TU WIEN Informatics

Erkennung von Mustern in EVM-Bytecode durch XQuery-Abfragen auf Ausführungspfaden im Kontrollflussgraphen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Master Software Engineering & Internet Computing

eingereicht von

Christian Sack, BSc
Matrikelnummer 01526277

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo
Mitwirkung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Wien, 30. April 2023

Christian Sack	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Detecting patterns in EVM bytecode by XQuery-ing execution traces in the control flow graph

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master Software Engineering & Internet Computing

by

Christian Sack, BSc
Registration Number 01526277

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo
Assistance: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Vienna, 30th April, 2023

Christian Sack	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Christian Sack, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. April 2023

Christian Sack

Danksagung

Diese Arbeit ist der Abschluss einer ganz großartigen Zeit. Zuerst möchte ich mich bei meiner Familie bedanken, insbesondere meiner Schwester Monika und meinen Eltern Brigitte und Heinz. Ihr habt mir das alles ermöglicht.

Ich möchte mich auch bei meinen Freunden und Wegbegleitern bedanken, die all die Jahre hinweg für mich da waren, speziell bei euch Anja, Michelle und Verena. Ich möchte mich bei meiner Freundin Kathi bedanken, ohne der ich wohl noch eine ganze Weile länger gearbeitet hätte. Danke, dass du in der verrückten Zeit immer für mich da warst und mir auch jetzt bei dieser Danksagung hilfst.

Brigitte, dir möchte ich auch noch einmal ganz im Speziellen danken! Ich weiß nicht, ob ich dein neues Bad so schön geformt habe, wie du meine Arbeit. Danke für deinen Einsatz bis spät in die Nacht und in die frühen Morgenstunden, du warst eine tolle Hilfe! Ich wünsche dir eine gute Besserung!

Und zu guter Letzt, einen riesigen Dank an meine beiden Betreuer Monika di Angelo und Gernot Salzer. Ihr zwei habt nicht nur diese Arbeit geprägt, ihr wart Zeit meines Studiums in allen erdenklichen Belangen an meiner Seite. Ich schätze euch als gute Freunde, tolle Arbeitskollegen und Betreuer dieser Arbeit. Danke für all das Vertrauen, das ihr über all die Jahre hinweg in mich hattet.

Kurzfassung

Blockchain-Programme, sogenannte Smart Contracts (SCs), sind eine Weiterentwicklung zu der im Jahr 2008 veröffentlichten Blockchain Bitcoin. Ethereum, eine der großen Blockchains, erlaubt es, diese quasi-Turing-vollständigen SCs autonom und durch die Blockchain gesichert laufen zu lassen. Die Ethereum-Virtual-Machine (EVM), ein zentraler Bestandteil von Ethereum, ermöglicht es diesen Programmen, viele neue Anwendungsgebiete zu erschließen. Prominente Beispiele hierfür sind Token, dAPPs, dezentrale Finanzsysteme (DeFi), Logistik und Protokollierung (Logging). Etwa 250.000 unterschiedliche derartige Programme wurden bereits auf Ethereum veröffentlicht.

In dieser Arbeit untersuche ich einen neuartigen Ansatz zur Klassifizierung von Smart-Contract-Mustern (SC-Mustern) basierend auf ihren EVM-Bytecodes (EVM-BCs) unter Verwendung einer allgemeinen Abfragesprache. Anhand verschiedener Smart-Contract Muster (SC-Muster) zeige ich mögliche Anwendungsgebiete meiner vorgeschlagenen Abfragesprache. Für die SC-Mustererkennung entwickle ich einen Prototyp, der gegebene EVM-BCs dekompiliert und in XML-Dokumente (Extensible Markup Language) umwandelt. Die heuristisch arbeitenden Erkennungsstrategien, die auf Extensible-Markup-Language-Query-Language (XQuery) basieren, werden schließlich gegen eine bereits klassifizierte Menge von SCs evaluiert.

Abstract

Blockchain programs, so-called smart contracts (SCs), are a further development of the blockchain Bitcoin, which was published in 2008. Ethereum, one of the big blockchains, allows these quasi-Turing-complete SCs to run autonomously and secured by the blockchain. The Ethereum Virtual Machine (EVM), a central part of Ethereum, enables these programs to open up many new areas of application. Prominent examples of these are tokens, dAPPs, decentralized finance (DeFi), logistics and logging. About 250,000 different such programs have already been published on Ethereum.

