Bachelorarbeit

im Studiengang
Bachelor of Computer Science

Erarbeitung eines metrikbasierten Qualitätsmodells für ERC-20 Token

von

Simon Worm

Erstprüfer:	Prof. Dr. rer.nat. Rainer Koschke
Zweitprüfer:	Prof. Dr. Andreas Breiter
Betreuer:	Prof. Dr. rer.nat. Rainer Koschke
Eingereicht am:	24.01.2020

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In traditionellen Finanzsystemen werden Transaktionen zwischen Benutzern von einer zentralen Institution, beispielsweise einer Bank, geprüft und bestätigt.
Die Sicherheit dieses System folgt daraus, dass die Nutzer der Institution vertrauen und diese mit Überweisungen beauftragen, die daraufhin ausgeführt werden.
Abgesehen davon, dass ein zentralisiert aufgebautes Finanzsystem als zuverlässig und vertrauenswürdig gilt, gibt es auch einige nennenswerte Nachteile. Die Ausführungszeit von Transaktionen kann zwischen Stunden und Tagen variieren, die zwischengeschaltete Institution kann eine vergleichsweise hohe Transaktionsgebühr verlangen und der Nutzer hat eingeschränkte Möglichkeiten, den Status seiner Überweisung zu verfolgen.
Um diesen Nachteilen der traditionellen Finanzsysteme zu begegnen, wurde der technologische Fortschritt in den Bereichen Peer-to-Peer Netzwerken und dezentraler Datenverwaltung vorangetrieben. Indem mithilfe der DLT-Technologie (Distributed Ledger Technologie) ein digitales und dezentrales, öffentliches Hauptbuch implementiert wird, hat sich die Blockchain-Technologie in den letzten Jahren als führende Technologie etabliert.
In dem verteilten Hauptbuch werden alle Kontostände und verifizierten Transaktionen von Kryptowährungen dokumentiert. Es entspricht einer verteilten Datenbank, welche zwischen den unterschiedlichen Knoten des Netzwerkes geteilt und in getakteten Abständen synchronisiert wird.
Durch diese Verteilung auf eine Vielzahl von Teilnehmern im Netzwerk erlangen die Daten Vertraulichkeit, Zugänglichkeit und Integrität [5].

In einem weiteren Entwicklungsschritt wurde die Blockchain-Technologie um die Funktion erweitert - abgesehen von der Dokumentation von Kryptowährungstransaktionen - auch Programme zu verwalten, welche auf Aufruf eines Nutzers im Netzwerk ausgeführt werden. Nach dem Aufruf werden die vom Entwickler programmierten Befehle chronologisch und autonom ausgeführt. Diese Programme werden Smart Contracts genannt.
Aufgrund ihrer autonomen Ausführung bieten Smart Contracts ein hohes disruptives Potential, sie werden in vielen Branchen getestet und eingesetzt.
Zu diesen Branchen gehören unter anderem die Finanzbranche [6] [7], die Gesundheitsbranche [8] [9], die Energiebranche [10] [11], Supply Chain Management [12] [13] und die Verwaltung in Regierungen [14] [15].

Mittlerweile existieren verschiedene und voneinander unabhängige Blockchain-Netzwerke, welche in der Lage sind, Smart Contracts bereitzustellen und auszuführen. Als erste Plattform hat Ethereum im Jahr 2017 [16] die Idee von Smart Contracts vorgestellt. Mittlerweile haben sich modifizierte Varianten wie EOS, Link, Hyperledger Fabric und

viele andere gebildet.
In dem hier beschriebenen Versuch werden nur Smart Contracts innerhalb der Ethereum-Blockchain untersucht. Smart Contracts werden in Ethereum in der Sprache Solidity geschrieben, welche der Sprache Javascript in ihrer Syntax ähnlich ist.
Möchte ein Entwickler seinen geschrieben Smart Contract in die Ethereum Blockchain laden, lädt er seinen compilierten Quellcode in den dafür vorgesehenen Speicher einer Adresse innerhalb des Netzwerkes. Da die Daten in der Blockchain unveränderbar gespeichert werden, kann der Quellcode im Nachhinein nicht mehr angepasst werden. Der Contract ist nun mit einer Adresse verknüpft und kann die Ether der Adresse verwalten und diese an andere Adressen, externe Nutzer oder weitere Contracts versenden oder von diesen weitere Ether erhalten.
Ethereums Smart Contracts sind also in der Lage, einen monetären Wert in Form der Kryptowährung Ether basierend auf der Programmierung in Solidity zu verwalten.
Da der Programmcode nach dem Abspeichern in der Blockchain nicht mehr veränderbar ist, muss die Implementierung fehlerfrei und korrekt sein. Dabei sind Entwickler mit mehreren Herausforderungen konfrontiert:

- Aufgrund der frühen Entwicklungsstufe der Blockchain-Technologie gibt es noch viele Wissenslücken über die Implementierung und Nutzung von Smart Contracts.
- Es gibt beschränkt viele *Best Practice*-Methoden für die Implementierung und das Testen von Smart Contracts.
- Werden Fehler erst nach der Bereitstellung entdeckt, können diese nicht wie in traditionellen Softwaresystemen mithilfe von Updates und Patches korrigiert werden. Der fehlerhafte Smart Contract muss normalerweise mit einer dafür vorgesehenen Funktion zerstört werden bevor eine korrigierte Version bereitgestellt werden kann.

Um das Aufkommen von Sicherheitsrisiken in Smart Contracts näher zu untersuchen und eine Hilfestellung bei einer korrekten Implementierung zu schaffen, soll in dieser Arbeit ein metrikbasiertes Qualitätsmodell entwickelt werden. Das Modell soll dabei helfen, einen Zusammenhang zwischen Sicherheitsrisiken und Softwaremetriken herzustellen.

Für die Identifizierung der Sicherheitsrisiken sollen verschiedene Sicherheitsscanner genutzt werden. Damit eine umfangreiche und zuverlässige Untersuchung jedes betrachteten Smart Contracts gewährleistet wird, sollen Sicherheitsscanner mit unterschiedlichen Methoden der Quellcodeanalyse einbezogen werden.

Um das Modell möglichst nutzbringend zu gestalten, sollen Metriken einbezogen werden, die in der Softwareentwicklung weit verbreitet sind.

Die Testobjekte für die Messungen - Smart Contracts aus der Ethereum Blockchain - werden aus einer öffentlichen Blockchain Suchmaschine gesammelt.

Nachdem zu Beginn der Arbeit auf die benötigten Grundlagen der Blockchain Technologie und die berücksichtigten Sicherheitsrisiken eingegangen wird, wird im Hauptteil der Versuchsaufbau beschrieben und das erstellte Modell aus verschiedenen Perspektiven bewertet. Abschließend werden die Ergebnisse im Fazit kompakt zusammengetragen.

2 Theoretische Grundlagen

2.1 Blockchain

2.1.1 Entwicklung

Auch wenn sich der Begriff der *Blockchain* erst 2009 mit der Veröffentlichung des *Bitcoin Whitepaper* [17] etabliert hat, wurde die Grundidee, das Vertrauen in eine singuläre Institution durch das Vertrauen in ein Netzwerk auszutauschen, bereits früher aufgegriffen.
1992 beschäftigten sich Stuart Haber and W. Scott Stornetta [18] mit dem Problem, im Internet eindeutig nachverfolgen zu können, wann ein Dokument erstellt oder zuletzt geändert wurde.
In einem ihrer Vorschläge wird eine zentrale Institution, der TSS (Timestamp-Service) genutzt, um Einträge von Nutzern zu speichern. Dabei beschreiben sie das Risiko, dass der TSS das gesamte Vertrauen der Nutzer genießt und somit ungehindert falsche Einträge erstellen könnte.
In dem Versuch, eine Methode zu erstellen, die es unmöglich macht, falsche Timestamp-Einträge festzuhalten, beschreiben Sie die Idee, das Vertrauen auf die Teilnehmer des Netzwerkes zu verteilen. In ihren Ansätzen zur Umsetzung greifen sie auf eine Hash-Funktion und auf eine digitale Signatur zurück. Technologien, die auch in modernen Blockchain-Systemen genutzt werden.

Abgesehen von früheren Ansätzen zur Entwicklung der Idee, Netzwerken zu vertrauen, wird die Entwicklung der Blockchain-Technologie häufig in drei Phasen unterteilt [19] [20]:

2.1.1.1 Blockchain 1.0 - Währung Blockchain 1.0 beschreibt die erste Generation von Systemen und startete mit der Veröffentlichung des *Bitcoin Whitepapers* [17] im Jahr 2008. Diese erste Phase bezieht sich auf die Bereitstellung der grundlegenden Technologie (beispielsweise Mining, Hashing und das öffentliche Hauptbuch), dem darauf aufbauenden Protokoll (Software zum Senden für Transaktionen) und der verknüpften digitalen Währung (Bitcoin oder andere Kryptowährungen). [21]
Als technologische Metapher wird gerne die Protokollschicht im Internet verwendet, so beschreibt Blockchain 1.0, wie die Protokollschicht im Web, die grundlegende Struktur auf der spezifische Anwendungen geschaffen werden können. [20]

2.1.1.2 Blockchain 2.0 - Wirtschaft

Der Start von Ethereum im Jahr 2017 und die damit verbundenen Möglichkeit zur Erstellung von Smart Contracts hat die Phase Blockchain 2.0 eingeläutet [22]. Ethereum baut auf der vorgegeben Blockchain-Struktur von Bitcoin auf, abstrahiert dabei aber von der Dokumentation von Transaktionen und ermöglicht eine Verwendung in vielen Domänen.
Dabei werden Smart Contracts verwendet, um eine Vielzahl an Vermögenswerten zu steuern - unter anderem Treuhandgeschäfte, Aktien, Immobilien, Patente oder Identifikationspapiere wie der Personalausweis.

2.1.1.3 Blockchain 3.0 - Gesellschaft

Da die Blockchain-Technologie eine grundlegend neue Möglichkeit zur Organisation mit mehr Transparenz und weniger Aufwand bietet, lässt sich das Konzept auch auf viele Organisationsformen unabhängig von Märkten und Wirtschaft übertragen. Diese Idee beschreibt die Phase Blockchain 3.0, in der die Technologie in Bereiche wie Kunst, Gesundheit, Wissenschaft, Bildung und öffentliche Güter integriert wird.
Ein Beispiel für die Verwaltung von öffentlichen Gütern ist die Organisation von *Smart Cities*, in denen mehrere Elemente wie *Smart Mobility*, *Smart Living*, und *Smart Economy* verbunden werden. [23] [21]
Ein Beispielprojekt für Blockchain 3.0 ist IOTA [24], eine angepasste Umsetzung einer Blockchain, die speziell auf die Vernetzung von Geräten im Internet of Things entwickelt wurde.

2.1.2 Ethereum

Die Idee für Ethereum wurde im Jahr 2013 von Vitali Buterin [16] mit dem Vorhaben veröffentlicht, ein verteiltes Computernetzwerk zu bauen, welches das Bitcoin-Protokoll um die Möglichkeit zur Speicherung von Programmcodes erweitert. Dieser Programmcode, Smart Contract genannt, wird mit bestimmten Adressen verknüpft und ist somit innerhalb des Netzwerkes von Teilnehmen addressierbar und ausführbar.
Die Idee wurde im Jahr 2014 durch ein Crowdfunding finanziert, im folgenden Jahr wurde das Netzwerk geschaffen. Seitdem wird das Netzwerk durch ständige Updates weiterentwickelt.
Neben den zuvor beschrieben Bestandteilen der Blockchain-Technologie wird im folgenden auf die Komponenten des Ethereum-Netzwerkes eingegangen, welche für die Funktionalität und Ausführung von Smart Contracts verantwortlich sind.

2.1.2.1 Smart Contracts

Ein Smart Contract ist ein ausführbarer Quellcode, der innerhalb einer Blockchain verwaltet wird. Die Kernaufgabe eines Smart Contracts ist es,

automatisch eine Reihe von Ausführungsschritten zu durchlaufen, sobald die dafür benötigten Vorbedingungen erfüllt sind. Ursprünglich wurde die Idee bereits im Jahr 1994 von Szabo [25] beschrieben, durch die Entwicklung der Blockchain Technologie hat sie eine realistische Anwendung gefunden.
In einer aktuelleren Definition vom Ethereum Gründer Vitalik Buterin [16] können Smart Contracts als ein System beschrieben werden, welches Mittel unter allen oder einigen der involvierten Parteien verteilt, sobald die dafür benötigten Voraussetzungen eingetreten sind.
In der Literatur sind allerdings unterschiedliche Definitionen zu finden, so wird teilweise auch zwischen *Smart Contracts* und *Smart legal Contracts* unterschieden. Dabei zeichnet einen Smart Contract Quellcode aus, der in einer Blockchain gespeichert, verifiziert und ausgeführt wird. [26]

2.1.2.2 Virtual Maschine (EVM)

Die EVM behandelt die Durchführung und den Zustand von Smart Contracts. Sie ist auf einer stapelbasierten (engl. Stack-based) Sprache mit einer vordefinierten Menge an Anweisungen (opcodes) mit entsprechenden Argumenten aufgebaut. [16]
Unabhängig von der individuellen Programmierung in der Sprache Solidity wird jeder Smart Contract mithilfe eines Solidity Übersetzers in eine sequentielle Menge von opcodes übersetzt, welche dann an einer entsprechenden Adresse in der Blockchain hinterlegt werden.
Wird ein Smart Contract mittels Transaktion zur Ausführung aufgerufen, werden die entsprechenden opcodes von der Ethereum Virtual Maschine nacheinander ausgeführt.

2.1.2.3 Globaler Zustand

Der globale Zustand der Ethereum Blockchain ist als Abbildung zwischen 160-bit kodierten Adressen auf Accountzustände definiert [16]. Für das Mapping nutzt das Ethereum Netzwerk den selbst definierten *modified Merkle Patricia Tree* [16], welcher Funktionen einen Prefix-Baumes und eines Hash-Baumes verbindet.
Der Baum wird nicht in der Blockchain, sondern in einer gesonderten Datenbank, der sogenannten *state database*, verwaltet.
Der globale Zustand der Blockchain wird durch einen Wurzel-Hash des Baumes identifiziert, alle Wurzel-Hashes werden wiederum in der Blockchain gespeichert. Der globale Zustand der Ethereum Blockchain ändert sich mit jeder Transaktion im Netzwerk.

