart conctractele ajută oamenii să atragă investitori şi să-şi dezvolte afacerea la un nivel cât mai ridicat.

Smart contractele au căpătat din ce în ce mai mult teren, evident, în industria financiară, companiile alegându-le în proiectele sale în detrimentul intermediarilor. Ele pot fi folosite pentru a înregistra şi stoca informaţii financiare, realizând tranzacţii şi plăţi, colectarea fondurilor, stocarea şi distribuirea resurselor, toate acestea realizându-se automat. Pe lângă toate acestea, frauda este imposibilă.

Mai jos avem un smart contract simplificat cu explicaţii în conformitate cu documentaţia oficială **Solidity**:

```
pragma solidity ^0.4.21;

contract Coin {
    address public minter;
    mapping (address => uint) public balances;

    function Coin() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        if (msg.sender != minter) return;
            balances[receiver] += amount;
    }
```

```
    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
}
```

Linia pragma solidity ^0.4.21; reprezintă versiunea de Solidity cu care să fie compilat contractul.

Linia address public minter; reprezintă declararea unei variabile de tip *address* care poate fi accesată public în afara contractului. Keywordul *public* generază automat o funcţie ce permite accesarea variabilei din afara contractului. Tipul de date *address* este o valoare pe 160 de biţi care nu suportă niciun tip de operaţie aritmetică; se utilizează pentru o stoca adresele contractelor sau ale persoanelor din exterior.

Linia mapping (address => uint) public balances; crează de asemenea o variabilă publică. Acest tip de date reprezintă o mapare a adreselor la numere intregi. Putem privi acest map ca mapurile din alte limbaje de programare (ex. Java - HashMap<address, uint>, C++ - map<address, uint> etc).

Funcţia Coin reprezintă constructorul contractului, fiind apelat la crearea obiectului. În general constructorul stochează adresa persoanei care a creat contractul (în caz contrar reprezintă un real pericol de a fi exploatat). msg este o variabilă globală cheie, care conţine proprietăţi care permit accesarea blockchainului. msg.sender reprezintă adresa de unde s-a apelat funcţia.

Funcţiile care pot fi apelate din exterior sunt mint şi send. mint poate fi apelată doar de persoana care a creat contractul; în caz contrar nu se va întampla nimic. send poate fi apelată de oricine (care are deja *coins*) pentru a trimite coins altora.

Verificarea din funcţia send e următoarea:

$if(balances[msg.sender] < amount)return;$ => în caz ca apelantul funcţiei nu deţine suma necesară pentru a o trimite.

O alta verificară ar fi overflowul, care se poate intampla la cel care primeşte.

1.2 Motivaţie

Smart contractele conţin cod şi memorie, pot interacţiona cu alte contracte şi pot trimite şi primi "ether" la şi de la alte conturi. Sunt imutabile, deci trebuie gândite foarte bine din start. Orice greşeala creează oportunităţi pentru răuvoitori. De aceea atunci când i se face deploy unui smart contract trebuie să ne asiguram că respectă anumite proprietăţi.

În continuare voi prezenta un contract vulnerabil care a cauzat o pagubă iniţială de 100 de milioane de dolari, ulterior dovedindu-se a fi mult mai mare. Conform `www.hackernoon.com`, atacatorul a recurs la doar 2 tranzacţii. Pentru simplificare voi prezenta o variantă simplificată a contractului cu pricina.

```
pragma solidity ^0.4.17;
contract SuperWallet {
```

```
    mapping (address => uint) balances;
    address owner;
    modifier onlyOwner {
        assert(msg.sender == owner);
        _;
    }

    function initWallet() {
        owner = msg.sender;
    }

    function kill() public onlyOwner {
        suicide(msg.sender);
    }

    function getBalance(address user) constant public returns(uint) {
        return balances[user];
    }
    function topup() public payable {
        if (msg.value == 0) {
            throw;
        }
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint amount) public {
        if (balances[msg.sender] == 0) {
            throw;
        }
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