In this thesis, I investigate a novel approach to classify smart contract patterns (SC patterns) based on their EVM bytecodes (EVM-BCs) using a general query language. Using different SC patterns, I show possible areas of application of my proposed query language. For SC pattern recognition, I develop a prototype that decompiles given EVM-BC and transforms it into Extensible Markup Language (XML) documents. The heuristically working detection strategies, which are based on Extensible-Markup-Language-Query-Language (XQuery), are finally evaluated against an already classified set of SCs.

Inhaltsverzeichnis

KAPITEL 1

Einführung

1.1 Motivation & Problemstellung

Bitcoin ist ein echtes Peer to Peer (P2P) Netzwerk, das elektronische Zahlungen sicher machen sollte, ganz ohne das Hinzuziehen einer zentralen Finanzinstitution [Nak09]. Damit war der Beginn der Blockchains eingeleitet, tatsächlich war dieses Netzwerk völlig vertrauenslos (trustless) aufgebaut und stellte sich mit der Zeit als relativ sichere verteilte Datenbank für Finanztransaktionen heraus. Das Netzwerk selbst besteht aus sich gegenseitig misstrauenden Minern und Akteuren, die Transaktionen in diesem Netzwerk versenden. Über einen Konsensmechanismus namens Proof-of-Work (PoW) war und ist es scheinbar bis heute möglich die Fälschungssicherheit, also die Grundlage dieser dezentralen Datenbank, sicherzustellen.

Das Konzept wurde adaptiert weiterentwickelt und mündete schließlich in verschiedenen Systemen, die Ähnliches versuchen. In dieser Arbeit möchte ich mich auf ein System namens Ethereum konzentrieren, einer Blockchain, die es ermöglicht quasi[1] Turing-vollständige Programme in solch einem System adressierbar und ausführbar zu machen [But15]. Solche Programme nennt man SCs, und sie folgen den durch die EVM bestimmten Regeln [Woo17]. Mit der Einführung dieser komplexen Programme kristallisierten sich verschiedene Anwendungsgebiete in verschiedenen Bereichen heraus. Beispiele für solche Bereiche sind Versorgungsketten, die digitale Rechteverwaltung und Versicherungen. [MPJ18]

[1] Turing-vollständig im Kontrast zu quasi Turing-vollständig, erlaubt es, Programme unlimitiert lange laufen zu lassen und dabei Ressourcen zu verbrauchen. In Ethereum verhindert die quasi-Turing-Vollständigkeit endlos laufende Programme durch den Einbau eines Konstrukts namens Gas. Dieses Gas wird bei dem Beginn der Ausführung eines Programms (innerhalb einer Transaktion) definiert [Woo17]. Es ist durch Schranken limitiert und beschreibt die Menge an Berechnungen die durchgeführt werden können. Ist die Menge erreicht, terminiert das Programm. Eine Terminierung ist somit in jedem Fall gewährleistet. Denial-of-Service (DOS) Angriffe sind somit über diesen Weg nicht mehr möglich.

SCs werden typischerweise in Solidity geschrieben und in EVM-BC kompiliert. Laut [SXY+22] steht für mehr als 94 % der in Ethereum veröffentlichten SCs kein Source Code unmittelbar zur Verfügung. Dabei stellt sich die Frage, welche Funktionalität diese SCs ohne offenen Source Code implementieren. Eine automatisierte Erkennung von zumindest Teilen der Funktionalität aus dem öffentlich verfügbaren EVM-BC wäre hilfreich. In meiner Arbeit stelle ich eine Lösung vor, mit der man frei definierbare Programmiermuster erkennen kann. Dazu verwende ich die allgemeine Abfragesprache XQuery und versuche ihre Eignung anhand von ausgewählten Mustern zu klären. In zukünftigen Arbeiten sollen so Untersuchungen ermöglicht werden, die anhand vieler XQueries-Abfragen Rückschlüsse auf den Zweck und die Art von SCs geben können.

