HAW HAMBURG

Bachelorarbeit

Assiel Taher

Verwaltung von Sicherheitszertifikaten für IoT-Geräte in einem Policy-basiertem-Access Control Framework auf Basis von Blockchain-Technologien

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Assiel Taher

Verwaltung von Sicherheitszertifikaten für IoT-Geräte in einem Policy-basiertem-Access Control Framework auf Basis von Blockchain-Technologien

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 12. April 2019

Assiel Taher

Thema der Arbeit

Verwaltung von Sicherheitszertifikaten für IoT-Geräte in einem Policy-basiertem-Access Control Framework auf Basis von Blockchain-Technologien

Stichworte

Blockchain, Internet der Dinge, Zugriffskontrolle, Ethereum, Sicherheitszertifikat

Kurzzusammenfassung

In den letzten Jahren hat das Internet of Things immer mehr an Aufmerksamkeit gewonnen. Nach Gartner, Inc. sollen bis 2020 weltweit 20.4 Milliarden „Dinge“ vernetzt sein. Dadurch stehen IoT-Systeme ständig vor neuen Herausforderungen in Bezug auf die Sicherheit. Immer wieder werden Angriffe auf IoT-Geräte beobachtet. Ein Mechanismus zum Schutz gegen Angriffe bieten Zugriffskontrollen. Zugriffskontrollen ermöglichen es den Zugriff anhand von Policies zu kontrollieren. Wegen der Ressourcenbeschränktheit von IoT-Geräten, haben sich Blockchain-basierte Zugriffskontrollen entwickelt, um die Sicherheit für die Geräte zu übernehmen. In der vorliegenden Arbeit wird ein System vorgestellt, dass Blockchain-basierte Zugangskontrollen sicherer macht, indem der Zustand von Geräten, basierend auf Sicherheitszertifikaten, mit in die Entscheidung des Zugangskontrollmechanismus einfließt. Dafür werden die Zertifikate sicher in der Blockchain gespeichert, sodass diese nicht manipuliert werden können und immer verfügbar sind.

Assiel Taher

Title of Thesis

Management of securtiy certificates for IoT devices in a Policy-Based-Access-Conrol framework based on blockchain technology

Keywords

Blockchain, Internet of Things, Access Control, Ethereum, security certificate

Abstract

The Internet of Things is becoming more popular. Garter, Inc. says there will be 20.4 billion connected devices in 2020. Therefore the IoT will be confronted with new security challenges and issues. Many attacks to compromise the security of devices have been observed. One way to prevent those attacks compromising the IoT devices comes with the use of access control systems. Access control systems restrict the access rights of requests based on policies. Due to the constrained nature of IoT devices, blockchain-based access control frameworks have been proposed recently. This document presents a system where compromised devices are detected by the access control mechanism. For this, security certificates indicating the security status of a device are stored in a blockchain preventing anyone from manipulating it. Additionally the security certificates are always available.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Technologie heutzutage ermöglicht es, die kleinsten und alltäglichsten „Dinge“ mit dem Interet zu verbinden. Die Stadt wird dadurch wachsamer und die reale und digitale Welt sind eng miteinander verbunden. Nach Gartner, Inc. sollen bis 2020 weltweit 20.4 Milliarden Geräte vernetzt sein. Die Geräte erleichtern zwar vielen das Leben, aber gleichzeitig greifen sie tief in unsere Privatsphäre ein. So kennt das smarte Auto immer unsere Position, das Alarmsystem im Haus weiß immer, wann wir außer Haus sind oder Gesundheitssysteme kennen unseren gesundheitlichen Zustand. Für Hacker bietet das IoT somit eine große Angriffsfläche an. So können z.B. private Daten ausgelesen werden, Daten manipuliert oder auch Denial-of-Service Angriffe gestartet werden. Gerade sicherheitskritische IoT-Anwendungen, wie E-Health oder automonomes Fahren, dürfen die Sicherheit ihrer IoT-Geräte nicht als „luxus“ ansehen. Kompromittierte Sensoren eines autonomen Autos, die nicht richtig funktionieren oder falsche Daten erzeugen, könnten schwerwiegende Folgen auch für die reale Welt haben. Eine Möglichkeit, um die Ressourcen vor unauthorisierten Zugriffen zu schützen, sind Zugriffskontrollen. Zugriffskontrollen kontrollieren anhand von Policies, ob ein Zugriff erlaubt werden soll oder nicht. Da die IoT-Geräte selbst, aufgrund ihrer ressourcenbeschränkten Natur, keine zuverlässigen Zugriffskontrollmechanismen umsetzen können, wird diese Funktionalität oftmals von Drittanbietern übernommen. Doch der Einsatz einer zentralen Instanz, die die Zugriffskontrolle übernimmt, erfordert ein beidseitiges Vertrauen. Mit der Zeit haben sich deshalb dezentrale Zugriffskontrollen entwickelt, die die Blockchain als Technologie nutzen. Damit ist es möglich, eine transparente und ohne auf Vertrauen basierte Zugriffskontrolle zu realisieren. Die meisten dieser Systeme nutzen für die Policies, die Informationen der Kommunikation selbst, z.B. die Identität des Anfragers, oder auch Informationen über die Umgebung selbst, z.B. der Ort oder der Zeitpunkt des Zugriffversuchs. Wie der Zustand des Gerätes selber ist, also ob dieser kompromittiert ist oder Sicherheitslücken

aufweist, wird nicht geprüft. Deshalb sollten diese Zugriffskontrollen erweitert werden, sodass auch der Sicherheitszustand von Geräten geprüft wird. Dafür werden Sicherheitszertifikate von Sicherheitsprüfern ausgestellt, die das Gerät geprüft haben. Solche Zertifikate kann man in den Entscheidungsprozess einer Zugriffskontrolle einfließen lassen, um die Sicherheit des anfragenden Gerätes zu gewährleisten. Doch die Zertifikate müssen vor Manipulationen geschützt werden. Die Verwaltung der Zertifikate und dessen Einsatz in Zugriffskontrollmechanismen kann durch die kryptographischen und dezentralen Eigenschaften der Blockchain implementiert werden. Dadurch sind die Zertifikate geschützt vor Veränderungen und sind immer verfügbar. Überlässt man die Verwaltung der Zertifikate einem Programm, dass dezentral auf einem Blockchain-Netzwerk läuft, erhält man einen kompletten dezentralen Zuganskontrollmechanismus, der von keiner einzelnen Instanz beeinflusst werden kann.

1.2 Zielsetzung

In dieser Arbeit soll ein Konzept entwickelt werden, das beschreibt, wie Sicherheitszertifikate mithilfe der Blockchain Technologie sicher verwaltet werden können. Desweiteren soll ein Zugangskontrollframework so erweitert werden, dass Policies definiert werden können, die anhand der Sicherheitszertifikate den Zugriff erlauben oder nicht. Das System soll so entwickelt werden, dass es geschützt ist vor Angriffen und Manipulationen der Zertifikate. Zum Überprüfen der Ergebnisse, soll das System auf einem privaten Ethereum-Netzwerk deployt werden.

1.3 Zielgruppe

Diese Arbeit setzt grundlegende Kenntnisse der Informatik voraus und ist deshalb für alle Informatiker geeignet. Da in dieser Arbeit mit Blockchains gearbeitet wird, einer neuen aber aufstrebenden Technologie, ist sie vorallem für Informatik-Studenten der höheren Semester geeignet, die eventuell ebenfalls eine Bachelorarbeit im Bereich der Blockchains schreiben wollen. Generell liegt der Fokus dieser Arbeit auf IT-Sicherheit, weshalb die Hauptzielgruppe alle Interessenten der IT-Sicherheit sind und diejenigen die sich schon mit IT-Sicherheit oder Blockchain-Techonologie beschäftigt haben.

1.4 Struktur der Arbeit

In Kapitel zwei werden die Grundlagen für die Arbeit vorgestellt. Zunächst wird eine Einführung in die Blockchain-Technologie gegeben. Dafür werden die kryptographischen Eigenschaften erklärt und wie die Blockchain funktioniert. Anschließend wird speziell die Ethereum Blockchain erklärt und welche Eigenschaften sie hat. Nach der Blockchaineinführung wird es einen Einblick in das Internet-of-Things geben. Fokussiert wird dabei auf die Herausforderungen für die Sicherheit und der damit verbundenen Wichtigkeit für den Einsatz von Zugriffskontrollen. Zum Schluss werden drei Blockchain-basierte Zugriffskontrollen vorgestellt und verglichen, wobei einer für die weitere Arbeit identifiziert wird.

In Kapitel 3 wird das Problem genauer erklärt, dass mit dieser Arbeit gelöst werden soll. Anhand von einem Anwendungsfall, der das Problem beschreibt, werden dann Anforderungen an das zu erschaffende System hergeleitet.

Kapitel 4 stellt das Konzept vor, dass das Problem aus Kapitel 3 lösen soll. Hier wird beschrieben wie das System auszusehen hat und wie es kompatibel ist mit dem Zugangskontrollframework.

In Kapitel 5 wird die konkrete Umsetzung des Systems aufgezeigt. Dabei werden die Entscheidungen der Umsetzung geklärt und wie sie in der Entwicklungsumgebung anzuwenden sind.

2 Grundlagen

2.1 Blockchain

In diesem Kapitel werden die Grundlegenden Konzepte einer Blockchain erläutert. Dabei wird auf die Wichtigkeit der Kryptographie und Dezentralität eingegangen. Zur Erklärung der Funktionsweisen eines Blockchain-Systems wird die Bitcoin-Blockchain als Beispiel genommen.

Im November 2008 erscheint das White Paper Nakamoto [2009], welches die Einführung eines dezentralen digitalen Währungssystem - Bitcoin - beschreibt [Mattila, 2016]. Die Blockchain ist die Technologie hinter diesem Währungssystem [Bogart und Rice, 2015]. Eine offizielle Definition der Blockchain gibt es derzeit nicht. Während einige den Begriff Blockchain nur im Kontext von Bitcoin für richtig halten, gibt es andere, die Bitcoin und Blockchain getrennt sehen [Mattila, 2016].

In Condos u. a. [2016] wird die Blockchain definiert als ein elektronisches Kontenbuch für digitale Datensätze, Ereignisse oder Transaktionen, die kryptographisch gehasht und authentifiziert werden. Es zeichnet sich durch ein verteiltes Netzwerk und einen Konsens-Mechanismus aus.

Walport [2016] definiert die Blockchain als eine Art Datenbank, die Datensätze in Blöcken sammelt. Jeder Block ist dabei mit seinem Nachfolger über kryptographische Signaturen verknüpft. Jeder Block kann über seinen Hashwert eindeutig identifiziert werden. Ein Block enthält immer den Hash seines Vorgängerblocks.

Die Bitcoin Blockchain, als Beispiel, enthält Informationen über alle getätigten Transaktionen innerhalb des Bitcoin-Netzwerks. Diese sind für **alle** einsehbar. Transaktionen, die erfolgreich in die Blockchain eingefügt und von allen Teilnehmern bestätigt worden sind, können **nicht** rückgängig gemacht oder verändert werden [Bogart und Rice, 2015]. Die Bitcoin Blockchain wird nicht zentral verwaltet, sondern von allen Teilnehmern des

Netzwerks [Yaga u. a., 2018]. Teilnehmer[1] sind diejenigen, die eine bestimmte Client-Software laufen lassen, welche mit anderen Clients im Netzwerk kommuniziert. Um neue Transaktionen zu verfizieren und sie in die Blockchain einzufügen, müssen die Nodes Rechenleistung bereitstellen, um ein kryptographisches Puzzle zu lösen. Diese Nodes werden auch *Miner* genannt. Das ist der Konsensmechanismus, durch den die Blockchain für alle Nodes den gleichen Zustand hat. Im Bitcoin-Netzwerk wird etwa alle zehn Minuten das Puzzle gelöst, indem offene Transaktionen gesammelt werden und daraus ein valider Block erzeugt wird, der den Kriterien des Puzzles entspricht. Der Miner, der als erstes das Puzzle gelöst hat, wird mit Bitcoins belohnt [Glaser und Bezzenberger, 2015]. Dadurch haben die Nodes einen Anreiz neue Blöcke zu „schürfen“. Zudem können die Miner nicht schummeln, denn der neue Block muss von allen anderen Nodes akzeptiert werden. Daher lohnt es sich nicht Rechenleistung aufzubringen, um einen falschen Block bekannt zu machen. Die Blockchain, also die gesamte Datenstruktur mit allen Blöcken, wird von jedem Teilnehmer mitgeführt und gespeichert.

2.1.1 Kryptographie

In Bitcoin kommen kryptographische Verfahren in 2 Formen zum Einsatz, als Hashfunktionen und als Digitale Signaturen. Im Folgenden wird das Konzept der Asymmetrischen Verschlüsselung erklärt, welches Digitale Signaturen ermöglicht. Anschließend werden Hashfunktionen und Digitale Signaturen erläutert.

Public-Key-Kryptographie

Das Konzept, das Digitale Signaturen ermöglicht ist die Public-Key-Kryptographie, auch Asymmetrische Verschlüsselung genannt. Bei einer Asymmetrischen Verschlüsselung wird ein Schlüsselpaar erzeugt, jeweils einen Private-Key und einen Public-Key. Zwischen Private-Key und Public-Key gibt es eine mathematische Beziehung, die es erlaubt einen mit Public-Key verschlüsselten Text über den Private-Key zu entschlüsseln. Umgekehrt kann auch ein mit Private-Key verschlüsselter Text über dessen zugehörigen Public-Key entschlüsselt werden. Während der Public-Key allen bekannt ist, bleibt der Private-Key nur dem Besitzer des Schlüsselpaars bekannt. Durch dieses Schlüsselpaar können nun drei Ziele erreicht werden:

[1] Diese werden auch Nodes genannt.

Authentizität

Der Absender einer Nachricht verschlüsselt diese mit seinem Private-Key. Der Empfänger nutzt den Public-Key des Absenders, um die Nachricht zu entschlüsseln. Da der Public-Key nur den Text entschlüsseln kann, der zuvor mit dem zugehörigen Private-Key verschlüsselt wurde, muss die Nachricht tatsächlich vom Absender stammen.

Integrität

Da die Nachricht mit dem Private-Key verschlüsselt wird und dieser nur dem Absender bekannt ist, kann die Nachricht nicht verändert werden.

Vertraulichkeit

Der Absender einer Nachricht verschlüsselt diese mit dem Public-Key des Empfängers. Diese Nachricht kann nur mit dem zugehörigen Private-Key entschlüsselt werden und sofern dieser nur dem Empfänger bekannt ist, ist die Vertraulichkeit gewährleistet.

Hashfunktionen

Ein weiteres kryptographisches Verfahren, welches in der Blockchain zum Einsatz kommt, ist die Hashfunktion. Eine Hashfunktion bildet einen Bereich, den Urbildbereich oder Universum genannt, auf einen Hashwert ab. Die Menge aller Hashwerte wird Adressbereich genannt [Eckert, 2014, S. 380]. Konkret nimmt die Funktion einen Text beliebiger Länge als Input und liefert als Output eine Zeichenfolge (Hashwert) fester Länge. Hashfunktionen sind deterministisch. Das bedeutet, dass die gleichen Eingabewerte immer die gleichen Ausgabewerte zur Folge haben. Außerdem führt die kleinste Änderung des Eingabetextes zu einen stark veränderten Hashwert [Badev und Chen, 2014]. Da der Adressbereich kleiner als das Universum ist, können Kollisionen auftreten. Das heißt zwei beliebige Texte werden auf den gleichen Hashwert abgebildet. Erforderlich ist also eine kollisionsresistente Funktion. Daher besitzt eine 'Stark Kollisionsresistente Hashfunktion die folgenden drei Eigenschaften [Eckert, 2014, S. 380 ff]:

1. Der Hashwert einer beliebigen Eingabe ist leicht zu berechnen. Umgekehrt darf aus einem Hashwert nicht die Eingabe effizient zu berechnen sein (Einwegfunktion) [Eckert, 2014, S. 383 f].

2. Zu einem gegebenen Hashwert darf es nicht effizient möglich sein, eine weitere Eingabe zu finden, die den gleichen Hashwert hat [Eckert, 2014, S. 383 f].

3. Es darf in effizienter Zeit nicht möglich sein, zwei Eingabewerte zu finden, deren Hashwerte übereinstimmen [Eckert, 2014, S. 383 f].

Eingesetzt werden Hashfunktionen bei der Integritätskontrolle und dem Signieren von Nachrichten, um die Urheberschaft nachzuweisen [Eckert, 2014, S. 384]:

Integritätskontrolle

Diese Methode dient dem Aufdecken von Übertragungsfehlern, die bei der Nachrichtenübertragung auftreten können. Der Absender berechnet den Hash seiner Nachricht und verschickt beides an den Empfänger. Der Empfänger berechnet ebenfalls den Hash der erhaltenen Nachricht und vergleicht diese mit dem von dem Absender versendeten Hashwert. Gibt es Unterschiede zwischen den Werten, muss beim Versenden ein Fehler aufgetreten sein und die Nachricht entspricht nicht mehr der Originalform [Eckert, 2014, S. 384]. Da jedoch ein Angreifer die versendete Nachricht abfangen und sowohl die Nachricht selbst, als auch den Hash verändern kann, gibt es die Möglichkeit als Absender die Nachricht zu signieren, um sicherzustellen, dass kein Unbefugter Zugriff stattgefunden hat.

Signatur

Hashfunktionen werden oftmals im Zusammenhang mit Signaturen eingesetzt, um zusätzlich zur Prüfung der Integrität, auch die Urheberschaft nachweisen zu können. Der Absender berechnet den Hashwert seiner Nachricht und signiert diese, bevor er die Nachricht und den signierten Hash versendet. Für das Signieren wird in den meisten Fällen die Asymmetrische Verschlüsselung verwendet. Der Absender signiert den Hash mit seinem Private-Key und der Empfänger nutzt den zugehörigen Public-Key, um die Signatur zu bestätigen und den Hash mit seinem berechneten Hash zu vergleichen [Eckert, 2014, S. 384 f].

Digitale Signatur

Digitale Signaturen sind das elektronische Äquivalent zu handschriftlichen Unterschriften. Sie müssen also die gleichen **Funktionen** erfüllen [Eckert, 2014, S. 396 f]:

Identifikation

„Die Unterschrift gibt Auskunft über die Person des Unterzeichners.“ [Eckert, 2014, S. 397]

Echtheit

Das Dokument hat dem Unterzeichner vorgelegen und wurde von ihm anerkannt [Eckert, 2014, S. 397].

Abschluss

„Die Unterschrift erklärt den Text für inhaltlich richtig und vollständig.“ [Eckert, 2014, S. 397]

Warnung

Dem Unterzeichner wurde die rechtliche Bedeutung des Dokuments aufgezeigt [Eckert, 2014, S. 397].

Daraus resultieren einige **Anforderungen**, welche Digitale Signaturen erfüllen müssen:

- Sie bezeugt die Identität des Unterzeichners [Eckert, 2014, S. 397].
- Sie darf nicht unautorisiert wiederverwendet werden [Eckert, 2014, S. 397].
- Das signierte Dokument darf nicht veränderbar sein [Eckert, 2014, S. 397].
- Der Unterzeichner kann die Signatur nicht abstreiten [Eckert, 2014, S. 397].

Angenommen, Alice möchte eine Nachricht an Bob senden, dann werden die folgenden Schritte zur Erstellung einer signierten Nachricht durchgeführt:

1. Alice bildet den Hashwert der Nachricht.
2. Alice verschlüsselt den Hashwert mit ihrem Private-Key. Dies ist die Signatur.
3. Alice sendet die Nachricht zusammen mit der Signatur an Bob.
4. Bob entschlüsselt die Signatur mit dem Public-Key von Alice und erhält einen Hashwert.
5. Bob bildet den Hashwert der erhaltenen Nachricht und vergleicht diese mit dem Hashwert aus der Signatur.

Stimmen die im letzten Schritt verglichenen Hashwerte überein, kann Bob davon ausgehen, dass die Nachricht wirklich von Alice stammt und nicht verändert wurde, da der Private-Key, mit dem die Nachricht signiert wurde, nur Alice bekannt ist. In Abbildung 2.1 werden die einzelnen Schritte zur Erstellung und Verifikation Digitaler Signaturen nocheinmal veranschaulicht.

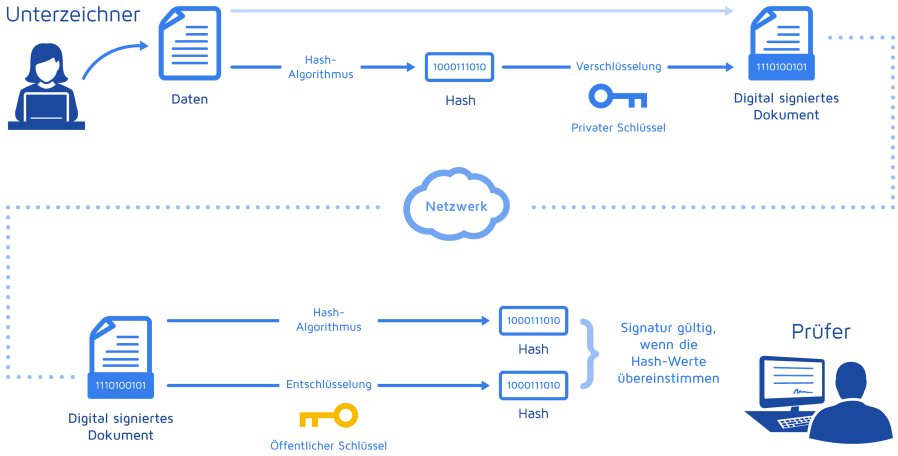


Abbildung 2.1: Erstellung und Verifikation Digitaler Signaturen[2]

Mit diesem Verfahren werden die vier Anforderungen an digitale Signaturen erfüllt:

Zweifelsfreie Identifikation

Da zum Entschlüsseln Alice' Public-Key benutzt wurde und nur sie den dazugehörigen Private-Key hat, mit dem die Nachricht signiert wurde, wird Alice als Urheberin der Nachricht bestätigt [Eckert, 2014, S. 399].