2.1.2.4 Solidity

Solidity ist eine high-level turingvollständige Programmiersprache, die für die Entwicklung von Ethereum Smart Contracts entwickelt wurde. Ihre Syntax ist ähnlich zu JavaScript und unterstützt gängige Eigenschaften verwandter high-level

Sprachen wie Vererbung, Polymorphie, Bibliotheken und benutzerdefinierte Typen. Abgesehen davon ähnelt ein in Solidity geschriebener Contract sehr einer Klasse in objektorientierten Sprache. Er besitzt eigene Variablen und Funktionen, welche diese Variablen manipulieren können.

In Abbildung 1 ist der Contract *AWallet* abgebildet, welcher eine simple Geldbörse für Ether implementiert, die einem Besitzer zugeordnet werden kann.
Der in Solidity geschriebene Contract ist in der Lage, Ether von anderen Accounts zu bekommen und der Besitzer kann beliebige Ether an andere Accounts versenden. Die Hashtabelle *outflow* fungiert als Übersicht über die versendeten Mittel, wobei jeweils die Adresse sowie der Etherbetrag gespeichert wird.
Für das Erhalten von Ether ist keine explizite Methode notwendig, der Wert wird automatisch im Feld *balance* gespeichert, einer speziellen Variable, die der Programmierer nicht verändern kann.
Nach der Deklarierung der Variablen steht der Konstruktor des Contracts. Er zeichnet sich dadurch aus, dass er der Funktionsname mit dem Contractnamen identisch ist. Der Konstruktor wird einmalig bei der Erstellung des Smart Contractes ausgeführt, im Nachhinein kann er nicht mehr angesprochen werden.
In Zeile 8 und 10 werden Exceptions geworfen, welche sich in Ethereum Smart Contracts von Exceptions in anderen Sprachen unterscheiden, da sie nicht aufgefangen werden können.
Wenn eine Exception geworfen wird, stoppt die Ausführung, alle bisherigen Ausführungsschritte werden rückgängig gemacht und die vom Sender gezahlte Ausführungsgebühr geht verloren.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner || msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Abbildung 1: AWallet, einfaches Solidity Smart Contract Beispiel [1]

2.1.2.5 Accounts Die minimal ansprechbare Komponente innerhalb des Ethereum-Netzwerkes ist ein Account. Im globalen Zustand des Ethereum-Netzwerkes ist jedem Account eine eindeutige Adresse, ein Wert an Ether, ein bestimmter EVM-Code (möglicherweise leer), ein eigener Speicher (möglicherweise leer) und eine Historie von Transaktionen zuzuordnen. [16]
Innerhalb des Ethereum-Netzwerkes kommunizieren Accounts mithilfe von Transaktionen miteinander. Dabei kann jeder Account anhand seiner Adresse, einer 160-Bit Identifizierung, angesprochen werden.
Der Zustand eines Accounts wird im globalen Zustand durch das Tupel (nonce, balance, storageRoot, codeHash) mit

nonce
: Ein Skalarwert der die Anzahl der gesendeten Transaktionen beschreibt. Für den Fall, das EVM Bytecode in dem Account hinterlegt ist, wird die Anzahl der Contract Erzeugungen gespeichert.

balance
: Der an der Adresse hinterlegte Wert an Ether.

storageRoot
: Ein 256-bit Hash für die Lokalisierung des zugehörigen Speichers.

codeHash
: Ein Hash des zugehörigen EVM Bytecodes. Die Codefragmente aller Contracts werden durch ein Mapping in einer separaten Datenbank (state database [16]) verwaltet. Hier wird zu jedem Hash der korrespondierende Bytecode gespeichert.

dargestellt.

2.1.2.6 Ether und Gas Ether (ETH) ist die primäre monetäre Einheit innerhalb des Netzwerkes und wird zum Wertaustausch zwischen verschiedenen Adressen genutzt. Transaktionen und Ausführungen eines Contracts sind mit individuellen Kosten verbunden, diese werden in Gas verwaltet. Ein Gas ist ein Bruchteil eines Ether. Jede Adresse, ob Nutzer- oder Contractadresse, kann Ether und Gas verwalten und mithilfe von Transaktionen versenden.
Wird ein Contract von einer Adresse aus aufgerufen und zur Ausführung gebracht, muss der Sender für die dabei entstehenden Kosten im Vorfeld aufkommen. Dafür spezifizieren die Felder *gasLimit* und *gasPrice* [16] der Transaktion, wie viel der Sender zu zahlen bereit ist. Hierbei kann der Sender die Werte für *gasLimit* und *gasPrice* frei wählen. Aus der Multiplikation der beiden Werte ergibt sich die Menge der Ether, die von seinem

Guthaben abgezogen und den ausführenden Netzwerkknoten (engl. Miner) gutgeschrieben wird.
Jede auszuführende EVM Operation (opcode) ist mit fest definierten Kosten verbunden. [16] Die Verknüpfung von Ausführungen mit direkten Kosten hat zwei Vorteile: Zum Einen wird ein Anreiz für Netzwerkknoten geschaffen, da diese die Kosten als Belohnung für die Ausführung ausgezahlt bekommen. Zum anderen wirken die anfallenden Kosten jeder Transaktion als Vorbeugung von Denial-of-Service-Angriffen auf das Netzwerk, da diese durch zu hohe Kosten unattraktiv werden.

2.1.2.7 Token und der ERC-20 Standard

Auf der Grundlage von Smart Contracts innerhalb von Ethereum implementieren viele Contracts einen eigenen *Token*. Dabei kann man einen Token eines Contractes als eigene, interne Währung verstehen. Der eigene Token kann gegen Ether getauscht werden und kann somit einen monetären Wert erhalten. Da Token inflationär implementiert werden können, werden sie häufig genutzt um diese an Investoren mit dem Versprechen zu verkaufen, dass ihr Wert steigt je stärker der Contract genutzt wird [27]. Der ERC-20 Standard wird durch ein Solidity Interface umgesetzt was die Kompatibilität zwischen verschiedenen Token sicherstellt.

2.1.2.8 Mining, Proof-Of-Work

Eine Herausforderung in der Umsetzung der Blockchain Technologie besteht darin, sicherzustellen, dass jeder Netzwerkknoten über denselben Datenbestand verfügt und dass dieser Datenbestand nur mit bestimmten Daten erweitert wird.
Damit die Netzwerkknoten einen Konsens über den aktuellen und zukünftigen globalen Zustand finden, werden in unterschiedlichen Projekten verschiedene Algorithmen eingesetzt. Da Ethereum aktuell den Proof-of-Work (PoW) Algorithmus verwendet, wird hier oberflächlich auf diesen eingegangen.
PoW wird genutzt, um unter allen Minern im Netzwerk einen Wettkampf zu schaffen, in dem jeder Miner versucht, als Erster den nächsten Block in der Kette zu finden und die damit verbundene Vergütung, in Form von Ether, ausgezahlt zu bekommen.
Neue Blöcke werden in Zeitintervallen an die bestehende Blockchain Kette angehängt, wobei der aktuellste Block stets die letzten und aktuellsten Transaktionen enthält.
Ein Block wird als gültig bezeichnet, wenn sein Hash-Wert eine bestimmte Anzahl an führenden Nullen hat. Es können nur gültige Blöcke an die Kette angehängt werden. Der Prozess, in dem ein neuer gültiger Block gesucht wird, wird *Mining* genannt und er besteht darin, einen Block solange mit einer neuen kleinen Modifikation zu hashen, bis ein gültiger Wert dabei entsteht. Diese Brute-Force-Methode sorgt dafür, dass hinter jedem gültigen Block in der Blockchain Kette eine aufgewendete Rechenkraft steht.

Zusätzlich zu der Liste von Transaktionen enthält jeder Block den Hash seines Vorgängers. Durch diese Verkettung der Blöcke entsteht aus der Gesamtzahl aller Blöcke eine Baum. Der aktuelle Zustand der Blockchain ist dabei immer durch die längste Kette im Baum repräsentiert, an dieser längsten Kette orientieren sich alle Miner, sodass ein Konsens entsteht. Die Belohnung für das Finden eines gültigen Hash-Wertes für einen Block steht einem Miner nur dann zu, wenn sein gefundener Block in der aktuell längsten Kette vorhanden ist.
Dieser Mechanismus sorgt dafür, dass es im Interesse der Miner liegt, immer die längste bestehende Kette fortzuführen. Durch dieses Bestreben entsteht ein Baum, der aus der längsten bestehenden Kette selbst und einigen Abzweigungen besteht, die im Normalfall schnell enden, da die Miner keine Belohnung ausgezahlt bekommen.
Entsteht eine Abzweigung an der Spitze der Kette, wird von einem *Fork* gesprochen. Die Menge der Miner kann sich nun auf die beiden möglichen Wege verteilen, der Konsens über die Daten in der Blockchain kann kurzzeitig gespalten sein. Doch nach dem Bestreben, stets an der längsten bestehenden Kette zu arbeiten wird nur der Zweig mit der Mehrzahl der Miner fortgeführt, wodurch der Konsens zeitnah wieder hergestellt wird.

2.2 Betrachtete Ethereum Smart Contract Schwachstellen

Im Folgenden werden die in dieser Arbeit betrachteten und einbezogenen Qualitäts- und Sicherheitslücken vorgestellt und an einem kurzen Beispiel erläutert. Die Beispiele und Erklärungen stammen dabei aus den Veröffentlichungen zu den Scannern Securify [3] und Oyente [4], sowie aus der öffentlichen Dokumentation des Scanners Slither [28].

2.2.1 State Variable Shadowing (Slither)

Globale Variablen eines Contracts werden an anderer Stelle (meist aufgrund von Vererbung) durch gleiche Namensgebung überdeckt. Die vorgesehene Funktionalität des Contracts kann stark eingeschränkt werden, indem bestimme Ausführungspfade nicht mehr erreichbar sind.
Beispiel:

```
1  contract BaseContract{
2      address owner;
3
4      modifier isOwner(){
5          require(owner == msg.sender);
6          _;
7      }
8  }
9
10 contract DerivedContract is BaseContract{
11     address owner;
12
13     constructor(){
14         owner = msg.sender;
15     }
16
17     function withdraw() isOwner() external{
18         msg.sender.transfer(this.balance);
19     }
20 }
```

Abbildung 2: Überdeckte Variable *owner* in BaseContract

2.2.2 State variable shadowing from abstract contracts (Slither)

Innerhalb einer Vererbung überschattet eine Variable durch gleiche Namensgebung eine Variable im vererbenden Smart Contract.
Beispiel:

```
1  contract BaseContract{
2      address owner;
3  }
4
5  contract DerivedContract is BaseContract{
6      address owner;
7  }
```

Abbildung 3: Überschattung von einer Variable innerhalb einer Vererbung durch gleiche Namensgebung

Die Variable *owner* in BaseContract wird nicht initialisiert und durch *owner* im geerbten Contract überschrieben. Die Abfrage *isOwner()* funktioniert nicht mehr wie vorgesehen, wodurch der Rumpf der Methode *withdraw()* nicht mehr erreicht wird.

2.2.3 Suicidal (Slither)

Die Funktionen *selfdestruct(msg.sender)* und *suicide(msg.sender)* zerstören den betroffenen Contract und senden alle enthaltenen Mittel zum Aufrufer. Die Methoden sind nicht rückgängig zu machen und deshalb mit höchsten Vorsicht einzusetzen. Die Sicherheitslücke *Suicidal* prüft, ob ein ungeschützter Ausführungspfad zu den Methoden existiert.
Beispiel:

```
1 contract Suicidal{
2     function kill() public{
3         selfdestruct(msg.sender);
4     }
5 }
```

Abbildung 4: Contract der von beliebigen Aufrufern zerstört werden kann

Die Methode *suicidal()* ist nicht geschützt und erlaubt es somit jedem öffentlichen Aufrufer den Contract zu zerstören.

2.2.4 Uninitialized state variable (Slither)

Es existieren uninitialisierte globale Variabeln, auf die an anderer Stelle zugegriffen wird. Während des Zugriffes auf die uninitialisierte Variable wird im Contract stets eine Exception geworfen, wodurch die Ausführung stoppt und alle Schritte rückgängig gemacht werden. Dadurch kann ein Teil des Contracts unausführbar sein.
Weitaus fataler ist es, wenn Ether an eine uninitialisierte Adresse gesendet werden. In diesem Fall wird keine Exception geworfen, sondern die Ether werden an die Adresse 0x0 gesendet, wo sie verloren gehen. Dieser Programmierfehler ist mit einer Nullpointer-Exception aus anderen Sprachen vergleichbar, er kann verhindert werden, indem die Variable explizit auf *zero* gesetzt wird.
Beispiel:

```
1 contract Uninitialized{
2     address destination;
3
4     function transfer() payable public{
5         destination.transfer(msg.value);
6     }
7 }
```

Abbildung 5: Unintialisierte Adresse führt zum Verlust von Ether

Die Adresse *destination* ist nicht initialisiert, doch es werden Ether an sie gesendet. Die Ether werden an die Adresse 0x0 gesendet und gehen verloren.

2.2.5 Uninitialized storage variable (Slither)

In Ethereum Smart Contracts referenzieren nicht initialisierte Speichervariablen die erste Stelle im Speicher. Wird eine Variable nicht initialisiert, kann dies dazu führen, dass eine andere Variable, welche zur Ausführungszeit an die erste Stelle im Speicher geschrieben wird, überschrieben wird.
Beispiel:

```
1  contract Uninitialized{
2      address owner = msg.sender;
3
4      struct St{
5          uint a;
6      }
7
8      function func() {
9          St st;
10         st.a = 0x0;
11     }
12 }
```

Abbildung 6: Die uninitialisierte Variable referenziert den Speicher der chronologisch ersten Variable im Quellcode

Bei der Erstellung des Contracts in Abbildung 5 werden die Variablen chronologisch in den Speicher geschrieben. Der Wert von *owner* wird an die erste Stelle geschrieben. Als zweites wird *a* bearbeitet, da die Variable nicht initialisiert ist, referenziert sie ebenfalls die erste Stelle im Speicher.
In *func()* wird *a* der Wert 0x0 zugeschrieben, wodurch der Wert von *owner* überschrieben wird.