Prima greşeala o găsim la nivelul constructorului. In *Solidity*, constructorul se defineşte ca fiind o funcţie care se va apela la crearea obiectului respectiv. În cele mai multe cazuri în constructor se iniţializează variabilă *owner*. Ca în majoritatea limbajelor de programare, convenţia este ca numele constructorului să aibă denumirea contractului. În cazul de faţă nu avem de-a face cu un constructor, ci cu o funcţie separată de iniţializare, numită *initWallet*, care nu are specificat niciun modificator de acces, Solidity tratând această funcţie ca fiind publică.

Astfel, odată creată o instanţă a contractului de mai sus, acesta va fi *owner-less*, întrucât nu avem un

constructor în care să se specifice ownerul. Deci, după ce i s-a facut deploy, oricine poate apela funcţia *initWallet*, devenind owner.

Odată ce un atacator devine owner pe contractul respectiv, el poate apela functia *kill()*, care conţine un apel catre *suicide()*. Instrucţiunea SUICIDE transferă toate fondurile contractului către apelant.

Cea de-a doua greşeala constă în lipsa verificării în funcţia *withdraw(uint amount)*, care ar putea produce underflow. Dacă valoarea variabilei *amount* este mai mare decât *balances[msg.sender]*, atunci se va produce underflow în variabila *balances[msg.sender]*. Cu toate că se va produce underflow, execuţia programului va continua cu valori arbitrare.

Prin urmare, atacatorul a apelat prima data funcţia *initWallet*, devenind owner. Apoi a apelat funcţia *kill* în cea de-a doua tranzacţie.

Observaţie!

Instrucţiunea SUICIDE este considerată deprecated, fiind înlocuită cu SELFDESTRUCT. Din exterior probabil această instrucţiune este inutilă, întrucât transferă toate fondurile la adresa dată ca parametru, făcând un contract foarte vulnerabil. Însă are două mari avantaje: SELFDESTRUCT este mult mai eficientă în ceea ce priveşte *gas-ul* faţă de *address.send(uint amount)*. De fapt, SELFDESTRUCT este foarte utilă, întrucât operaţia elimină spaţiu în blockchain eliminând datele contractului.

1.3 Contribuţii

Pentru a putea demonstra că un smart contract respectă anumite proprietăţi trebuie modelată semantica pentru EVM (Ethereum Virtual Machine). Pe baza acestei semantici se vor executa o serie de programe şi se vor analiza rezultatele experimentelor. Mai precis se vor rula programe cu complexitate progresivă pentru a ne asigura atât de corectitudinea operaţiilor simple, cât şi a celor mai complexe.

Mai jos voi prezenta principalele contribuţii:

1. Principala contribuţie este scrierea semanticii pentru instrucţiunile EVM, iar apoi scrierea lemelor care verifică anumite proprietăţi ale contractelor. Astfel putem rula contractul la cel mai de jos nivel, la nivel de bytecode (opcode), simulând un mediu virtual (stivâ, memorie, storage etc) şi verificând corectitudinea funcţionala a contractului şi descoperind vulnerabilităţile care reprezintă un pericol din punct de vedere al securităţii.

2. Prelucrarea conţinutului obţinut de pe aplicatia web Remix astfel încât să reprezinte o sintaxă validă pentru Coq.

3. La nivel practic am rulat programe cărora le-am verificat diverse proprietăţi si am făcut un tabel cu statistici.

1.4 Structura lucrării

Lucrarea prezentă este împărţită în 5 capitole:

1. Introducere
   (a) Context
   (b) Motivaţie
   (c) Contribuţii
   (d) Structura lucrării
2. Verificarea în contextul smart contracts
   (a) Problema
   (b) Abordări şi soluţii existente
3. O semantică formală pentru Ethereum Virtual Machine
   (a) Prezentarea generală a contribuţiilor
   (b) Modelarea execuţiei expresiilor aritmetice
   (c) Modelarea execuţiei comparaţiilor şi a operaţiilor logice pe biţi
   (d) Informaţii de mediu
   (e) Modelarea execuţiei operaţiilor pe stivă, memorie, storage şi a operaţiilor de flow
   (f) Modelarea execuţiei operaţiilor de PUSH, SWAP şi DUPLICATE
4. Experimente şi evaluarea soluţiei
   (a) Experimente realizate
   (b) Concluzii
5. Concluzii
   (a) Rezumat
   (b) Îmbunătăţiri
6. Appendix
7. Bibliografie

Verificarea în contextul smart contracts

2.1 Problema

Principiul EVM-ului (Ethereum Virtual Machine) este următorul: toate metodele sau funcţiile unui smart contract sunt executate ca şi tranzacţii. Este nevoie de un program pentru a compila, a face deploy şi a executa funcţiile şi metodele din smart contract. O soluţie este aplicaţia web Remix, care ne ajută în ceea ce priveşte compilarea. După ce i s-a făcut deploy smart contractului, userul poate interacţiona cu acesta folosind diverse portofele de monede virtuale (MEW, Ethereum Wallet/MIST, Parity). Limbajul în care este scris smart contractul este Solidity.