Keine Wiederverwendbarkeit

Die Signatur wurde in Abhängigkeit vom Hashwert der Nachricht erstellt und kann somit, aufgrund der Eigenschaften von Hashfunktionen, nicht auf andere Nachrichten übertragen werden [Eckert, 2014, S. 399].

Unveränderbarkeit

Die Signatur wurde mit dem Private-Key erstellt, der nur Alice bekannnt ist. Dadurch wird eine veränderte Nachricht bei der Verifikation auffallen [Eckert, 2014, S. 399]

Verbindlichkeit

Da der Private-Key nur Alice bekannt ist, kann sie die Signatur nicht abstreiten [Eckert, 2014, S. 399].

2.1.2 Aufbau einer Blockchain

Nach Antonopoulos und Klicman [2018] ist die Blockchain eine geordnete, rückwärts verlinkte Liste von Blöcken mit Transaktionen. Jeder Block ist über seinen Hash eindeutig identifizierbar und speichert den Hash des vorherigen Blocks, auch Parent-Block genannt. Somit ist in jedem Hash eines Blocks der Hash des Parent-Blocks enthalten. So entsteht eine kryptographisch gesicherte, verlinkte Kette von Blöcken. In Bitcoin besteht ein Block aus den folgenden Feldern:

Blockgröße (4 Bytes)
Gibt die größe des Blocks an, die auf dieses Feld folgt.

Block-Header (80 Bytes)
Enthält Metadaten.

Transaktionszähler (1-9 Bytes)
Anzahl an Transaktionen im Block.

Transaktionen
Die Liste der Transaktionen, die in diesem Block enthalten sind.

Eine Transaktion ist im Kontext der Blockchain nur eine Nachrichtenaktivität, die im Netzwerk stattgefunden hat. Je nach Blockchain-Anwendung, kann eine Transkation eine andere Form und andere Daten enthalten. In Abbildung 2.2 sieht man wie eine Transaktion in Bitcoin aussieht. Eine Transaktion der Bitcoin-Blockchain und der Ethereum-Blockchain unterscheiden sich deutlich. In Kapitel 2.2.2 sieht man wie die Transaktionen in Ethereum aussehen. Erfolgreiche Transaktionen werden in der Blockchain, jeweils in den Blöcken, zusammen mit anderen Transaktionen gespeichert.

Der Block-Header hat die folgenden Felder:

Version (4 Bytes)
Versionsnummer für Software-/Protokoll-Updates.

Vorheriger Block-Hash (32 Bytes)
Der Hash des Parent-Block-Headers.

[2] https://www.docusign.de/wie-es-funktioniert/elektronische-signatur/digitale-signatur/digitale-signatur-faq(06.03.2019)

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig" :
      "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad
      530a863ea8f53982c09db8f6e3813[ALL]
      0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a
      10fa336a8d752adf",
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```


Abbildung 2.2: Inhalt einer Transaktion in der Bitcoin-Blockchain

Merkle Root (32 Bytes)

Der Hash des Baums aller Transaktionen.

Timestamp (4 Bytes)

Ungefähre Erstellungszeit des Blocks.

Difficulty Target (4 Bytes)

Schwierigkeit des PoW Algorithmus' für diesen Block.

Nonce (4 Bytes)

Vom PoW Algorithmus verwendeter Zähler.

Der Hash eines Blocks wird nicht direkt im Block gespeichert. Lediglich der Hash seines Parent-Blocks ist in der Datenstruktur enthalten. Der Hash eines Blocks bezieht sich auch nicht auf den gesamten Block, sondern nur der Block-Header wird gehasht. Dies ist ausreichend, denn dieser enthält den Hash des vorherigen Blocks, was eine Verlinkung bis zum ersten Block, dem sogenannten Genesis-Block, zur Folge hat. Außerdem ist die Merkle Root enthalten, welche alle Transaktionen zusammenfasst.

2.1.3 Mining und Konsens

Das Mining ist die Grundlage für das Funktionieren einer Blockchain. Nur durch sie ist es möglich, in einem dezentralisierten Peer-to-Peer Netzwerk einen Konsens zu erreichen und auf eine zentrale Kontrollinstanz zu verzichten.

Der Konsens entsteht durch die Interaktion aller Nodes, die alle den gleichen Konsensregeln folgen [Antonopoulos und Klicman, 2018]. Nach [Antonopoulos und Klicman, 2018, S. 220] gibt es 4 Prozesse, die unabhängig voneinander auf den Nodes laufen und zum Konsens führen:

Verifikation jeder Transaktion

Wird eine Transaktion erstellt, muss diese an alle Nodes im Netzwerk gesendet werden. Erhält eine Node eine Transaktion, prüft sie diese anhand einer Checkliste. Ist die Transaktion valide, wird sie weitergeschickt. Ist sie fehlerhaft, wird sie verworfen. So werden fehlerhafte Transaktionen frühzeitig erkannt [Antonopoulos und Klicman, 2018, 221].

Zusammenfassen von Transaktionen in Blöcke

Mining-Nodes halten unbestätigte Transaktionen in einen extra Pool bereit, um aus ihnen einen neuen Block zu „schürfen“. Sie sammeln die Transaktionen in einen Anwärterblock. Ein Anwärterblock ist ein Block der noch nicht valide ist und nicht zur Blockchain gehört, da für diesen noch kein Proof-of-Work (PoW) gefunden wurde. Die Mining-Node erstellt den Anwärterblock und füllt die Header Informationen mit den entsprechenden Werten [Antonopoulos und Klicman, 2018, 229]. Nur die Nonce wird mit Nullen initialisiert, denn diese gilt es zu finden, um den PoW zu erhalten. Im Mining-Prozess wird nun die Nonce mit einem beliebigen Wert belegt und der Block-Header gehasht. Ist der Hash kleiner als der Target, der im Block-Header steht, hat man einen validen Block geschürft. Andernfalls wird die Nonce verändert und der Prozess beginnt von neuem. Dies führt man solange durch, bis der Hash kleiner ist als das Target [Antonopoulos und Klicman, 2018, 234]. Ist das Rätsel gelöst, wird der Block an alle Nodes versendet, damit diese den Block validieren können [Antonopoulos und Klicman, 2018, S. 222 ff].

Verifikation der neuen Blöcke

Jede Node, die einen Block empfängt, prüft zunächst, anhand einer Reihe von Kriterien, ob dieser valide ist. In Bitcoin umfassen diese Kriterien folgendes [Antonopoulos und Klicman, 2018, S. 241]:

- Syntaktische Korrektheit der Datenstruktur.
- Korrektheit des Proof-of-Work.
- Der Zeitstempel des Blocks liegt weniger als 2 Stunden in der Zukunft.

- Die Blockgröße liegt im erlaubten Bereich
- Nur die erste Transaktion ist eine Coinbase-Transaktion.
- Alle Transaktionen innerhalb des Blocks sind valide.

Ist der Block valide, sind damit alle enthaltenen Transaktionen bestätigt. Anschließend schickt er diesen weiter. Ist der Block nicht valide, hat der Miner, der diesen Block erstellt hat, seine Rechenleistung verschwendet und der Block wird verworfen. Da alle Nodes nach den gleichen Regeln validieren, kann niemand Schummeln.

Alle Mining-Nodes, die versucht haben den gleichen Block zu schürfen, machen mit dem nächsten Block weiter. Dabei werden alle Transaktionen, die im neuen Block enthalten sind, aus dem eigenen Transaktionspool entfernt, sodass nur noch unbestätigte Transaktionen als Block zusammengefasst werden. Der empfangene Block wird zudem in das *Vorheriger Block-Hash* Feld gesetzt [Antonopoulos und Klicman, 2018, 240 f].

Wahl der richtigen Blockchain

Valide Blöcke werden nun mit der Blockchain verbunden. Dazu wird das *Vorheriger Block-Hash* Feld genutzt, um zu bestimmen, an welchen Block der neue Block angehängt werden soll. Es kann vorkommen, dass der Vorgänger des neuen Blocks nicht die Spitze der aktuellen Blockchain ist. Man fügt ihn dann an die Stelle ein wo er hingehört und behält sich diese Abzweigung vor. Diese Abzweigung wird Sekundär-Chain genannt. Möglich ist es, dass die Mehrheit der anderen Nodes diese Sidechain als Vorgänger ihrer neuen Blocks verwenden. Dann wird diese Chain zur Haupt-Chain. Dadurch wählen alle Nodes früher oder später die gleiche Chain und temporäre Unterschiede werden aufgelöst. Mit dem Schürfen eines neuen Blocks, stimmt man der Wahl einer Haupt-Blockchain zu, denn im Feld *Vorheriger Block-Hash* des zu schürfenden Blocks, steht dessen Vorgänger und somit die Blockchain, an die der neue Block angehängt werden soll. [Antonopoulos und Klicman, 2018, S. 241 - 243].

2.2 Ethereum

Die für diese Arbeit verwendete Blockchain wird die Ethereum-Blockchain sein. Die Eigenschaften und Vorteile von Ethereum werden deshalb in diesem Abschnitt aufgezeigt.

Eine der Hauptgründe ist das Konzept von Smart-Contracts, die es so z.B. im Bitcoin-System nicht gibt.

Ethereum ist eine Plattform, auf der man Programme deployen kann, die dezentral ausgeführt werden [Buterin, 2014]. Sie bietet, wie Bitcoin auch, eine Blockchain-Datenstruktur und einen Konsensmechanismus an. Darüber hinaus bietet Ethereum allerdings eine Turing-Vollständige Sprache an, die es einem erlaubt, jedes beliebige Programm zu schreiben [Swan, 2015]. Diese Programme in Ethereum werden Smart-Contracts genannt. Sie werden auf bestimmten Nodes, den sogenannten Full-Nodes, im Ethereum-Netzwerk in der *Ethereum Virtual Machine* (EVM) ausgeführt [Swan, 2015]. Die Währung in Ethereum heißt *ether* (ETH) und wird für das Betätigen von Transaktionen verwendet. Man kann Ethereum auch als einen „Welt-Computer“ ansehen [Antonopoulos und Klicman, 2018]. Dieser Welt-Computer hat einen einzigen globalen Zustand.

Technisch gesehen kann Ethereum auch als Zustandsautomat betrachtet werden. Die EVM ist dafür verantwortlich den Zustandsautomaten zu verändern und einen Zustand in einen anderen zu überführen. Ein Zustand ist definiert durch eine Menge von Key-Value Paaren, wobei der Key die Ethereum Adresse eines Accounts ist und der Value ist der Account selbst. Jeder kann sich durch die Erzeugung eines Schlüsselpaars, dem Private- und Public-Key, einen Account erstellen und über Transaktionen den Zustandsautomaten anstoßen. Die Transaktion beschreibt wie der Zustand verändert wird. Es kann sich um eine einfache Überweisung von ETH handeln oder auch um das Ausführen von einem Smart-Contract.

2.2.1 Ethereum Accounts

In Ethereum gibt es zwei Arten von Accounts: **'externally owned account'** (EOA) und **'contract'** (auch *Smart Contract* genannt). Zu jedem EOA gehört ein eindeutiger Private-Key, der für digitale Signaturen verwendet wird. Jeder, der den Private-Key besitzt, hat damit die Kontrolle über den Account. Ein Contract Account hingegen wird von keiner Person gesteuert, daher braucht dieser keinen Private-Key. Ein Contract Account kann nur über eine Transaktion erzeugt werden. Diese spezielle Transaktion wird auch „Contract Creation“ genannt. Dazu muss man ein Programm schreiben, welches an diesen Account gebunden werden soll und anschließend eine Transaktion an die Ethereum Adresse *0x0* gesendet werden. Bei erfolgreicher Transaktion, wird ein neuer Account mit einer eigenen Ethereum Adresse erzeugt. Der Besitzer des Accounts ist keine Person,

wie es bei EOAs der Fall ist. Der neue Contract Account wird lediglich von dem mit ihm verbundenen Programmcode gesteuert.

Beide Account-Arten haben eine öffentliche Adresse, über die sie adressierbar sind. Bei EOAs wird diese Adresse direkt von dem Private-Key abgeleitet. Zwischen EOAs kann man ETH, die Währung der Plattform Ethereum, über Transaktionen transferieren. Ist das Ziel einer Transaktion ein Contract Account, so wird dessen Programmcode ausgeführt. Contracts sind also autonome Agenten, die auf erhaltenen Transaktionen reagieren können [Antonopoulos und Wood].

Die Menge der Zustände aller Accounts definiert den globalen Zustand der Ethereum Blockchain. Ändert sich der Zustand eines Accounts, ändert sich damit auch der globale Zustand. Der Zustand eines Accounts ist durch folgende Felder definiert und wird in jedem Block der Blockchain festgehalten:

- **balance**: Menge an ETH im Besitz des Accounts.
- **nonce**: Die Anzahl an erfolgreichen Transaktionen.
- **storage-root**: Der Hash des root-nodes vom Merkle Patricia Baum der den World State codiert. Da nur Smart-Contracts einen eigenen Speicher habe, bleibt bei EOAs das Feld leer.
- **code-hash**: Der Hash des Ethereum Codes, der mit diesem Account verbunden ist. Da nur Smart-Contracts einen Programmcode besitzen, bleibt bei EOAs das Feld leer.

2.2.2 Transaktionen

Transaktionen stoßen Ethereum als Zustandsautomaten an und verändern den Zustand. Nur EOAs können eine Transaktion initiieren. Contracts jedoch können auf Transaktionen reagieren und dann interne-Transaktionen starten, sogenannte *Message-Calls*. Jede valide Transaktion wird in die Blockchain aufgenommen und ist permanent gespeichert.

Eine Transaktion kann nur von einem EOA initiiert werden, da Transaktionen signiert werden müssen. Weil Contracts keinen Private-Key haben, können sie keine Signatur erstellen und somit keine Transaktion starten. Transaktionen können zur Überweisung von ETH verwendet werden oder um Code eines Contracts auszuführen. Es ist auch möglich,

gleichzeitig einen Contract zu adressieren, um Code auszuführen, und diesem Contract-Account ETH zu überweisen. Ohne Transaktionen würde die Plattform Ethereum still stehen und nichts würde passieren, bzw. der globale Zustand von Ethereum würde sich nicht ändern.

Eine zusätzliche Einheit in Ethereum, neben ETH, ist „Gas“ . „Gas“ wird verwendet, um den Berechnungsaufwand für das Ausführen von Smart-Contracts zu messen. Sie wird nur in der EVM genutzt und hat darüberhinaus keine Bedeutung. Bei jeder Transaktion muss angegeben werden, wieviel „Gas“ -Einheiten man bereit ist zu kaufen (***gaslimit***) und für wieviel ETH man eine „Gas“ -Einheit kaufen will (***gasprice***). Während einfache Transaktionen zwischen EOAs ein fixes *gaslimit* von 21000 haben, kann dies bei Transaktionen an Smart-Contracts, je nach implementierten Code variieren. Jede Operation im Code hat einen bestimmten Verbrauch an „Gas“ (***gascost***) und dieser wird jedesmal vom, in der Transaktion angegebenen *gaslimit* (Tank), abgezogen. Fällt der Tank auf 0 bevor das Programm erfolgreich terminiert ist, wird die Ausführung des Smart-Contracts abgebrochen und alle Zustandsänderungen, die bis dahin stattgefunden haben, werden zurückgesetzt. Dem Transaktionsinitiator wird das verbrauchte „Gas“ in Form von ETH abgezogen und auf das Konto des Miners, der den Block mit dieser Transaktion „geschürft“ hat, addiert (**Transaktionsgebühr**). Der Transaktionsinitiator hat somit ETH verschwendet. Wenn das Programm erfolgreich terminiert und noch „Gas“ übrig bleibt, erhält der Miner nur das verbrauchte Gas in Form von ETH. Der Transaktionsinitiator muss also als *gaslimit* mindestens soviel angeben, dass das Programm terminiert. Der Preis den der Transaktionsinitiator an den Miner zahlen muss, bildet sich also wie folgt:

$gascost \cdot gasprice =$ Transaktionsgebühr

Das „Gas“ dient hauptsächlich der Lösung des „Halteproblems“ . Dies besagt, dass es unmöglich ist im voraus zu wissen, ob ein Programm terminiert oder nicht. Die einzige Möglichkeit es herauszufinden, ist das Programm laufen zu lassen, bis es terminiert. Da eine Endlosschleife, als Beispiel, nie terminieren wird, muss man ewig warten, um es herauszufinden. Damit ein Programm also nicht ewig läuft, wird ein *gaslimit* als eine Art „Tank“ mitgegeben und das Programm läuft solange bis es terminiert oder der „Tank“ leer geht.

Das *gasprice* hat eine weitere Funktion in Ethereum, die allerdings nur auf ökonomische Gründe zurückzuführen ist. Ein hoher *gasprice* führt zu einer schnelleren Bestätigung der Transaktion. Sie wird also schneller in die Blockchain aufgenommen und somit erfolgreich

ausgeführt. Das liegt daran, dass Miner sich aussuchen können welche Transaktionen als nächstes in einen Block zusammengefasst werden sollen. Da ein höherer *gasprice* bei gleichem *gaslimit* zu höheren Transaktionsgebühren führt, die der Miner erhält, werden diese von Minern bevorzugt.

Transaktionen enthalten die folgenden Informationen:

- **Nonce**: Eine Zahl, die bei jeder Transaktion, von diesem Account aus, erhöht wird.
- **gasprice**: Der Preis, den man bereit ist zu zahlen für Gas.
- **gaslimit**: Die Menge an Gas, die man bereit ist zu kaufen.
- **Recipient**: Der Empfänger der Transaktion.
- **Value**: Die Menge an Ether, die man dem Empfänger überweisen möchte.
- **Data**: Payload, der bei der Transaktion versendet wird.
- **Signatur**: Signatur des Senders.

Die Nonce dient dem Schutz gegen Replay Angriffe. Indem mit jeder Transaktion die Nonce erhöht wird und diese mit in die Signatur einfließt, können doppelt übertragene Transaktionen erkannt werden.

Jede Transaktion verändert den Zustand eines Accounts. Einfache Überweisungen zwischen EOAs verändern z.B. dessen *balance* und die *nonce*. Jede ausgeführte Operation eines Smart-Contract, kann den Zustand des Smart-Contracts ändern indem z.b. Variablen belegt werden. [Antonopoulos und Wood]

Jede Transaktion wird durch den Initiator der Transaktion digital signiert. Als Algorithmus für Digitale Signaturen verwendet Ethereum den *'Elliptic Curve Digital Signature Algorithm'* (ECDSA). Es ist ein asymmetrisches Verfahren und nutzt ein Schlüsselpaar, bestehend aus dem Private-Key und dem Public-Key [Antonopoulos und Klicman, 2018, S. 140].

2.2.3 Smart Contracts

Contracts oder auch Smart Contracts sind Accounts, die nicht durch die User gesteuert werden, wie es bei EOAs der Fall ist. Sie haben keinen Private-Key und können somit auch keine Transaktion starten. Nur der Programmcode hat die Kontrolle über den Contract-Account.

Im Kontext Ethereum ist ein Smart Contract definiert, als ein unveränderliches, deterministisches Computerprogramm, das auf der EVM läuft [Antonopoulos und Wood].

Für das Erstellen von Smart Contracts gibt es verschiedene High-Level Programmiersprachen. Eine davon ist Solidity. Ein in Solidity geschriebenes Programm muss zunächst in den für die EVM lesbaren bytecode kompiliert werden. Anschließend muss der Code deployt werden, indem man eine Transaktion an die *Contract-Creation-Address* sendet. Der Contract ist dann über seine Ethereum-Adresse erreichbar und kann von EOAs durch Transaktionen aufgerufen werden. [Antonopoulos und Wood]

2.3 Internet of Things

Der Begriff *'Internet of Things'* (IoT) wurde erstmals 1999 von Kevin Ashton benutzt [Ashton]. Er bringt den Begriff in Verbindung mit Computern, die ohne menschlichen Einfluss Informationen aufnehmen und prozessieren können. Allerdings gibt es bis heute keine genaue und einheitliche Definition für das IoT [Litzel]. Singh und Singh [2015]; Litzel; Atzori u. a. [2010] beschreiben das IoT, als eine weltweite Vernetzung von Alltagsgegenständen (*IoT-Devices*) über das Internet. Jeder der Gegenstände hat eine eigene IP und kann mit allen im Netzwerk kommunizieren. Das Ziel liegt im Austausch von Informationen über unser tägliches Leben [Li u. a., 2013], um die Wahrnehmung der Umgebung zu verbessern [Zhang und Wen, 2017].

Heute generieren IoT-Devices enorme Mengen an Daten, die sowohl sicherheitskritische Daten, als auch private und sensible Daten enthalten können. Das macht es unter anderem zu einem attraktiven Ziel für Cyber-Angriffe [Sadeghi u. a., 2015]. Nach [Gartner, 2017] steigt die Anzahl der IoT-Devices, im Jahr 2020, auf 20 Milliarden an.