2.2.6 Incorrect erc20 interface (Slither)

Durch das Nutzen eines veralteten erc20 Interfaces sind Contracts, die mit einer Compilerversion höher als 0.4.22 erstellt wurden, nicht in der Lage, bestimmt Funktionen auszuführen. Dies liegt daran, dass der Interfacefunktion *transfer()* im Laufe der Entwicklung des Compilers und der Sprache Solidity ein boolean Rückgabewert hinzugefügt wurde. Dieser Rückgabewert wird von Contracts mit Compilerverison höher 0.4.22

erwartet.
Beispiel:

```
1 contract Token{
2     function transfer(address to, uint value) external;
3     //...
4 }
```

Abbildung 7: Veraltetes erc20 Interface ohne boolean Rückgabewert

Durch den fehlenden boolean Rückgabewert ist eine Interaktion mit vielen Contracts nicht möglich.

2.2.7 Incorrect erc721 interface (Slither)

Durch eine veraltete genutze Interfaceversion des erc721 Interfaces gibt die Interfacemethode *ownerOf()* keinen Rückgabewert zurück. Contracts, die mit einer Compilerversion höher 0.4.22 compiliert wurden, erwarten eine Adresse als Rückgabewert, wodurch eine Kommunikation zwischen den Versionen nicht möglich ist.
Beispiel:

```
1 contract Token{
2     function ownerOf(uint256 _tokenId) external view returns (bool)
          ;
3     //...
4 }
```

Abbildung 8: Veraltetes erc721 Interface ohne Adresse als Rückgabewert

2.2.8 Dangerous strict equalities (Slither)

Im Quellcode werden Vergleichsoperatoren unbedacht eingesetzt, sodass der Contract von einem Angreifer in einen unbeabsichtigten Zustand gebracht wird.
Beispiel:

```
1 contract Crowdsale{
2     function fundReached() public returns(bool){
3         return this.balance == 100 ether;
4     }
```

Abbildung 9: unbedachter Einsatz von Vergleichsoperatoren

Der Smart Contract *Crowdsale* nutzt die Funktion *fundReached()*, um zu erkennen, wann die Sammlung von Ether abgeschlossen ist. Sendet ein Angreifer einen Wert, sodass das Guthaben des Contracts höher als 100 ist, gibt die Methode immer false zurück und das Spendensammeln endet nie.

2.2.9 Right-To-Left-Override-Character (Slither)

Ein Angreifer nutzt die Unicode Zeichenkette *Right-To-Left Override* (U+202E) und erzwingt damit ein Rendering des Textes in entgegengesetzt Richtung. Dadurch können die Übergabeparameter an eine Methode böswillig übergeben werden, sodass sie in dem Smart Contract in falscher Reihenfolge interpretiert werden.
Beispiel:

```
function withdraw() external returns(uint)
{
    uint amount = tokens[msg.sender];
    address payable d = msg.sender;
    tokens[msg.sender] = 0;
    _withdraw(/*owner*/o ,d /*destination*/
                    /*value */, amount);
}
```

Abbildung 10: In dem Aufruf von _withdraw() wird ein der RTLO übergeben, wodurch die Parameter *o* und *d* beim Ziel vertauscht interpretiert werden

2.2.10 Locked Ether (Slither, Securify)

Im November 2017 hat eine suicidal Sicherheitslücke in der weit verbreiteten Multi-Sig Bibliothek dazu geführt, dass der Contract von einem Nutzer zerstört werden konnte.
Die Bibliothek wurde währenddessen von 587 Smart Contracts genutzt, um eigene Ether zu versenden. Dadurch, dass diese Smart Contracts auf die Bibliothek vertrauten, um Ihre Ether aus dem Smart Contract zu ziehen und diese Bibliothek zerstört wurde, wurden 513.774,16 Ether unzugänglich. In den betroffenen Smart Contracts, die keine eigene Funktion zum Versenden von Ether eingebaut haben, wurde zum Zeitpunkt des Vorfalls am 6. November 2017 etwa 98 Millionen US-Dollar [1] eingefroren.

Daraufhin wurde eine Sicherheitseigenschaft definiert, die untersucht, ob Smart Contracts Ether über eine Funktion erhalten können und ob sie eine eigene Methode haben,

[1] Am 7. November hatte ein Ether nach coinmarketcap.com [29] einen Wert von 191,50$

um die eigenen Ether zu versenden. Ist dies nicht der Fall, wird der Smart Contract auch *greedy* genannt. [30]

2.2.11 Reentrancy (Oyente), RepeatedCall (Securify)

Smart Contracts können sich mithilfe der Methode *call()* untereinander aufrufen, dabei schickt der Sendende mit *call()* einen Aufruf an den empfangenden Contract.
Der Kontrollfluss des Senders steht nun still. Er wartet darauf, dass der Aufgerufene seine Ausführungsschritte abgeschlossen hat. Bei einer fehlerhaften Implementierung kann sich der Aufgerufene den Stillstand von Sender zunutze machen.
Am 18. Juni 2016 gelang es einem Angreifer, etwa 60 Millionen US-Dollar zu stehlen, indem er einen Reentrancy/RepeatedCall-Angriff auf den Smart Contract *The DAO* ausführte [1].
Das vereinfachte Beispiel SimpleDAO veranschaulicht die Sicherheitslücke.