Zusammenfassend kann diese Arbeit bei der Abklärung helfen, inwiefern XQuery als mögliche neue universale Sprache dienen kann, bestimmte Eigenschaften von SC beschreibbar zu machen. Diese Eigenschaften sollen aus dem kompilierten EVM-BC des jeweiligen SC abgeleitet werden können. So soll schlussendlich die Identifizierung von ähnlichem Verhalten unterschiedlicher SCs automatisiert ermöglicht werden, ohne dabei auf die Veröffentlichung des Codes des SC angewiesen zu sein.

1.2 Methodisches Vorgehen

- **Literaturrecherche**
  Aufbau einer fundierten theoretischen Basis über wiederkehrende SC-Muster im kompilierten EVM-BC sogenannte EVM-Bytecode-Muster (EVM-BC-Muster) und damit einhergehend ein tiefes Verständnis für Architektur und Arbeitsweise der EVM. Dieses Wissen fließt dann direkt in die Implementierung einer symbolisch arbeitenden EVM ein. Sie stellt die Basis für den zu erstellenden PoC dar.
- **Recherche und Dokumentation typischer SC-Muster.**
  - Was sind typische EVM-BC-Muster oder Instruktionsfolgen in kompilierten SCs, die von Interesse sein können?
  - Welche Ziele verfolgen solche wiederkehrenden Sequenzen und wie könnten sie mittels XQuery beschrieben werden?

  Dabei werden bestehende bekannte SC-Muster in EVM-BC kompiliert. Der erhaltene Output wird auf die wesentlichen Merkmale des SC-Muster reduziert. Da SC-Muster verschiedene Ausprägungen aufweisen können, werden EVM-BC Kompilate auch zu den anderen Ausprägungen angefertigt. Die so erhaltenen EVM-BCs werden manuell auf Gemeinsamkeiten durchsucht, um EVM-BC-Muster abzuleiten. Diese EVM-BC-Muster werden als Input für die zu entwickelnden Transformationsregeln des EVM-BC nach Extensible Markup Language (XML) herangezogen. Dadurch soll das erhaltene XML-Dokument als mächtige Basis für die Abfragesprache XQuery schlussendlich anwendbar werden.
- **Symbolische Ausführung und bestehende Tools.**
  Da es als Zwischenschritt notwendig ist, einen Control-Flow-Graph (CFG) des zu analysierenden SC zur Verfügung zu haben, wird ein geeignetes frei verfügbares

Tools eingesetzt. Außerdem sollen bewährte Praktiken für die Entwicklung des entsprechenden PoC abgeleitet werden.

- **Entwicklung des PoC.**
  Als PoC wird ein Prototyp konzipiert und implementiert, der alle möglichen Programmausführungen iteriert und in die entsprechenden XML-Dokumente transformiert, damit diese dann mittels XQuery-Abfragen evaluiert werden können.
- **Evaluation**
  - Heranziehen der zuvor entdeckten SC-Muster als Grundwahrheit.
  - Definition der entsprechenden XQuery-Abfragen zur Erkennung ausgewählter EVM-BC-Muster.
  - Ausführung der Abfragen gegenüber der Grundwahrheit zur Ermittlung der Qualität von XQuery als universale Abfragesprache zur Erkennung bestimmter EVM-BC-Muster bzw. SC-Muster.

1.3 Stand der Technik

Wie bereits erwähnt, haben SCs in Ethereum typische Anwendungsgebiete [MPJ18]. Feinere Strukturen dieser SCs, also beispielsweise Funktionen, verfolgen dann mitunter auch ähnliche Ziele [Opea]. Über die Zeit haben sich typische SC-Muster, aber auch Standards entwickelt [Opeb].

In einer empirischen Analyse [BP17] wurden 811 SCs nach SC-Muster klassifiziert. Dabei wurde sie festgestellt, dass nur etwa 20 % der 811 SCs nicht in eine der Kategorien Token, Autorisation, Oracle, Randomness, Poll, Time constraint, Termination, Math oder Fork check zugeordnet werden kann. Die drei größten Kategorien sind Autorisation mit 61 % Time constraint mit 33 % und Termination mit 22 %.

In einer weiteren Studie [SHS22] wurden SC-Muster in vier verschiedene Unterkategorien unterteilt: Management (12 verschiedene, darunter am prominentesten der Proxy Pattern), Sicherheit (11, Beispiel: Emergency Stop), Effizienz(15, Beispiel: Incentive execution pattern) und Zugangsberechtigungspattern(9, Beispiel: Ownership).