2.3.1 Herausforderungen für die Sicherheit

Bei solch einer hohen Anzahl an IoT-Devices darf die Sicherheit der Geräte nicht außen vor gelassen werden. Leider wird heutzutage nicht viel Wert auf die Sicherheit gelegt, sondern eher auf die Funktionalität des Produkts und die schnelle Vermarktung [Rouse, 2018].

Da IoT-Devices meistens relativ klein sind und Batterie betrieben, sind sie **hardwaretechnisch** stark eingeschränkt und haben nicht die nötige Rechenpower, um neben dem eigentlichen Zweck auch Sicherheitsfunktionen auszuführen [Hossain u. a., 2015; Rouse, 2018]. Auch der Energieverbrauch muss gering gehalten werden und Sicherheit verbraucht viel Energie[Dean und Agyeman, 2018]. Deshalb können vor allem Asymmetrische Verfahren nicht angewendet werden, da diese zu viel Rechenpower benötigen [Zhang u. a., 2014]. Der niedrige Speicher ist ebenfalls ein Problem. Herkömmliche Algorithmen sind nicht für diese Art von Speicherkapazität zugeschnitten [Hossain u. a., 2015].

Aber nicht nur **hardwaretechnisch** gibt es Herausforderungen zu bewältigen, sondern auch **softwaretechnisch**. Die eingebetteten Betriebssysteme können meist nicht mehr als die eigentliche Funktionalität ausführen. Hier müssen Mechanismen gefunden werden, mit denen trotz der geringen Operationsmöglichkeiten des Betriebssystems, Sicherheit in das System integriert werden kann. Ebenfalls sollten die IoT-Geräte regelmäßige Sicherheitsupdates erhalten können [Hossain u. a., 2015].

2.3.2 Zugriffskontrolle

Durch die Einschränkungen von IoT-Geräten ist es Angreifern möglich, die Kontrolle über diese zu übernehmen oder eigene IoT-Geräte ins vorhandene System zu deployen, um z.B. sensible Daten abzugreifen [Zhang u. a., 2019]. Um den Zugriff auf IoT-Geräte zu schützen und zu kontrollieren ist ein Zugriffskontrollmechanismus entscheidend für die Sicherheits des Gerätes. Im Folgenden werden verschiedene Zugriffskontrollmodelle erklärt.

Die Zugriffskontrolle eines IT-Systems stellt Mechanismen zur Vergabe von Zugriffsrechten bereit und prüft die Autorisierung beim Zugriff auf die zu schützenden Objekte [Eckert, 2014].

Es gibt vier Hauptklassen der Zugriffskontrolle:

	Datei 1	Datei 2	....
Bill	owner, r, w, x		
Joe	r, x	w	
Anne		owner, r, x	
...			

Tabelle 2.1: Zugriffsmatrix (nach [Eckert, 2014, S. 651])

- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC)
- Role-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC)

DAC

Beim DAC entscheidet der Besitzer des zu schützenden Objekts, wer auf das Objekt zugreifen darf und vor allem, mit welchen Rechten [Sandhu und Munawer, 1998]. Der Anfrager muss also zunächst identifiziert werden bevor er autorisiert wird [Sandhu und Samarati, 1994]. Die weitverbreitetste Methode zur Verwaltung der Rechte ist die **Zugriffsmatrix**.

In 2.1 sieht man, dass eine Zugriffsmatrix ein 2 dimensionales Array ist, welches jede Datei für jedes Subjekt beschreibt und umgekehrt. Diese Matrix hat im normalfall allerdings viele freie Felder und ist deshalb nicht zu Implementation geeignet. Deshalb haben sich 2 Konzepte entwickelt, dessen Kombination zu einer Zugriffsmatrix wird, jedoch ohne die freien Felder: Zugriffskontrolllisten und Zugriffsausweise [Eckert, 2014].

Zugriffskontrolllisten

Zugriffskontrolllisten bilden sich aus den Spalten der Matrix. Sie sind einem Objekt zugeordnet und speichern die Rechte für jedes Subjekt [Samarati und de Vimercati, 2001]. Damit fallen die leeren Felder weg, denn jeder Eintrag in der Liste besteht aus dem *Subjekt-Namen*, sowie seinen *Zugriffsrechten* [Eckert, 2014, S. 651]. Die Zugriffskontrollliste, die der *Datei 1* aus 2.1 zugeordnet ist, enthält also die Einträge [(Bill -> owner, r, w, x), Joe -> r, x].

Zugriffsausweise

Zugriffsausweise bilden sich aus den Zeilen der Matrix. Jedem User ist ein Liste zugeteilt, in der die Rechte für jedes Objekt definiert sind [Samarati und de Vimercati, 2001]. *Bills* Liste (Zugriffsausweis) enthält die Einträge [(Datei 1 -> owner, r, w, x)]. Da Bill laut Tabelle 2.1 keine weiteren Zugriffsrechte auf anderen Objekten hat, steht auch nichts weiteres in seinem Ausweis. Ein großer Vorteil der Zugriffsausweise ist, dass man den Ausweis an andere Subjekte vergeben kann, da dieser nicht direkt an ein Subjekt gebunden ist. Es ist lediglich eine Liste mit Objekte und den Zugriffsrechten auf die Objekte [Eckert, 2014, S. 658].

MAC

Mit systembestimmte Zugriffskontrolle (engl. mandatory access control (MAC)) kann man den Informationsfluss kontrollieren [Eckert, 2014, S. 692]. Es gibt eine Menge von Sicherheitsklassen, die partiell geordnet sind und den Subjekten und Objekten zugeordnet wird. Über systemglobale Regeln wird der Informationsfluss zwischen den Klassen definiert. Ein bekanntes Modell ist das Bell-LaPadula-Modell. Dies legt fest, dass man mindestens die Sicherheitsklasse des Objektes haben muss, um darauf zuzugreifen. Möchte ein Subjekt mit einer geringen Sicherheitsstufe ein Dokument lesen, das eine hohe Sicherheitseinstufung hat, wird ihm der Zugriff verweigert. Umgekehrt, wenn ein Subjekt mit einer hohen Sicherheitsklasse auf ein Objekt mit einer geringen Sicherheitseinstufung zugreifen will, wird der Zugriff genehmigt [Eckert, 2014, S. 692].

RBAC

RBAC reguliert den Zugriff auf Basis der Rolle des Anfragers. Dazu muss zunächst definiert werden welche Rollen es gibt und welche Subjekte zu den Rollen gehören. Jeder Rolle wird eine Menge von Rechten zugeschrieben, die beschreiben auf welche Objekte die Rolle mit welchen Operationen zugreifen darf. Ein Subjekt einer Rolle, darf also auf alles zugreifen, die der Rolle gestattet ist. Dadurch muss man nicht jedem Subjekt einzeln, die Rechte zuteilen, sondern kann Gruppen bilden. Es ist dadurch auch möglich einem Subjekt mehrere Rollen zu vergeben [Sandhu und Samarati, 1994].

ABAC

ABAC führt die Zugriffskontrolle, anders als die drei Vorgänger, nicht anhand der Identität des Subjekts durch, sondern anhand von vorher definierten Regeln.

In Hu u. a. [2014] wird ABAC wie folgt definiert:

> An access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions.

Die definierten Regeln basieren auf Attributen, die Servos und Osborn [2017] in die folgenden Kategorien einteilt:

Benutzerattribute
: Dies sind Attribute des Subjekts, welches die Anfrage stellt und autorisiert werden soll. Die Attribute können das Alter, Name, Adresse, Rolle, Job etc. sein.

Objektattribute
: Attribute des Objekts auf das zugegriffen werden möchte. Es kann sich um Meta-Daten des Objekts handeln wie die Größe, das Datum der Erstellung, der Typ oder der Besitzer. Es kann aber auch der Inhalt des Objekts sein.

Umgebungsattribute
: Attribute der Umgebung sind z.B. das aktuelle Datum, die Netzwerkauslastung etc.

Verbindungsattribute
: Verbindungs Attribute beziehen sich auf die Eigenschaften der aktuellen Verbindung. Die IP-Adresse, welche die Anfrage stellt, der Ort von dem die Anfrage gestellt wird oder auch bestimmte Daten, die mit der Session versendet wurden.

Administrative Attribute
: Diese Attribute beziehen sich auf Konfigurationen des Systems, die durch einen Administrator getätigt worden sind.

Ein ABAC-Framework sollte zum Definieren der Regeln eine geeignete Sprache mitliefern. Die 'extensible Access Control Markup Language' (XACML) ist eine der gängigsten Sprachen zur beschreibung von Regeln [Servos und Osborn, 2017]. Mit einer Sprache ist es möglich die verschiedene Attribute zu kombinieren, um daraus eine Regel zu erzeugen.

Viele Online-Bibliotheken nutzen eine Regel, um den Zugriff von Hochschulen aus zu erlauben. Solch eine Regel könnte wie folgt aussehen: 'Erteile Zugriff, wenn IP-Adresse zum Netz X gehört.', wobei X das Netz einer Hochschule ist.

Die meisten ABAC Modelle haben die folgenden Komponenten [Servos und Osborn, 2017]:

User
Die Benutzer, die eine Anfrage stellen, um den Zugriff auf Objekte zu erhalten

Objekt
Die Objekte, die durch das ABAC System geschützt werden sollen.

Attribute
Die Attribute, die man verwenden kann.

Rechte
Die Art der Rechte, die man nach der Autorisierung erhält.

Regeln
Die Regeln, die beschreiben, wie man den Zugriff erhält.

2.4 Blockchain-basierte Access Control Frameworks

In diesem Abschnitt wird erläutert, was Blockchain-basierte Access Control Frameworks sind, wie sie arbeiten und weshalb sie eine wichtige Rolle bei der Zugriffskontrolle für IoT-Geräte spielen. Anschließend werden drei Frameworks miteinander verglichen.

Aufgrund der geringen Rechenpower von IoT-Geräten, sind diese oftmals abhängig von Serviceleistungen über Third-Parties [Zhou u. a., 2017]. Mit dem Auslagern sicherheitskritischer Funktionen an zentrale Instanzen ist man nicht mehr auf die eigenen Ressourcen beschränkt und hat dementsprechend leistungsfähigere Servcies. Allerdings bringt dieser Ansatz auch Nachteile mit sich. Ein Nachteil dieses Ansatzes ist die Abhängigkeit, des Besitzers eines IoT-Gerätes, vom Vertrauen des Service-Providers. Die sensitiven Daten die vom IoT-Gerät erzeugt werden und die geheimen Schlüssel, die für Sicherheitsfunktionen notwendig sind, werden beim Service-Provider verarbeitet und gespeichert. Die Sicherheit und die Privatsphäre liegt damit bei der Third-Party, welcher zusätzlich einen

Single-Point-of-Failure darstellt [Wei u. a., 2014]. Ein weiterer Nachteil ist, dass es keine Ende-zu-Ende Sicherheit gibt, da die Kommunikation zunächst über die Third-Party geht [Zhang u. a., 2019].

Daher haben sich einige dezentrale Modelle entwickelt, die als Grundlage die Blockchain nutzen. Die Blockchain hat die Eigenschaft, dass sie dezentral ist und aufgrund ihres Konsensmechanismus alle Peers zu einer einheitlichen Lösung kommen. Damit wird der Zuganskontrollmechanismus zu einer dezentralen Anwendung und eine zentrale Instanz ist nicht mehr nötig. Zudem kann man Zugriffsaktivitäten in der Blockchain speichern, um sie jederzeit einsehen zu können und die Anwendung vollständig transparent zu machen. Ein entscheidender Nachteil dieses Ansatzes ist, dass das „Schürfen“ eines Blocks in Ethereum ca. 10 - 20 Sekunden dauert [Siriwardena, 2017] und somit das System langsamer läuft als gewöhnliche Zugangskontrollmodelle.

Im Folgenden werden 3 Blockchain-basierte Access Control Frameworks vorgestellt und verglichen. Eines der Frameworks wird dabei für die weitere Arbeit ausgesucht.

2.4.1 RBAC-SC

Das in [Cruz u. a., 2018] beschriebene Framework ist ein Rollen-basiertes Access Control Framework. Eine Organisation, welches ihren Mitgliedern bzw. Usern Rollen zuweisen möchte, erstellt einen Smart-Contract und deployt diesen auf die Ethereum-Blockchain. Die Adresse des Contracts kann man über unterschiedliche Wege erreichbar machen, z.B. auf der eigenen Homepage. Der Smart-Contract muss dabei die folgenden Funktionen bereitstellen:

- addUser(user.EOA, user.role, user.notes): Diese Funktion kann nur von der Organisation aufgerufen werden, die den Smart-Contract deployt hat und die Rollen verteilen will. Die Methode weist dem User mit dem Account „user.EOA“ die Rolle „user.role“ zu. „User.notes“ kann verwendet werden, um zusätzliche Informationen an diesen User zu binden.
- removeUser(user.EOA): Diese Methode kann nur vom Ersteller des Smart-Contracts ausgeführt werden. Sie löscht einen User und nimmt ihm damit die Rolle weg.
- addEndorsee(endorsee.EOA, endorsee.notes): Diese Funktion kann nur von einem bestehenden User der Organisation aufgerufen werden. Der User kann damit einen

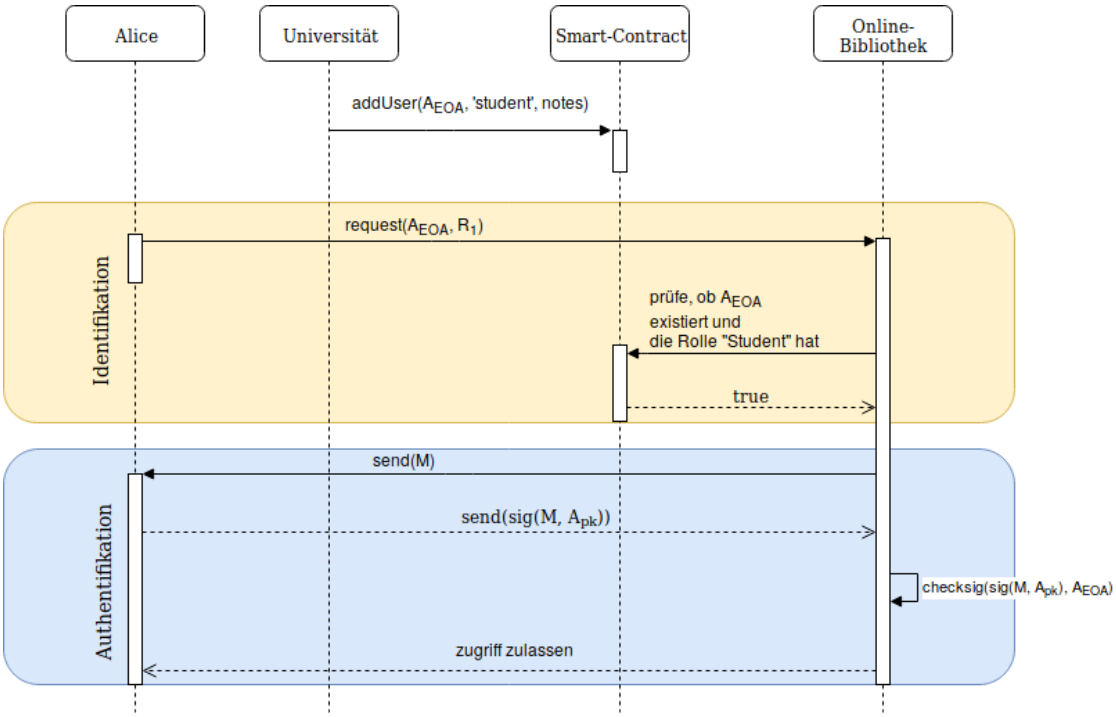
anderen User (endorsee.EOA), der noch nicht Teil der Organisation ist, unterstützen.

- removeEndorsee(endorsee.EOA): Diese Funktion kann nur von einem bestehenden User der Organisation aufgerufen werden. Sie löscht die „Unterstützung“ für den User *endorsee.EOA*.
- changeStatus(): Diese Methode kann nur vom Ersteller des Smart-Contracts ausgeführt werden. Sie deaktiviert den Smart-Contract.

Eine Organisation kann nun über diesen Smart-Contract seine User verwalten und ihnen verschiedene Rollen zuweisen. Solch eine Organisation kann z.B. eine Universität sein, die ihre User in „Student“ und „Professor“ einteilt. Möchte ein Student auf eine Ressource zugreifen, identifiziert er sich zunächst mit seinem EOA. Der Anbieter dieser Ressource (RO) prüft daraufhin, ob der Anfrager autorisiert ist. Dafür schaut dieser in den Smart-Contract der Universität, auf der der Student behauptet zu sein und prüft, ob der angegebene EOA tatsächlich ein Student ist. Ist dieser User vorhanden, wurde der Anfrager erfolgreich identifiziert und muss sich nur noch authentifizieren. Der RO sendet dem Anfrager eine zufällig gewählte Nachricht, welche dieser mit dem Private-Key signieren muss, der zu seinem EOA passt. Die signierte Nachricht wird zurück an den RO gesendet und wird von ihm verifiziert. Dazu nutzt er den Public-Key des Anfragers, mit dem sich dieser identifiziert hat. Abbildung 2.3 verdeutlich den Ablauf anhand eines Beispiels, indem Alice, eine Studentin, auf eine Online-Bibliothek zugreifen möchte, die nur für Studenten zugelassen ist.

Ein Vorteil ist, dass die Online-Bibliothek Alice identifizieren kann, ohne die Universität zu benachrichtigen. Sie muss lediglich in die Blockchain schauen und prüfen, ob es einen Eintrag der Universität für Alice gibt. Hier muss allerdings gleichzeitig angenommen werden, dass der Smart-Contract mit dem die Online-Bibliothek kommuniziert, wirklich von der Universität stammt und keine falsche Smart-Contract Adresse benutzt wurde. Außerdem kann die Authentifikation auch offline, z.B. über NFC oder QR, stattfinden. Desweiteren ist es der Universität ohne großen Aufwand möglich, einem Studenten die Rolle zu entziehen.

Das Framework bringt allerdings auch Probleme mit sich. Die Identifikation ist nur an die Private-Keys der User gebunden. Sollte dieser verloren gehen, kann die eindeutige Identifikation nicht mehr gewährleistet werden. Es ist zwar möglich für den Studenten einen neuen EOA zu erstellen und diesen bei der Universität anzumelden, aber es muss



A_{pk}: Alice' Private-Key

A_{EOA}: Alice' Ethereum Adresse (Public-Key)

M: beliebige Nachricht

R_1: Ressource die nur für Studenten zugelassen ist

sig(): Funktion zur Erstellung einer Signatur

checksig(): Funktion zur Verifikation einer Signatur

Abbildung 2.3: Beispielablauf einer erfolgreichen Zugriffskontrolle nach [Cruz u. a., 2018]

gleichzeitig auch nachgewiesen werden, dass der Private-Key komprommitert ist. Andernfalls könnte ein Angreifer behaupten der Private-Key von Alice sei verloren gegangen, um einen neuen zu erstellen.

2.4.2 FairAccess

FairAccess [Ouaddah u. a., 2016] ist ein Framework, welches zunächst die Bicoin-Blockchain als Grundlage hatte. Wegen den Limitierungen der Script-Sprache von Bitcoin, wurde ein weiteres Konzept vorgestellt, das Ethereums Smart-Contracts zur Definition von Policies einsetzt [Ouaddah u. a., 2017].

In diesem Framework erstellt ein Ressource Owner (RO) für jede Ressource, die er schützen möchte, einen Smart-Contract. Dieser Smart-Contract dient als Policy und muss von einem Anfrager (RQ), der auf die Ressource zugreifen möchte erfüllt werden. Erfüllt der RQ die Bedingungen im Smart-Contract, dann wird ihm ein Token ausgestellt, das er der Ressource vorweisen kann, um den Zugriff zu erhalten. Das Framework beschreibt drei Arten von Nachrichten, die ausgetauscht werden:

- GrantAccess: Diese Nachricht wird vom RO erstellt, um ein Smart-Contract auf Ethereum zu deployen.
- RequestAccess: Diese Nachricht wird von einem RQ an den Smart-Contract der Ressource gesendet, auf die der RQ zugreifen möchte.
- GetAccess: Diese Nachricht wird vom RQ an das Gerät gesendet, welches die Ressource bereitstellt.

Es gibt sechs Phasen in diesem Framework:

1. Systeminitialisierung
Der RO identifiziert die Ressourcen, die er schützen möchte und richtet Adressen für sie ein, damit sie im Ethereum-Netzwerk identifiziert werden können.

2. Policy Erstellung
Der RO erstellt Smart-Contracts für seine zu schützenden Ressourcen und deployt diese auf die Etherum-Plattform. Im Smart-Contract muss definiert werden, unter welchen Bedingungen ein Zugriff zu erhalten ist. Hier wird die *GrantAccess* Nachricht vom RO an das Ethereum-Netzwerk gesendet.

3. Anfrage an Ressource

Der RQ möchte auf eine Ressource zugreifen und kontaktiert den zugehörigen Smart-Contract, der zuvor vom RO deployt wurde. Hier wird die *RequestAccess* Nachricht genutzt und muss entsprechende Daten beinhalten, damit der Smart-Contract den RQ authentifizieren kann.

4. Evaluation der Anfrage

Der Smart-Contract entscheidet, basierend auf den Informationen, die der RQ mitgesendet hat, ob dieser autorisiert ist auf die Ressource zuzugreifen. Bei Erfolg wird ein Autorisierungstoken ausgestellt. Andernfalls wird die Nachricht verworfen.

5. Zugriff auf Ressource

Der RQ schickt das erhaltene Token an das Gerät, welches die Ressource bereitstellt.