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
12
13 contract Mallory {
14     SimpleDAO public dao = SimpleDAO(0x354...);
15     address owner;
16     function Mallory(){owner = msg.sender; }
17     function() { dao.withdraw(dao.queryCredit(this)); }
18     function getJackpot(){ owner.send(this.balance); }
19 }
```

Abbildung 11: Ausnutzung der Reentrancy/RepeatedCall Schwachstelle in SimpleDAO [1]

Wie *The DAO* hat der Contract SimpleDAO die Funktion, dass Nutzer Ether als Spenden an die Contracts ihrer Wahl senden können. Daraufhin können die betroffenen Contracts ihre bezogenen Spenden beanspruchen.
Durch die Reentrancy/RepeatedCall Sicherheitslücke ist es für einen Angreifer möglich, alle Spenden des Contractes zu entziehen. Dafür veröffentlicht er den Contract *Mallory*. Er sendet Ether an seinen gerade erstellten Contract, wodurch die namenlose fallback-Methode (Zeile 17) ausgeführt wird.
Hier wird die Methode *dao.withdraw()* aufgerufen. Wird nun die if-Anweisung in Zeile

8 betreten, wird wieder eine *call()*-Methode an den Contract *Mallory* gesendet. An dieser Stelle wird der Angriff ausgenutzt, da nun wieder die fallback-Methode (Zeile 17) ausgeführt, die *dao.withdraw()* ein zweites Mal aufruft.
Das Problem hierbei ist, dass die Anweisung in Zeile 10 nie erreicht wurde, *Mallory* kann also wieder die if-Anweisung durchlaufen und mittels der *call()*-Methode Ether erhalten.
Diese Schleife kann solange fortgeführt werden bis entweder die Transaktionsgebühr für den Miner ausgelaufen ist (diese kann zu Beginn allerdings beliebig hoch gesetzt werden), der Speicher des Contracts überläuft (dann wird eine Exception geworfen) oder der Contract SimpleDAO keine Ether mehr besitzt.
In der Reentrancy/RepeatedCall Schwachstelle wird ein Programmierfehler ausgenutzt, der es Angreifern ermöglicht, einen Ausführungspfad in verschiedenen Transaktionen mehrmals zu betreten. Diese Sicherheitslücke kann geschlossen werden, indem alle nötigen Anpassungen von Variablen vor dem Aufruf der *call()*-Methode durchgeführt werden.

2.2.12 Unrestricted Write (Securify)

Diese Sicherheitslücke besteht, falls es eine Variable im Code gibt, die von beliebigen Nutzern beschrieben werden kann.

Die Aufdeckung dieser Sicherheitslücke geht auf Juli 2017 zurück, als es einem Angreifer gelang, 153.037 Ether, rund 25 Millionen[2] US-Dollar, zu entwenden [31], indem er die *owner*-Variable eines weit verbreiteten Contracts beschreiben konnte ohne die Berechtigung zu besitzen. Zur Vorbeugung wird die Sicherheitslücke behandelt, indem geprüft wird, ob ein uneingeschränkter Schreibzugriff für eine Variable im Quellcode existiert. Der Scanner Securify prüft dabei für jede Variable eines Smart Contractes, ob einen Nutzer gibt, der diese Variable nicht beschreiben kann.

2.2.13 Unrestricted Ether Flow (Securify)

Hier wird der Zugriff auf den EVM *call*-Befehl untersucht, da mittels dieses Befehls Ether aus dem Contracts entwendet werden können. Es wird geprüft, ob es einen *call*-Befehl gibt, der von jedem Nutzer ausgeführt werden kann.
Diese Eigenschaft ist besonders hilfreich bei der Identifikation von Schneeballsystemen innerhalb von Smart Contracts. [32]

[2] Am 27. Juli 2017 hatte ein Ether nach coinmarketcap.com [29] einen Wert von etwa 220,00$

2.2.14 Missing Input Validation (Securify)

Ruft ein Nutzer eine Methode eines Contractes auf, so ist er auch für die Übergabe der Parameter verantwortlich. Es ist aus zwei Gründen notwendig, die übergebenen Parameter vor der Verarbeitung zu validieren: Zum einen kann der Aufrufer unabsichtlich falsche Werte übergeben, zum anderen könnte er dies mit Absicht tun, um die Ausführung der Methode zu verfälschen.
An dieser Stelle prüft Securify, ob der Wert einer Variable überprüft wurde bevor sie durch Aufruf des EVM *sstore*-Befehls in den Speicher des Smart Contractes geschrieben wird.

2.2.15 Unhandled Exception (Securify), Misshandled Exception (Oyente)

Smart Contracts können sich mithilfe der Methode *call()* untereinander selbst aufrufen. Dabei schickt der sendende Contract A mit *call()* eine Transaktion an den empfangenden Contract B.
Contract A wartet nun die Ausführung im Contract B ab, kann aber nicht wissen, ob diese erfolgreich verläuft oder ob innerhalb von Contract B eine Exception geworfen wird. Sollte es innerhalb von Contract B zu einer Exception kommen, werden alle ausgeführten Schritte rückgängig gemacht und Contract A wird der Rückgabewert *false* als Hinweis auf den Fehler gesendet.
In einer mangelhaften Programmierung überprüft Contract A nicht den Rückgabewert der call()-Methode und nimmt naiverweise an, dass es zu keinem Fehler gekommen ist. Aufgrund eines unbehandelten Rückgabewertes der *call()*-Methode musste der bekannte Smart Contract *KingOfEther* öffentlich darum bitten, dass Nutzer keine Ether mehr an den Contract senden [33].

2.2.16 Transaction Ordering Dependency (Securify, Oyente)

In der Ethereum Blockchain wird etwa alle 12 Sekunden ein neuer Block hinzugefügt [34]. Dieser Block enthält alle Transaktionen, die seit dem letzten Block in dem Netzwerk getätigt wurden. Dabei muss die genaue Reihenfolge innerhalb des Blocks nicht der tatsächlichen Reihenfolge entsprechen.
Der bearbeitende Miner kann die Transaktionen innerhalb des Blockes nach Belieben anordnen. Da sich der Zustand des Smart Contracts abhängig von den Transaktionen verändert, kann ein Miner die Transaktionen so anordnen, dass er von einer bestimmten Anordnung profitiert. An dieser Stelle könnte er dann seine eigene Transaktion einbinden.

Transaction-Ordering-Dependency Angriffe wurden häufig in *Initial Coin Offerings (ICOs)* beobachtet. In ICOs bieten Start-Ups eine Kryptowährung zum Verkauf an, um finanzielle Mittel zu sammeln. Dabei bieten sie - solang die Nachfrage geringe ist - einen hohen Bonus an, der Bonus sinkt mit dem Anstieg der Nachfrage. Miner haben dieses Verhalten ausgenutzt, indem sie ihre Transaktionen zum Zeitpunkt einer niedrigen Nachfrage platziert haben [3].

2.2.17 Timestamp Dependency (Oyente)

Da die Ausführung der Ethereum Virtual Machine deterministisch ist, ist es eine besondere Herausforderung, innerhalb der Ausführung von Smart Contracts zufällig generierte, nichtdeterministische Werte zu erzeugen. Deswegen wird häufig auf pseudo-zufällige Zahlen zurückgegriffen, die mit dem Verlauf der Blockchain zusammenhängen.
Als beliebte Methode hat sich das Wählen des Hash-Wertes oder des Timestamp-Wertes eines Blocks erwiesen, der zu einem bestimmten Zeitpunkt in der Zukunft erstellt wird. Da nicht vorherzusagen ist, welche Transaktionen zu dieser Zeit ausgeführt werden, ist auch der zugehörige Hash-Wert oder Timestamp nicht vorhersehbar.
Da Miner die Transaktionen und die Reihenfolge dieser innerhalb eines Blockes festlegen können, ist es für einen böswilligen Miner möglich, den Inhalt seines Blockes so zu strukturieren, dass der Hash-/ oder Timestamp-Wert mit der pseudo-zufälligen Zahl im Smart Contract übereinstimmt.
Eine Studie [35] belegt, dass dafür sogar nur begrenzte Ressourcen notwendig sind.

3 Versuchsaufbau, Durchführung und Auswertung

3.1 Fragestellung

Wie bereits gezeigt, haben Ethereum Smart Contracts in vielen Bereichen potentielle Einsatzmöglichkeiten. Dabei übernehmen sie komplexe Aufgaben, die mit einem hohen Risiko verbunden sein können.
Aufgrund des jungen Alters der Technologie ist es in der Vergangenheit immer wieder zu Situationen gekommen, in denen ein Smart Contract fehlerhaft entwickelt wurde, sodass ein oft hoher Schaden entstanden ist.
Auch wenn es bereits eine Vielzahl an Programmen gibt, die bei der Erstellung eines sichereren Quellcodes behilflich sein sollen, müssen diese meist über eine komplexe Kommadozeile gesteuert werden oder sie sind anderweitig schwer in den Entwicklungsprozess einzubinden.
Die Syntax der Programmiersprache Solidity ähnelt in vielen Aspekten der Syntax anderer verbreiteter, objektorientierter Programmsprachen. Dadurch lassen sich viele, für Entwickler bekannte, objektorientierte Metriken auf die Programmiersprache Solidity übertragen.

Aufgrund dieser Tatsache soll in dieser Arbeit ein Modell entwickelt werden, welches - basierend auf gängigen Quellcodemetriken - für die Solidity Programmiersprache eine Aussage über das potentielle Sicherheitsrisiko eines betrachteten Smart Contracts treffen kann.
Dadurch könnte Entwicklern ein Rahmen gegeben werden, der ihnen bei der Entwicklung von Smart Contracts eine Orientierung gibt, sodass es in Zukunft zu weniger Sicherheitslücken in, und Schäden aufgrund von, Smart Contracts kommt.

3.2 Hypothesen

In den Messungen der verwendeten Smart Contracts werden Komplexitäts- und Strukturmetriken erhoben. Ausgehend von der Annahme, dass die Fehleranfälligkeit und das Sicherheitsrisiko in Programmen mit ansteigender Komplexität und abnehmender Struktur zunimmt, wird dieses Verhalten hier auch in Smart Contracts erwartet.

Daraus ergeben sich die zwei Hypothesen **Hypothese 1** und **Hypothese 2** mit den entsprechenden Gegenhypothesen:

Hypothese 1: „Ein Smart Contract mit einer hohen Komplexität und niedrigen Struktur - widergespiegelt in den Messwerten der Metriken - wird einer Kategorie mit einem hohen Sicherheitsrisiko zugeordnet.“

Gegenhypothese Hypothese 1: „Ein Smart Contract mit einer hohen Komplexität und niedrigen Struktur - widergespiegelt in den Messwerten der Metriken - wird einer Kategorie mit einem niedrigem Sicherheitsrisiko zugeordnet. “

Hypothese 2: „Ein Smart Contract mit einer niedrigen Komplexität und einer hohen Struktur - widergespiegelt in den Messwerten der Metriken - wird einer Kategorie mit einem niedrigen Sicherheitsrisiko zugeordnet. “

Gegenhypothese Hypothese 2: „Ein Smart Contract mit einer niedrigen Komplexität und einer hohen Struktur - widergespiegelt in den Messwerten der Metriken - wird einer Kategorie mit einem hohem Sicherheitsrisiko zugeordnet. “

Darüber hinaus davon wird aufgrund der hohen positiven Korrelation (siehe Abbildung 23) zwischen den Korrelationsmetriken angenommen, dass hier nur eine Teilmenge für den Entscheidungsbaum relevant sein wird.

3.3 Verwendete Technologien

3.3.1 Sicherheits-Scanner

Für die Risikobewertung der gesammelten Smart Contracts wurden die drei statischen Analysewerkzeuge Slither [36], Securify [38] und Oyente [52] gewählt.
Sie sind für die Aufgabe besonders gut geeignet, da sie für die Bewertung der Objekte verschiedene Methoden der Analyse anwenden und dafür unterschiedlichen Input nehmen. Ausgehen davon decken sie eine große Menge an aktuellen Sicherheitsrisiken ab.
Im Folgenden werden die Scanner und ihre jeweilige Methode der Sicherheitsanalyse genauer beschrieben.

3.3.1.1 Slither Slither [36] ist ein open-source Werkzeug zur statischen Quellcodeanalyse von Ethereum Smart Contracts, das von dem Unternehmen Trail of Bits [37] entwickelt wurde. Nach Angaben der Entwickler kann Slither zur automatisierten Erkennung von Sicherheitslücken, zur Quellcodeoptimierung, als Hilfe zum Verständnis von Quellcode sowie als Hilfe beim Codereview eingesetzt werden [2].

Da in dieser Arbeit die automatisierte Erkennung von Sicherheitslücken genutzt wird, wird im Folgenden - ergänzend zu Abbildung 12 - auf den technischen Ablauf dieser Funktion eingegangen.

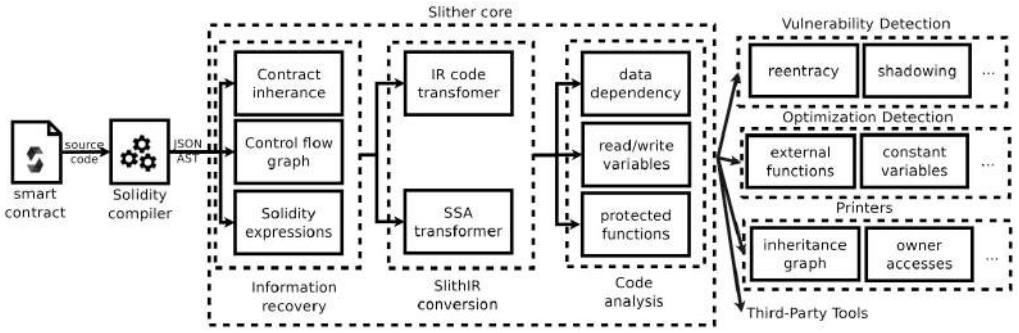

Abbildung 12: Oberflächlicher Ablauf der automatisierten Erkennung von Sicherheitslücken mittels Slither [2]

Als Input nutzt Slither den Abstrakten Syntax Baum (AST) des zu untersuchenden Smart Contractes. Dieser wird vom Solidity Compiler während der Übersetzung erstellt und kann somit verwendet werden. Im ersten Schritt der Analyse erhebt Slither weitere Daten über den Smart Contract, dazu gehören der Vererbungsbaum, die Kontrollflussgraph sowie eine Liste der verwendeten Ausdrücke.