```
pragma solidity ^0.4.0;
contract Counter {
    int private count = 0;
    function incrementCounter() public {
        count += 1;
    }
    function decrementCounter() public {
        count -= 1;
    }
    function getCount() public constant returns (int) {
        return count;
    }
}
```

Avem un contract cu variabila privată count, care nu poate fi accesată de cineva din afara contractului si 3 funcţii. Prima funcţie, *incrementCounter*, modifică valoarea variabilei count, incrementând-o. Cea de-a doua funcţie, decrementCounter, modifică valoarea variabilei count, decrementând-o. Cea de-a treia funcţie, *getCount*, returnează valoarea variabilei count oricui apelează această funcţie. După cum am spus, după ce s-a făcut deploy nu mai putem modifica smart contractul. Facem următoarea modificare contractului.

```
pragma solidity ^0.4.0;
contract Counter {
    int private count = 0;
```

```
    function incrementCounter() public {
        count += 1;
    }
    function decrementCounter() public {
        count -= 1;
    }
    function getCount() public constant returns (int) {
        return count;
    }

    function test() public returns(int){
        incrementCounter();
        incrementCounter();
        incrementCounter();
        decrementCounter();
        return count;
    }
    int test_var = test();
}
```

La finalul execuţiei contractului, valoarea variabilei *test_var* va trebui să fie 2. Acesta este un exemplu banal, ţinând cont că un smart contract este mult mai complex în realitate. Însă prin banalitatea acestui exemplu este uşor de demonstrat principiul. Raportându-ne la exemplul de mai sus, la finalul execuţiei ne aşteptăm să avem în storage la adresa 1 valoarea 2. În primă instanţă pentru a rezolva o asemenea problemă trebuie să putem executa bytecodul generat din Solidity. Ca să îl putem executa trebuie să dăm semantică fiecărei instrucţiuni în parte. Pe baza acestei semantici vom putea demonstra că î]n storage avem valoarea 2 la adresa 1. Astfel putem demonstra anumite proprietăţi ale unui smart contract. Acest exemplu simplu poate fi extins la contracte mult mai complexe.

2.2 Abordări şi soluţii existente

1. Alloy : dezvoltat de Daniel Jackson la MIT. Oferă posibilitatea programatorului de a explora modelul într-un mod interactiv şi de a verifica instrucţiunile logice complexe în mod automat.

2. K Framework : într-o colaborare cu Grigore Roşu, Runtime Verification (RV) a folosit frameworkul K pentru a construi şi testa un model matematic al EVM-ului (Ethereum Virtual Machine), ceea ce face posibilă verificarea formală a acurateţii smart contractelor.

3. Isabelle : contribuţiile generale sunt formalizarea EVM in frameworkul Isabelle, aratând corectitudinea proprietăţilor unui smart contract; Isabelle oferă o altă abordare conceptului de *gas*; conceptul se bazează pe generarea automată a condiţiilor de verificare folosindu-se de regulile logicii.

O semantică formală pentru Ethereum Virtual Machine

3.1 Prezentarea generală a contribuţiilor

Mai jos voi prezenta principalele contribuţii:

1. Principala contribuţie este scrierea semanticii pentru instrucţiunile EVM, iar apoi scrierea lemelor care verifică anumite proprietăţi ale contractelor. Astfel putem rula contractul la cel mai de jos nivel, la nivel de bytecode (opcode), simulând un mediu virtual (stivâ, memorie, storage etc) şi verificând corectitudinea funcţionala a contractului şi descoperind vulnerabilităţile care reprezintă un pericol din punct de vedere al securităţii.

2. Prelucrarea conţinutului obţinut de pe aplicatia web Remix astfel încât să reprezinte o sintaxă validă pentru Coq.

3. La nivel practic am rulat programe cărora le-am verificat diverse proprietăţi si am făcut un tabel cu statistici.

Modelul execuţiei specifică cum ar trebui să se altereze starea sistemului dându-se o serie de instrucţiuni şi un tuplu de date de mediu. Acest model este cunoscut sub numele de ***Ethereum Virtual Machine*** (EVM). Maşina virtuală este limitată de parametrul *gas*, care limitează numărul total de calcule.

În conformitate cu **Ethereum Yellow Paper**, EVM este o arhitectură bazată pe stivă. Elementele de pe stivă sunt pe 256 de biţi. Memoria este modelată ca o funcţie ce primeşte ca parametru adresa unui element şi returnează elementul de la adresa dată ca parametru. Storage-ul este modelat similar memoriei; în comparaţie cu memoria, care este volatilă, storage-ul nu este volatil şi este menţinut ca parte a sistemului. Toate locaţiile din storage, respectiv memorie sunt iniţializate cu 0.

Pe lângă starea sistemului, σ şi gasul rămas pentru calcule, g, mai sunt câteva elemente importante folosite la execuţie. Ele sunt conţinute în tuplul $\boldsymbol{I}$:

1. I_a, adresa contului care deţine codul care se execută.

2. I_o, adresa emiţătorului tranzacţiei care a generat execuţia.

3. I_p, preţul gasului din tranzacţia care a generat execuţia.

4. I_d, vectorul de bytes care este input pentru execuţie.

5. I_s, adresa contului care a generat execuţia codului.
6. I_v, valoarea, în Wei, trimisă contului ca parte a aceleiaşi proceduri ca execuţia.
7. I_b, vectorul de bytes ce conţine codul ce urmează a fi executat.
8. I_H, headerul blocului curent.
9. I_e, "adâncimea" contractului (de câte ori s-a executat CALL sau CREATE în prezent).
10. I_w, permisiunea de a face modificări stării.

    O stare este definită ca un tuplu (g, pc, m, i, s) care reprezintă, în ordine, gasul disponibil, contorul program, memoria, numărul de words din memorie şi stiva.

3.2 Modelarea execuţiei expresiilor aritmetice

Operaţiile aritmetice şi de oprire implementate sunt următoarele:

1. STOP, opreşte execuţia
2. ADD, $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$ operaţia de adunare
   Exemplu: presupunând ca elementele de pe stivă sunt, de la stânga la dreapta:

   $$1 \ :: \ 2 \ :: \ 3 \ :: \ 4 \ :: \ 5\,,$$

   după execuţia instrucţiunii *ADD* stiva va fi modificată astfel:

   $$1 \ :: \ 2 \ :: \ 3 \ :: \ 9$$

3. MUL, $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$ operaţia de înmulţire
4. SUB, $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$ operaţia de scădere
5. DIV, operaţia de împărţire

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[1] = 0, \\ \mu_s[0] \div \mu_s[1] & otherwise \end{cases}$$

6. SDIV, operaţia de împărţire a numerelor întregi cu semn

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[1] = 0, \\ -2^{255} & if \mu_s[0] = -2^{255} \wedge \mu_s[1] = -1, \\ sgn(\mu_s[0] \div \mu_s[1])|(\mu_s[0] \div \mu_s[1])| & otherwise \end{cases}$$

7. MOD, operaţia modulo

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[1] = 0, \\ \mu_s[0] \ mod \ \mu_s[1] & otherwise \end{cases}$$

8. SMOD, operaţia modulo a numerelor cu semn

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[1] = 0, \\ sgn(\mu_s[0])(|\mu_s[0] \ \ mod \ \ \mu_s[1]|) & otherwise \end{cases}$$

9. ADDMOD, operaţia de adunare modulo

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[2] = 0, \\ \mu_s[0] + \mu_s[1] \ \ mod \ \ \mu_s[2] & otherwise \end{cases}$$

10. MULMOD, operaţia de înmultire modulo

$$\mu'_s[0] \equiv \begin{cases} 0 & if \mu_s[2] = 0, \\ \mu_s[0] \times \mu_s[1] \ \ mod \ \ \mu_s[2] & otherwise \end{cases}$$

11. EXP, operaţia de ridicare la putere $\mu'_s[0] \equiv \mu_s[0]^{\mu_s[1]}$

Definiţia inductivă a instrucţiunilor:

```
Inductive instr : Set :=
(* STOP AND ARITHMETIC OPERATIONS *)
| STOP : instr
| ADD : instr
| MUL : instr
| DIV : instr
| SDIV : instr
| SUB : instr
| MOD : instr
| SMOD : instr
| ADDMOD : instr
| MULMOD : instr
| EXP : instr
...
```

Execuţia instrucţiunilor aritmetice:

```
Fixpoint run_instr (instruction : instr) (state: State) (gas : nat) (I : I) : State :=
match state with
| myState g pc m s i stack =>
match stack with
| myStack stck =>
match instruction with
```