6. Evaluation des Tokens

Das Gerät, welches die Ressource bereitstellt prüft, ob der Autorisierungstoken tatsächlich vom RO and den RQ ausgestellt wurde, indem der entsprechende Eintrag in der Blockchain gesucht wird.

Ein typischer Ablauf dieses Frameworks wird in Abbildung 2.4 dargestellt.

In der Abbildung 2.4 kann man sehen, dass der RQ zunächst das Endgerät kontaktiert und dieser ihn anschließend zum Smart-Contract weiterleitet, um sich dort zu authentifizieren. Dieser Schritt ist nicht essentiell, da der RQ, falls dieser den zugehörigen Smart-Contract kennt, direkt den Smart-Contract kontaktieren kann.

Wie man sieht, liegt die letzte Entscheidung beim Endgerät. Dieses könnte auch wenn der Token gültig ist, den Zugriff ablehnen. Ein weiterer Aspekt ist, dass auch hier der RO niemals vom RQ kontaktiert werden muss, sondern der Smart-Contract die Verteilung der Autorisierungstoken übernimmt. Ein Vorteil dieses Frameworks ist, dass alle Zugriffsversuche und Ausstellungen von Autorisierungstoken in der Blockchain gespeichert werden, da diese immer gleichzeitig an eine Transaktion innerhalb des Ethereum-Netzwerks gekoppelt sind. Dadurch kann man den Verlauf von Zugriffen, die Häufigkeit und die Quelle der Zugriffsversuche jederzeit einsehen.

Das Framework schreibt keine konkrete Implementierung vor. Es bietet einen Zugriffskontrollmechanismus an, statt einem Zugriffskontrollmodell. Als Zugriffskontrollmodell könnte man in Phase 5 und Phase 6 des Frameworks z.B. eine Rollen-basierte Zugriffskontrolle, oder auch eine Attributs-basierte Zugriffskontrolle implementieren.

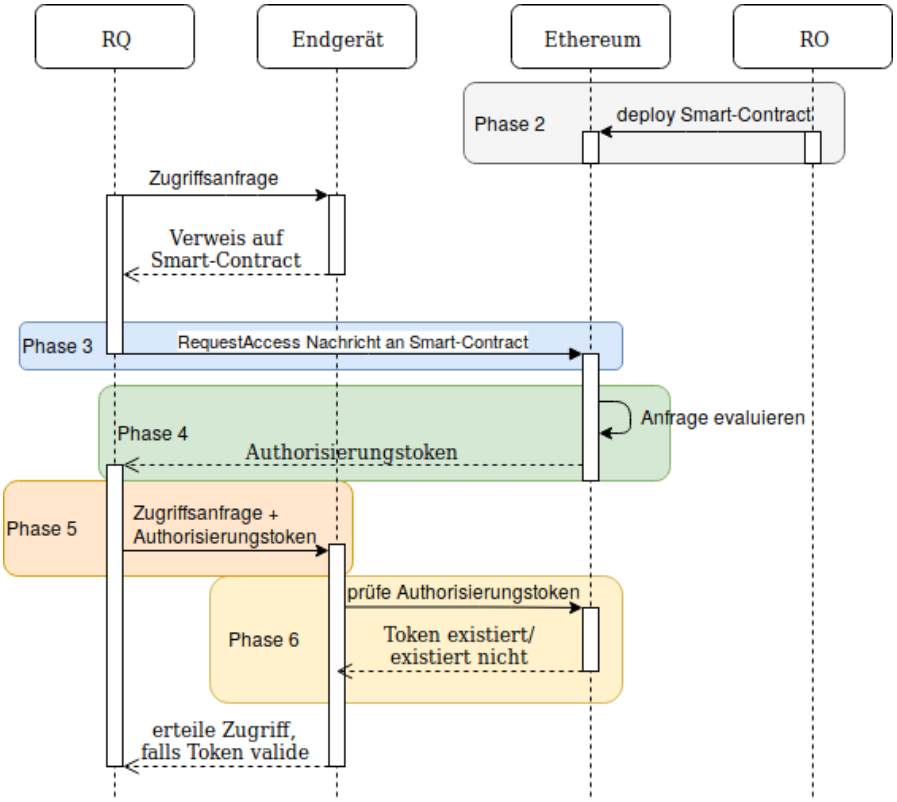


Abbildung 2.4: FairAccess Sequenzdiagramm nach [Ouaddah u. a., 2017]

2.4.3 Smart-Contract-basierte Zugriffskontrolllisten

In [Zhang u. a., 2019] wird eine Smart-Contract-basierte Zugriffskontrolle für IoT-Systeme beschrieben. Als Blockchain dient hier die Ethereum-Blockchain. Das Framework besteht aus 3 Arten von Smart-Contracts, wie man in Abbildung 2.5 sehen kann. Die Access-Control-Contracts (ACC) kontrollieren den Zugriff von Subjekten für eine bestimmte Ressource. Jeder ACC ist dabei an ein Subjekt-Objekt Paar gebunden. Möchte man dem Subjekt Zugriff auf ein neues Objekt gewähren, muss man einen weiteren ACC dafür erstellen und auf die Blockchain deployen. Die Zweite Art ist der Judge-Contract (JC). ACCs die ein Fehlverhalten eines Subjekts erkennen, senden diese Informationen an den JC. Dieser bewertet das Fehlverhalten und legt eine Sanktion fest. Der Register-Contract (RC) dient der Verwaltung von deployten JCs und ACCs. Hier können die Informationen der Smart-Contracts, wie Adresse, Name, Subjekt etc. deployt werden, damit diese für alle einsehbar sind.

Access-Control-Contract (ACC)

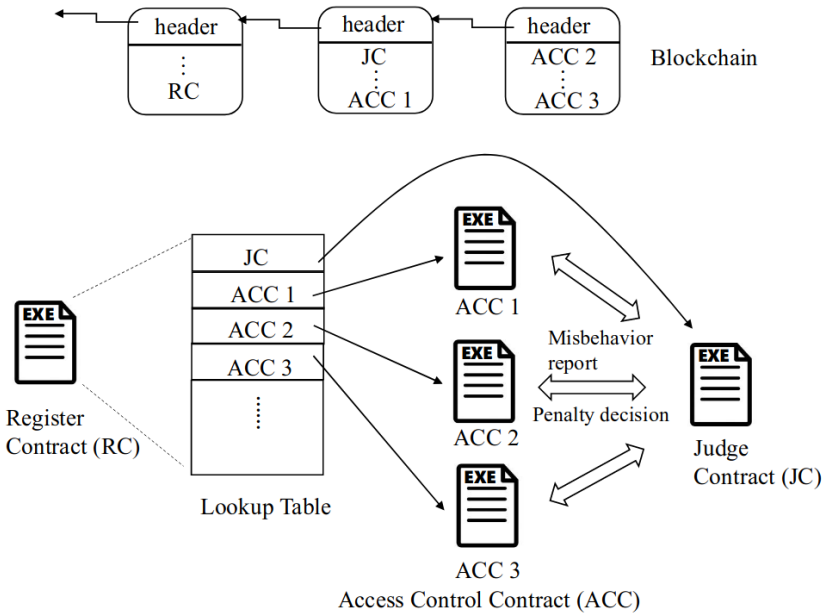


Abbildung 2.5: Framework Funktionsweise [Zhang u. a., 2019]

Ressource	Berechtigung	Zugriff erlaubt?	ToLR
Datei A	lesen	ja	2017-12-1116:19
Datei B	schreiben	nein	2017-12-1220:34
Programm A	ausführen	nein	2017-12-1116:19

Tabelle 2.2: Beispiel einer Policyliste (nach [Zhang u. a., 2019])

Ein ACC beschreibt eine Methode zwischen einem Subjekt, dem Anfrager, und einem Objekt, die Ressource die angefragt wird. Sie wird vom Besitzer des Objekts, dem Ressource Owner, erstellt und auf die Blockchain deployt. Dieser Smart-Contract enthält die Policies, die ein Anfrager erfüllen muss, um Zugriff auf die Ressource zu erhalten. Der ACC implementiert sowohl eine statische Kontrolle, durch vordefinierte Policies, als auch dynamische Kontrolle, indem das Verhalten des Anfragers beurteilt wird. Statische Policies sind, vom RO, vordefinierte Policies, die gelten müssen. Abbildung 2.2 zeigt ein Beispiel einer statischen Policyliste. Demnach dürfte das Subjekt des ACC auf die Datei A lesend zugreifen. Die *ToLR* (Time of Last Request) ist eine zusätzliche Information, die bei der dynamischen Kontrolle zum Einsatz kommt. Die dynamische Kontrolle eines ACC beobachtet das Verhalten des Anfragers und entscheidet darauf basierend, ob ein Zugriff erlaubt

wird oder nicht. Wenn ein Subjekt z.B. zuviele Anfragen in geringer Zeit sendet wird dies als Fehlverhalten interpretiert und als Reaktion kann dem Subjekt eine Sperre erteilt oder sonstige Sanktionen erteilt werden. Dazu sendet das ACC die erforderlichen Informationen zum JC, welches das erkannte Fehlverhalten bewertet und als Antwort eine Sanktion liefert.

Ein ACC muss die folgenden Schnittstellen bereitstellen:

- policyAdd(Policy): Diese Funktion wird vom RO aufgerufen und fügt eine neue Policy hinzu.
- policyDelete(PolicyId): Diese Funktion wird vom RO aufgerufen und löscht eine bestehende Policy.
- policyUpdate(Policy): Diese Funktion wird vom RO aufgerufen und aktualisiert eine bestehende Policy.
- accessControl(Information): Diese Funktion wird vom Subjekt aufgerufen und trifft die Entscheidung, ob das Subjekt den Zugriff erhalten darf oder nicht.
- setJC(JC): Diese Funktion wird vom RO aufgerufen und setzt den zugehörigen JC fest.
- deleteACC()c: Diese Funktion wird vom RO aufgerufen und löscht den ACC.

Judge-Contract (JC)

Dieser Smart-Contract bewertet das Fehlverhalten eines Subjekts und entscheidet die Sanktion. Als Beispiel einer Sanktionierung könnte man festlegen, dass das Subjekt eine Stunde für weitere Zugriffe gesperrt ist.

Die folgenden Funktionen muss ein JC bereitstellen:

- misbehaviourJudge(Information): Diese Funktion wird von einem ACC aufgerufen und bewertet das Fehlverhalten, welches der ACC als Information angibt. Daraufhin antwortet der JC dem ACC mit einer Sanktion die für das Subjekt gelten soll.
- deleteJC(): Diese Funktion wird vom RO aufgerufen und löscht den JC.

Register-Contract (RC)

Der RC dient der Verwaltung und dem Überblick von ACCs und JCs. Hier können deployte ACCs registriert werden, um sie für andere auffindbar zu machen. Subjekte die auf eine Ressource zugreifen wollen, können in der RC sehen, welcher Smart-Contract (ACC) die Zugriffskontrolle für die Ressource übernimmt. Auch die JCs werden hier registriert und sind somit für alle auffindbar. Die Informationen können in einer Art Tabelle festgehalten werden, wie in Abbildung 2.3 zu sehen ist.

Methodenname	Subjekt	Objekt	ScName	Ersteller	ScAdresse	ABI
Methode 1	Server A	Sensor B	ACC 1	Sensor B	0xca35b7d915458ef540ade6068dfe2f44e8fa733c	accessControl(),...
Methode 2	Server A	Sensor B	ACC 2	Sensor B	0xab072c469475346532bf47aea86df6176104956	accessControl(),...
Methode 3	Server B	Server A	ACC 3	Server A	0xb51f6d86d4c998531056a501344060fbafc32a48	accessControl(),...
JC			Judge		0x3f23c7b929cced4191ef6064ffcb33902ea1d92b	misbehaviorJudge()...

Tabelle 2.3: Beispiel einer RC Tabelle (nach [Zhang u. a., 2019])

Der **Methodenname** muss ein eindeutiger Name sein, der bisher nicht registriert wurde. **Subjekt** ist das Gerät, wessen Zugriff man kontrollieren möchte. **Objekt** ist die Ressource die geschützt werden soll. **ScName** ist der Name des Smart-Contracts. Der **Ersteller** ist der Ersteller des Smart-Contracts. **ScAdresse** ist die Ethereum-Adresse des registrierten Smart-Contracts. **ABI** ist das *Application-Binary-Interface*, also die Schnittstelle des Smart-Contracts.

Die folgenden Funktionen müssen bereitgestellt werden und können von jedem aufgerufen werden, die in der RC ihre ACCs und JCs verwalten möchten:

- methodRegister(Methodenname, Subjekt, Objekt, ScName, Ersteller, ScAdresse, ABI): Diese Funktion registriert eine neue Methode.
- methodUpdate(Methodenname, Subjekt, Objekt, ScName, Ersteller, ScAdresse, ABI): Diese Funktion aktualisiert den Eintrag der Methode *Methodenname*. Nur der Ersteller der Methode kann diese aktualisieren.
- methodDelete(Methodenname): Diese Funktion löscht den Eintrag der Methode *Methodenname*. Nur der Ersteller der Methode kann diese löschen.
- getContract(Methodenname): Diese Funktion liefert die Adresse der ACC und der JC von *Methodenname*.

Hier ist anzumerken, dass die Methoden *methodRegister*, *methodUpdate* und *methodDelete* nur von den Erstellern des Smart-Contracts ausführbar sind. Sollte Alice

den Smart-Contract ACC1 deployen, dann ist es nur ihr erlaubt den entsprechenden Eintrag in der RC zu erstellen, zu aktualisieren und zu löschen.

Ein typischer Ablauf des Frameworks, um auf eine geschützte Ressoource zuzugreifen könnte wie folgt aussehen:

1. Das Subjekt ruft die ***getContract*** Methode des RCs auf, um die Informationen über den entsprechenden ACC zu erhalten.
2. Der RC antwortet mit der Adresse des ACC und der ABI.
3. Das Subjekt sendet eine Transaktion an den ACC, um die *accessControl* Funktion aufzurufen. Dabei werden Informationen des Subjekts mitgegeben, um den Policies des ACC gerecht zu werden.
4. Sollte ein Fehlverhalten des Subjekts entdeckt worden sein, wird diese Nachricht an den JC weitergeleitet.
5. Der JC beurteilt das Fehlverhalten und legt eine Bestrafung fest. Diese sendet das JC an den ACC, damit dieser die Bestrafung umsetzen kann.
6. Das Ergebnis wird dem Subjekt mitgeteilt.

Wie schon erwähnt gilt ein ACC immer für ein Subjekt-Objekt Paar. Es kann auch mehrere ACCs für das gleiche Subjekt-Objekt Paar geben. Dies kann der Fall sein, wenn man sich dazu entscheidet, den Smart-Contract zu aktualisieren, aber die alte Version beizubehalten. Da man Smart-Contracts nicht verändern kann, aufgrund der Blockchain Eigenschaften, muss man einen neuen Smart-Contract erstellen. Das gleiche gilt für JCs. Ein RO kann sich dazu entscheiden, seinen JC zu aktualisieren, um z.B. mehr Informationen in die Bewertung des Fehlverhaltens einfließen zu lassen. Auch hier muss ein neuer Smart-Contract dafür erstellt werden und die Referenzen der ACCs auf den JC entsprechend neu gesetzt werden.

2.4.4 Vergleich

Die beide Modelle Zhang u. a. [2019] und Ouaddah u. a. [2017] unterscheiden sich in ihrer grundlegenden Funktionsweise nur sehr leicht. Beide nutzen Smart-Contracts zur Definition der Policies und den Bedingungen unter denen ein Subjekt den Zugriff erhält oder nicht. Ouaddah u. a. [2017] schreibt **einen** Smart-Contract pro Ressource vor. Dieser

Smart-Contract kontrolliert den Zugriff von allen Subjekten. Im Gegensatz dazu ist ein Smart-Contract (ACC) in Zhang u. a. [2019] immer an ein Subjekt-Objekt Paar gebunden. Für jeden weiteren Anfrager der auf die gleiche Ressource zugreifen will, muss ein neuer Smart-Contract erstellt und deployt werden. Zhang u. a. [2019] bietet allerdings zusätzlich eine dynamische Kontrolle an, die das Verhalten des Subjekts bewerten kann und darauf basierend eine Entscheidung trifft und sogar eine Sanktion erteilen kann. Das Rollenbasierte Modell Cruz u. a. [2018] hingegen unterscheidet sich deutlich von den anderen beiden. Auch wenn Smart-Contracts zum Einsatz kommen, dient das Framework in erster Linie der Verwaltung von Rollen und dem Nachweis der Rollen. Die Blockchain wird nicht als dezentralisiertes System zur Entscheidung eines Zugriffs genutzt, sondern die Evaluierung des Zugriffs liegt weiterhin beim Objekt. Zhang u. a. [2019] und Ouaddah u. a. [2017] verwenden die Blockchain als dezentralen Mechanismus zur Evaluierung der Anfrage und Authentifikation. In IoT-Systemen ist das von Vorteil, da die energiebeschränkten Geräte dann keine eigenen Zugriffskontrollen bereitstellen müssen.

Für die weitere Arbeit wird das Framework Zhang u. a. [2019] genommen, da dieses bereits eine konkrete Implementation [Scbimpl] vorgibt und somit dieses einfach erweiterbar ist.

3 Anforderungsanalyse

3.1 Problemanalyse

Das Framework Zhang u. a. [2019] bietet bereits einen Mechanismus zur Zugriffskontrolle an, indem die Ethereum-Blockchain Zugriffsanfragen annimmt und anhand von Policies entscheidet, ob der Zugriff gewährt wird oder nicht. Bisher wird über vordefinierte Policies für jedes Gerät entschieden, ob dieses für eine bestimmte Aktion autorisiert ist. Dabei gibt es Policies für eine statische Kontrolle, wie sie in Tabelle 2.2 zu sehen sind, die entscheiden, ob die Aktion erlaubt wird und es gibt eine dynamische Kontrolle, die anhand des Verhaltens des Anfrager, entscheidet, ob ein Zugriff erlaubt wird. Für keine der beiden Kontrollen werden Informationen des zugreifenden Gerätes beachtet. Ob das Gerät kompromittiert ist oder Sicherheitslücken aufweist, sollte ebenfalls mit in die Entscheidung eines Zugangskontrollmechanismus einfließen. Die Abbildung 3.1 veranschaulicht das Problem.

In Schritt 1 deployt der Ressource Owner des IoT-Gerätes (Objekt A) ein Smart-Contract, um den Zugriff von Subjekt B auf Objekt A zu schützen und zu kontrollieren. Angenommen, der Ressource Owner möchte Subjekt B den Lesezugriff auf das Objekt A gewähren. Um den Zugriff auf Objekt A zu erhalten, muss Subjekt B sich zunächst, über die Blockchain, authentifizieren (Schritt 2). Anschließend entscheidet die Blockchain, bzw. der Smart-Contract, das dabei ausgeführt wird, ob Subjekt B autorisiert ist für den Lesezugriff (Schritt 3). Dazu werden, die vom Ressource Owner definierten Policies geprüft. Da der Ressource Owner die Policies so definiert hat, dass Subjekt B den Lesezugriff erhalten darf, wird Subjekt B zugelassen und eine Benachrichtigung über den Zugriffsversuch an das zugehörige Objekt gesendet (Schritt 4). An dieser Stelle weiß der Ressource Owner allerdings nicht, dass der Laptop mit Viren infiziert ist und der Zugriff deshalb ein Risiko für sein eigenes IoT-Gerät darstellt. Es könnte auch sein, dass der Besitzer von Subjekt B nichts von den Viren auf seinem Gerät weiß. Um diesem Problem entgegenzukommen, soll es einen zusätzlichen Mechanismus geben, mit dem das Framework Zhang u. a.

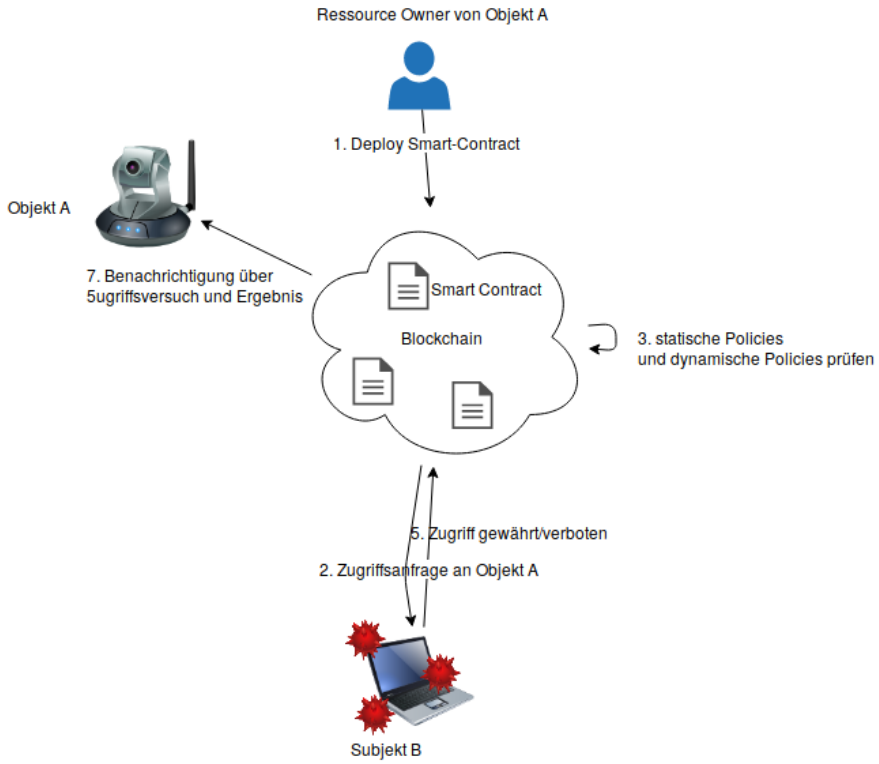


Abbildung 3.1: Access-Control Ablauf nach Zhang u. a. [2019]

[2019], bei der Evaluierung der Policies zusätzlich den Sicherheitsstand des Anfragers berücksichtigt.