Daraufhin wird der Quellcode in die SSA-Form (engl.*static-single assignment*) übersetzt, dafür wird die interne Repräsentationsform *SlithIR* genutzt.
In der dritten Stufe, der eigentlichen Code-Analyse, durchläuft Slither drei vordefinierte Analysen, die den verschiedenen Modulen ihre Informationen liefern. Zu den Analysen gehören **schreibende/lesende Zugriffe von Variablen** (engl. read/write variables), **Datenabhängigkeiten** (engl. data dependency) und **Geschützte Funktionen** (engl. protected functions).

schreibende/lesende Zugriffe von Variablen

Innerhalb des Kontrollflussgraphes werden für jeden Knoten die lesenden oder schreibenden Zugriffe der enthalten Variablen untersucht, wobei zusätzlich zwischen lokalen und globalen Variablen unterschieden wird. Damit kann beispielsweise untersucht werden, von welchen Stellen im Quellcode aus eine bestimmte Variable angesprochen werden kann.

Datenabhängigkeiten

Mithilfe der SSA-Form berechnet Slither die Datenabhängigkeiten für alle Variablen im Quellcode. Dabei werden die Abhängigkeiten zunächst im Rahmen der

Funktionen ermittelt, bevor dann Abhängigkeiten darüber hinaus betrachtet werden. So können Abhängigkeiten zwischen mehreren Aufrufen erkannt werden.
Eine Variable wird zusätzlich markiert, falls eine Abhängigkeit zu einer Variable besteht, die vom Nutzer beeinflusst werden kann.
Zuletzt berücksichtigen die erhobenen Datenabhängigkeiten die Informationen über geschützte Funktionen, wodurch die Abhängigkeiten ausgehend von den Rechten der spezifischen Nutzer untersucht werden können.

Geschützte Funktionen

Eine weit verbreitete Zugangsbeschränkung in Ethereum Smart Contracts ist die Verwendung von Eigentumsrechten für Funktionen.
Durch diese Funktion kann ein Nutzer oder eine Gruppe von Nutzer als Eigentümer der Funktion festgelegt werden, wodurch er bestimmte privilegierte Ausführungspfade durchlaufen kann.
Durch die Analyse von geschützten Funktionen kann Slither die Anzahl an *false positives*, Meldungen von Fehlern die eigentlich keine Fehler sind, verringern [2].

3.3.1.2 Securify Securify [38] ist ein statisches open-source Analysewerkzeug, das zur Erkennung von Sicherheitslücken genutzt werden kann.
Securify wurde ausgehend von der Feststellung entwickelt, dass es möglich ist, präzise Muster zu entwerfen, die auf dem Datenfluss von Smart Contracts basieren. Dabei kann bei der Untersuchung spezifischer Contracts eine Übereinstimmung oder eine Verletzung des Musters festgestellt werden, wodurch eine Sicherheitseigenschaft als bestätigt oder widerlegt angesehen werden kann.
Die grundlegende Idee besteht also darin, für jede betrachtete Sicherheitseigenschaft ein Übereinstimmungsmuster sowie ein Verletzungsmuster zu entwickelt. Um diese Muster zu prüfen, wird der Abhängigkeitsgraph für einen Smart Contract aufgebaut, in einem *geschichteten Datalog* (engl. *stratified Datalog*) [39] Programm kodiert und daraufhin mithilfe von *Datalog-Solvern* analysiert.
Um die Menge der betrachteten Sicherheitseigenschaften problemlos erweitern zu können, werden die Muster in einer domänenspezifischen Sprache (DSL, engl. *domain-specific language*) definiert.

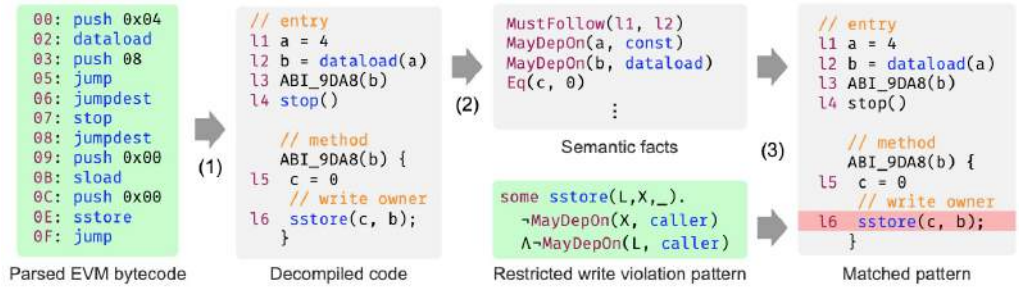

Abbildung 13: Oberflächlicher Ablauf einer Erkennung der *unrestricted-write*-Schwachstelle mithilfe von Securify [3]

Input

Als Input für die Analyse benötigt Securify den EVM Bytecode sowie eine Menge an Sicherheitsmustern in der dafür vorgesehenen DSL. Securify kann als Input ebenfalls den Solidity Quellcode eines Contractes erhalten, dann wird dieser zuerst in den EVM Bytecode übersetzt bevor die Analyse beginnt.

Dekompilierung des EVM Bytecodes

Der als EVM Bytecode übergebene Quellcode wird im ersten Schritt mithilfe der SSA-Form in eine vom Speicher unabhängige Darstellung transformiert. Dabei wird der Quellcode um Hilfsvariablen erweitert, sodass keine spezifischen Adressen im Speicher mehr vorhanden sind.

Abgesehen davon werden Methoden im Quellcode identifiziert und gekennzeichnet.

Erfassung von semantischen Fakten

Im nächsten Schritt werden aus der SSA-Form des Quellcodes semantische Eigenschaften ermittelt. Dazu zählen Daten- und Kontrollfluss-Abhängigkeiten, die das Verhalten und die Struktur des Smart Contractes beschreiben. Als Beispiel für erhobene Datenabhängigkeiten sind *MayDepOn*(b, dataload) und *MustFollow*(l1, l2) in Abbildung 12 zu sehen. Dabei beschreibt *MayDepOn*, dass der Wert der Variable b eventuell vom Rückgabewert des Befehls dataload abhängt. *MustFollow* hält fest, dass die Instruktion l2 immer nach Instruktion l1 ausgeführt wird.

Prüfung der Muster

Im letzten Schritt der Analyse prüft Securify die als Input übergebenen Sicherheitsmuster mithilfe der semantischen Fakten. Ein Sicherheitsmuster, definiert in seiner spezifischen DSL, besteht dabei aus einer Menge von semantischen Fakten, welche mithilfe von logischen Operatoren zu Ausdrücken verbunden werden. Das Verletzungsmuster für die Schwachstelle *unrestricet-write* ist wie folgt definiert:

$$somesstore(L,X,\_).\neg MayDepOn(X,caller) \wedge \neg MayDepOn(L,caller)$$

Das Muster wird gegen jeden *sstore*-Befehl im Quellcode geprüft. Dabei wird untersucht, ob der Zugriff auf die Speicheradresse des Befehls oder die Ausführung des Befehls abhängig vom Rückgabewert der *caller*-Instruktion ist. In dem Muster wird die Speicheradresse mit X und die Ausführung mit L beschrieben.
Da die Instruktion *caller* die Adresse des Aufrufers der Transaktion abruft, bedeutet eine Erfüllung des Musters, dass der betroffene *sstore*-Befehl von jedem Nutzer ausgeführt werden kann.

Output

Nach der Analyse macht Securify für jede untersuchte Eigenschaft eine Aussage. Die Aussagen unterteilen sich dabei in die drei Möglichkeiten Erfüllung, Verletzung und Warnung.
Im Fall der Erfüllung wurde das Übereinstimmungsmuster der Eigenschaft erfüllt, im Fall der Verletzung wurde das Verletzungsmuster erfüllt. Wird keines der beiden Muster erfüllt, wird eine Warnung für die Eigenschaft ausgegeben; hier kann Securify keine klare Aussage treffen und bittet den Nutzer um manuelle Überprüfung.

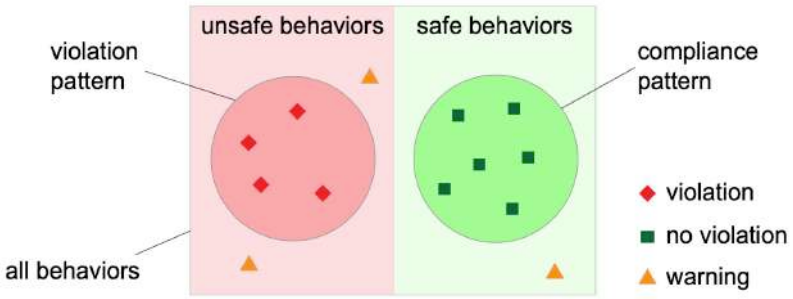

Abbildung 14: Output von Securify, unterteilt in Erfüllung (compliance), Verletzung (violation) und Warnung (warning) [3]

3.3.1.3 Oyente Oyente [52] ist ein open-source Programm, das symbolische Analyse [40] verwendet, um Sicherheitslücken in Ethereum Smart Contracts zu identifizieren. In der symbolischen Ausführung werden symbolische Ausdrücke, an Stelle von konkreten Eingabewerten genutzt, um die Ausführungspfade des Quellendes zu ermitteln. Dabei steht der symbolische Ausdruck für eine Menge an Eingabewerten, die den Quellcode einen bestimmten Ausführungspfad durchlaufen lassen.
Die Pfade werden dabei auch symbolische Pfade genannt. Als Eingabe verwendet Oyente zwei Parameter, zum einen den EVM Bytecode des zu untersuchenden Smart Contractes, zum anderen den momentanen globalen Zustand der Ethereum Blockchain. Die Analyse (siehe Abbildung 15) beruht auf den Komponenten *CFGBuilder*, *Explorer*, *Core-Analysis* und *Validator*.

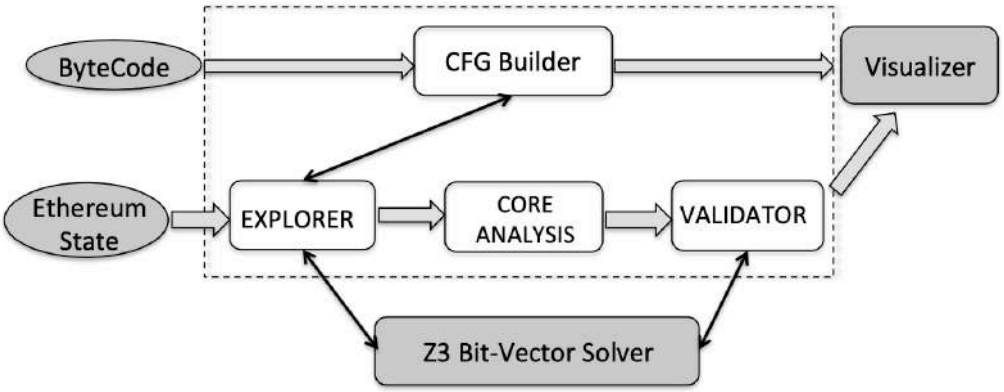

Abbildung 15: Oberflächer Ablauf einer Sicherheitsanalyse mittels Oyente [4]

CFG Builder Zu Beginn der Analyse wird die Komponente *CFG Builder* genutzt, um den Kontrollflussgraphen des Smart Contractes aufzubauen. Dabei werden Basisblöcke gebildet, die Teile des Quellcodes enthalten. Diese werden über Kanten verbunden, welche den Kontrollfluss repräsentieren.
Die Kanten werden im Laufe der Analyse weiterhin vervollständigt.

Explorer Der *Explorer* startet am Eingangsknoten des zuvor erstellen Kontrollflussgraphen. Von hier aus läuft er in einer Schleife, in der er von einem Zustand aus eine Instruktion symbolisch ausführt und so in einen Folgezustand kommt. Die Schleife läuft so lange bis keine neuen Zustände hinzugefügt werden oder eine vom Nutzer definierte Zeitgrenze erreicht ist.
Zur Auswertung von bedingten Sprüngen im Kontrollflussgraphen wird Z3 [41] genutzt. Kann Z3 eine Bedingung ausgehend vom bisher gelaufenen Pfad bestimmen, wird der Pfad in diese Richtung erweitert. Ist dies nicht der Fall, werden beide Bedingungen als möglich gesehen und beide Pfade werden nach der Tiefensuche durchlaufen.
Nachdem die Schleife des Explorers terminiert ist, wird die erstellte Liste an symbolischen Pfaden ausgegeben. Dabei werden nur Pfade mit unterschiedlichen Flüssen an Ether ausgegeben, sodass die folgende Analyse optimiert wird.

Core Analysis In *Core Analysis* werden eigene Komponenten zur Identifizierung der Sicherheitslücken *Transaction Ordering Dependency*, *Timestand dependecy*, *Mishandled Exception* und *Reentrancy* genutzt, die wie folgt arbeiten.

- *Transaction Ordering Dependency:* Es werden die vom Explorer gelieferten Ausführungspfade zusammen mit ihren jeweiligen Ether-Transaktionen betrachtet. Es wird verglichen, ob zwei unterschiedliche Ausführungspfade einen unterschiedlichen Verlauf an Ether im Smart Contract ausweisen. Ist dies der Fall wird die Schwachstelle vermerkt.

- *Timestand dependecy:* Der Zeitstempel von Blöcken wird in Oyente durch eine spezielle symbolische Variable repräsentiert.
  Ist ein beliebiger Ausführungspfad von dieser Variable abhängig, wird die Schwachstelle vermerkt.

- *Mishandled Exception:* Es wird nach der Ausführung der EVM Instruktion ISZERO gesucht. Eine Exception schreibt den Wert 0 auf den Speicher der Aufrufers. Die EVM-Instruktion ISZERO prüft, ob dies geschehen ist.
  Die Schwachstelle wird vermerkt, falls die Instruktion in keinem Ausführungspfad vorhanden ist, da dann an keiner Stelle auf eine Exception geprüft wurde.

- *Reentrancy:* In der Erkennung dieser Schwachstelle werden die bedingten Pfade im Kontrollflussgraphen genutzt. Für jede CALL-Instruktion im Bytecode wird die bedingte Kante ausfindig gemacht, von der das Erreichen der Instruktion abhängig ist. Nun wird überprüft, ob die Bedingung auch nach der Ausführung der CALL-Instruktion noch gilt und die Instruktion so ein zweites Mal erreichbar ist.

Validation Im letzten Schritt wird die Komponente *Validation* genutzt, um mögliche falsche Fehlermeldungen zu erkennen. Da der gesamte globale Zustand der Ethereum Blockchain während der Analyse nicht berücksichtigt werden kann, beschränkt sich diese Komponente auf manuelle Überprüfungen.

3.3.2 SolMet Solidity Parser

Zur Erhebung der Codemetriken wurde das Programm SolMet [42] gewählt. Es wurde im Rahmen einer wissenschaftlichen Untersuchung [43] zur Komplexität von Ethereum Smart Contracts entwickelt.
Die durch statische Analyse erhobenen Metriken werden in der Untersuchung wie folgt vorgestellt:

Nummer	Metrik	Beschreibung
1	SLOC (souce code lines)	Anzahl der Zeilen welche Quellcode enthalten
2	LLOC (logical code lines)	Anzahl der Zeilen welche logische Anweisungen enthalten (Zeilen die nicht leer sind oder keine Kommentare sind)

3	CLOC (comment code lines)	Anzahl der Zeilen welche Kommentare sind
4	NF (number of functions)	Anzahl der Funktionen in Quellcode
5	McCC (McCabe)	Komplexität nach McCabe [44]. Für einen Smart Contract wird der durchschnittliche McCabe Wert aller Funktionen gemessen
6	WMC (weighted methods McCabe)	Die Summe der Gewichte aller Funktionen, gewichtet nach McCabe
7	NL (nesting level)	Die Summe der tiefsten Verschachtelungsebene von allen Funktionen im Smart Contract
8	NLE (nesting level else-if)	Die Summe der tiefsten Verschachtelungsebene von allen Funktionen im Smart Contract, else-if Pfade ausgeschlossen
9	NUMPAR (number of parameters)	Die durchschnittliche Anzahl an Parametern die Funktionen im Smart Contract übergeben bekommen
10	NOS (number of statement)	Die durchschnittliche Anzahl der Anweisungen innerhalb aller Funktionen des Smart Contractes