In [dAS20] wurden SC anhand derer Schnittstellen gruppiert, wobei angenommen wurde, dass gleichlautende Funktionen auch gleiche Funktionalität implementieren.

[HHY+21] versuchen, angreifbare SCs zu finden, indem sie aus dem EVM-BC einen CFG konstruieren. Sie wenden Grapheneinbettung an, um aus den Graphen Vektoren für einen Vergleich zu erhalten.

[NPS+17] versuchen, die SCs auf Ethereum anhand ihrer Funktionalität zu kennzeichnen, indem sie Fuzzy-Hashing (ssdeep) verwenden, um Ähnlichkeiten in EVM-BC zu finden.

Gigahorse wurde in 2019 als ein Dekompilierungs-Framework veröffentlicht zur Dekompilierung von EVM-BC in eine explizite und höhere Repräsentation des Codes [GBSS19]. Es benutzt einen on-the-fly-CFG Kontruktions-Methode. Ein ähnlicher Ansatz wird auch in dieser Arbeit benutzt.

KAPITEL 2

Technischer Hintergrund

Dieses Kapitel soll den Lesenden zuerst einen grundlegenden Einblick in die Blockchain Technology geben und darauf aufbauend, detailliert in die Funktionsweise der EVM eintauchen.

Die EVM und deren Funktionsweise ist zentraler Teil dieser Arbeit, da bei der statischen Code-Analyse neben der Detektion der Blöcke die Dekompilation des EVM-BC notwendig ist. Bereiche, die keinen oder wenig Bezug zu dieser Arbeit haben, werden hervorgehoben und durch Verweise auf Arbeiten, die sich damit beschäftigen, kenntlich gemacht.

Als Grundlage für diese Arbeit werden zwei bereits vorhandene Tools verwendet, die sich im Laufe der Recherche als wertvoll für diese Arbeit herausgestellt haben:

- EtherSolve: Aufgrund der intrinsischen Komplexität des EVM-BC ist das Konstruieren eines CFG mit dem Überwinden von vielen Hürden verbunden [CCCP21]. EtherSolve wird in dieser Arbeit zum Konstruieren eines CFG herangezogen. Dieser CFG hilft bei der Enumeration der möglichen Ausführungsstränge.
- Saxon CS: C# oder genauer das .net Framework bietet leider kein aktuelles XQuery Framework an. Daher wurde für die Prozessierung der XQueries-Abfragen Saxon CS als Prozessor gewählt [Sax]. Das Framework wurde dankenswerterweise von der Fa. Saxonica für diese Arbeit bereitgestellt.

Hintergründe zu den jeweiligen Tools und deren Eingliederung in diese Arbeit werden im Folgenden besprochen.

2.1 Blockchain Grundlagen

Nakamoto hat mit seinem Paper [Woo17] die Idee von Blockchains der breiten Masse in Form von Bitcoin zugänglich gemacht. An sich ist die Idee eines manipulationssicheren und verifizierbaren Timestamp-Services nicht neu. Erste technologische Spuren führen sogar

bis in die 90er Jahre zurück [HS91]. In Ethereum wurde die Idee von Bitcoin, nämlich unverbrauchte Transaktionen zu speichern, hin zu einem Account-System abgeändert. Dabei blieb der Konsensmechanismus zunächst relativ ähnlich [Nak09] [But15].

Durch die Verkettung der historischen Daten lässt sich der Systemzustand anhand im Programmcode festgeschriebener Regeln jederzeit validieren. Allgemein haben Blockchains durch ihren Konsensmechanismus einen Weg gefunden, Sybil-Attacken zu verhindern und gleichzeitig eine Lösung für das double spending Problem in einem P2P System zu bieten [Nak09].

Im Folgenden möchte ich auf die zuvor verwendeten wesentlichen Begriffe bzw. Problemstellungen rund um Blockchains eingehen, bevor ich mich weiter spezifisch mit Blockchains auseinandersetze.