```
(* STOP AND ARITHMETIC OPERATIONS *)
| STOP => match stck with
| s' => myState (g-C(ADD)) (pc+1) m s i (myStack (s'))
end
| ADD => match stck with
| n1 :: n2 :: s' => myState (g-C(ADD)) (pc+1) m s i (myStack (Z.add n1 n2 :: s'))
| _ => state
end
| MUL => match stck with
| n1 :: n2 :: s' => myState (g-C(MUL)) (pc+1) m s i (myStack (Z.mul n1 n2 :: s'))
| _ => state
end
| SUB => match stck with
| n1 :: n2 :: s' => myState (g-C(MUL)) (pc+1) m s i (myStack (Z.sub n1 n2 :: s'))
| _ => state
end
| DIV => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.div n1 n2 :: s'))
| _ => state
end
| SDIV => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.div n1 n2 :: s'))
| _ => state
end
| MOD => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.modulo n1 n2 :: s'))
| _ => state
end
| SMOD => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.modulo n1 n2 :: s'))
| _ => state
end
| ADDMOD => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.add n1 n2 :: s'))
| _ => state
end
| MULMOD => match stck with
| n1 :: n2 :: s' => myState (g-C(DIV)) (pc+1) m s i (myStack (Z.mul n1 n2 :: s'))
```

```
| _ => state
end
| EXP => match stck with
| n1 :: n2 :: s' => myState (g–C(EXP)) (pc+1) m s i (myStack (Z.pow n1 n2 :: s'))
| _ => state
end
| SIGNEXTEND => state
(* STOP AND ARITHMETIC OPERATIONS /*)
...
```

3.3 Modelarea execuţiei comparaţiilor şi a operaţiilor logice pe biţi

1. LT, comparaţia aritmetică "less-than"

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] < \mu_s[1], \\ 0 & otherwise \end{cases}$$

2. GT, comparaţia aritmetică "greater-than"

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] > \mu_s[1], \\ 0 & otherwise \end{cases}$$

3. SLT, comparaţia aritmetică "less-than" pe numere cu semn

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] < \mu_s[1], \\ 0 & otherwise \end{cases}$$

, unde toate valorile sunt considerate pozitive.

4. SGT, comparaţia aritmetică "greater-than" pe numere cu semn

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] > \mu_s[1], \\ 0 & otherwise \end{cases}$$

5. EQ, comparaţia aritmetică de egalitate

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] = \mu_s[1], \\ 0 & otherwise \end{cases}$$

6. ISZERO, operatorul NOT

$$\mu'_s[0] \equiv \begin{cases} 1 & if \mu_s[0] = 0, \\ 0 & otherwise \end{cases}$$

7. AND, AND logic pe biţi
   $\forall i \in [0..255] : \mu'_s[0]_i \equiv \mu_s[0]_i \wedge \mu_s[1]_i$
8. OR, OR logic pe biţi $\forall i \in [0..255] : \mu'_s[0]_i \equiv \mu_s[0]_i \vee \mu_s[1]_i$
9. XOR, XOR logic pe biţi $\forall i \in [0..255] : \mu'_s[0]_i \equiv \mu_s[0]_i \oplus \mu_s[1]_i$
10. NOT, NOT logic pe biţi

$$\forall i \in [0..255] : \mu'_s[0]_i \equiv \begin{cases} 1 & if \mu_s[0]_i = 0, \\ 0 & otherwise \end{cases}$$

```
Definition lt (x : Z) (y : Z) : Z :=
match Z.ltb x y with
| false => 0
| true => 1
end.

Definition gt (x : Z) (y : Z) : Z :=
match Z.gtb x y with
| false => 0
| true => 1
end.

Definition eq (x : Z) (y : Z) : Z :=
match Z.sub x y with
| 0%Z => 1
| _ => 0
end.



Fixpoint run_instr (instruction : instr) (state: State) (gas : nat) (I : I) : State :=
match state with
| myState g pc m s i stack =>
match stack with
| myStack stck =>
match instruction with
(* COMPARISON & BIWISE LOGIC OPERATIONS *)
```

```
| LT => match stck with
| n1 :: n2 :: ss' => myState (g-C(LT)) (pc+1) m s i (myStack ((lt n1 n2) :: ss'))
| _ => state
end
| GT => match stck with
| n1 :: n2 :: tt' => myState (g-C(GT)) (pc+1) m s i (myStack ((gt n1 n2) :: tt'))
| _ => state
end
| SLT => match stck with
| n1 :: n2 :: uu' => myState (g-C(SLT)) (pc+1) m s i (myStack ((lt n1 n2) :: uu'))
| _ => state
end
| SGT => match stck with
| n1 :: n2 :: vv' => myState (g-C(SGT)) (pc+1) m s i (myStack ((gt n1 n2) :: vv'))
| _ => state
end
| EQ => match stck with
| n1 :: n2 :: rr' => myState (g-C(EQ)) (pc+1) m s i (myStack ((eq n1 n2) :: rr'))
| _ => state
end
| ISZERO => match stck with
| n1 :: s' => myState (g-C(ISZERO)) (pc+1) m s i (myStack (eq 0 n1 :: s'))
| _ => state
end
| AND => match stck with
| n1 :: n2 :: s' => myState (g-C(AND)) (pc+1) m s i (myStack ((Z.land n1 n2) :: s'))
| _ => state
end
| OR => match stck with
| n1 :: n2 :: s' => myState (g-C(OR)) (pc+1) m s i (myStack ((Z.lor n1 n2) :: s'))
| _ => state
end
| XOR => match stck with
| n1 :: n2 :: s' => myState (g-C(XOR)) (pc+1) m s i (myStack ((Z.lxor n1 n2) :: s'))
| _ => state
end
| NOT => match stck with
| n1 :: n2 :: ss' => myState (g-C(NOT)) (pc+1) m s i (myStack ((Z.lnot n1) :: ss'))
```