11	DIT (depth of inheritance Tree)	Die Tiefe des vorhandenen Vererbungsbaumes
12	NOA (number of ancestors)	Die Anzahl der Vorgänger im Vererbungsbaum
13	NOD (number of descendants)	Die Anzahl der Nachfolger im Vererbungsbaum

14	CBO (coupling between object classes)	Die Anzahl der externen Objekte, die im Smart Contract innerhalb von Attributen, Funktionsparametern oder Rückgabewerten genutzt werden
15	NA (number of attributes)	Die Anzahl der globalen Variablen im Smart Contract
16	NOI (number outgoing invocations)	Durchschnittlicher Wert, wie viele externe Funktionsaufrufe innerhalb aller Funktionen im Smart Contract getätigt werden. Kopplung durch Funktionsaufrufe soll widergespiegelt werden.

3.4 Aufbau und Durchführung

3.4.1 Beschaffung und Aufbau der Testdaten

Aufgrund der Transparenz der Daten in der Ethereum Blockchain gibt es einige Internetseiten, die als Suchmaschine fungieren. Hier können die gesamte Entwicklung der Blockchain sowie alle enthaltenen Accounts durchsucht werden. Eine der bekanntesten dieser Suchmaschinen ist etherscan.io, hier werden alle aktuell verifizierten ERC-20 Token Smart Contracts der Blockchain gelistet [45].
Für die in dieser Liste enthaltenen Smart Contracts ist die Adresse in der Blockchain hinterlegt, an der der Quellcode hinterlegt ist. Folgt man dieser Adresse in der Suchmaschine, kann man den Quellcode lesen.
Für die Sammlung des Quellcodes aller gelisteten Elemente wurde ein eigenes Java-Programm implementiert, welches durch die Liste wandert, für jedes Element die Quellcodeseite öffnet und den Quellcode in einer lokalen Datei speichert.
Durch dieses Verfahren konnten ursprünglich 798 Quellcodedateien von unterschiedlichen Smart Contracts gesammelt werden. Diese wurden in einer Ordnerstruktur angelegt, sodass eine Automatisierung der Messung gut umsetzbar war. Nach den ersten Evaluationen der Testdaten mithilfe der Sicherheitsscanner musste die Menge der Testobjekte auf eine Anzahl von 470 Quellcodedateien verringert werden.

Dies lag daran, dass der Quellcode von 328 Dateien für die Sicherheitsscanner fehlerhaft war und diese keine verwertbaren Ergebnisse liefern konnten.
Die verwertbaren Dateien wurde in einer Ordnerhierarchie strukturiert. Dabei wurde in einem Oberordner für jedes Objekt ein Unterordner angelegt, der den Namen des Smart Contracts trägt und in dem die Quellcodedatei platziert wurde.
Diese Struktur hilft, im Verlauf der Messungen eine eindeutige Zuordnung zwischen Testobjekt und Analyseergebnis zu bewahren.

3.4.2 Strukturierung der Messungen und Sammlung der Ergebnisse

Der Ablauf der Messungen sowie die folgende Sammlung der Ergebnisse wurde mithilfe von Shell-Skripten automatisiert. Dafür wurde für jeden Scanner ein Skript geschrieben, welches durch die Ordnerstruktur läuft und für jedes Testobjekt eine Analyse durchführt. Die Ergebnisse wurden in einer JSON-Datei festgehalten. Die Analysen der drei Sicherheitsscanner wurden parallel durchgeführt.
Nachdem die Analysen abgeschlossen waren, wurde das Skript *categorizeContracts.sh* (Abbildung 16) ausgeführt, welches ebenfalls die Ordnerstruktur durchläuft und dabei die Ergebnisse interpretiert. In diesem Skript wurde eine if-then-else-Struktur durchlaufen, welche die gefundenen Sicherheitslücken abfragt und das Testobjekt, einer Kategorie zuordnet. Das Ergebnis wurde wieder in einer Datei festgehalten.
Neben den Analysen wurden die OO-Metriken der Testobjekte mithilfe des Skriptes *solMet.sh* (Abbildung 16) erfasst. Das Skript läuft durch die Ordnerstruktur, erfasst für jedes Testobjekt seine Metriken und speichert diese in einer Datei.
Nachdem nun für jedes Objekt die entsprechende Kategorie und die OO-Metriken erfasst wurden, wurden alle Daten in Tabellenform gesammelt. Dabei enthält eine Zeile den Smart Contract Namen, die Metriken und die erfasste Kategorie.

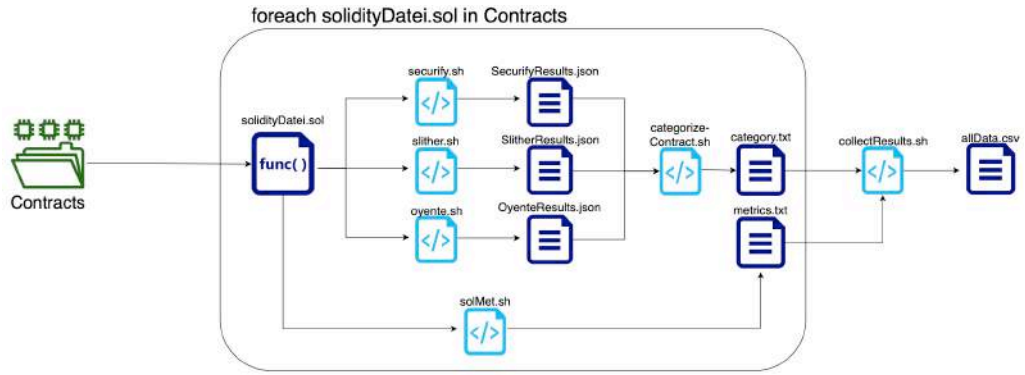

Abbildung 16: Ablauf der Analysen und Sammlung der Ergebnisse

3.4.3 Kategorisierung der Smart Contracts

Um eine Aussage über die Sicherheit und Qualität der Smart Contracts machen zu können, werden diese, ausgehend von den Ergebnissen der Sicherheitsscanner, verschiedenen Kategorien zugeteilt.
Die drei geformten Sicherheitskategorien sind *Ohne Warnung*, *Warnung vorhanden* und *Bekannte Warnung vorhanden*.

Nummer	**Name**	**Beschreibung**
1	Ohne Warnung	In dem Smart Contract hat kein verwendeter Sicherheitsscanner eine potentielle Sicherheitslücke vermerkt
2	Warnung vorhanden	Mindestens einer der Sicherheitsscanner hat in dem Smart Contracts eine potentielle Sicherheitslücke vermerkt
3	Bekannte Warnung vorhanden	Mindestens einer der Sicherheitsscanner hat im Smart Contract eine Warnung für eine der bekannten Sicherheitslücken ausgegeben

Die drei Kategorien können auf einer Ordinalskala wie folgt sortiert werden:

$$\textit{Ohne Warnung} < \textit{Warnung vorhanden} < \textit{Bekannte Warnung vorhanden}$$

Die am wenigsten gewichtete Kategorie ist **Ohne Warnung**, da hier keiner der Sicherheitsscanner ein potentielles Risiko entdecken konnte.
Die stärker gewichtete Kategorie heißt **Warnung vorhanden**. Hier hat mindestens ein Scanner eine potentielle Gefahr erkannt. Da die verwendeten Scanner öffentlich verfügbar sind, ist es jeder beliebigen Person möglich, diese potentielle Gefahr zu erkennen und auszunutzen. Dadurch geht von den Smart Contracts dieser Kategorie ein höheres Risiko aus.
Die dritte und schwerwiegendste Kategorie ist **Bekannte Warnung vorhanden**. Ein Smart Contract wird dieser Kategorie zugeordnet, wenn ein Scanner eine Warnung für die Sicherheitslücken *Reentrancy*, *LockedEther*, *RestrictedWrite*, *Unhandled / Misshandled Exception* oder *Suicidal* ausgegeben hat.
Diese Sicherheitslücken zeichnen sich dadurch aus, dass sie in der Vergangenheit mit

bekannten Angriffen [31] [46] [1] [47] [33] ausgenutzt wurden, wodurch ein Gesamtschaden von über 240 Millionen US-Dollar [3] entstanden ist.
Es sind öffentliche Erklärungen [31] [1] verfügbar, die das genaue Vorgehen zum Ausnutzen der genannten Sicherheitslücken beschreiben. So sind einem potentiellen Angreifer massive Hilfestellungen bei der Ausnutzung der Sicherheitslücken gegeben, wodurch ein höheres Risiko im Smart Contract besteht.

Einige der betrachteten Sicherheitslücken werden von mehr als einem Scanner gesucht. Dazu gehören *Locked Ether*, *Reentrancy*, *Unhandled / Misshandled Exception* und *Transaction Ordering Dependency*. Um dem Aufkommen von falschen Fehlermeldungen entgegenzuwirken, werden diese Sicherheitslücken nur berücksichtigt, wenn mindestens zwei der Scanner ein potentiellen Risiko erkannt haben.

3.4.4 Aufbau des Entscheidungsbaumes

Zum Aufbau des Entscheidungsbaumes wurde das Paket *rpart* [48] der Programmiersprache R genutzt. Der Datensatz wurde in die Untergruppen **Trainingssatz**, **Validierungssatz** und **Testsatz** unterteilt. Dabei wurden 70 Prozent (328 Objekte) dem Trainingssatz, 10 Prozent (48 Objekte) dem Validierungssatz und 20 Prozent (93 Objekte) dem Testsatz zugeordnet.
Nachdem der Baum mit dem Trainingssatz trainiert wurde, wurde der Validierungssatz genutzt, um eine optimale Performanz zu erzielen und einem over-fitting vorzubeugen. Dabei wurden Pre-/ und Post-Pruning [49] Methoden verwendet.

- **Pre-Pruning** Es wurden verschiedene Bäume erstellt, wobei bestimmte Eigenschaften variiert wurden. Dabei wurden die Parameter *maxdepth*, die maximale Tiefe, die der Baum annehmen kann, *minsplit*, die minimale Anzahl an Verzweigungen, die existieren müssen, sodass ein neuer Zweig erstellt wird, und *minbucket*, die minimale Anzahl an Werten in einem Blatt des Baumes, sodass dieses erzeugt wird, variiert.
  Nachdem die Bäume mit den verschiedenen Variationen an Eigenschaften erstellt wurden, wurde ihre Performanz auf dem Validierungsset gemessen und im Anschluss verglichen. Beim Parameter *maxdepth* wurden die Werte von 1 bis 8 untersucht, wobei das Optimum bei bei den Werten 5 bis 8 mit einer Performanz von 0,6667 liegt.
  Minsplit wurde mit den Werten von 1 bis 60 untersucht. Das Optimum liegt zwischen den Werten 15 und 20, hier hat jeder Wert ebenfalls eine Performanz von 0,6667.

[3] Die Zahl ergibt sich aus der Anzahl der betroffenen Ether multipliziert mit dem Ether-Preis zur Zeit der Angriffe

Im letzten Parameter *minbucket* wurden ebenfalls die Werte zwsichen 1 und 60 untersucht, das Optimum liegt hier bei den Werten zwischen 26 und 49, jeder Wert in diesem Intervall hat eine Performanz von 0,75.

- **Post-Pruning** Hier wurde der Baum einmal ohne eine vorher definierte Einschränkung an Parametern erstellt. So hatte der Algorithmus die freie Wahl und konnte einen, aus seiner Sicht, optimalen Baum erstellen.
  Nachdem der Baum erstellt wurde, wurde sein Verhalten mithilfe des Komplexität-Parameters untersucht. Dabei wurde der minimale *Cross Validation Error* bestimmt. Dieser wurde genutzt, um den Baum zu beschneiden. Die Performanz des dabei entstehenden Baumes wurde auf dem Validierungsset gemessen, wobei eine Performanz von 0,6667 gemessen wurde.

Durch den Vergleich der Performanz der verschieden konfigurierten Entscheidungsbäume wurde der Entscheidungsbaum mit dem Attribut *minbucket* - belegt mit einem Wert zwischen 26 und 49 - als optimaler Baum gewählt. Dieser wird in Abbildung 17 dargestellt.

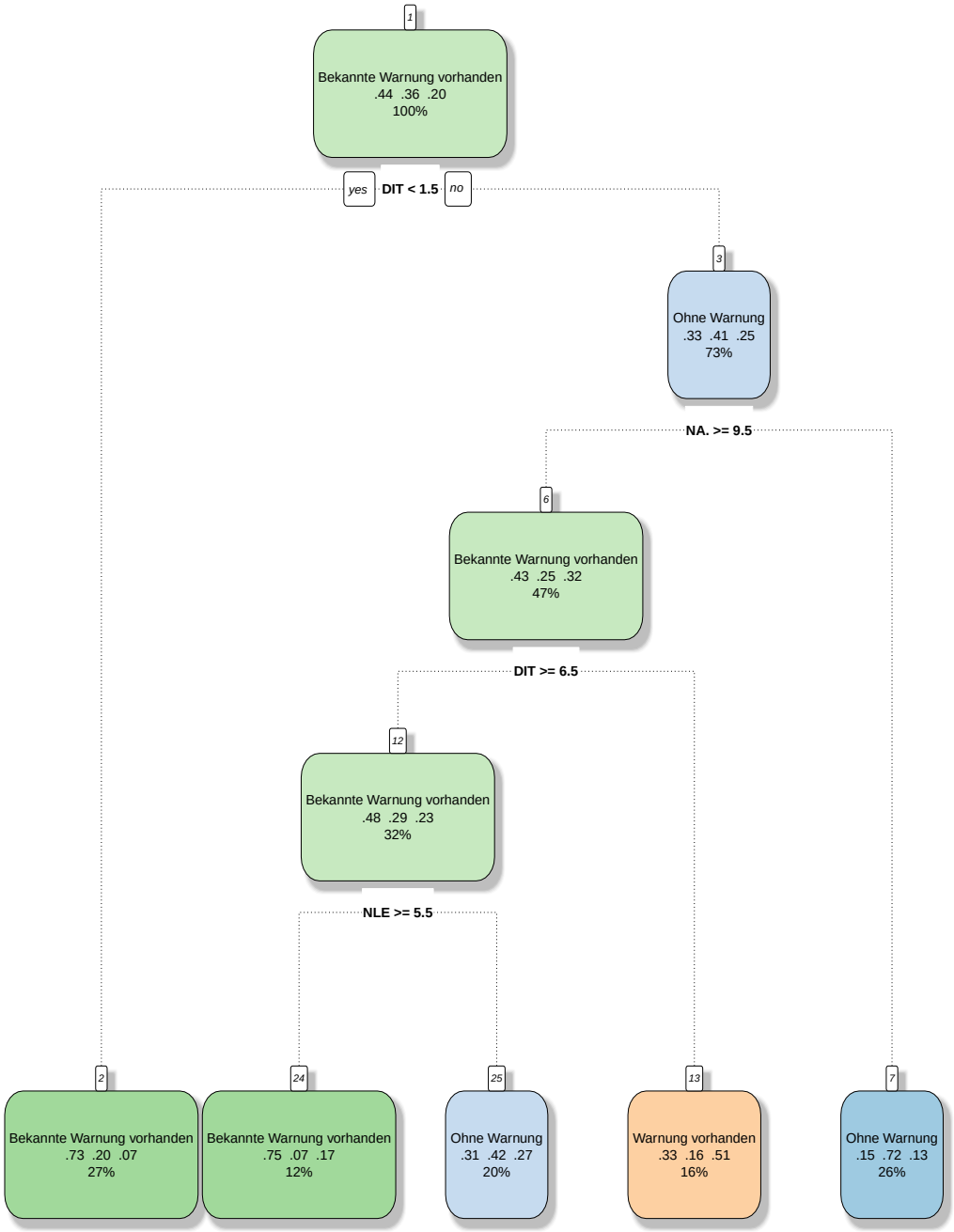

Abbildung 17: Entscheidungsbaum für eine Sicherheitskategorisierung für Ethereum Smart Contracts

3.5 Quantitative Analyse

Nachdem der Entscheidungsbaum mit dem Trainingssatz gebaut und mithilfe des Validierungssatzes optimiert wurde, wurde er auf dem Testsatz evaluiert. Die Auswertung der Ergebnisse sind in Abbildung 18 in einer Konfusionsmatrix dargestellt.

		TestSatz		
		Bekannte Warnung vorhanden	**Ohne Warnung**	**Warnung vorhanden**
Vorhersage	**Bekannte Warnung vorhanden**	29	12	3
	Ohne Warnung	3	24	3
	Warnung vorhanden	10	2	7

Abbildung 18: Konfusionsmatrix des Entscheidungsbaumes

Im Folgenden werden die Metriken *precision* und *recall* für die drei Methoden berechnet. Dabei werden die Formeln

$$precision = \frac{true\ positives}{true\ positives\ +\ false\ positives}\ ,$$
$$recall = \frac{true\ positives}{true\ positives\ +\ false\ negatives}$$

genutzt.

- **Bekannte Warnung vorhanden**
  In dem Testsatz waren 44 Objekte mit einer Warnung für eine bekannte Sicherheitslücke vorhanden. Davon hat das Modell 29 korrekt erkannt. Es ergeben sich $precision = \frac{29}{42} = 0,69$ und $recall = \frac{29}{44} = 0,65$.

- **Ohne Warnung**
  In dem Testsatz waren 30 Objekte ohne Warnung vorhanden, von welchen das Modell 24 korrekt erkannt hat. Daraus ergeben sich $precision = \frac{24}{38} = 0,63$ und $recall = \frac{24}{30} = 0,8$.

- **Warnung vorhanden**
  Bei 19 Objekten des Testsatzes waren Warnungen vorhanden. Das Modell hat 7 davon identifiziert. Es ergeben sich $precision = \frac{7}{13} = 0,53$ und $recall = \frac{7}{19} = 0,36$.

In den Ergebnissen ist zu erkennen, dass eine Zuordnung in die Kategorie *Warnung vorhanden* im Vergleich deutlich unzuverlässiger ist. In der folgenden qualitativen Analyse werden Ursachen hierfür genannt.

3.6 Qualitative Analyse

Es werden die Eigenschaften des Entscheidungsbaums, sowie die Ergebnisse der quantitativen Analyse, genauer untersucht. Dabei liegt der Fokus auf der Eigenschaft, dass eine Zuordnung in die Kategorie *Warnung vorhanden* am unzuverlässigsten ist.