- **peer-to-peer**:
  In P2P Systemen können Rechner innerhalb eines Netzwerks sich direkt, also ohne Umwege über einen Server (zentralisiert), untereinander austauschen. In solchen Systemen tritt typischerweise das sogenannte Byzantine-Generals-Problem auf, das etwas weiter unten beschrieben ist.
- **double spending problem**:
  Beim double spending problem handelt es sich um ein in digitalisierten und dezentralisierten Systemen auftretendes Problem, in denen Aktionen oder wie hier in den behandelten Blockchains Transaktionen ausgeführt werden. Nach dem Ausführen einer digitalen Transaktion muss sichergestellt werden, dass diese Transaktion kein zweites Mal ausgeführt wird bzw. so verschobene Token kein zweites Mal ausgegeben werden können.
  Im dezentralen digitalen Umfeld muss sichergestellt werden, dass ein bestimmter Token nicht im Austausch eines anderen Gutes von zwei verschiedenen Parteien gleichzeitig empfangen werden kann bzw. dass ein digitaler Token mehrfach kopiert und für eine Transaktion verwendet werden kann. Andernfalls könnten sich alle beteiligten Akteure Kopien eines Tokens behalten, und diese gegebenenfalls weitere Male verwenden [Cho17].
- **byzantine generals problem**:
  Bei diesem Problem handelt es sich um ein Problem in der Entscheidungsfindung/-Synchronisierung, wenn durch mehrere Komponenten in einem System verschiedene Antworten gegeben werden können und daraufhin eine konsolidierte Sicht auf den globalen Systemzustand erarbeitet werden muss. Der Einfluss des Problems nimmt noch einmal zu, wenn mit boshaften Komponenten, also Komponenten, die sich nicht im Sinne des Systems verhalten, gerechnet werden muss [LSP82].

Zur Lösung dieser Probleme kommen viele Teilaspekte zusammen: Zentral in all diesen Überlegungen steht bei Blockchains der sogenannte Konsensmechanismus. In Bitcoin ist die Basis dieses Mechanismus der PoW Algorithmus. In Ethereum seit 2022 nun Proof-of-Stake (PoS). Davor verwendete Ethereum ebenfalls einen PoW als Basis.

Der Konsensmechanismus stellt selbst wiederum die Basis des sogenannten verteilten

Timestampservers dar, der jedem einzelnen Block in dieser Kette einen eindeutigen Zeitpunkt und Position zuordnet. Schlussendlich trägt er Sorge dafür, dass jeder Rechner des P2P Netzwerks am Ende eine gemeinsame Sicht auf den globalen Systemzustand hat. Dabei kann das Ende der Kette zwischenzeitlich divergieren, nachdem aber ausreichend "proofs" gefertigt wurden, und die längste Kette bestimmt wurde, haben alle Rechner dieselbe Sicht auf die Kette und all ihrer bisher publizierten Blöcke.

Die Blockchain in Ethereum ist eine öffentliche, verifizierbare und verteilte Datenbank, die die Liste aller jemals getätigten Transaktionen speichert. Der globale Systemzustand ist eine ständig aktualisierte Sicht auf den letztgültigen Kontostand aller Akteure. Da die Liste aller Transaktionen stets auflistbar ist, kann der Kontostand zu jedem Zeitpunkt nachvollzogen und somit auch verifiziert werden.

Ein markanter Unterschied von Bitcoin zu Ethereum ist die Möglichkeit, dass sich hinter Adressen statt externer Akteure Programme befinden können. Sie besitzen neben einem Kontostand auch einen eigenen Programmzustand. Transaktionen, die solche sogenannten SCs als Ziel haben, können dadurch den neuen Systemzustand in einer quasi Turing-vollständigen Art und Weise anpassen. Denkbar ist hierzu zum Beispiel in einem sehr einfachen Fall das Weiterüberweisen des eingehenden Betrags an mehrere Personen.

2.1.1 Konsensmechanismus

Typischerweise wird der Begriff synonym für PoW, PoS oder Proof-of-Authority (POA) verwendet. Diese drei Mechanismen schützen vor Sybil-Attacken, während der Konsensmechanismus umfassender alle Ideen, Bereiche und Systeme abdeckt, die bei den einzelnen Peers eines P2P Netzwerks für eine konsolidierte gemeinsame Sicht auf den Gesamtsystemzustand sorgen.