3.2 Anwendungsfall

Um den Sicherheitsstand festzustellen, muss es zunächst eine Sicherheitsprüfung geben, welche die Geräte testet und bewertet. Ein solcher Prüfprozess kann z.B. das Gerät auf Sicherheitslücken untersuchen oder ob das Gerät kompromittiert ist. Die Prüfung kann auch dazu dienen, die Softwareversionen eines Gerätes zu prüfen. Das Prüfen der Geräte soll in dieser Arbeit nicht behandelt werden. Vielmehr wird davon ausgegangen, dass dies existiert und von einer Third-Party übernommen wird. Sowohl der Ressource-Owner, der seine Geräte mit einer Zugangskontrolle schützen will, als auch das Subjekt, das den Zugang zu einer dieser geschützten Geräte haben möchte, müssen der Third-Party und seinen Prüfprozessen vertrauen. Als Resultat der Sicherheitsprüfung muss der Prüfer einen Nachweis bereitstellen können, welcher den Sicherheitsstand des geprüften Gerätes

wiedergibt und von davon abhängigen Instanzen geprüft werden kann. Dieser Nachweis soll anschließend von der Zugangskontrolle genutzt werden, um eine Entscheidung treffen zu können. Abbildung 3.2 zeigt einen Anwendungsfall, indem das Subjekt B geprüft wird und das Ergebnis in die Entscheidung der Zugangskontrolle mit einfließt. Das Ergebnis wird im Folgenden Sicherheitszertifikat genannt. Die Abbildung soll den groben Ablauf der Zugangskontrolle zeigen, ohne dabei auf die technische Umsetzung einzugehen. Es soll dazu dienen, Anforderungen an das System herauszuarbeiten. Im folgenden Anwendungsfall gibt es den Ressource-Owner, der den Zugriff von Subjekt B auf die Ressource Objekt A kontrollieren möchte. Subjekt B ist ein mit Viren infizierter Laptop.

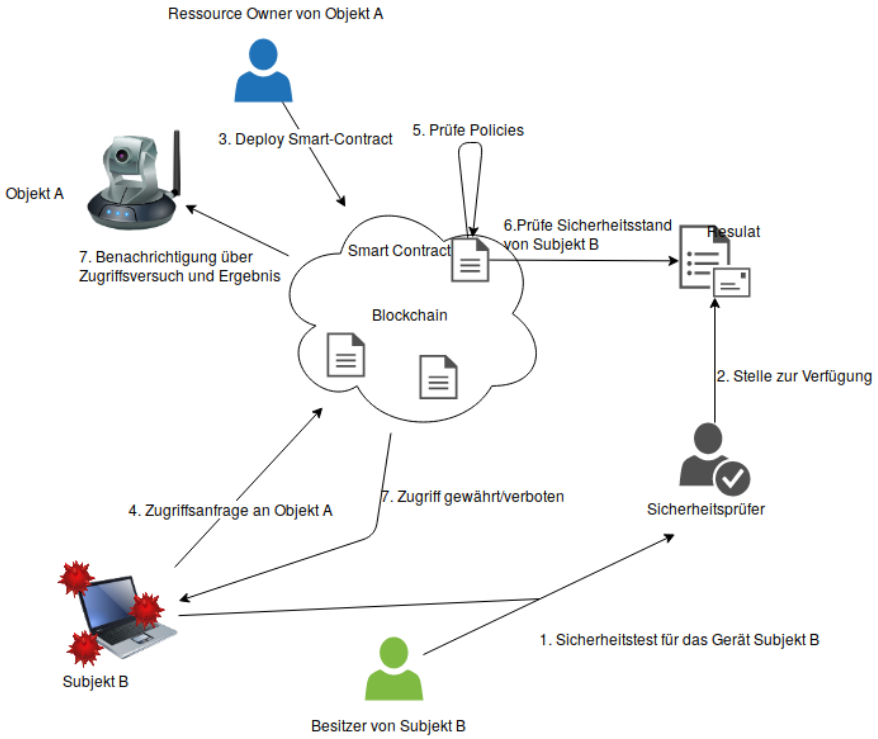


Abbildung 3.2: Der Zugriffskontrollen Ablauf mit Prüfung des Sicherheiststand des anfragenden Gerätes

Zunächst muss der Besitzer eines Gerätes, der mit dem Gerät auf eine Ressource zugreifen will, das Gerät einem Sicherheitstest unterziehen (Schritt 1). Dieser Sicherheitstest wird von einer Trusted-Third-Party durchgeführt. Sowohl der Ressource Owner, auf dessen Gerät zugegriffen werden soll, als auch der Besitzer von Subjekt B, der mit seinem Gerät auf Objekt A vom Ressource Owner zugreifen möchte, vertrauen dem Sicherheitsprüfer. In Schritt 2 stellt der Sicherheitsprüfer das Sicherheitszertifikat zur Verfügung, sodass

dieses vom Zugangskontrollmechanismus abgerufen werden kann. In Schritt 3 erstellt der Ressource-Owner den Smart-Contract, welches den Zugriff von Subjekt B auf Objekt A schützen soll. Beim Definieren der Policies innerhalb des Smart-Contracts, muss neben der statischen und dynamischen Kontrolle zusätzlich eine Kontrolle des Sicherheitszertifikats vorgesehen werden. Dafür werden Policies geschrieben, die sich auf Informationen des Sicherheitszertifikats beziehen. Bei einer Zugangsanfrage von Subjekt B an Objekt A wird wie gewohnt das entsprechende Smart-Contract aufgerufen (Schritt 4). Dieses führt die statische und dynamische Kontrolle durch, indem es jeweils die vordefinierten Policies prüft und das Verhalten des Anfragers (Schritt 5). Zusätzlich wird das Sicherheitszertifikat für Subjekt B abgerufen und ebenfalls mit den Policies verglichen (Schritt 6). Zuletzt wird, in Schritt 7, das Ergebnis dem Subjekt B und dem Objekt A mitgeteilt.

Je nachdem, wie die Policies definiert wurden, wird der Zugang zugelassen oder nicht. Entscheidend ist in diesem Anwendungsfall, dass im Sicherheitszertifikat von Subjekt B vermerkt ist, dass das Gerät mit Viren infiziert ist. Der Ressource-Owner hätte also eine einfache Policy definieren können, die prüft, ob das Gerät mit Viren infiziert ist oder nicht und dementsprechend jeweils den Zugang verbieten oder gewähren.

3.3 Anforderungen

Aus dem geschilderten Anwendungsfall (siehe 3.2) und der Problemanalyse (siehe 3.1) sollen nun die Anforderungen an das System herausgearbeitet werden.

Abbildung 3.2 beschreibt bereits aus abstrakter Sicht, das erweiterte System. Allerdings müssen die konkreten Abläufe zwischen den Entitäten festgelegt werden und vor allem in welcher Form die Sicherheitszertifikate den vorhanden Smart-Contracts zur Verfügung gestellt werden. Deswegen muss zunächst geklärt werden, welchen Anforderungen ein solches Zertifikat unterliegt. Ein Problem ist, dass die Zertifikate an das Subjekt gebunden werden müssen. Einem Angreifer darf es nicht möglich sein, das Zertifikat eines anderen Benutzers zu stehlen, um es wiederzuverwenden. Außerdem muss die Herkunft des Zertifikats bestätigt werden können. Das bedeutet, dass zu jeder Zeit geprüft werden kann, dass der Sicherheitsprüfer, dem beide Parteien vertrauen, dieses Zertifikat ausgestellt hat. Ein Angreifer darf also keine eigenen Zertifikate erstellen können, die vom System als valide erkannt werden. Desweiteren muss ein Zertifikat vor Änderungen durch Unbefugte geschützt sein. Zuletzt sollte ein Zertifikat jederzeit abrufbar sein. Es sollte also

in einer Form zur Verfügung gestellt werden, dass dieser keinen Single-Point-of-Failure darstellt.

Zusammengefasst müssen die folgenden Anforderungen für ein Sicherheitszertifikat gelten:

- Es muss eindeutig einem Objekt zugeordnet werden können.
- Es darf nicht wiederverwendet werden können.
- Die Herkunft muss eindeutig bestimmt werden können.
- Es darf nicht verändert werden können.
- Die Zertifikate müssen zu jeder Zeit verfügbar sein.
- Das Datum der Ausstellung muss bekannt sein.

4 Konzept

In diesem Kapitel soll das Konzept vorgestellt werden, wie das Framework erweitert werden kann, um die Anforderungen zu erfüllen.

Da das Grundgerüst aus Zhang u. a. [2019] schon auf Ethereum basiert, wird auch für die Erweiterung um die Sicherheitszertifikate die Ethereum-Plattform als Fundament genutzt. Das neue System ist aufgezeigt in Abbildung 4.1. Alle Geräte müssen sich einen Account in Ethereum erstellen, indem sie ein Private-Public-Schlüsselpaar erzeugen. Auch die Ressource Owner, die ihre Geräte schützen wollen, erstellen einen Account, um die zugehörigen Smart-Contracts zu erzeugen. Zusätzlich kommen nun die Sicherheitsprüfer hinzu, die sich ebenfalls mit der Erzeugung eines Schlüsselpaars einen Account in Ethereum erstellen.

4.1 Verwaltung von Sicherheitszertifikaten in der Blockchain

In Abbildung 3.2 wurde bereits die Idee des Systems eingeführt. Es soll eine dritte Instanz geben, die den Sicherheitsstand eines Gerätes bestätigt, indem es ein Sicherheitszertifikat dazu ausstellt. Damit die Anforderungen 3.3 an die Sicherheitszertifikate erfüllt werden, sollen die Sicherheitszertifikate mithilfe der Blockchain verwaltet werden. Durch die kryptographischen Eigenschaften der Blockchain und dem Einsatz von Konsensregeln, durch die alle Nodes der Blockchain den gleichen Regeln befolgen müssen, ist eine Manipulation des Inhalts der Blockchain praktisch unmöglich. Damit ein Sicherheitsprüfer mit der Ethereum-Blockchain arbeiten kann, muss dieser sich erst einmal einen Account erstellen, bestehend aus dem Private-Public-Schlüsselpaar. Somit ist der Sicherheitsprüfer eindeutig über seinen Public-Key identifizierbar.

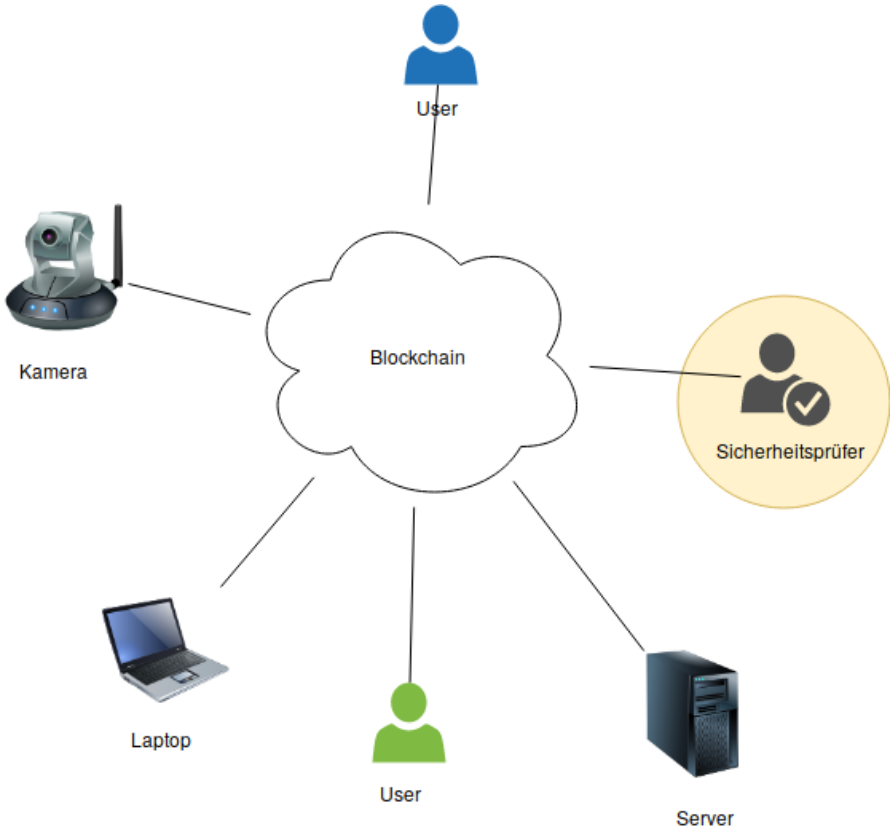


Abbildung 4.1: P2P-Netzwerk über Ethereum

4.1.1 Sicherheitszertifikat

Ein Sicherheitszertifikat soll das Ergebnis eines Prüfprozesses repräsentieren. In der weiteren Arbeit wird einfachheitshalber von einem Prüfprozess ausgegangen, der das Gerät auf Viren untersucht. Es könnte aber auch eine Prüfung auf Sicherheitslücken oder Softwareversionen sein. Im Prinzip kann das Zertifikat das Ergebnis eines beliebigen Prüfprozesses repräsentieren.

Ein Zertifikat hat den folgenden Inhalt:

- Identität des Prüfers.
- Identität des geprüften Gerätes.
- Zeitpunkt der Ausstellung des Zertifikats.
- Das Ergebnis der Prüfung. (Hat es Viren oder nicht?)

- Zusätzliche Informationen vom Prüfer. Eventuell die Art der Prüfung oder was genau untersucht wurde.

4.1.2 Sicherheitsprüfer-Smart-Contract

Für die Verwaltung der Sicherheitszertifikate soll ein Smart-Contract zuständig sein, dass der Sicherheitsprüfer erstellt. Dieser Smart-Contract läuft dann als Programm in der Blockchain, unabhängig von seinem Ersteller, dem Sicherheitsprüfer. Dadurch liegt die Verwaltung der Sicherheitszertifikate nicht mehr in der Hand einer zentralen Instanz, sondern wird dezentral durch die Ethereum-Blockchain und dem Programmcode des Smart-Contract gesteuert. Sobald der Smart-Contract einmal erstellt wurde, kann man mit ihr interagieren. Eine Funktion des **Sicherheitsprüfer-Smart-Contract** (SPSC) ist das Persistieren von Sicherheitszertifikaten. Die zweite Funktion des SPSC ist das Bereitstellen von Sicherheitszertifikaten. Diese Funktion dient auch als Schnittstelle für Zugangskontrollen, die das Zertifikat abrufen wollen, um es für die Evaluierung von Policies zu nutzen.

Wichtig ist zu beachten, dass nur der Sicherheitsprüfer, der den SPSC erstellt hat, auch Zertifikate einspeichern kann. Dadurch ist jeder SPSC einem Prüfer zugeordnet, der seine Zertifikate dort verwalten lassen kann. Allerdings müssen die verwalteten Zertifikate für jeden zugänglich sein.

4.1.3 Sicherheit durch Blockchain

Die Sicherheit dieses Systems ist durch den Einsatz der Blockchain gewährleistet. Ein in Ethereum deployter Smart-Contract kann nicht mehr entfernt werden, außer es wird explizit vom Programmierer des Smart-Contract implementiert. Da sich die Smart-Contracts in der Blockchain befinden und jeder dessen Inhalt lesen kann, kann man sehen was das Programm, also der Smart-Contract, tut. Durch den Einsatz von Smart-Contracts und der dezentralen Natur von Blockchains, ist die Komponente zur Verwaltung von Sicherheitszertifikaten immer verfügbar. Die Zertifikate sind nicht an den Prüfer selbst gebunden, der als zentrale Instanz ein Single-Point-of-Failure wäre. Sollte der Prüfer nicht mehr erreichbar sein, sind die Zertifikate trotzdem in der Blockchain verfügbar und können zu jeder Zeit abgerufen werden, indem man mit dem Smart-Contract interagiert. Da das Hinzufügen von Zertifkaten immer mit einer Ethereum-Transaktion verbunden ist, ist

die Herkunft des Zertifikats gleichzeitig auch bewiesen, denn nur der Sicherheitsprüfer soll in der Lage sein, Zertifikate hinzuzufügen. Das muss explizit in den Smart-Contract implementiert werden. Dadurch wird die Sicherheit des Systems auf die Geheimhaltung der Private-Keys reduziert. Solange die Private-Keys, mit denen man sich im Ethereum-Netzwerk authentifiziert und die der Schlüssel zu einem Ethereuem Account sind, nicht kompromittiert sind, kann kein Angreifer die Rolle von z.B. einem Prüfer übernehmen. Da die Smart-Contracts unveränderlich in der Blockchain gespeichert sind und **nur** durch ihren eigenen Programmcode gesteuert werden, gibt es auch hier keine Möglichkeit für Angreifer, dem System zu schaden. Der Inhalt der Blockchain kann nicht mehr verändert werden. Sie erlaubt es nur neuen Inhalt hinzuzufügen. Dadurch können gespeicherte Zertifikate nachträglich nicht mehr verändert werden, da sie mit der Transaktion, die das Einspeichern verursacht hat, für immer in der Blockchain persistiert sind.

4.1.4 SPSC-Schnittstellen

Damit man mit dem Smart-Contract interagieren kann, muss dieser Schnittstellen nach außen anbieten, die im Rahmen einer Transaktion aufgerufen werden können. Auch wenn ein Smart-Contract technisch gesehen keinen Besitzer hat, sondern ein eigener Account ist, wird im Folgenden der Sicherheitsprüfer, der den SPSC deployt hat, als Besitzer bezeichnet. Die Zugriffsrechte auf Funktionen müssen vom Besitzer des SPSCs explizit implementiert werden.

Im Folgenden werden die Schnittstellen des Smart-Contract beschrieben.

Objektmengen

- **Address**: Repräsentiert die Ethereum Adresse.
- **Time**: Repräsentiert die Zeit.
- **Boolean**: True oder False.
- **String**: Eine beliebige Zeichenkette.
- **Certificate**: Tupel in der Form (Address, Address, Time, Boolean, String). Repräsentiert das Sicherheitszertifikat aus 4.1.1.

In dieser Darstellung wird das *Certificate* Objekt ohne die Information der Identität des Prüfers und des Subjekts dargestellt, da es hier um die technische Schnittstelle geht, wie sie im Smart-Contract zu implementieren ist. Die Herkunft des Zertifikats ist implizit durch die signierte Transkation gegeben, da der Aufruf der Schnittstellen immer mit einer Transkation verbunden ist.

addCertificate

Diese Funktion kann nur vom Besitzer des Smart-Contracts ausgeführt werden. Sie bekommt die Informationen eines Sicherheitszertifikats als Input und speichert diese persistent.

Signatur: *addCertificate(Certificate certificate)*

Input

- **certificate**: Das Zertifikat, das gespeichert werden soll.

getCertificate

Diese Funktion kann von jedem Ethereum-Account aufgerufen werden. Sie bekommt als Input eine Ethereum Adresse und liefert das aktuellste Zertifikat, das zu der Adresse ausgestellt wurde.

Signatur: *getCertificate(Address dest)* → Certificate

Input

- **dest**: Die Adresse des Gerätes, dessen aktuellstes Sicherheitszertifikat man haben möchte.

Output

- **Certificate**: Das aktuellste Zertifikat zu der angegeben Adresse

4.2 Zugangskontrollen

4.2.1 Policies

Im Folgenden wird erläutert, welche neuen Policies von einer Zugangskontrolle eingesetzt werden können.

Sicherheitsprüfer

Der RO kann eine Liste von Sicherheitsprüfern definieren, dessen Zertifikate akzeptiert werden. Da die Zertifikate von Prüfern auf deren SPSCs liegen und dort verwaltet werden, muss die Zugangskontrolle nur die Adresse der SPSCs kennen. Da die SPSCs vom Sicherheitsprüfer selbst erstellt worden sind und nur dieser Zertifikate in den SPSC einspeichern kann, ist die Herkunft aller Zertifikate im SPSC bestätigt. So kann die Zugangskontrolle davon ausgehen, dass beim Abruf eines Zertfikates kein gefälschtes Zertifikat geliefert wird.

Datum des Sicherheitszertifikats

Der RO kann in seiner Zugangskontrolle definieren, wie alt ein Sicherheitszertifikat maximal sein darf, um noch akzeptiert zu werden. Wenn dieser nur ein Monat alt sein darf, das Subjekt jedoch zuletzt vor zwei Monaten geprüft wurde, wird der Zugang unabhängig vom Status des Zertifikats nicht erteilt.

Sicherheitszustand

Der RO kann nach dem Status des Geräts prüfen. Der Status ist entweder *TRUE*, wenn das Gerät sicher ist, oder *FALSE*, wenn der Test Sicherheitslücken, Viren etc. festgestellt hat. In Zukunft kann hier eine Struktur entworfen werden, mit der man auch komplexere Zustände als Policy definieren kann. Da in dieser Arbeit von einem Prüfprozess ausgegangen wird, der nur nach Viren untersucht, reicht ein einfacher *boolean*-Flag mit *true* oder*false*.