```
| _ => state
end
(* COMPARISON & BIWISE LOGIC OPERATIONS */)
```

3.4 Informaţii de mediu

1. ADDRESS, adresa contului curent ce execută

$$\mu'_s[0] \equiv I_a$$

2. CALLVALUE, valoarea depozitată de instrucţiunea/tranzacţia responsabilă execuţiei

$$\mu'_s[0] \equiv I_v$$

3. CALLDATALOAD, inputul pentru mediul curent

$$\mu'_s[0] \equiv I_d[\mu_s[0]...(\mu_s[0]+31)] \;\; cu \;\; I_d[x] = 0 \;\; daca \;\; x \geq ||I_d||$$

4. CALLDATASIZE, dimensiunea inputului în mediul curent

$$\mu'_s[0] \equiv ||I_d||$$

5. GASPRICE, preţul gasului în mediul curent (este specificat în tranzactia originară)

$$\mu'_s[0] \equiv I_p$$

3.5 Modelarea execuţiei operaţiilor pe stivă, memorie, storage şi a operaţiilor de flow

1. POP, elimină un element de pe stivă
2. MLOAD, încarcă un word din memorie

$$\mu'_s[0] \equiv \mu_m[\mu_s[0]...(\mu_s[0]+31)]$$

3. MSTORE, salvează un word în memorie

$$\mu'_m[\mu_s[0]...(\mu_s[0]+31)] \equiv \mu_s[1]$$

4. MSTORE8, salvează un byte în memorie

$$\mu'_m[\mu_s[0]] \equiv (\mu_s[1] \ \ mod \ \ 256)$$

5. SLOAD, încarcă un word din storage

$$\mu'_m[\mu_s[0]] \equiv \sigma[I_a]_s[\mu_s[0]]$$

6. SSTORE, salvează un word în storage

$$\sigma'[I_a]_s[\mu_s[0]] \equiv \mu_s[1]$$

7. JUMP, modifică contorul program (sare la o locaţie precizată). Are ca efect modificarea valorii μ_{pc}.

$$J_{JUMP}(\mu) \equiv \mu_s[0]$$

8. JUMPI, modifică contorul program (sare la o locaţie precizată după ce se verifică o anumită condiţie). Are ca efect modificarea valorii μ_{pc}.

$$\mu'_s[0] \equiv \begin{cases} \mu_s[0] & if \ \ \mu_s[1] \neq 0, \\ \mu_{pc} + 1 & otherwise \end{cases}$$

9. PC, valoarea contorului program

$$\mu'_s[0] \equiv \mu_{pc}$$

10. GAS, gasul disponibil

$$\mu'_s[0] \equiv \mu_g$$

11. JUMPDEST, marchează o destinaţie ca fiind validă pentru jumpuri. Această instrucţiune nu are niciun efect asupra stării maşinii pe durata execuţiei.

12. CREATE, creează un nou cont. Pune pe stivă adresa contului creat.

13. KECCAK256, calculează funcţia hash Keccak-256 în funcţie de ultimele 2 valori de pe stivă. În implementarea de faţă am prezentat o variantă simplificată.

3.6 Modelarea execuţiei operaţiilor de PUSH, SWAP şi DUPLICATE

1. PUSH1 *item*, plasează un byte pe stivă

$$\mu'_s[0] \equiv item$$

2. PUSH2 *item*, plasează un număr pe 2 bytes pe stivă

$$\mu'_s[0] \equiv item$$

...

3. PUSH32 *item*, plasează un număr pe 32 bytes (full word) pe stivă

$$\mu'_s[0] \equiv item$$

Observaţie: pentru simplificarea operaţiilor nu am mai tratat aparte fiecare PUSH, deoarece bytecode-ul generat din Solidity va preciza exact instrucţiunea (ex: PUSH1 2, PUSH2 300 etc).

4. DUP1, duplică primul item de pe stivă

$$\mu'_s[0] \equiv \mu_s[0]$$

5. DUP2, duplică al doilea item de pe stivă

$$\mu'_s[0] \equiv \mu_s[1]$$

...

6. DUP16, duplică al 16-lea item de pe stivă

$$\mu'_s[0] \equiv \mu_s[15]$$

7. SWAP1, interschimbă primul cu al doilea item de pe stivă

$$\mu'_s[0] \equiv \mu_s[1]$$

$$mu'_s[1] \equiv \mu_s[0]$$

8. SWAP2, interschimbă primul cu al treilea item de pe stivă

$$\mu'_s[0] \equiv \mu_s[2]$$

$$mu'_s[2] \equiv \mu_s[0]$$

...

9. SWAP16, interschimbă primul cu al 17-lea item de pe stivă

$$\mu'_s[0] \equiv \mu_s[16]$$

$$mu'_s[16] \equiv \mu_s[0]$$

Experimente si evaluarea solutiei

Înainte de a putea experimenta diverse programe a trebuit să modelez inputul (bytecode-ul) pentru a putea fi în forma acceptată de frameworkul Coq.

Aplicaţia web Remix îmi punea la dispoziţie bytecodeul şi lista de instrucţiuni, care au trebuit identificate şi separate prin *::* (semantica Coq), iar apoi, pentru fiecare program în parte a trebuit să analizez bytecode-ul în paralel cu codul assembly pentru a putea corecta valorile de la instrucţiunea de JUMP (dificultatea pe care am întâlnit-o a fost că program counterul nu îşi modifică valoarea conform aşteptărilor - primeam o adresă de jump eronată, astfel a trebuit să verific manual fiecare jump şi să mă asigur că se face la adresa corectă).
După corectarea adreselor de jump am prelucrat inputul folosind un script Python, care genera bytecodeul conform cu semantica Coq.

Specificaţiile sistemului pe care au fost rulate experimentele: Procesor Intel i7 7700HQ, 4 nuclee, 8GB DDR4 memorie.

4.1 Experimente realizate

1. Operaţii aritmetice

```
contract Ballot {
    function foo(){
        int a = 5;
        a += 10;
    }
}
```

JUMPDEST	function f(){\n int a =...
PUSH1 0x0	int a
PUSH1 0x5	5
SWAP1	int a = 5
POP	int a = 5
PUSH1 0xA	10
DUP2	a += 10
ADD	a += 10

SWAP1	a += 10
POP	a += 10
POP	function f(){\n int a =...

Secvenţa de bytecode prezentată mai sus este doar apelul funcţiei *foo* din cadrul contractului *Ballot*. Am extras doar această porţiune pentru a evidenţia şirul de bytecodes generaţi de cele 2 instrucţiuni din funcţia *foo*.