Seit 2022 hat sich die Basis des Konsensmechanismus des Ethereum Netzwerks von dem allgemein bekannten PoW- zu einem PoS-Mechanismus geändert. Seither beruht die cryptoökonomische Sicherheit auf Ether, also der Währung in Ethereum, welches durch "Staker" unter Verwahrung gestellt wurde. Diese Staker werden dann auf Basis eines Regelwerks bestraft bzw. belohnt. Sie erfüllen unter anderem die Aufgabe der Validierung im Netzwerk. In Ethereum ist ein Konsens im Netzwerk erreicht, wenn mindestens zwei Drittel der beteiligten Parteien sich auf einen gemeinsamen globalen Systemzustand geeinigt haben.

Im Folgenden wird auf die verschiedenen Basisideen des Konsensmechanismus eingegangen:

- PoS: Aufgrund des hohen Energieverbrauchs in Ethereum [McD21] sowie auch in Bitcoin [de 18] ist man auf der Suche nach Alternativen zum bekannten PoW. Obwohl Bitcoin die ältere Währung ist, hat inzwischen lediglich Ethereum den Weg hin zu PoS eingeleitet. Im PoS-Verfahren unter Ethereum muss man einen bestimmten Betrag an Ether entrichten "staken", um als Validator an dem Netzwerk teilzunehmen. Sobald nun ein neuer Block durch den sogenannten Block Proposer

bekannt wird, ist die Aufgabe des Validators, diesen Block zu validieren und zu bestätigen. Dieser Block Proposer wird pseudo-randomisiert ausgewählt (RANDAO), er baut den neuen Block auf und verschickt ihn im Netzwerk. Die beteiligten Validatoren einigen sich nun auf einen neuen Block und dessen Gültigkeit. An der Stelle sei erwähnt, dass es sich hierbei um eine Zusammenfassung des PoS-Mechanismus handelt. Viele weitere Details sind notwendig, um dieses System sicher betreiben zu können.

- PoW: Sogenannte "Miner" stehen in Konkurrenz zueinander, um den nächsten Block in einer Kette zu minen. Bei diesem Prozess gibt es typischerweise einen Gewinner, der als erstes erfolgreich solch einen Block finalisieren kann. Wird dieser Block Teil der Kette, so gewinnt der Miner Ether in Form eines "Blockrewards" und Transaktionsgebühren. Der Prozess des Minens beschreibt im Fall von Ethereum das Errechnen eines besonderen Hashes. Allgemeiner kann es sich aber auch um ein anderes mathematisches Verfahren handeln, das leicht zu validieren, aber schwer zu berechnen ist.
  Ohne erneutes Aufwenden von zuvor geleistetem Rechenaufwand und damit einhergehend hohen Kosten kann also ein Block nicht mehr geändert werden. Die Anordnung als Kette kommt durch das Inkludieren des Ergebnisses des Vorgängerblocks in den Input zum mathematischen Verfahren des Nachfolgenden zustande [Nak09] [Woo17]. Dadurch wird außerdem sichergestellt, dass ein Umordnen gültiger Blöcke ausgeschlossen bzw. nur unter Leistung von sehr hohem Rechenaufwand möglich ist.
  Bei dem mathematischen Verfahren kann es konkret wie in Ethereum darum gehen, dass der Hash eines Blockes eine bestimmte Anzahl an führenden Nullen aufweisen muss. Solche Hashes sind im Allgemeinen nicht zu errechnen und nach heutigem Wissensstand nur durch Ausprobieren (sogenanntes brute force) ermittelbar.
  Da es immer wieder zu sogenannten Forks kommen kann - das heißt, mehr als ein Block kommt als neues Glied in der Kette infrage -, müssen sich die Peers des Netzwerks für einen der beiden Zweige als allgemeinen Systemzustand entscheiden. Hier gibt es erstmal keine Lösung, und es kommt zu einer Spaltung des Netzwerks. Erst durch das Bekanntwerden neuer Glieder stellt sich mit der Zeit eine längste Kette mit den meisten Blöcken heraus. Diese längste Kette entspricht dabei dem allgemein gültigen Systemzustand, der nun von allen Peers akzeptiert wird.
  Da das Validieren des mathematischen Puzzles sehr einfach möglich ist, ist es auch immer möglich, die immer länger werdende Kette an Blöcken jederzeit zu validieren. Gleichzeitig aber ist nun ein immer höher werdender Rechenaufwand notwendig, um eine ähnlich lange Kette nachzubilden bzw. einzelne Segmente darin zu ändern. Die Kriterien, die zum Erkennen einer gültigen Kette notwendig sind, sind von Kette zu Kette unterschiedlich. Beispielhaft sei das Regelwerk von Bitcoin [Nak09] oder Ethereum [But15] genannt.
  Durch die im System zunehmende und abnehmende Rechenpower wird die Komplexität des mathematischen Verfahrens von der gesamten Rechenkraft der Miner im System abhängig gemacht. Der Parameter, der die Komplexität beschreibt, wird