4.2.2 Zugangsanfrage

Nachdem ein Prüfer das Zertifikat in die Blockchain speichert, kann zu jederzeit darauf zugegriffen werden. Da die Zugangskontrolle bei einer Zugangskontrolle nicht weiß, welches Zertifikat es abrufen soll und von welchem Prüfer, muss das Subjekt, das die Anfrage stellt, die nötigen Informationen der Zugangskontrolle übergeben. Die Informationen müssen enthalten:

- Die Adresse des SPSC, welches das Zertifikat hält.
- Einen Identifikator, um das richtige Zertifikat zu prüfen.

Der Identifikator ist deshalb wichtig, weil ein Subjekt viele Zertifikate haben kann, aber ein Zertifikat nur einem Subjekt zugeordnet ist. Der Identifikator kann z.B. der Zeitpunkt der Ausstellung sein oder ein anderer eindeutiger Wert. In der weiteren Arbeit wird als Identifikator der Zeitstempel des Zertifikats genommen.

4.3 Architektur des Systems

4.3.1 Kontextsicht

Wie sich der neue Smart-Contract SPSC in die vorhanden Struktur einordnet, wird in Abbildung 4.2 gezeigt. Die grün markierten Elemente sollen die neuen Funktionalitäten

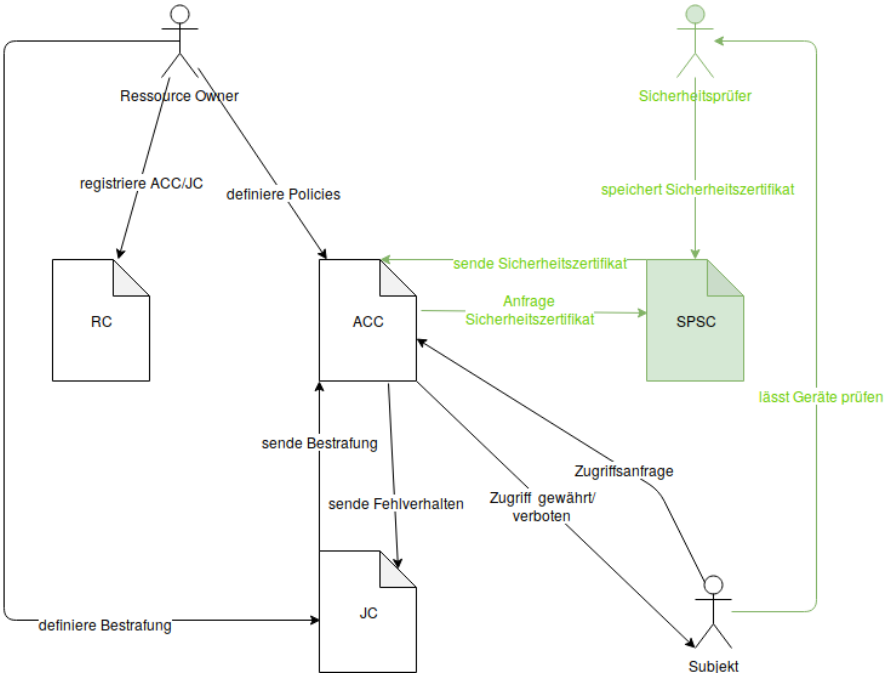


Abbildung 4.2: Erweiterte Kontextsicht des Systems

des Systems darstellen. Das Subjekt lässt seine Geräte beim Sicherheitsprüfer prüfen. Der Prüfer speichert das Ergebnis (Sicherheitszertifikat) in der Blockchain mithilfe des

SPSC. Ein ACC kann dann beim Zugriffskontrollprozess das Sicherheitszertifikat des anfragenden Gerätes über den SPSC prüfen.

4.3.2 Laufzeitsicht

Die Abbildung 4.3 zeigt das Prüfen eines Gerätes und das Speichern des daraus resultierenden Zeritifikates in der Blockchain. Der Besitzer eines Gerätes, der auf eine geschütze

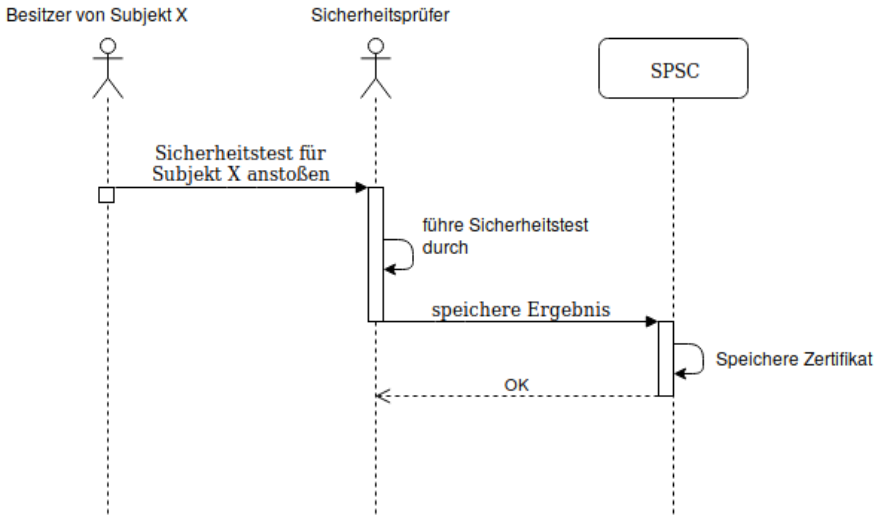


Abbildung 4.3: Laufzeitsicht: Erstellung von Sicherheitszertifikaten

Ressource zugreifen möchte, lässt das Gerät, das den Zugriff durchführen soll, zunächst bei einem Sicherheitsprüfer testen. Dieser speichert das Ergebnis über den SPSC in die Blockchain.

Die Abbildung 4.4 zeigt, wie eine Zugriffsanfrage verarbeitet wird. Zunächst wird vom Subjekt die Adresse des ACCs, das den Zugriff auf die Ressource schützt, ermittelt. Dazu fragt man beim RC nach, welches alle ACCs registriert hat und wie man diese erreicht. Dieser Schritt muss nicht ausgeführt werden, wenn dem Subjekt die Adresse des ACCs bereits bekannt ist. Anschließend wird die Zugriffsanfrage an den ACC gesendet, indem die dafür zuständige Funktion des Smart-Contracts aufgerufen wird. Hier muss die Ethereum-Adresse des Smart-Contracts mitgesendet werden, von der man das Sicherheitszertifikat erhält. Der ACC prüft, ob die statischen Policies erfüllt sind und führt dann die dynamische Kontrolle durch. Wird ein Fehlverhalten dabei erkannt, wird der JC benachrichtigt. Dieser bewertet das Fehlverhalten und legt eine Sanktionierung fest, die

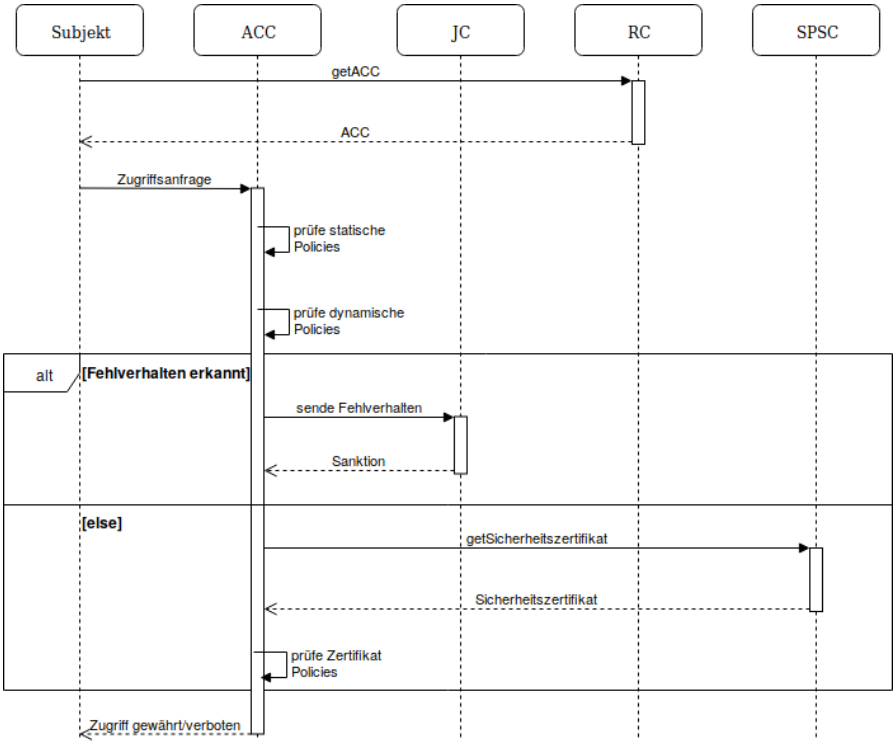


Abbildung 4.4: Laufzeitsicht: Zugriffskontrolle

er dem ACC mitteilt. Die Sanktionierung wird vom ACC an das Subjekt gesendet und damit der Zugriff verboten. Wird kein Fehlverhalten erkannt, holt der ACC das letzte Sicherheitszertifikat, das vom Sicherheitsprüfer ausgestellt wurde und prüft die Policies gegen das Zertifikat. Dafür wird zunächst anhand der Adresse des Sicherheitsprüfers, die das Subjekt mitgesendet hat, geprüft, ob dieser akzeptiert wird. Danach wird geprüft, ob das zuletzt ausgestellte Zertifikat maximal in einer vom Ressource Owner definierten Zeit liegt. Anschließend wird noch geschaut, welches Ergebnis das Zertifikat hat. Also, ob es aussagt, dass das Gerät sicher ist oder nicht. Zuletzt teilt der ACC dem Subjekt das Ergebnis mit, ob der Zugang erteilt wird oder nicht.

5 Umsetzung

In diesem Kapitel wird die Umsetzung des Systems erläutert. Dabei wird hauptsächlich die Umsetzung des erweiterten Systems aufgezeigt, ohne auf die Umsetzung des Grund-Frameworks ([Zhang u. a., 2019]) einzugehen. Zuerst wird auf Events eingegangen, die vom System erzeugt werden. Anschließend wird erklärt, wie die Funktionalitäten des Systems implementiert wurden.

Die Smart-Contracts werden in der Programmiersprache *Solidity* [Solidity] implementiert und als Blockchain wird die Ethereum-Blockchain benutzt. Als Entwicklungsumgebung wird die Online-IDE Remix [Remix] verwendet, mit der man Smart-Contracts in der Sprache Solidity entwickeln kann.

5.1 Events

Events sind in Ethereum gleichzustellen mit Logs. Mit Events kann man beliebige Aktivitäten als permanente Logs festhalten, die für immer in der Blockchain gespeichert werden. Anwendungen außerhalb von Ethereum, können diese Logs lesen und sie verarbeiten. Die folgenden Logs werden vom System erzeugt.

UpdateTester(addressTester, valid)

Wird beim Hinzufügen eines Sicherheitsprüfers ausgelöst.

- **addressTester:** Die Adresse des Sicherheitsprüfers.
- **valid:** Ein Boolean der *true* ist, wenn der Prüfer hinzugefügt worden ist und *false*, wenn dieser entfernt wurde.

AddCertificate(addressSubject, time, result, notes)

Wird beim Hinzufügen eines Zertifikats ausgelöst.

- **addressSubject:** Die Adresse des geprüften Gerätes.

- **time:** Der Zeitpunkt der Ausstellung. Wird in Unix-Format angegeben.
- **result:** Ein Boolean der *true* ist, wenn das Gerät sicher ist. Andernfalls *false*.
- **notes:** Ein String der zusätzliche Informationen zum geprüften Gerät enthalten kann.

5.2 Implementation

5.2.1 Zugriffsberechtigung auf Funktionen

Einige Funktionen, wie z.B. das Ausstellen von Zertifikaten, Hinzufügen von Policies oder Hinzufügen von validen Sicherheitsprüfern, dürfen nur von bestimmten Accounts durchgeführt werden. So darf das Subjekt z.B. keine eigenen Policies in einem ACC definieren. Das Beschränken des Zugriffs muss explizit vom Ersteller des Samrt-Contracts implementiert werden. Um solche Zugriffe zu verbieten, gibt es in Solidity die Möglichkeit *modifier* zu erstellen, die dann an die Signatur der Funktion gehängt werden. Beim Aufruf der Funktion, wird dann zunächst der *modifier* ausgeführt, bevor der Inhalt der Funktion ausgeführt wird. Der folgende *modifier* soll den Zugriff auf Funktionen schützen.

```
modifier onlyBy(address _account)
{
        require(msg.sender == _account);
        _;
}
```

Der *modifier* prüft, ob der Sender der Transaktion, der Adresse entspricht, die in *account* steht.

Der Modifier kann anschließend an die Funktion gehängt werden wie im folgenden Beispiel.

```
function testerAdd(address _tester) external onlyBy(owner){
        testers[_tester] = true;
}
```

Hier wird der Rumpf der Funktion nur ausgeführt, wenn der Sender der Transaktion der *owner* ist. Alternativ könnte man, statt der *owner* Variable, direkt die Adresse des Smart-Contract Besitzers schreiben.

5.2.2 SPSC

Der SPSC gehört immer zu einem Prüfer. Daher muss der Prüfer beim deployen des SPSCs sicherstellen, dass die Zugriffsrechte auf die Funktionen geregelt sind, sodass nur er selber Zertifikate hinzufügen kann. Dafür kann er den Modifier *onlyBy* (5.2.1) nutzen. Der SPSC speichert in einer Map alle ausgestellten Zertifikate. Die Map hat zwei *keys*, um ein Zertifikat zu finden. Damit hat sie die Form:

```
address => (time => certificate)
```

Der erste *key* ist die Adresse des Gerätes und der zweite *key* ist das Ausstellungsdatum des Zertifikats. Alternativ könnte man das Ausstellungsdatum als *key* weglassen und die Form

```
address => [certificate_1, certificate_2, ..., certificate_N]
```

realisieren. Hier zeigt der *key* auf alle Zertifikate, die an die Adresse adressiert sind. Das Problem ist, dass das Ausführen von Smart-Contracts Gaskosten verursachen, die der Sender einer Transaktion zahlen muss. Da beim Suchen eines Zertifikats in der Liste ein unvorhersehbarer Aufwand entsteht und das Überprüfen jedes einzelnen Zertifikats weitere Gaskosten verursacht, werden zwei *keys* bestehend aus der Adresse und dem Ausstellungsdatum eingesetzt. Die Kombination von beiden zeigt auf ein eindeutiges Zertifikat, da der Prüfer keine zwei Zertifikate zur selben Zeit an die gleiche Adresse ausstellen kann [1].

Ein Zertifikat *certificate* hat die Struktur:

```
struct Certificate{
        uint time;
        bool result;
        string notes;
}
```

[1] Dies wird ebenfalls durch die Implementation der Funktion *addCertifcate* vorgegeben.

Der SPSC muss zwei Funktionen implementieren, die öffentlich zugänglich gemacht werden. Die erste Funktion dient dem Veröffentlichen von Zertifikaten. Diese Funktion kann nur vom Prüfer aufgerufen werden, dem der SPSC gehört. Die Signatur sieht wie folgt aus:

```
function addCertificate(address _subject, uint _time,
bool _status, string _notes) external onlyBy(owner)
```

Die Funktion prüft erst, ob schon ein Zertifikat mit dem Zeitstempel *time* für das Subjekt existiert. Ist dies nicht der Fall, wird das Zertifikat gespeichert und ein *AddCertificate* Event wird ausgelöst. Andernfalls wird ein Error geworfen.

Die zweite Funktion ist das Abrufen von Zertifikaten. Diese Funktion kann von jedem aufgerufen werden, um ein Zertifikat zu einem Subjekt zu erhalten. Dabei wird der Zustand der Blockchain nicht geändert, also keine Variablen neu belegt. Die Signatur sieht wie folgt aus:

```
function getCertificate(address _address, uint _time)
external constant returns (bool, uint, bool, string)
```

Als Input bekommt die Funktion die beiden *keys* zum Identifizieren des Zertifikats. Die Ausgabe ist ein vierer Tupel. Der erste Wert des Tupels, gibt an, ob das Zertifkat existiert oder nicht. Die restlichen drei repräsentieren das Zertifikat, jeweils mit dem Zeitstempel, dem Status und Notizen.

5.2.3 Verwaltung von Sicherheitsprüfern

Der Ressource Owner kann festlegen, welchen Sicherheitsprüfern er vertraut und somit dessen Sicherheitszertifikate akzeptiert. Dafür wird der ACC um eine weitere Funktion erweitert, die das Hinzufügen von Sicherheitsprüfern umsetzt. Konkret werden die Ethereum Adressen der SPSCs gespeichert. Die Adressen werden in einer *Map* gespeichert, da der Zugriff effizienter ist als das Iterieren über eine Liste. Der *key* ist die Ethereum Adresse des Prüfers und der *value* ist *TRUE*, wenn dieser akzeptiert wird vom Ressource Owner. Andernfalls ist dieser Wert *FALSE*. Die Signatur in Solidity sieht wie folgt aus:

```
function testerAdd(address _tester) external onlyBy(owner)
```

Außerdem wird eine Methode implementiert, die das Löschen von verifizierten Prüfern ermöglicht. Auch diese Funktion darf nur von dem Besitzer des ACCs, also dem Ressource Owner, ausgeführt werden. Die Signatur sieht wie folgt aus:

```
function testerDelete(address _tester) external onlyBy(owner)
```

Zuletzt muss noch eine Methode implementiert werden, mit der man sehen kann, ob ein Prüfer anerkannt ist oder nicht. Dazu wird eine einfache Methode *isValidTester(Address adress)* implementiert, die zur Adresse eines Prüfers *TRUE* liefert, wenn der Ressource Owner die Adresse hinzugefügt hat. Diese Funktion hat keine Zugriffseinschränkungen, sodass jeder sehen kann, welche Prüfer akzeptiert werden.

```
function isValidTester(address _tester) external constant
returns (bool)
```

5.2.4 Sicherheitszertifikat als Policy

Damit ein Sicherheitszertifikat gegen Policies geprüft werden kann, wird ein Feld *ToLC* (Time of Last Certificate) eingeführt, das der Ressource Owner defnieren kann. Dieser gibt an wie alt ein Zertifikat maximal sein darf. Die Signatur der Funktion sieht wie folgt aus:

```
function setToLC(uint _time) external onlyBy(owner)
```

Ethereum nutzt für die Zeit das Unix-Format. Die Zeit wird also in Anzahl an **Sekunden** ab 1970-01-01 gespeichert. Der Wert von *time* muss also in Sekunden angegeben werden. Möchte der Ressource Owner als Policy definieren, dass ein Zertifikat nur maximal eine Woche alt sein darf, muss er als time den Wert *604800* angeben. Die Funktion muss ausßerdem prüfen, ob der Wert nicht negativ ist, da es keine negativ verstrichene Zeit gibt.

Für das Feld gibt es ebenfalls einen einfachen *Getter*, damit jeder sehen kann, wie alt ein Zertifikat maximal sein darf. Der Output ist die Zeit in Sekunden. Der Getter hat die folgende Signatur:

```
function getToLC() external constant returns (uint)
```

5.2.5 Zugriffskontrolle

Die definierte Policy muss nun in der Zugangskontrolle auch eingesetzt werden. Dazu wird die bereits vorhandene *accessControl()* Methode erweitert.

Die erste Erweiterung die hinzu kommt, ist ein Paramter, den das Subjekt der Funktion übergibt. Das Subjekt teilt dem Smart-Contract mit, an welcher Ethereum Adresse das Zertifikat zu finden ist. Zuerst wird die statische und dann die dynamische Kontrolle durchgeführt. Anschließend prüft der Smart-Contract, ob der angegebene Prüfer anerkannt wird. Dazu wird geschaut, ob der Ressource Owner, diesen hinzugefügt hat oder nicht. Hat der Ressource Owner den Prüfer nicht hinzugefügt, dann ist die Zugangskontrolle beendet und das Subjekt erhält keinen Zugriff. Ist der Prüfer anerkannt, wird die *getCertificate* Methode des angegeben SPSCs aufgerufen, um das zuletzt ausgestellte Zertifikat abzurufen. Die Struktur des Zertifikats hat die Form (time, result, notes). *time* ist das Datum der Ausstellung, *result* ist *TRUE*, falls das geprüfte Gerät sicher ist und *notes* sind Hinweise, die der Aussteller hinzugefügt hat. Die Notes haben zunächst keinen Einfluss auf die Zugangskontrolle, können aber in zukünftigen Arbeiten genutzt werden, um weitere Kontexinformationen zu geben, die geprüft werden sollen. Nun wird das Datum des Zertifikats mit dem *ToLC* verglichen. Wenn das Datum der Ausstellung weiter zurückliegt als die ToLC erlaubt, dann wird der Zugang nicht erteilt. Zuletzt wird noch das Ergebnis des Zertifikats geprüft. Ist dieser *TRUE*, dann wird der Zugang erteilt. Andernfalls wird der Zugang nicht gewährt.

6 Evaluation

Der in dieser Arbeit entwickelte Prototyp soll in einem privaten Ethereum-Netzwerk zum Einsatz kommen, um die Funktionalitäten zu testen.

6.1 Testumgebung

Für das Testen der Funktionalitäten soll ein privates Ethereum-Netzwerk erstellt werden. Zum Vereinfachen wurde die dynamische Kontrolle aus dem ACC entfernt, damit auch der damit verbundene JC entfällt. Die dynamische Kontrolle hat keinen Einfluss auf das System das im Folgenden getestet werden soll. Da sie lediglich zusätzliche Policies enthalten, hat es keinen Einfluss auf das neue System.