```
myEnv (myState 9990 10 (fun _: Z => 0%Z) (fun _ : Z => 0%Z) 0
(myStack nil)) 100
(myI 673 2356 7834 (23 :: 12 :: nil) 6745 5362
(PUSH1 0
:: PUSH1 5
:: SWAP1 :: POP :: PUSH1 10 :: DUP2 :: ADD :: SWAP1 :: POP :: POP :: nil)
nil 674 35643)
: Environment
```

Se observă că stiva este vidă. Acest lucru s-a întâmplat din cauza faptului că am făcut POP valorii variabilei *a*. Făcând abstracţie de ultimul POP, se observă mai jos că în stivă avem valoarea corectă a variabilei *a*.

```
myEnv (myState 9991 9 (fun _: Z => 0%Z) (fun _ : Z => 0%Z) 0
(myStack (15%Z :: nil))) 100
(myI 673 2356 7834 (23 :: 12 :: nil) 6745 5362
(PUSH1 0
:: PUSH1 5
:: SWAP1 :: POP :: PUSH1 10 :: DUP2 :: ADD :: SWAP1 :: POP :: nil)
nil 674 35643)
: Environment
```

Aceasta a fost prima metodă cu ajutorul căreia verificam iniţial programele: verificam elementele de pe stivă.

2. Operaţii cu storage

Referindu-ne la exemplul anterior, vom salva valoarea obţinută în poziţia 0 din storage.

JUMPDEST	function f(){\n int a =...
PUSH1 0x0	int a
PUSH1 0x5	5
SWAP1	int a = 5
POP	int a = 5

```
PUSH1 0xA          10
DUP2               a += 10
ADD                a += 10
SWAP1              a += 10
POP                a += 10
PUSH1 0x0
SSTORE
```

Evident, verificările iniţiale ale corectitudinii le făceam interogând storage-ul şi comparând valorile. Ulterior am încercat să formalizez analiza proprietăţilor.
În cazul de faţă ne interesează să demonstrăm formal că în poziţia 0 din storage avem valoarea 15.

```
Lemma expected_value_at_address_0:
    forall E state g I g' pc m s i stack,
       runpgm initialEnv 12 = E ->
          E = myEnv state g I ->
             state = myState g' pc m s i stack ->
                s 0 = 15


Proof.
    intros.
    simpl in *.
    subst.
    inversion H0.
    unfold updateStorage.
    simpl.
    reflexivity.
Qed.
```

(a) simpl : Când avem de-a face cu un termen complex folosim *simpl*, pentru a-l "simplifica". Spre exemplu, dacă avem **add 6 2**, aplicând *simpl* obţinem 8. Se foloseşte pentru a simplifica expresiile.

(b) subst : Când avem un identificator care este egal cu o expresie putem folosi *subst* pentru a substitui identificatorul cu acea expresie.

(c) inversion : Uneori avem o ipoteză care nu poate fi adevarată pana nu demonstrăm că alte lucruri sunt adevărate. Putem folosi *inversion* pentru a descoperi ce alte condiţii trebuie să fie îndeplinite şi adevărate pentru ca ipoteza iniţiala să fie adevărată.

(d) unfold : Se foloseşte atunci când avem nevoie să apelăm la o definiţie (în cazul de faţă *updateStorage*).

(e) reflexivity : Se foloseşte când avem de demonstrat că un lucru este egal cu el însuşi, finalizând demonstraţia.

În figura de mai jos este arătat statusul execuţiei demonstraţiei înainte de a se executa *reflexivity*.

3. Număr prim:

```
pragma solidity ^0.4.21;
contract Ballot {
    function is_prime(int n) public returns(int){
        for(int i = 2; i < n/2; i++)
            if(n%i == 0)
                return 0;
        return 1;
    }
    int result = is_prime(17);
}
```

În programul de mai sus avem funcţia *is_prime* din cadrul contractului *Ballot*, care primeşte ca parametru un număr întreg şi returnează 1 dacă numărul este prim şi 0 altfel. Experimentul a constat prin a verifica valoarea variabilei *result*, aflată la poziţia 0 în storage. A se vedea rezultatul experimentului în figura **6.1**.

```
Lemma function_result_at_address_0:
    forall E state g I g' pc m s i stack,
        runpgm initialEnv 272 = E ->
            E = myEnv state g I ->
                state = myState g' pc m s i stack ->
                    s 0 = 1
```

4. Factorial

```
pragma solidity ^0.4.21;
contract Ballot{
    function factorial_recursive(int n) public returns(int){
        if(n==0 || n==1)
            return 1;
        return n*factorial_recursive(n-1);
    }
    int product = factorial_recursive(5);
}
```

În programul de mai sus avem funcţia *factorial_recursive* din cadrul contractului *Ballot*, care calculează în mod recursiv factorialul unui număr dat ca parametru. În cazul nostru parametrul este 5, deci ne aşteptăm să avem 5! (120) în locaţia 0 in storage. A se vedea rezultatul experimentului în figura **6.2**.

```
Lemma function_result_at_address_0:
```

```
    forall E state g I g' pc m s i stack,
      runpgm initialEnv 206 = E ->
        E = myEnv state g I ->
          state = myState g' pc m s i stack ->
            s 0 = 120
```

5. Structuri de date (I)

```
pragma solidity ^0.4.21;
contract Ballot {
    struct Test{
        int a;
        int b;
        int c;
    }
    function struct_test(int n) public returns (int){
        Test t1;
        t1.a = n;
        t1.b = n*2;
        t1.c = n*3;
        return t1.a + t1.b + t1.c;
    }
    int result = struct_test(10);
}
```

Programul de mai sus este puţin mai complex, întrucât am introdus o structură de date, *Test*. După apelul funcţiei *struct_text* cu parametrul 10, ne aşteptăm să fie creată o structură *Test*, cu câmpurile a=10, b=20, c=30 şi să fie returnată suma $a+b+c$, adică 10+20+30=60. Acest rezultat îl stocăm în variabila *result*. Ne aşteptăm deci să avem în storage la adresa 0 valoarea 60. A se vedea rezultatul experimentului în figura **6.3**.