Die Blätter des Baums enthalten mehrere Attribute, welche nacheinander betrachtet werden. Wie in Abbildung 19 zu erkennen ist, ist jedes Blatt mit einer Nummerierung versehen, daraufhin folgt ein Text, der die Kategorie angibt, von der die meisten Objekte diesem Blatt zugeordnet wurden. Darunter stehen drei Kennzahlen, die die Wahrscheinlichkeit angeben, dass das hier gelandete Objekt einer jeweiligen Kategorie angehört [50]. Dabei gehört die linke Zahl zur Kategorie *Bekannte Warnung vorhanden*, die mittlere Zahl zur Kategorie *Ohne Warnung* und die rechte Zahl zur Kategorie *Warnung vorhanden*. Die letzte, zentral dargestellte Prozentzahl im Blatt gibt Auskunft über die Anzahl der Objekte von der Gesamtmenge im Testdatensatz, die in diesem Blatt gelandet sind.
Um die Plausibilität zu bewerten, werden die Verteilungen der entscheidenden Metriken im Testdatensatz, dargestellt in Abbildung 20, 21 und 22, berücksichtigt.

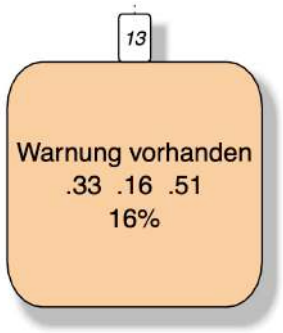

Abbildung 19: Beispiel eines Blattes im Vererbungsbaum

- **Blatt 2**
  Ein Element, das im ersten Blatt von links landet, gehört mit einer Wahrscheinlichkeit von 73 Prozent zur Kategorie *Bekannte Warnung vorhanden*. Zu 20 Prozent ist es in der Kategorie *Ohne Warnung*, zu 7 Prozent in *Warnung vorhanden*. Die Zuteilung hängt nur von der Entscheidung ab, ob die Vererbungstiefe kleiner als 1,5 ist. Schaut man sich die Verteilung der Metrik *DIT* zur Kategorie *Bekannte Warnung vorhanden* in Abbildung 20 an, so sind die Elemente in dem Bereich kleiner 1,5 schnell wiederzufinden.

- **Blatt 24** Hier werden 75 Prozent der Objekte der Kategorie *Bekannte Warnung vorhanden* zugeordnet. Mit einer Wahrscheinlichkeit von 17 Prozent werden Objekte der Kategorie *Bekannte Warnung* und mit nur 7 Prozent der Kategorie *Ohne*

Warnung zugeordnet.
Ausschlaggebend dafür ist der Wert der Metrik *NLE*, der über 5 liegen muss. Ein Blick auf die Verteilung innerhalb der Kategorien in Abbildung 22 zeigt, dass diese Werte plausibel sind. Bei einem Wert von über 5 kommen in der Kategorie *Ohne Warnung* nur 3 Objekte in Frage. In den anderen beiden Kategorien sind es deutlich mehr.

- **Blatt 25**
  Auch wenn dieses Blatt mit der Kategorie *Ohne Warnung* betitelt ist, stimmt die Zuteilung nur zu einer Wahrscheinlichkeit von 42 Prozent. Da in diesem Blatt 20 Prozent der ausgewerteten Elemente landen, besteht eine hohe Wahrscheinlichkeit, dass ein Element fälschlicherweise der Kategorie *Ohne Warnung* zugeordnet wird. Wir folgen dem Entscheidungsbaum von der Wurzel an in Richtung Blatt 25 und betrachten dabei die Verteilung der Metriken in den einzelnen Kategorien. Im ersten Knoten werden alle Elemente mit geringer Vererbung (DIT < 1,5) entfernt. Im darauffolgenden Knoten 3 werden 26 Prozent der Objekte der Kategorie *Ohne Warnung* zugeteilt. Sie landen in Blatt 7, da ihre Anzahl der Attribute kleiner oder gleich 9,5 ist. In Abbildung 21 sind diese Elemente gut erkennbar.
  Die verbleibenden Elemente wandern weiter den Baum entlang, sodass bei Knoten 6 ein weiteres Mal die Entscheidungsbaumstiefe betrachtet wird. Hier werden alle Elemente mit einer Tiefe kleiner gleich 6,5 der Kategorie *Warnung vorhanden* zugeordnet (Blatt 13).
  Betrachtet man die DIT-Verteilung (Abbildung 20), so ist zu erkennen, dass es einige Ausreißer gibt, die einen DIT-Wert über 6,5 haben. Sie werden nicht zugeordnet und laufen im Entscheidungsbaum weiter. Sie verursachen falsche Positivmeldungen in den Blättern 24 und 25.

- **Blatt 13**
  Neben Blatt 25 ist Blatt 13 ein weiterer Verursacher von vielen falschen Positivmeldungen im Entscheidungsbaum. Die Elemente werden zu einer Wahrscheinlichkeit von 51 Prozent korrekt der Kategorie *Warnung vorhanden* zugeordnet. Dies deckt sich mit den Messwerten der quantitativen Analyse. Hier landen alle Elemente mit einer Vererbungstiefe zwischen 1,5 und 6,5 und einer Anzahl an Attributen größer gleich 9,5.
  In der Verteilung in Abbildung 21 ist zu erkennen, dass viele Elemente der Kategorie *Bekannte Warnung vorhanden* einen Wert *NA* größer gleich 9,5 haben. In der Vererbungstiefe der Kategorie *Bekannte Warnung vorhanden* in Abbildung 20 sind ebenfalls einige Elemente zu finden, die im betroffenen Intervall liegen. Diese machen die 33 Prozent im Blatt 13 aus und verursachen falsche Zuteilungen.

- **Blatt 7**

  Die Objekte, die in Blatt 7 landen, gehören mit einer Wahrscheinlichkeit von 72 Prozent zu der Kategorie *Ohne Warnung*. Wenn man den vorherigen Pfad im Baum betrachtet, sind die Elemente, deren Vererbungstiefe größer gleich 1,5 ist und welche weniger als 9,5 Attribute besitzen hier zu finden.

  Wirft man einen Blick auf die Verteilungen der Metriken innerhalb der Kategorie *Ohne Warnung*, so fällt schnell auf, dass ein Großteil der Element eine höhere Vererbungstiefe als 1,5 und gleichzeitig unter 9,5 Attribute besitzt. Es besteht eine Wahrscheinlichkeit von 28 Prozent, dass hier ein Objekt fälschlicherweise zugeordnet wird.

  Aus dem Weg im Baum von der Wurzel zu diesem Blatt, den damit verbundenen Eigenschaften der Objekte und der hohen Wahrscheinlichkeit einer korrekten Zuordnung, lässt sich das Muster erkennen, dass viele Objekte ohne Sicherheitswarnung eine bestimmt Vererbungstiefe und eine begrenzte Anzahl an globalen Attributen besitzt.

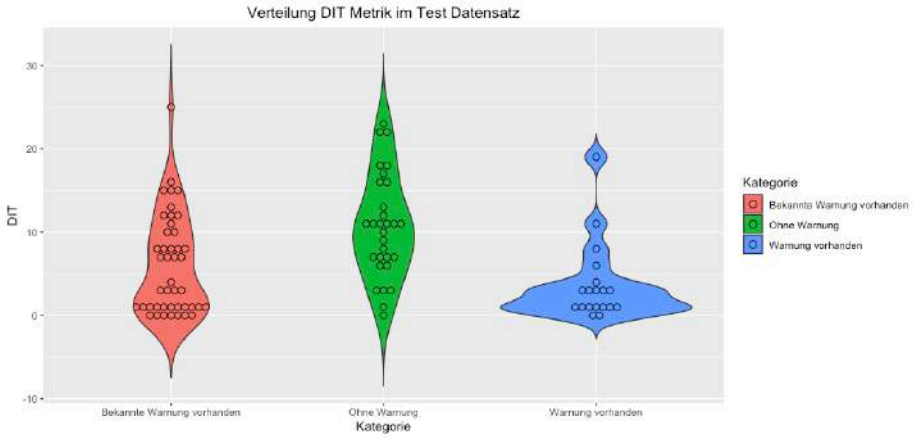

Abbildung 20: Verteilung der Metrik DIT im Testdatensatz

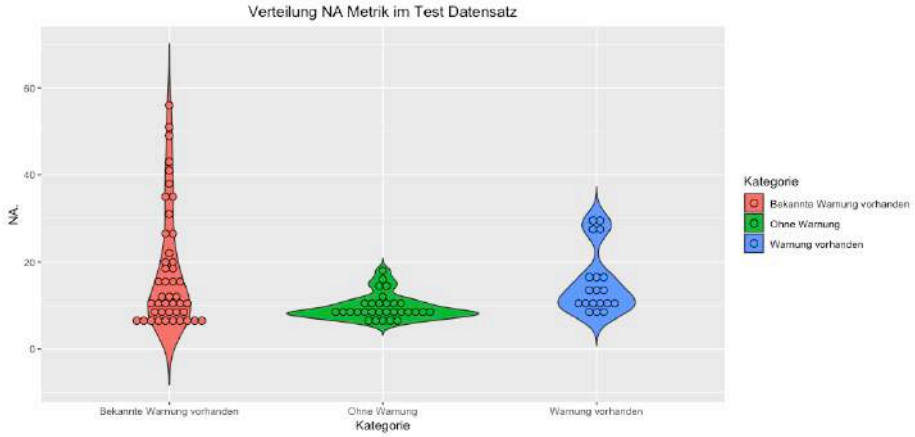

Abbildung 21: Verteilung der Metrik NA im Testdatensatz

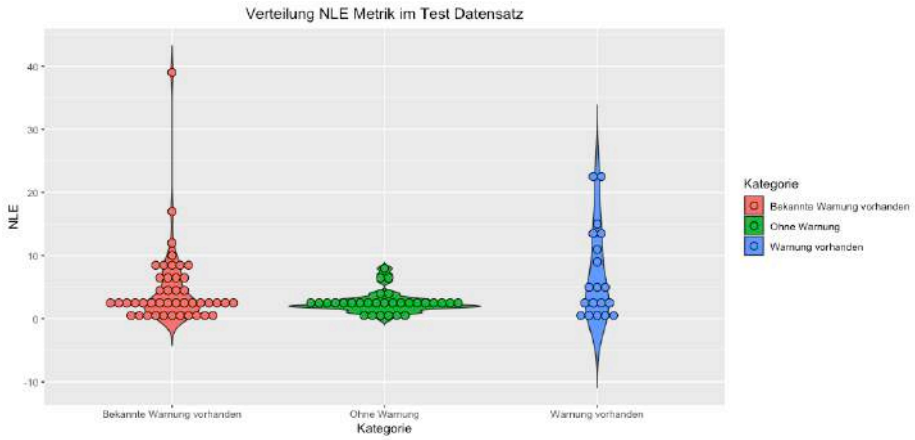

Abbildung 22: Verteilung der Metrik NLE im Testdatensatz

3.7 Diskussion

3.7.1 Korrelation zwischen Metriken

Abbildung 23 zeigt die paarweise Korrelationsmatrix nach Spearman der 19 verwendeten Software-Metriken. Dabei ist zu erkennen, dass nur positive Korrelationen zwischen den Metriken vorkommen, viele davon höher als 0,8. Zwischen den Metriken *{DIT, NOA, NOD}*, sowie *{NL, NLE}* und *{LLOC, NOS}* ist eine Korrelation von 1 zu erkennen.

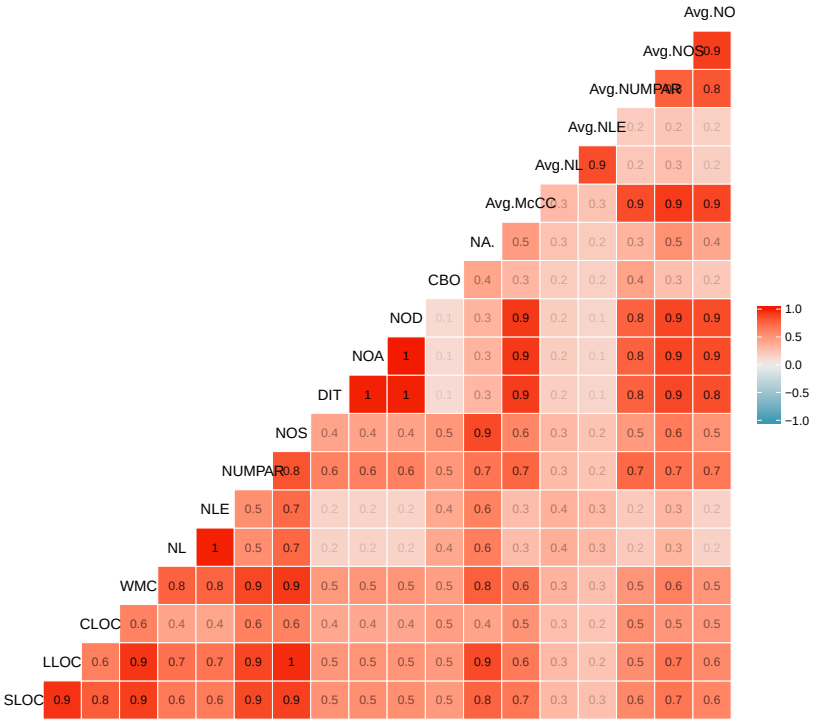

Abbildung 23: Korrelationsmatrix der OO-Metriken auf dem Datensatz

3.7.2 Struktur des Entscheidungsbaumes

Der mit dem Trainingssatz trainierte und mit dem Validierungssatz angepasste Entscheidungsbaum aus Abbildung 17 hat eine maximale Tiefe von 4.
Aus den 19 verfügbaren Metriken sind lediglich *DIT*, die Tiefe des Vererbungsbaumes, *NA*, die Anzahl an globalen Attribute, und *NLE*, die Summe der tiefsten Verschachtelungsebenen der Funktionen - else-if-Anweisungen ausgeschlossen - vertreten. Aufgrund der hohen Rate an stark positiven Korrelation zwischen den Metriken war es zu erwarten, dass viele davon nicht berücksichtigt werden.
Zu Beginn des Entscheidungsbaumes wird *DIT*, die Tiefe des Vererbungsbaumes, betrachtet. Dies ist somit die aussagekräftigste Metrik. Hier werden bereits 27 Prozent der Objekte der Kategorie **Bekannte Warnung vorhanden** zugeteilt, falls ihr Vererbungsbaum eine Tiefe kleiner oder gleich 1 hat. Daraus lässt sich bereits ableiten, dass eine ausführlich eingesetzte Vererbung zu einer geringeren Fehleranfälligkeit führen kann.
Im nächsten Schritt wird die Anzahl der globalen Attribute (NA) der verbleibenden 73 Prozent der Objekte untersucht. Hier wird bei einem Wert von 9,5 unterschieden, Objekte mit weniger als 10 globalen Attributen werden der Kategorie **Ohne Warnung** zugeteilt. Diesen Weg durchliefen 26 Prozent der gesamten Objekte im Testsatz und aus ihm lässt sich ableiten, dass eine korrekte Kopplung der Funktionalität durch richtig einge-

setzte Vererbung, sowie eine möglichst geringe Anzahl an globalen Variablen zu einer Implementierung ohne Sicherheitswarnung führen kann.
Bei den Objekten mit 10 oder mehr globalen Attributen wird ein zweites Mal die Tiefe des Vererbungsbaumes betrachtet. Hier wird am Wert 6,5 unterschieden. Objekte mit einem Vererbungsbaum unter diesem Wert werden der Kategorie **Warnung vorhanden** zugeteilt. Diesen Weg durchliefen 16 Prozent der Daten im Testsatz.
Die restlichen Objekte werden einem letzten Vergleich unterzogen, in dem die Verschachtelungstiefen betrachtet werden. Hier weist ein Wert über 5,5 darauf hin, dass eine bekannte Warnung vorhanden ist. Einem darunter liegenden Wert folgt eine Zuteilung der Kategorie **Ohne Warnung**. Hieraus kann man ableiten, dass eine geringe Verschachtelungstiefe angestrebt werden sollte. Aus einer tieferen Verschachtelung im Quellcode folgt häufig in hohe Anzahl an möglichen Ausführungspfaden im Programm, wodurch wiederum die Komplexität erhöht wird. Mit ansteigender Komplexität steigt das Risiko fehlerhafter Implementierungen. [51]

3.7.3 Untersuchung der Hypothesen

Nachdem der Entscheidungsbaum erfolgreich aufgebaut und untersucht wurde, werden hier noch einmal die zuvor definierten Hypothesen betrachtet. Diese werden unter Verwendung der entsprechenden Gegenhypothese angenommen oder abgelehnt.

Hypothese 1 Die Hypothese wird angenommen, wenn ihre Gegenhypothese widerlegt werden kann. Hierfür muss untersucht werden, ob Smart Contracts mit einer hohen Komplexität oder niedrigen Struktur nach dem Entscheidungsbaum einer Kategorie mit niedrigem Sicherheitsrisiko zugeordnet werden. Es gibt zwei Wege im Baum, nach denen ein Objekt der Kategorie *Ohne Warnung* zugeordnet wird:
Im ersten Fall hat das Objekt eine Vererbungstiefe von mindestens zwei und unter zehn globalen Attributen und landet im Blatt 7. Da sich die Vererbungstiefe auf die Struktur und die Anzahl der Attribute auf die Komplexität beziehen, handelt es sich hierbei um Objekte mit einer begrenzten Komplexität und einer gewissen Struktur. Folglich kann die Gegenhypothese in diesem Fall widerlegt werden.
Im zweiten Fall werden Objekte im Blatt 25 der Kategorie *Ohne Warnung* zugeordnet. Diese Objekte zeichnet eine Anzahl an Argumenten über zehn, eine Vererbungstiefe über 7 und ein Verschachtelungstiefe kleiner 5 aus.
Auch wenn die Anzahl an Argumenten und die Vererbungstiefe keine Begrenzung haben, zeichnet sie die begrenzte Verschachtelungstiefe aus. Dadurch wird die Komplexität eingegrenzt und eine Struktur bewahrt.
Da Objekte ohne diese Eigenschaft in der schwerwiegendsten Kategorie *Bekannte*

Warnung vorhanden landen, kann auch hier die Gegenhypothese widerlegt werden.

Hypothese 2 Die Hypothese wird angenommen, wenn ihre Gegenhypothese widerlegt werden kann. Hierfür muss untersucht werden, ob Smart Contracts mit einer niedrigen Komplexität oder einer hohen Struktur nach dem Entscheidungsbaum einer Kategorie mit hohem Sicherheitsrisiko zugeordnet werden. Es gibt zwei Fälle, nach denen ein Objekt der Kategorie mit hohem Risiko zugeordnet wird:
Zu Beginn werden Objekte mit einer Vererbungstiefe von weniger oder gleich eins sofort der Kategorie *Bekannte Warnung* (Blatt 2) zugeteilt. In diesen Objekten wird die Vererbung - ein Methode zur Strukturierung und Trennung von Aufgaben - wahrscheinlich nicht ausreichend benutzt. Es ist also davon auszugehen, dass eine mangelhafte Struktur vorhanden ist. Demnach handelt es sich hierbei nicht um Objekte mit einer hohen Struktur, die Gegenhypothese ist widerlegt.
Im zweiten Fall landen Objekte mit einer Anzahl von Attributen über oder gleich 9, einer Vererbungstiefe über 6 und einer Verschachtelungstiefe über 5 in der Kategorie *Bekannte Warnung vorhanden* (Blatt 24). Es handelt sich hierbei also um Objekte mit keiner Begrenzung an Attributen und einer beliebig hohen Vererbungs- und Verschachtelungstiefe. Im Entscheidungsbaum landen hier die Objekte mit der höchsten Komplexität, da es keine obere Begrenzung der Attribute gibt. Die Nullhypothese ist widerlegt.

Durch die Widerlegung beider Gegenhypothesen werden die aufgestellten Hypothesen bestätigt.

3.8 Threats to Validity

3.8.1 Interne Validität

Die Klassifizierung der Smart Contracts basiert auf der Vertrauenswürdigkeit der Ergebnisse der drei genutzten Sicherheitsscanner. Dabei ist zu berücksichtigen, dass die Warnung eines Scanners immer eine false-positive Warnung sein kann.