6.1.1 Client

Für das Teilhaben an einem Netzwerk, wird ein Client benötigt, der nach den Konsensregeln von Ethereum arbeitet. Hier wurde die Software Geth [Geth] genutzt. Der Sourcecode wurde so geändert, dass die *difficulty*, also die Schwierigkeit des PoW konstant bleibt, sodass keine langen Mining-Zeiten auftreten.

6.1.2 Vorbereitung

Bootstrapping

Mittels *geth* wurden auf einem Rechner vier Full-Nodes (Peers) gestartet und jeweils miteinander verbunden, sodass ein Ethereum-Netzwerk entsteht. Auch wenn die Nodes

auf demselben Rechner laufen, laufen sie unabhängig voneinander und haben alle eine eigene Kopie der Blockchain. Die Nodes kommunizieren über RLPx [RLPx], einem Transportprotokoll, dass auf TCP basiert.

Die Nodes sollen jeweils, den RO, das Subjekt, das Objekt und den Prüfer repräsentieren.

EOAs

Die folgenden EOAs (externally owned account) wurden erzeugt:

RO
: Der RO ist derjenige, der die Zugangskontrolle nutzt, um seine Ressourcen zu schützen. Er ist der Besitzer des ACC.
Adresse: 0xbed1222446db6bdb2ade44f9d472a2ee68e5f628

Objekt
: Das Objekt gehört dem RO und soll durch die Zugangskontrolle geschützt werden.
Adresse: 0x14f8460658d3c0bb56a3cd7462653d216cbd059d

Subjekt
: Das Subjekt ist das Gerät, das versucht auf das Objekt zuzugreifen. **Adresse: 0xbacae6ebe39523d429c41699378350a0f60516f1**

Prüfer
: Der Prüfer prüft die Subjekte und veröffentlicht dann das Ergebnis als Zertifikat.
Adresse: 0xb2546bf3fd9812b189bc858eff787c7b230a2069

Mit jedem EOA wurde zunächst eine Menge ETH „geschürft“. Ohne ETH können die Accounts keine Transaktionen senden.

Smart-Contracts

Als nächstes wurde mit dem Account des Prüfers der SPSC deployt. Dieser ist an der Adresse „**0xef05133f4d8eb9d2b87c8ec11fd5bb77eff77a7c**“ zu finden.

Mit dem Account vom RO wurde ein ACC deployt, welches an der Adresse „**0x27b31277a0395a4149e37d32101453da577ef5d0**“ liegt. Ein ACC ist immer einem Subjekt-Objekt Paar zuzuorden, da es den Zugriff von einem einzigen Subjekt auf ein einziges Objekt schützt. Als Subjekt-Objekt Paar wurden die oben beschriebenen Adressen in den Smart-Contract implementiert.

Events

Zusätzlich wurde ein weiterer Node aufgesetzt mit dem Account „**0x66d7139efe28ec0a5ace972397e3bfc17b93ac65**“. Dieser Knoten dient als Beobachter, der die Events abgreift. Der Beobachter repräsentiert eine Anwendung, die über die Ereignisse informiert werden will. So kann der Beobachter ein Subjekt sein, der das Ergebnis einer Zugriffsanfrage erhält, oder auch das Objekt, das ebenfalls das Ergebnis der Zugangskontrolle mitgeteilt bekommt.

Für das Empfangen von Events wurden, für die Smart-Contracts, die Events erzeugen, ein Python-Script geschrieben. Dieses Script ist mit einer Node - in dem Fall der Beobachter Node - verbunden und durchläuft eine Endlosschleife. Dabei prüft sie jedesmal, ob von der Adresse eines Smart-Contracts ein Event ausgelöst wurde. Wurde ein Event ausgelöst, dann wird das Event einfach auf der Konsole ausgegeben. Abbildung 6.1 zeigt wie das *AddCertificate* Event (siehe 5.1) aussieht. Das Event enthält zunächst die Informationen,

```
AttributeDict({'args': AttributeDict({'_addressSubject': '0xBAcae6Ebe39523D429c4
1699378350A0f60516f1', '_time': 1554926400, '_result': True, '_notes': 'test cer
tificate'}), 'event': 'AddCertificate', 'logIndex': 0, 'transactionIndex': 0, 't
ransactionHash': HexBytes('0x006bcfb5d21c2ccae8cc000627c0a57a00acf9593b06c17b1f5
8cd7f34ffad8b'), 'address': '0xEf05133F4D8eb9d2b87C8eC11Fd5bB77EfF77a7c', 'block
Hash': HexBytes('0xe86bd2b5606967ce7f104a06c73737afe50c94c34cd901f1d7c93170281a7
fed'), 'blockNumber': 16})
```

Abbildung 6.1: Inhalt eines Events

die explizit vom Smart-Contract eingefügt worden sind, also die Adresse des Subjekts, der Zeitstempel im Unix-Format, der Status des Gerätes und beliebige Notizen. Das Event enthält zudem noch weitere Metadaten, z.B. der Hash der Transaktion, welche das Event ausgelöst hat, der Hash des Blocks, in dem sich die Transaktion befindet und die Nummer des Blocks, in der sich das Event und die Transaktion befindet.

6.2 Funktionale Tests

Die folgenden Tests sollen die Funktionalität sowie die Sicherheit des Systems testen.

6.2.1 Verwaltung von Sicherheitszertifikaten

Der Test soll zeigen, dass nur der Prüfer, der einen SPSC erstellt hat, Zertifikate an den SPSC senden kann.

Testszenario - Erfolgreich

Als erstes soll getestet werden, ob der Prüfer die Berechtigung hat, Zertifikate in den SPSC zu speichern. Dazu wurde die ***addCertificate(subjectAddress, time, status, notes)*** Methode des SPSC, mit den folgenden Werten, vom Account des Prüfers aufgerufen:

- subjectAddress: <Die Adresse des Subjekts (siehe oben)>
- time: 1554926400
- status: true
- notes: test certificate

Abbildung 6.1 zeigt die Ausgabe des Beobachters. Das Ergebnis zeigt die gleichen Werte wie die Eingabewerte. Die Transaktion wurde also erfolgreich durchgeführt und das Zertifikat ist im SPSC gespeichert. Ein Abruf des Zertifikats (Abbildung 6.2) bestätigt dies.

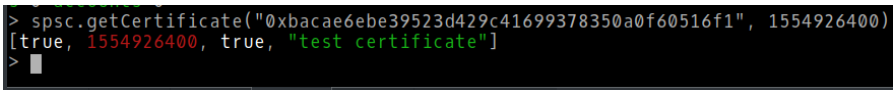


Abbildung 6.2: Abruf des Zertifikats.

Testszenario - Fehlschlag

Das zweite Testszenario soll zeigen, dass das Hinzufügen von einem anderen Account als dem des Prüfers zu einem Fehler führt und das Zertifikat nicht gespeichert wird. Dafür wird vom Account des RO die Transaktion gestartet mit den gleichen Werten wie im ersten Szenario. Lediglich der Zeitstempel wurde geändert, da der Zeitstempel zusammen mit der Adresse des Subjekt den eindeutigen Schlüssel bilden. Die folgenden Werte wurde benutzt:

- subjectAddress: <Die Adresse des Subjekts (siehe oben)>
- time: 1554926123
- status: true
- notes: test certificate

Nachdem der nächste Block mit der Transaktion geschürft wurde, hat der Beobachter keine Benachrichtigung erhalten. Ein Blick in den sogenannten „Transaction-Receipt“ aus Abbildung 6.3 zeigt, dass keine Logs (Events) erzeugt wurden. Außerdem ist der status=0x0, was darauf hinweist, dass die Transaktion mit einem Error beendet wurde.


```
{
  blockHash: "0x85e8e8754ade98afde6c4d94060efe03ac485823ce3c11ed4cbe48d063a1032d
",
  blockNumber: 17,
  contractAddress: null,
  cumulativeGasUsed: 25562,
  from: "0xbed1222446db6bdb2ade44f9d472a2ee68e5f628",
  gasUsed: 25562,
  logs: [],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000",
  status: "0x0",
  to: "0xef05133f4d8eb9d2b87c8ec11fd5bb77eff77a7c",
  transactionHash: "0x9de2a22543bd0e79e68c3e9ca7539a0b9c2a5fcddcf3f8f4f061ec5117
444c62",
  transactionIndex: 0
```


Abbildung 6.3: Transaction Receipt

Der Versuch, das Zertifkat abzurufen scheitert, wie man in Abbildung 6.4 sehen kann.

```
> spsc.getCertificate("0xbacae6ebe39523d429c41699378350a0f60516f1", 1554926123)
[false, 0, false, ""]
```

Abbildung 6.4: Abruf des Zertfikats

Resultat

Das Ergebnis zeigt, dass nur der Prüfer die Berechtigung hat, Zertifikate hinzuzufügen. Dadurch ist beim Abruf eines Zertifikats, die Herkunft implizit nachgewiesen. Außerdem sind dadurch gefälschte Zertfikate ausgeschlossen, da der SPSC nur Zertifkate speichert, die vom Prüfer erstellt wurden.

6.2.2 Verwaltung der Policies

Der Test soll zeigen, dass nur der RO die Policies für die Sicherheitskontrolle definieren kann.

Testszenario - Erfolgreich

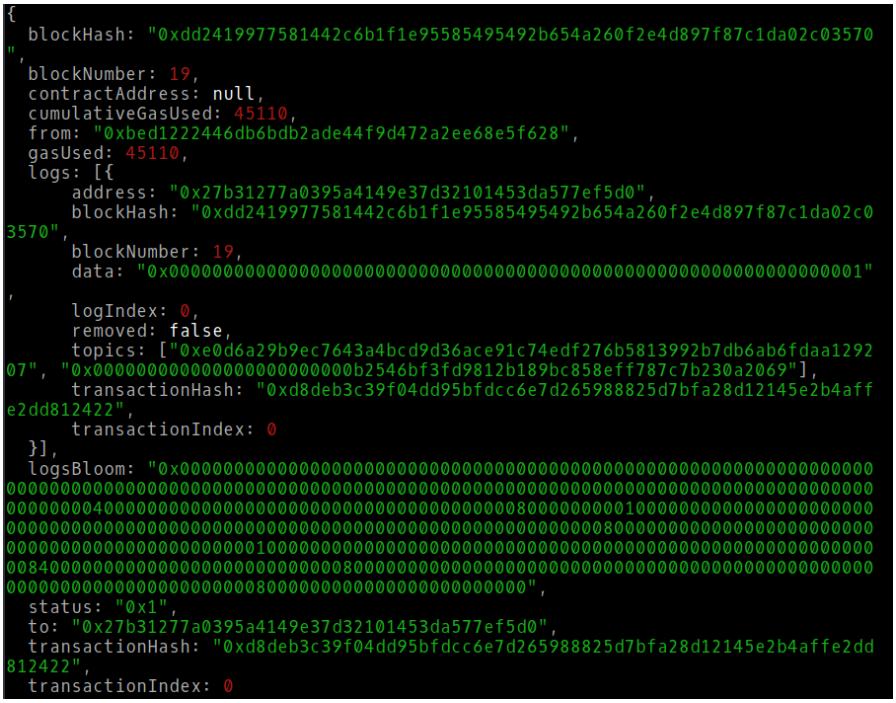
Der erste Test soll zeigen, dass der Besitzer des ACC die Policies hinzufügen und verändern kann.

Zunächst wird die Adresse des SPSC hinzugefügt. Abbildung 6.5 zeigt, dass der Status der Transaktion 0x1, also „OK“ ist.

Der Beobachter erhält dabei die Nachricht aus Abbildung 6.6

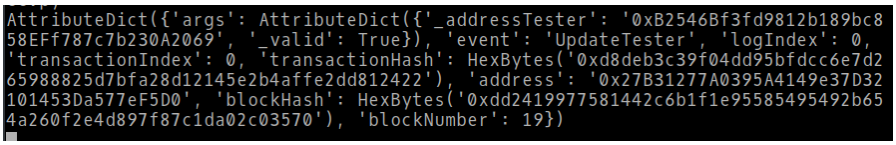
Beim Prüfen, ob ein SPSC von der Zugangskontrolle akzeptiert wird (Abbildung 6.7), wird ein *true* geliefert.

Als nächstes wird das Ablaufdatum eines Zertifikats auf 1 Woche (604800 im Unix-Format) gesetzt. Auch hier sieht man anhand des Status in Abbildung 6.8, dass die Transaktion erfolgreich durchgeführt wurde. Abbildung 6.9 bestätigt, dass die Zeit gestzt wurde.



```
{
  blockHash: "0xdd2419977581442c6b1f1e95585495492b654a260f2e4d897f87c1da02c03570
",
  blockNumber: 19,
  contractAddress: null,
  cumulativeGasUsed: 45110,
  from: "0xbed1222446db6bdb2ade44f9d472a2ee68e5f628",
  gasUsed: 45110,
  logs: [{
      address: "0x27b31277a0395a4149e37d32101453da577ef5d0",
      blockHash: "0xdd2419977581442c6b1f1e95585495492b654a260f2e4d897f87c1da02c0
3570",
      blockNumber: 19,
      data: "0x0000000000000000000000000000000000000000000000000000000000000001"
,
      logIndex: 0,
      removed: false,
      topics: ["0xe0d6a29b9ec7643a4bcd9d36ace91c74edf276b5813992b7db6ab6fdaa1292
07", "0x000000000000000000000000b2546bf3fd9812b189bc858eff787c7b230a2069"],
      transactionHash: "0xd8deb3c39f04dd95bfdcc6e7d265988825d7bfa28d12145e2b4aff
e2dd812422",
      transactionIndex: 0
  }],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000040000000000000000000000000000000000000800000000010000000000000000000000
0000000000000000000000000000000000000000000000000000000800000000000000000000000
0000000000000000000001000000000000000000000000000000000000000000000000000000000
0084000000000000000000000000000800000000000000000000000000000000000000000000000
0000000000000000000800000000000000000000000",
  status: "0x1",
  to: "0x27b31277a0395a4149e37d32101453da577ef5d0",
  transactionHash: "0xd8deb3c39f04dd95bfdcc6e7d265988825d7bfa28d12145e2b4affe2dd
812422",
  transactionIndex: 0
```


Abbildung 6.5: Transaction Receipt



```
AttributeDict({'args': AttributeDict({'_addressTester': '0xB2546Bf3fd9812b189bc8
58EFf787c7b230A2069', '_valid': True}), 'event': 'UpdateTester', 'logIndex': 0,
'transactionIndex': 0, 'transactionHash': HexBytes('0xd8deb3c39f04dd95bfdcc6e7d2
65988825d7bfa28d12145e2b4affe2dd812422'), 'address': '0x27B31277A0395A4149e37D32
101453Da577eF5D0', 'blockHash': HexBytes('0xdd2419977581442c6b1f1e95585495492b65
4a260f2e4d897f87c1da02c03570'), 'blockNumber': 19})
```


Abbildung 6.6: UpdateTester Event

Testszenario - Fehlschlag

Dieser Test soll zeigen, dass ein anderer Account als der des RO, keine Berechtigungen hat, Policies zu definieren.

Für diesen Test wurden die gleichen Transaktionen wie im ersten Szenario gesendet, jedoch mit anderen Paramtern. Für das Hinzufügen eines Sicherheitsprüfer, wurde die Adresse des Subjekts selbst genommen. Das Ergebnis ist in Abbildung 6.10 zu sehen. Der Status ist 0x0 und bedeutet, dass die Transaktion fehlschlug. Auch der Beobachter hat keine Benachrichtung bekommen, dass ein neuer Sicherheitsprüfer hinzugefügt worden ist.

Die Prüfung, ob der Prüfer „0xbacae6ebe39523d429c41699378350a0f60516f1“ valide ist

```
> accesscontrolmethod.isValidTester("0xb2546bf3fd9812b189bc858eff787c7b230a2069")
true
> 
```

Abbildung 6.7: Transaction Receipt


```
{
  blockHash: "0xb9f098b1e2b4757149d4caf4e6961db1267cca54c1fb6e32fa6733176fefd03c",
  blockNumber: 20,
  contractAddress: null,
  cumulativeGasUsed: 42349,
  from: "0xbed1222446db6bdb2ade44f9d472a2ee68e5f628",
  gasUsed: 42349,
  logs: [],
  logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000",
  status: "0x1",
  to: "0x27b31277a0395a4149e37d32101453da577ef5d0",
  transactionHash: "0xb502e25284897a1950af4e5228d815a36be15ae41c3ab5a02cb64fb3aac0
b4c7",
  transactionIndex: 0
}
```


Abbildung 6.8: Transaction Receipt

(Abbildung 6.11), gibt ein false zurück, was bedeutet dieser wird nicht akzeptiert. Die Policy wurde also nicht hinzugefügt.

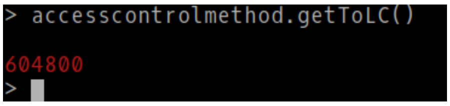
Beim Versuch die Zeitpolicy zu verändern, bekommt man ebenfalls einen Error, wie man in Abbildung 6.12 sehen kann.

Resultat

Hier sieht man anhand der Ergebnisse, dass es nicht möglich ist die Policies zu fälschen. Lediglich der RO bzw. der Besitzer des ACC, hat die Berechtiung die Policies zu verändern.

6.2.3 Zugangskontrolle

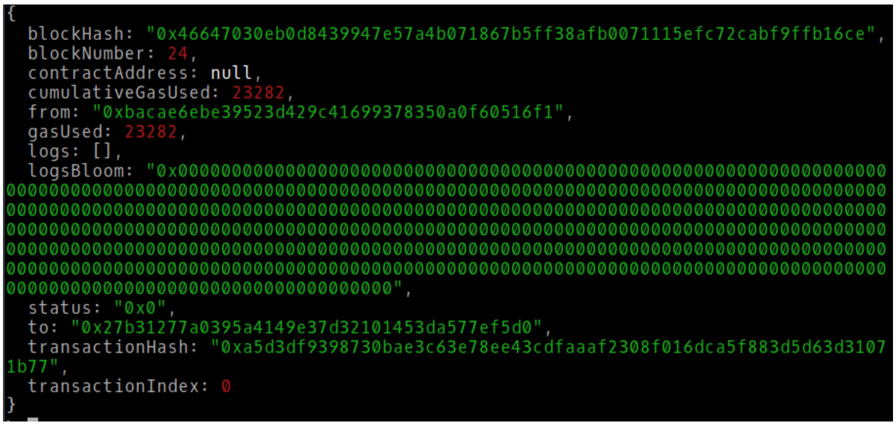
Dieser Test soll zeigen, dass die Zugriffskontrolle korrekt arbeitet und die Policies richtig eingestzt werden.



```
> accesscontrolmethod.getToLC()

604800
> 
```

Abbildung 6.9: Abruf der Zeitpolicy




```
{
  blockHash: "0x46647030eb0d8439947e57a4b071867b5ff38afb0071115efc72cabf9ffb16ce",
  blockNumber: 24,
  contractAddress: null,
  cumulativeGasUsed: 23282,
  from: "0xbacae6ebe39523d429c41699378350a0f60516f1",
  gasUsed: 23282,
  logs: [],
  logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000",
  status: "0x0",
  to: "0x27b31277a0395a4149e37d32101453da577ef5d0",
  transactionHash: "0xa5d3df9398730bae3c63e78ee43cdfaaaf2308f016dca5f883d5d63d3107
1b77",
  transactionIndex: 0
}
```


Abbildung 6.10: Transaction Receipt - Status ist 0x0, da es einen Error gab

Testszenario - Erfolgreich

Hier soll gezeigt werden, dass bei korrekten Zertifikaten der Zugang gewährt wird, wenn Subjekt die Anfrage startet. Zunächt wird die Policy wie folgt definiert:

- Der SPSC <Adresse des SPSC siehe oben> wird vom RO hinzugefügt, damit die von ihm zur Verfügung gestellten Zertifikate akzeptiert werden.
- Die maximale Zeit wie alt ein Zertfikat sein darf wird auf 2 Wochen gesetzt, also 1.209.600 im Unix-Format.
- Das Zertifikat muss selbstverständlich den Status „OK“ bzw. *true* haben.

Als nächstes stellt der Prüfer dem Subjekt ein Zertifikat mit den folgenden Werten aus:

- Die Adresse muss die Adresse des Subjekts sein.
- Der Zeitstempel beträgt 1554753600, was dem 08.04.2019 um 20 Uhr entspricht.
- Der Status des Zertifikats ist *true*
- Als Notiz fügt der Prüfer hinzu: „Sicherheitstest“.

```
> acontrol.isValidTester("0xbacae6ebe39523d429c41699378350a0f60516f1")
false
>
```

Abbildung 6.11: Überprüfung, ob der Tester vom RO akzeptiert wird oder nicht.


```
{
  blockHash: "0x88a34957c07c9ccf2d6c4f3308993f758767b4d9d1615936d0ca8a60fc40e4ce",
  blockNumber: 25,
  contractAddress: null,
  cumulativeGasUsed: 22298,
  from: "0xbacae6ebe39523d429c41699378350a0f60516f1",
  gasUsed: 22298,
  logs: [],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000",
  status: "0x0",
  to: "0x27b31277a0395a4149e37d32101453da577ef5d0",
  transactionHash: "0x0eb4956918cea109f0f5735102da50ad4938f1e8074b81a3c429036998a5
5c4b",
  transactionIndex: 0
}
>
```


Abbildung 6.12: Transaction Receipt - Status ist 0x0, da es einen Error gab

Nun muss das Subjekt die Zugriffskontrolle starten. Dabei ruft er die dafür zuständige *accessControl* Funktion des ACCs auf und übergibt die folgenden Parameter:

- Die Adresse des SPSC, welches das Zertifikat hält.
- Der Zeitstempel des Zertifikats

Mit der Kombination aus den beiden Parametern, kann die Zugangskontrolle das Zertifikat eindeutig identifizieren, da jede Adresse einzigartig ist und ein Zertifikat nicht zweimal zur selben Zeit vom Prüfer ausgestellt werden kann.