```
Lemma function_result_at_address_0:
    forall E state g I g' pc m s i stack,
      runpgm initialEnv 87 = E ->
        E = myEnv state g I ->
          state = myState g' pc m s i stack ->
            s 0 = 60
```

6. Obiecte

```
pragma solidity ^0.4.21;
```

```
contract A{
    function sum(int a, int b) public returns(int){
        return a+b;
    }
}

contract Ballot{
    A a = new A();
}
```

În exemplul de mai sus am creat două clase, dintre care *Ballot* este cea principală. După rularea contractului, ne aşteptăm să avem în storage la pozitţia 0 adresa noului cont creat după execuţia instrucţiunii *CREATE*. Ţinând cont că toate testele sunt făcute local instrucţiunea *CREATE* va genera o adresă fictivă, hardcodată ca 4007355.

```
Lemma expected_account_address_at_address_0:
    forall E state g I g' pc m s i stack,
        runpgm initialEnv 87 = E ->
            E = myEnv state g I ->
                state = myState g' pc m s i stack ->
                    s 0 = 4007355
```

Pentru toate programele de mai sus, secvenţa care testează condiţiile este aceiaşi.

```
Proof.
    intros.
    simpl in *.
    subst.
    inversion H0.
    unfold updateStorage.
    simpl.
    reflexivity.
Qed.
```

7. Structuri de date (II)

```
pragma solidity ^0.4.21;
pragma experimental ABIEncoderV2;
contract Ballot {
    struct Test{
        int a;
```

```
        int b;
        int c;
    }
    Test test_parameter = Test({a:13, b:223, c:454});
    function struct_test(Test test) public returns (Test){
        test.a = test.a * 2;
            return test;
    }
    Test result = struct_test(test_parameter);
}
```

În exemplul de mai sus am folosit de asemenea o structură de date, *Test*, pe care am dat-o ca parametru funcţiei *struct_test*. Creăm prima structură, *test_parameter*, care va ocupa primele 3 câmpuri din storage, apoi creăm cea de-a doua structură, *result*, care va fi o copie a primei structuri, doar că valoarea câmpului *a* va fi dublată. Astfel, ne aşteptăm la următoarea structură o storage-ului:
{(0, 13), (1, 223), (2, 454), (3, 26), (4, 223), (5, 454)}.

```
Lemma conjunction_property:
    forall E state g I g' pc m s i stack,
      runpgm initialEnv 9 = E ->
        E = myEnv state g I ->
            state = myState g' pc m s i stack ->
                s 0%Z = 13%Z /\ s 1%Z = 223%Z /\ s 2%Z = 454%Z /\
                s 3%Z = 26%Z /\ s 4%Z = 223%Z /\ s 5%Z = 454%Z.


Proof.
    intros.
    simpl in *.
    subst.
    inversion H0.
    unfold updateStorage.
    simpl.

    split.
    reflexivity.

    split.
```

```
    reflexivity.

    split.
    reflexivity.

    split.
    reflexivity.

    split.
    reflexivity.

    reflexivity.
Qed.
```

Atunci când avem de demonstrat o conjuncţie cu mai mult de un predicat vom împărţi conjuncţia folosind split până ajungem la un singur predicat.

8. Balances

```
pragma solidity ^0.4.21;
contract Ballot{
    mapping (address => uint) balances;
    function updateBalances() public returns(uint){
        balances[0] = 16;
        balances[1] = 56;
        return balances[0];
    }
    uint value = updateBalances();
}
```

De această dată am testat o structură de date mai complexă, şi anume *mapping (address => uint)*, în care avem funcţia *updateBalances()*, care inserează valori pentru adresele 0 şi 1 şi returnează fondurile pentru adresa 0. Această valoare este în storage la adresa 1 (variabila *value*).

9. Variabila *msg.sender*