Während des Trainings und der Anpassung der Modelle wurde die optimale Konfiguration von dem Messwerten *precision* und *recall* ausgemacht. Die Wahl eines anderen Vergleichswertes führt unter Umständen zu anderen Ergebnissen.

3.8.2 Externe Validität

Das in dieser Arbeit entwickelte Modell basiert auf den Eigenschaften von Smart Contracts in der High-Level-Programmiersprache Solidity und innerhalb der Ethereum Blockchain.
Damit unterliegt es dem technischen Rahmen der Technologie, wodurch die Ergebnisse nicht auf Smart Contracts in anderen Programmiersprachen oder anderen Blockchains übertragbar sind.
Abgesehen davon wird die Ethereum Blockchain fortlaufend weiterentwickelt, wodurch auftretende Sicherheitslücken obsolet werden können. In diesem Fall muss das Modell an den aktuellen Entwicklungsstand angepasst werden, bevor ein valides Ergebnis erzielt werden kann.
Für die Klassifizierung der Ergebnisse wurde ein Entscheidungsbaum genutzt, andere Verfahren führen eventuell zu anderen Ergebnissen.

4 Fazit und Ausblick

4.1 Fazit

In der Arbeit wurden 469 aktuelle Ethereum Smart Contracts mithilfe der drei Analysewerkzeuge Securify [38], Slither [36] und Oyente [52] auf Sicherheitsrisiken untersucht. Die Ausgabe der verschiedenen Quellcodeanalysen wurde genutzt, um die Kategorisierung *Ohne Warnung*, *Warnung vorhanden* und *Bekannte Warnung vorhanden* aufzustellen. Jeder betrachtete Smart Contract wurde untersucht und, abhängig von den Ergebnissen der Analysewerkzeuge, einer Kategorie zugeordnet.
Im nächsten Schritt wurde der Solidity Parser SolMet [42] genutzt, um 16 verschiedene Quellcodemetriken zu erheben, die zum Teil aus den objektorientierten Metriken heraus entwickelt wurden. Die 16 verfügbaren Metriken wurden für jeden Smart Contract erhoben und entsprechend festgehalten.
Im finalen Schritt wurden die gesammelten Quellcodemetriken, sowie die vorher durchgeführte Sicherheitskategorisierung, genutzt, um mithilfe des Paketes rpart [50] eine Regressionsanalyse durchzuführen. Als Ergebnis wurde ein Entscheidungsbaum für die Kategorisierung von Ethereum Smart Contracts in die zuvor genannten Kategorien entwickelt.
Dafür wurde die Menge der betrachteten Smart Contracts in einen Trainingssatz, mithilfe dessen der Entscheidungsbaum aufgebaut wurde, einen Validierungssatz, mithilfe dessen der erstellte Baum optimiert wurde, und einen Testsatz, mithilfe dessen der Baum bewertet wurde, aufgeteilt.
Zur Bewertung des Entscheidungsbaumes wurden die Kennzahlen *precision* und *recall* verwendet. Eine Zuordnung in die Kategorie *Warnung vorhanden* ist mit den Werten $precision = \frac{7}{13} = 0,53$ und $recall = \frac{7}{19} = 0,36$ am ungenauesten, danach folgen *Bekannte Warnung vorhanden* mit $precision = \frac{29}{42} = 0,69$ und $recall = \frac{29}{44} = 0,65$ und *Ohne Warnung* mit $precision = \frac{24}{38} = 0,63$ und $recall = \frac{24}{30} = 0,8$.

Indem man die Wege von der Wurzel zu den Blättern des Entscheidungsbaumes verfolgt, können Orientierungen zur Entwicklung von Ethereum Smart Contracts gewonnen werden, indem man die Wege von der Wurzel zu den Blättern läuft.
So kann man beispielsweise sehen, dass Smart Contracts mit einer dürftigen oder nicht vorhandenen Vererbungsstruktur mit hoher Wahrscheinlichkeit der Kategorie *Bekannte Warnung vorhanden* angehören. Smart Contracts mit einer Vererbungstiefe über oder gleich zwei und einer Anzahl von globalen Attributen kleiner neun gehören wiederum mit einer hohen Wahrscheinlichkeit der Kategorie *Ohne Warnung* an.

Das Modell wurde aus der Motivation heraus entwickelt, fehlerbehaftete Implementierungen und Wertverluste in Smart Contracts, wie sie in der jüngeren Vergangenheit aufgetreten sind, vorzubeugen. Es kann auf die beschriebene Weise bei der Entwicklung und Bewertung von Smart Contract Quellcode genutzt werden, ohne dass eine tiefe Quellcodeanalyse durchgeführt werden muss.

4.2 Ausblick

Da Blockchain-Systeme und Smart Contracts in Zukunft mit hoher Wahrscheinlichkeit weiterentwickelt werden, ergeben sich viele Möglichkeiten, die hier durchgeführte Forschung zu übernehmen oder zu nutzen.
Das hier entwickelte Modell und die Ergebnisse lassen sich nur auf den aktuellen Entwicklungsstand der Ethereum Blockchain und nur auf Smart Contracts in der Solidity Programmiersprache übertragen.
Sobald die Ethereum Blockchain weiterentwickelt wird oder Solidity eine Änderung erhält, wäre es sehr interessant, zu untersuchen, wie sich diese neuen Gegebenheiten auf das Modell auswirken. Abgesehen vom technologischen Entwicklungsstand konnten auch nur aktuell bekannte und verbreitete Sicherheitslücken berücksichtigt werden. Wird eine neue Sicherheitslücke aufgedeckt, kann man sie in die Forschung aufnehmen.
Neben der Ethereum Blockchain existieren bereits verschiedene, weniger populäre Blockchain Systeme mit Unterstützung von eigenen Smart Contracts. Eine Wiederholung der Forschung innerhalb einer anderen Umgebung ist in Zukunft sehr gut durchführbar.

Literatur

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[2] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, pages 8–15, Piscataway, NJ, USA, 2019. IEEE Press.

[3] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 67–82, New York, NY, USA, 2018. ACM.

[4] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[5] Gareth William Peters and Efstathios Panayi. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money. *CoRR*, abs/1511.05740, 2015.

[6] Ittay Eyal. Blockchain technology: Transforming libertarian cryptocurrency dreams to finance and banking realities. *Computer*, 50:38–49, 01 2017.

[7] Philip Treleaven, Richard Brown, and Danny Yang. Blockchain technology in finance. *Computer*, 50:14–17, 01 2017.

[8] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. pages 25–30, 08 2016.

[9] Matthias Mettler. Blockchain technology in healthcare: The revolution starts here. pages 1–3, 09 2016.

[10] Fabian Knirsch, Andreas Unterweger, Günther Eibl, and Dominik Engel. *Privacy-Preserving Smart Grid Tariff Decisions with Blockchain-Based Smart Contracts*, pages 85–116. 01 2018.

[11] Esther Mengelkamp, Benedikt Notheisen, Carolin Beer, David Dauer, and Christof Weinhardt. A blockchain-based smart grid: towards sustainable local energy markets. *Computer Science - Research and Development*, pages 1–8, 08 2017.

[12] Saveen Abeyratne and Radmehr Monfared. Blockchain ready manufacturing supply chain using distributed ledger. *International Journal of Research in Engineering and Technology*, 05, 09 2016.

[13] Si Chen, Rui Shi, Zhuangyu Ren, Jiaqi Yan, Amy Shi, and Jinyu Zhang. A blockchain-based supply chain quality management framework. 11 2017.

[14] Svein Ølnes, Jolien Ubacht, and Marijn Janssen. Blockchain in government: Benefits and implications of distributed ledger technology for information sharing. *Government Information Quarterly*, 34, 10 2017.

[15] Mark Staples, Shiping Chen, Sara Falamaki, Alexander Ponomarev, Paul Rimba, An Binh Tran, Ingo Weber, Xiwei Xu, and John Zhu. Risks and opportunities for systems using blockchain and smart contracts, 06 2017.

[16] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: 2016-08-22.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *https://bitcoin.org/bitcoin.pdf*.

[18] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *J. Cryptol.*, 3(2):99–111, January 1991.

[19] J. Zhao, Shaokun Fan, and Jiaqi Yan. Overview of business innovations and research opportunities in blockchain and introduction to the special issue. *Financial Innovation*, 2, 12 2016.

[20] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 1st edition, 2015.

[21] Dmitry Efanov and Pavel Roschin. The all-pervasiveness of the blockchain technology. 2018.

[22] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *CoRR*, abs/1802.06993, 2018.

[23] Jianjun Sun, Jiaqi Yan, and Kem Zhang. Blockchain-based sharing services: What blockchain technology can contribute to smart cities. *Financial Innovation*, 2, 12 2016.

[24] Roman Alexander. *IOTA - Introduction to the Tangle Technology: Everything You Need to Know About the Revolutionary Blockchain Alternative*. Independently published, 2018.

[25] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[26] J. Stark. Making sense of blockchain smart contracts. *http://www.coindesk.com/making-sense-smart-contracts/*, Gelesen am 24.11.2019.

[27] Laurin Arnold, Martin Brennecke, Patrick Camus, Gilbert Fridgen, Tobias Guggenberger, Sven Radszuwill, Alexander Rieger, André Schweizer, and Nils Urbach. *Blockchain and Initial Coin Offerings: Blockchains Implications for Crowdfunding*. 08 2018.

[28] Josselin Feist. Slither wiki. *https://github.com/crytic/slither/wiki*, 26.11.2019.

[29] Price per Ether 07.11.2017. https://coinmarketcap.com/de/currencies/ethereum/historical-data/.

[30] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.

[31] Lorenz Breidenbach, Philip Daian, Ari Juels, and EG Sirer. An in-depth look at the parity multisig bug. *Appeared at Hacking, Distributed http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug*, 2016.

[32] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *CoRR*, abs/1703.03779, 2017.

[33] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. Vultron: Catching vulnerable smart contracts once and for all. pages 1–4, 05 2019.

[34] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10, Sep. 2013.

[35] Cécile Pierrot and Benjamin Wesolowski. Malleability of the blockchain's entropy. *Cryptography Commun.*, 10(1):211–233, January 2018.

[36] Slither repo github. *https://github.com/crytic/slither*, Abgerufen 05.12.2019.

[37] Trail of bits website. *https://www.trailofbits.com/services/blockchain-security/*, Abgerufen 05.12.2019.

[38] Securify github repository. *https://github.com/eth-sri/securify*, Abgerufen 05.12.2019.

[39] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[40] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[41] Microsoft-Research. Z3 theorem prover. *https://rise4fun.com/z3/*, Abgerufen: 09.12.2019.

[42] Hegeds P. Solmet github quellcode verzeichnis. *https://github.com/chicxurug/SolMet-Solidity-parser*, Abgerufen 07.12.2019.

[43] Péter Hegeds. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. *Technologies*, 7(1), 2019.

[44] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[45] Etherscan ERC-20 Liste verfizierte Token. https://etherscan.io/tokens. Abgerufen 07.12.2019.

[46] Wai Choy and Teng Pengtao. When smart contracts are outsmarted: The parity wallet freeze and software liability in the internet of value. *Lexology, Dec. 2017.*, URL: www.lexology.com/library/detail.aspx?g=33a0af51-80f3-4643-8536-db9a0d9ba2c7, Abgerufen: 10.12.2019.

[47] Pete Humiston. Smart contract attacks [part 2] - ponzi games gone wrong. *Hackernoon*, https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8, Aufgerufen 10.12.2019.

[48] rpart funktion in r. *https://www.rdocumentation.org/packages/rpart/versions/4.1-15/topics/rpart*, Abgerufen: 14.12.2019.

[49] Nikita Patel and Saurabh Upadhyay. Study of various decision tree pruning methods with their empirical comparison in weka. *Int. J. Comput. Appl.*, 60:20–25, 12 2012.

[50] Dokumentation rpart package. *https://www.rdocumentation.org/packages/rpart/versions/4.1-15/topics/predict.rpart*, Abgerufen 10.01.2020.

[51] Say-Wei Foo and A. Muruganantham. Software risk assessment model. In *Proceedings of the 2000 IEEE International Conference on Management of Innovation and Technology. ICMIT 2000. 'Management in the 21st Century' (Cat. No.00EX457)*, volume 2, pages 536–544 vol.2, Nov 2000.

[52] Oyente github repository. *https://github.com/melonproject/oyente*, Abgerufen: 11.01.2020.

Erklärung

Ich versichere, die Bachelorarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäSS aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 22. Januar 2020

(Simon Worm)