Das Ergebnis sehen wir beim Beobachter, der das Event (Abbildung 6.13) als Benachrichtigung bekommen hat.


```
AttributeDict({'args': AttributeDict({'_from': '0xBAcae6Ebe39523D429c41699378350A0
f60516f1', '_errmsg': 'Access authorized!', '_result': True}), 'event': 'ReturnAcc
essResult', 'logIndex': 0, 'transactionIndex': 0, 'transactionHash': HexBytes('0xb
b251b42e335fabddd7b8df7f8bd1c7f7a818fabe3c2393d7122677d66338c5a'), 'address': '0x2
7B31277A0395A4149e37D32101453Da577eF5D0', 'blockHash': HexBytes('0xe8635540229c28f
d63c3ca4c2fb29d5fe8ecf06f1548e4ce454e2ca7a0820ef9'), 'blockNumber': 31})
```


Abbildung 6.13: ReturnAccessResult Event - Die Zugriffskontrolle war erfolgreich und der Zugriff wurde gewährt.

In einem realen Beispiel müssen sowohl Subjekt als auch Objekt die Events des ACCs

beobachten, um das Ergebnis der Zugangskontrolle zu erfahren. Ein Problem ist dabei die Dauer der Zugangskontrolle. Die Benachrichtigung wird erst versendet, wenn die Transaktion, welche die Zugangskontrolle ausgelöst hat, erfolgreich in einen Block „geschürft“ wurde. Das Finden von neuen validen Blöcken in Ethereum dauert im Durchschnitt fünfzehn Sekunden, was je nach Anwendung und dessen Anforderungen zu hoch sein kann. Dafür hat man enorme Sicherheitsvorteile gegenüber anderen Zugangskontrollen, die nicht auf Blockchain basieren.

Testszenario - Fehlschlag

In diesem Test soll gezeigt werden, dass das Subjekt nicht ohne ein korrektes Zertifikat zugelassen werden kann.

Als erstes soll gezeigt werden, dass das Subjekt einen validen SPSC angeben muss, der vom RO akzeptiert wird. Gibt das Subjekt einen falschen SPSC an, wird das Event aus Abbildung 6.14 ausgelöst.

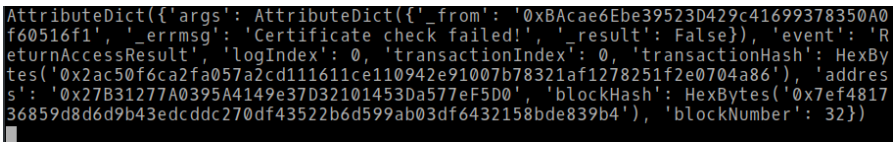


Abbildung 6.14: ReturnAccessResult Event - Die Zugriffskontrolle war nicht erfolgreich.

Die Funktion ist allerdings trotzdem erfolgreich terminiert und hat keine Error geworfen, wie man in Abbildung 6.15 sehen kann. Würde man einen Error erzeugen lassen, könnte man keine Events auslösen und es bliebe unbekannt, was das Ergebnis der Zugangskontrolle ist.

Einen Error würde die *accessControl* Methode auslösen, wenn ein falschen Subjekt die Zugriffskontrolle starten würde, da die Zugriffsberechtigung der Funktion nur auf das eine vom RO definierte Subjekt gilt.

In Abbildung 6.16 sehen wir das Ergebnis, wenn das Zertifikat abgelaufen ist, das Zertifikat nicht existiert oder der Status nicht ok ist.

```
{
  blockHash: "0x7ef481736859d8d6d9b43edcddc270df43522b6d599ab03df6432158bde839b4",
  blockNumber: 32,
  contractAddress: null,
  cumulativeGasUsed: 41515,
  from: "0xbacae6ebe39523d429c41699378350a0f60516f1",
  gasUsed: 41515,
  logs: [{
      address: "0x27b31277a0395a4149e37d32101453da577ef5d0",
      blockHash: "0x7ef481736859d8d6d9b43edcddc270df43522b6d599ab03df6432158bde839
b4",
      blockNumber: 32,
      data: "0x0000000000000000000000000000000000000000000000000000000000000040000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000019436572746966696361746520636865636b20666
1696c65642100000000000000",
      logIndex: 0,
      removed: false,
      topics: ["0x2a3c461514290b1711a7fc50547acb9a196e291478e1a3a03596f1c263c47681
", "0x000000000000000000000000bacae6ebe39523d429c41699378350a0f60516f1"],
      transactionHash: "0x2ac50f6ca2fa057a2cd111611ce110942e91007b78321af1278251f2
e0704a86",
      transactionIndex: 0
  }],
  logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000
0000200000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000100000000000000000000000000000000000000000001000000000000000000000001000
0000000000000000000000000000000000000004000000000000000000000000000000000000000000
8000000000000001000000000000000000000000000000000000000000000000000000080000000000
0000000000000000000008000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000",
  status: "0x1",
  to: "0x27b31277a0395a4149e37d32101453da577ef5d0",
  transactionHash: "0x2ac50f6ca2fa057a2cd111611ce110942e91007b78321af1278251f2e070
4a86",
  transactionIndex: 0
}
```


Abbildung 6.15: ReturnAccessResult Event - Die Zugriffskontrolle war nicht erfolgreich.


```
AttributeDict({'args': AttributeDict({'_from': '0xBAcae6Ebe39523D429c41699378350A0
f60516f1', '_errmsg': 'Certificate check failed!', '_result': False}), 'event': 'R
eturnAccessResult', 'logIndex': 0, 'transactionIndex': 0, 'transactionHash': HexBy
tes('0xed57622c7f34c890a88501b8cfdd52af5437ae7a5c5a856494e9310361aaf933'), 'addres
s': '0x27B31277A0395A4149e37D32101453Da577eF5D0', 'blockHash': HexBytes('0x0f152b5
6e10aa45856883b777ddd263de24ddd8780ef799a24c1b3adf0f78fcd'), 'blockNumber': 33})
```


Abbildung 6.16: ReturnAccessResult Event - Die Zugriffskontrolle war nicht erfolgreich.

Resultat

Die Ergebnisse zeigen, dass ein Subjekt die Zugriffskontrolle nicht umgehen kann. Ein falscher Prüfer wird sofort erkannt, da der RO die eindeutige Adresse der SPSCs als Policy definiert, die anerkannt sind. Da die Zertfikate in der Blockchain durch die Kombination von SPSC, Zeitstempel und der Adresse des Subjekts, identifiziert werden können, können keine falschen Zertifikate genommen werden.

6.3 Vorgenommene Vereinfachungen

Das entworfene Konzept fokussiert sich hauptsächlich auf die sichere Verwaltung der Sicherheitszertifikate und wie man sie in einer Zugangskontrolle nutzen kann, um den Zugangskontrollmechanismus zu stärken bzw. die Policies zu verstärken. Dabei wurde der vorangegange Prüfprozess auf die Sicherheit des zu prüfenden Gerätes reduziert. Also, ob das Gerät mit Viren infiziert ist oder nicht. Dementsprechend enthält das Ergebniszertifikat einen binären Wert, der angibt, ob das Gerät sicher war oder nicht. Es wurde deshalb eine einfache Struktur für das Zertifikat entworfen.

Alternativ könnte man ein Gerät auch anhand der Softwareversionen prüfen oder anderen Kriterien, die alle ein anderes Ergebnisformat erzeugen. In dieser Arbeit wurden jedoch die Policies der Zugangskontrolle auf ein sehr spezielles Format von Zertifikaten zugeschnitten, nämlich dem Zeitstempel und einem binären Wert, der angibt, ob das Gerät sicher ist oder nicht. Doch auch mit anderen Formaten ist das Konzept an sich weiterhin Kompatibel. Es gilt dann nur eine einheitliche Struktur zu entwerfen, mit der Zertifikate verschiedene Formen, wie z.B. Sicherheitsprüfung, Betriebssystemversion etc. annehmen können.

7 Fazit

Im Folgenden soll eine Zusammenfassung die Arbeit und die Ergebnisse kurz erläutern. Anschließend gibt es einen Ausblick, der beschreiben soll, wie das System um weitere Funktionen ergänzt werden kann.

7.1 Zusammenfassung

In dieser Arbeit wurde ein System konzipiert, das Sicherheitszertifikate sicher verwaltet, um sie bei Bedarf für Zugangskontrollmechanismen einzusetzen. Durch den Einsatz von Blockchain Technologien und Smart-Contracts, konnte die Verwaltung der Zertfikate einer dezentralen Anwendung überlassen werden, das nur durch sein eigenen Programmcode gesteuert wird. Durch die Transparenz der Blockchain, kann jeder den Programmcode einsehen und selbst entscheiden, ob sie dem Verwalter der Zertfikate vertrauen oder nicht. Zugriffskontrollmechanismen können ihre Policies erweitern, indem sie die Sicherheitszertifikate mit in den Entscheidungsprozess einfließen lassen. Besitzer von Geräten, die ihre Ressourcen schützen wollen, können den Status eines Zertifikats prüfen und den Zeitpunkt der Ausstellung als Policy definieren.

Das System wurde so konzipiert, dass es kaum eine Angriffsfläche bietet. Die Smart-Contracts und deren Speicher sind durch die Eigenschaften der Blockchain gegen Manipulationen geschützt. Jede Transaktion im System kann nachverfolgt werden und jeder Zugriffsversuch wird für immer in der Blockchain gespeichert. Auch die Zertifikate sind für immer in der Blockchain. Da die Blockchain dezentral läuft, sind die Zertifikate und die Zugangskontrollmechanismen zu jeder Zeit verfügbar und können nicht durch Denial-of-Service Angriffe abgeschossen werden.

7.2 Ausblick

Das vorgestellte Konzept beschreibt bisher nur die Verarbeitung von Zertifikaten, die anhand des Sicherheitsstatus ausgestellt wurden. Um einen wirklichen Vorteil in der Welt der IoT zu bringen, muss das System Zertifikate verschiedener Arten unterstützen, wie Softwareversionsnummer, Umweltverträglichkeit oder den Stand der Updates. Durch solche Erweiterungen wird das System nicht nur für Zugangskontrollen interessant, sonder für ganz neue Bereiche.

Eventuell kann das System in wirtschaftlicher Hinsicht erweitert werden, indem Zahlungen getätigt werden müssen, um an die Informationen eines Zertifikats zu gelangen. Blockchains bieten alle eine eigene Kryptowährung an, mit der man innerhalb des Netzwerkes zahlen kann. So können z.B. Smart-Contracts anhand der mit der Transaktion versendeten Menge an ETH, ein Zertifikat ausgeben.

Eine weitere Möglichkeit zur Erweiterung wäre, eine Art Reputationsstatus für die Subjekte einzuführen. So können Ressource Owner, den Zugangskontrollprozess bewerten und ein Feedback über die Subjekte geben, die ebenfalls mittels Blockchain festgehalten werden und immer abrufbar sind. Diese Reputation kann dann ebenfalls den Entscheidungsprozess von Zugangskontrollen beeinflussen.

Literaturverzeichnis

[Geth] : *Geth - a command line interface for running a full ethereum node.* – URL https://github.com/ethereum/go-ethereum/wiki/geth. – Zugriffsdatum: 2019-09-03

[Scbimpl] : *Implement access control in a simple iot system using ethereum smart contracts*

[Remix] : *Remix.* – URL https://remix.ethereum.org. – Zugriffsdatum: 2019-09-03

[RLPx] : *RLPx - a TCP-based transport protocol.* – URL https://github.com/ethereum/devp2p/blob/master/rlpx.md. – Zugriffsdatum: 2019-09-03

[Solidity] : *Solidity - an object-oriented, high-level language for implementing smart contracts.* – URL https://solidity.readthedocs.io/en/v0.4.24/index.html. – Zugriffsdatum: 2019-09-03

[Antonopoulos und Wood] Antonopoulos, A.M. ; Wood, G.: *Mastering Ethereum.* – URL https://github.com/ethereumbook/ethereumbook. – Zugriffsdatum: 2019-03-06

[Antonopoulos und Klicman 2018] Antonopoulos, Andreas M. ; Klicman, Peter: *Bitcoin & Blockchain - Grundlagen und Programmierung - Die Blockchain verstehen, Anwendungen entwickeln.* 2. Aufl. Dpunkt.Verlag GmbH, 2018. – ISBN 978-3-960-09071-7

[Ashton] Ashton, Kevin: *That 'Internet of Things' Thing.* – URL https://www.rfidjournal.com/articles/view?4986.. – Zugriffsdatum: 2019-02-01

[Atzori u. a. 2010] Atzori, Luigi ; Iera, Antonio ; Morabito, Giacomo: The Internet of Things: A survey. In: *Computer Networks* 54 (2010), Nr. 15, S. 2787 – 2805. – URL http://www.sciencedirect.com/science/article/pii/S1389128610001568. – ISSN 1389-1286

[Badev und Chen 2014] Badev, Anton ; Chen, Matthew: *Bitcoin: Technical Background and Data Analysis*. 2014

[Bogart und Rice 2015] Bogart, Spencer ; Rice, Kerry: *The blockchain report: welcome to the internet of value*. 2015

[Buterin 2014] Buterin, Vitalik: *Ethereum: A next-generation smart contract and decentralized application platform*. 2014. – URL https://github.com/ethereum/wiki/wiki/White-Paper. – Accessed: 2019-01-01

[Condos u. a. 2016] Condos, James ; Sorrell, William H. ; Donegan, Susan L.: *Blockchain Technology: Opportunities and Risks*. 01 2016

[Cruz u. a. 2018] Cruz, J. P. ; Kaji, Y. ; Yanai, N.: RBAC-SC: Role-Based Access Control Using Smart Contract. In: *IEEE Access* 6 (2018), S. 12240–12251. – ISSN 2169-3536

[Dean und Agyeman 2018] Dean, Andrew ; Agyeman, Michael O.: A Study of the Advances in IoT Security. In: *Proceedings of the 2Nd International Symposium on Computer Science and Intelligent Control*. New York, NY, USA : ACM, 2018 (ISCSIC '18), S. 15:1–15:5. – URL http://doi.acm.org/10.1145/3284557.3284560. – ISBN 978-1-4503-6628-1

[Eckert 2014] Eckert, Claudia: *IT-Sicherheit - Konzepte - Verfahren - Protokolle*. 9th updated edition. Berlin/Boston : De Gruyter Oldenbourg, 2014. – ISBN 978-3-486-77848-9

[Gartner 2017] Gartner: *Gartner Says 8.4 Billion Connected "Things"Will Be in Use in 2017, Up 31 Percent From 2016*. 2017. – URL https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016. – Zugriffsdatum: 2019-02-01

[Glaser und Bezzenberger 2015] Glaser, Florian ; Bezzenberger, Luis: *Beyond Cryptocurrencies - A Taxonomy of decentralized consensussystems*. 2015

[Hossain u. a. 2015] Hossain, M. M. ; Fotouhi, M. ; Hasan, R.: Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things. In: *2015 IEEE World Congress on Services*, June 2015, S. 21–28. – ISSN 2378-3818

[Hu u. a. 2014] Hu, Vincent ; Ferraiolo, David ; Kuhn, D ; Schnitzer, A ; Sandlin, K ; Miller, R ; Scarfone, Karen: Guide to attribute based access control (ABAC) definition and considerations. In: *National Institute of Standards and Technology Special Publication* (2014), 01, S. 162–800

[Li u. a. 2013] Li, Hongyu ; Tian, Ye ; Liu, Yang ; Li, Tingli ; Mao, Wei: UAI-IOT Framework: A Method of Uniform Interfaces to Acquire Information from Heterogeneous Enterprise Information Systems, 08 2013, S. 724–730

[Litzel] Litzel, Nico: *Was ist das Internet of Things?*. – URL https://www.bigdata-insider.de/was-ist-das-internet-of-things-a-590806/. – Zugriffsdatum: 2019-02-01

[Mattila 2016] Mattila, Juri: *The Blockchain Phenomenon - The Disruptive Potential of Distributed Consensus Architectures*. 05 2016

[Nakamoto 2009] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system*. 2009. – URL http://www.bitcoin.org/bitcoin.pdf

[Ouaddah u. a. 2016] Ouaddah, Aafaf ; Abou Elkalam, Anas ; Ait Ouahman, Abdellah: FairAccess: a new Blockchain-based access control framework for the Internet of Things. In: *Security and Communication Networks* 9 (2016), Nr. 18, S. 5943–5964. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1748

[Ouaddah u. a. 2017] Ouaddah, Aafaf ; El Kalam, Anas A. ; Ouahman, Abdellah A.: Harnessing the power of blockchain technology to solve IoT security & privacy issues. In: *ICC*, 2017, S. 7–1

[Rouse 2018] Rouse, Margaret: *IoT security (internet of things security)*. 2018. – URL https://internetofthingsagenda.techtarget.com/definition/IoT-security-Internet-of-Things-security. – Zugriffsdatum: 2019-05-01

[Sadeghi u. a. 2015] Sadeghi, A. ; Wachsmann, C. ; Waidner, M.: Security and privacy challenges in industrial Internet of Things. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, S. 1–6. – ISSN 0738-100X

[Samarati und de Vimercati 2001] Samarati, Pierangela ; Vimercati, Sabrina C. de: Access Control: Policies, Models, and Mechanisms. In: Focardi, Riccardo (Hrsg.) ; Gorrieri, Roberto (Hrsg.): *Foundations of Security Analysis and Design*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, S. 137–196. – ISBN 978-3-540-45608-7

[Sandhu und Samarati 1994] Sandhu, R. S. ; Samarati, P.: Access control: principle and practice. In: *IEEE Communications Magazine* 32 (1994), Sep., Nr. 9, S. 40–48. – ISSN 0163-6804

[Sandhu und Munawer 1998] Sandhu, Ravi ; Munawer, Qamar: How to Do Discretionary Access Control Using Roles. In: *Proceedings of the Third ACM Workshop on Role-based Access Control*. New York, NY, USA : ACM, 1998 (RBAC '98), S. 47–54. – URL http://doi.acm.org/10.1145/286884.286893. – ISBN 1-58113-113-5

[Servos und Osborn 2017] Servos, Daniel ; Osborn, Sylvia L.: Current Research and Open Problems in Attribute-Based Access Control. In: *ACM Comput. Surv.* 49 (2017), Januar, Nr. 4, S. 65:1–65:45. – URL http://doi.acm.org/10.1145/3007204. – ISSN 0360-0300

[Singh und Singh 2015] Singh, S. ; Singh, N.: Internet of Things (IoT): Security challenges, business opportunities amp; reference architecture for E-commerce. In: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, Oct 2015, S. 1577–1581

[Siriwardena 2017] Siriwardena, Prabath: *The Mystery Behind Block Time*. 2017. – URL https://medium.facilelogin.com/the-mystery-behind-block-time-63351e35603a?gi=ced020b045ad. – Zugriffsdatum: 2019-03-30

[Swan 2015] Swan, Melanie: *Blockchain: Blueprint for a New Economy*. 1st. O'Reilly Media, Inc., 2015. – ISBN 1491920491, 9781491920497

[Walport 2016] Walport, Mark: *Distributed Ledger Technology: beyond block chain*. 01 2016

[Wei u. a. 2014] Wei, Lifei ; Zhu, Haojin ; Cao, Zhenfu ; Dong, Xiaolei ; Jia, Weiwei ; Chen, Yunlu ; Vasilakos, Athanasios V.: Security and privacy for storage and computation in cloud computing. In: *Information Sciences* 258 (2014), S. 371 – 386. – URL http://www.sciencedirect.com/science/article/pii/S0020025513003320. – ISSN 0020-0255

[Yaga u. a. 2018] Yaga, Dylan ; Mell, Peter ; Roby, Nik ; Scarfone, Karen: *Blockchain Technology Overview*. 2018. – URL https://doi.org/10.6028/NIST.IR.8202

[Zhang u. a. 2019] Zhang, Y. ; Kasahara, S. ; Shen, Y. ; Jiang, X. ; Wan, J.: Smart Contract-Based Access Control for the Internet of Things. In: *IEEE Internet of Things Journal* (2019), S. 1–1. – ISSN 2327-4662

[Zhang und Wen 2017] Zhang, Yu ; Wen, Jiangtao: The IoT electric business model: Using blockchain technology for the internet of things. In: *Peer-to-Peer Networking and Applications* 10 (2017), Jul, Nr. 4, S. 983–994. – URL https://doi.org/10.1007/s12083-016-0456-1. – ISSN 1936-6450

[Zhang u. a. 2014] Zhang, Z. ; Cho, M. C. Y. ; Wang, C. ; Hsu, C. ; Chen, C. ; Shieh, S.: IoT Security: Ongoing Challenges and Research Opportunities. In: *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, Nov 2014, S. 230–234. – ISSN 2163-2871

[Zhou u. a. 2017] Zhou, J. ; Cao, Z. ; Dong, X. ; Vasilakos, A. V.: Security and Privacy for Cloud-Based IoT: Challenges. In: *IEEE Communications Magazine* 55 (2017), January, Nr. 1, S. 26–33. – ISSN 0163-6804

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original