```
pragma solidity ^0.4.21;
contract Ballot{
    address owner;
    function Ballot(){
        owner = msg.sender;
    }
```

}

4.2 Concluzii

În urma experimentelor realizate am arătat că se pot demonstra anumite proprietăţi ale unor programe, pe baza semanticii create pentru EVM.

În ceea ce priveşte procesul de rulare a programului ce verifică proprietăţile unui contract, am observat un lucru constant, mai exact memoria folosită este între 500 si 600 MB. De asemenea, durata de execuţie variază în funcţie de numărul instrucţiunilor:

Index	**Program**	**Instrucţiuni**	**Timp**
1	Suma a două numere	52	5h
2	Număr prim	272	11h
3	Inversul unui număr	201	8.5h
4	Counter	136	7h
5	Structuri de date (I)	87	6h
6	Factorial recursiv	206	9h
7	Palidrom	256	12h
8	Obiecte (I)	70	5h
9	GCD recursiv	238	15h
10	Fibonnacci recursiv	464	$> 40h$

Concluzii

5.1 Rezumat

Smart contractele conţin cod şi memorie, pot interacţiona cu alte contracte şi pot trimite şi primi "ether" la şi de la alte conturi. Sunt imutabile, deci trebuie gândite foarte bine din start. Orice greşeala creează oportunităţi pentru răuvoitori. De aceea atunci când i se face deploy unui smart contract trebuie să ne asiguram că respectă anumite proprietăţi.

Pentru a verifica aceste proprietăţi vrem să analizăm codul la cel mai de jos nivel, adică la nivel de bytecode.

Am prezentat o semantică a EVM-ului, sunt prezentate bucăţile de semantică implementate, am experimentat pe nişte programe şi am reuşit să fac nişte demonstraţii.

Am peste 20 de programe rulate, care validează şi întăresc corectitudinea semanticii create în Coq. Acestea reprezintă o temelie pentru a avansa la următorul nivel, mai exact pentru a valida proprietăţi în contracte mult mai complexe.

5.2 Îmbunătăţiri

1. Automatizarea prelucrării bytecode-ului fără intervenţia manuală asupra modificării poziţiilor pentru jump.

2. Îmbunătăţirea timpului de execuţie; trebuie micşorat mult mai mult, întrucât un smart contract real ajunge la peste 1000 de instrucţiuni, ceea ce, după statisticile actuale, ar însemna că validarea lor să ajunga la câteva săptămâni.

3. Acceptarea contractelor mult mai complexe. Acest lucru va fi făcut consolidând baza deja existentă, implementând instrucţiunile mai detaliat. Astfel, se va putea analiza overflowul sau alte programe cu buguri.

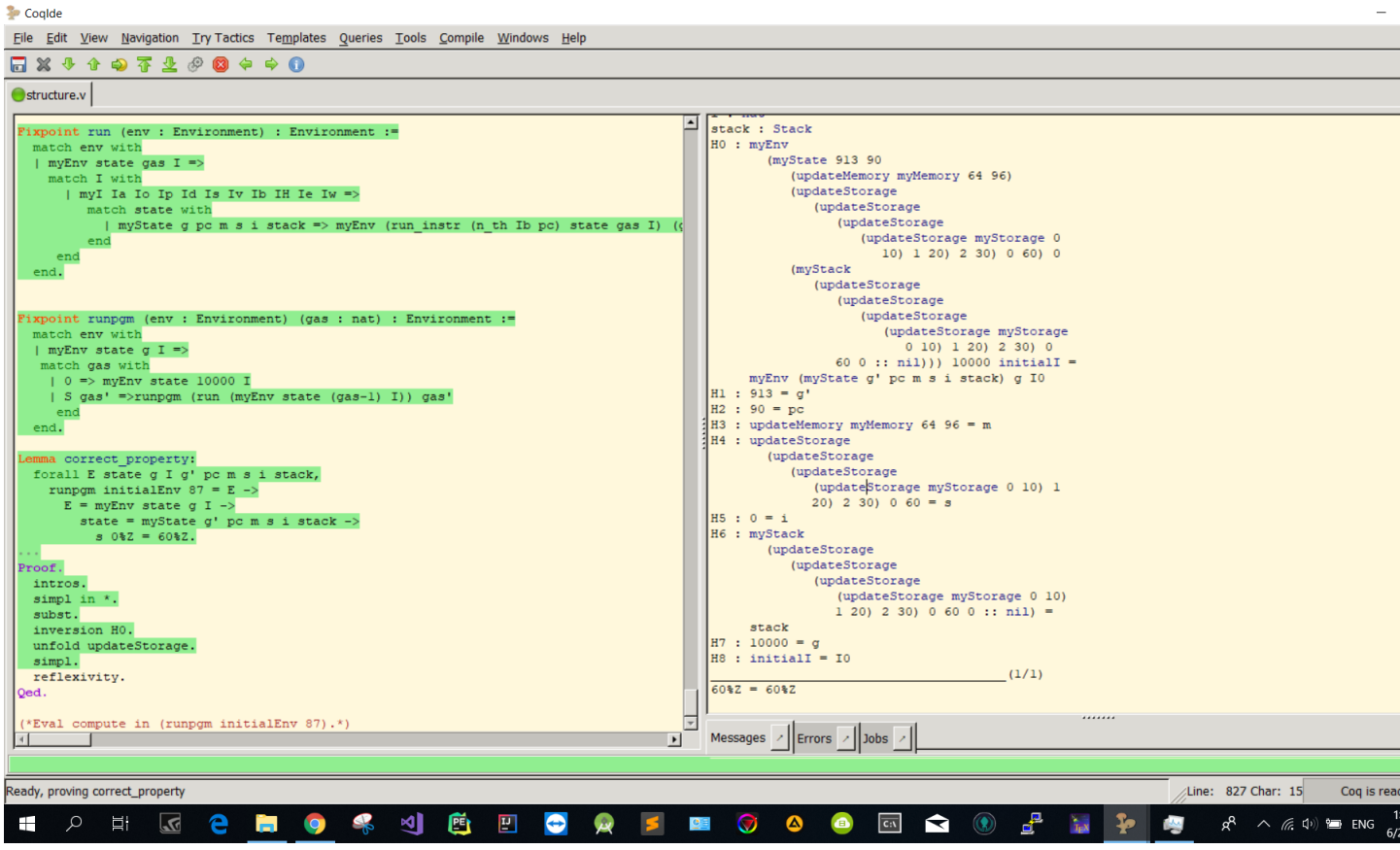
Appendix

```
1 subgoal
g : nat
I0 : I
g', pc : nat
m : memory
s : storage
i : nat
stack : Stack
H0 : myEnv (myState 728 96 (updateMemory myMemory 64 96) (updateStorage myStorage 0 1) 0 (myStack nil))
       10000 initialI = myEnv (myState g' pc m s i stack) g I0
H1 : 728 = g'
H2 : 96 = pc
H3 : updateMemory myMemory 64 96 = m
H4 : updateStorage myStorage 0 1 = s
H5 : 0 = i
H6 : myStack nil = stack
H7 : 10000 = g
H8 : initialI = I0
______________________________________(1/1)
1%Z = 1%Z
```

6.1 Număr prim

```
1 subgoal
g : nat
I0 : I
g', pc : nat
m : memory
s : storage
i : nat
stack : Stack
H0 : myEnv
       (myState 794 81 (updateMemory myMemory 64 96) (updateStorage myStorage 0 120) 0 (myStack nil))
       10000 initialI = myEnv (myState g' pc m s i stack) g I0
H1 : 794 = g'
H2 : 81 = pc
H3 : updateMemory myMemory 64 96 = m
H4 : updateStorage myStorage 0 120 = s
H5 : 0 = i
H6 : myStack nil = stack
H7 : 10000 = g
H8 : initialI = I0
______________________________________(1/1)
120%Z = 120%Z
```

6.2 Factorial recursiv



6.3 Structuri de date (I)

Bibliografie

1. Documentaţie Coq: https://coq.inria.fr/refman/
2. Jello Paper: https://jellopaper.org/
3. Yellow Paper: https://github.com/ethereum/yellowpaper
4. Documentatie Solidity: https://solidity.readthedocs.io/en/v0.4.24/
5. Formal Verification of Smart Contracts (F*): https://www.cs.umd.edu/~aseem/solidetherplas.pdf
6. KEVM: Semantics of EVM in K: https://github.com/kframework/evm-semantics
7. Formalization of Ethereum Virtual Machine in Lem : https://github.com/pirapira/eth-isabelle
8. Formal Verification of Ethereum Contracts (Yoichi's attempts) : https://github.com/pirapira/ethereum-formal-v