"difficulty" genannt [Nak09] [But15]. Sie kann dabei innerhalb von Schranken beim Minen eines Blocks für den darauffolgenden angepasst werden. Das erlaubt es dem P2P-System die Blockgenerationszeit, also jene Zeit, die benötigt wird, um einen neuen Block zu minen, in einem für das Netzwerk optimalen Rahmen zu halten. Kurz stellt PoW im Fall von Bitcoin [Nak09] und Ethereum [But15] sicher, dass nur boshafte Akteure, die mehr als 50 % der gesamten Rechenkraft über einen entsprechend langen Zeitraum im Netzwerk halten, die Möglichkeit haben, vergangene Daten zu ändern. Eine solche Monopolisierung der Hashingpower ist in der Regel aber nicht erwünscht, da damit das Vertrauen in die Unveränderbarkeit verloren gehen würde und damit auch der Wert der Cryptocurrency als solche selbst verloren gehen würde. Ein Angriff wäre damit auch ein Angriff auf die eigene Infrastruktur, die womöglich ausschließlich für den Einsatz des PoW gedacht war.

- POA: Hier entscheidet ein Konsortium, das aus einem oder mehreren sogenannten Validierern bestehen kann, über die Gültigkeit eines neuen Blocks. Nach dem Erreichen einer Mehrheit innerhalb dieses Konsortiums wird ein neu generierter Block Teil der Blockchain. Im Allgemeinen benötigt man für dieses Verfahren ein Set an vertrauenswürdigen Validierern, die aktiv am Validierungsprozess beteiligt sind. Durch verschiedene Anreize oder Strafen wird sichergestellt, dass Validierer ihr Stimmrecht auch benutzen. Andernfalls könnte es dazu führen, dass kein weiterer Block mehr validiert wird und somit Transaktionen nicht mehr abgearbeitet werden können. Im Vergleich zu PoW verbraucht dieses System keine physischen Ressourcen in Form von Rechenkraft und ist somit deutlich Energie effizienter. Gleichzeitig aber liegt das Vertrauen im Extremfall an nur einem Validierer, auf jeden Fall aber wieder zentralisiert bei einem Set an vertrauenswürdigen Agents.

2.1.2 Transaktionen

Im Allgemeinen führen Transaktionen zu einer Zustandsänderung in der Blockchain. In Ethereum gibt es verschiedene Arten von Transaktionen darunter Message Calls und Contract Creations.

- Message Call: Man unterscheidet zwei verschiedene Arten von Message Calls:
  - Externer Call: Externe Calls sind von externen Akteuren, also Akteuren, die außerhalb der EVM mit der EVM über kryptografisch signierte Transaktionen interagieren. Diese Transaktionen stellen einen Input (meist Funktionsaufrufe) für, zum Beispiel, SCs dar. Nachdem ein Peer so eine Transaktion empfangen und erfolgreich validiert hat, schickt er diese weiter durch das Netzwerk, wodurch die Transaktion rasch an möglichst alle Peers weitergeleitet wird. Nachdem das Netzwerk erfolgreich einen PoW, POA oder PoS erbracht hat, wird der zusammengestellte Block mit der entsprechenden Transaktion im Netzwerk veröffentlicht. [Woo17].
  - Interner Call: Interne Calls nehmen ihren Ursprung immer in externen Transaktionen allerdings ist der direkte Aufrufende nicht ein externen Akteur, sondern zum Beispiel ein SC innerhalb der EVM. Das bedeutet, dass auch SCs, in

folge einer externen Transaktion, andere SCs aufrufen können. [Woo17]. Von sich aus selbst kann ein SC nicht aktiv werden.

- Contract Creation: Contract Creation Calls führen zu der Erstellung neuer Accounts, und typischerweise der Hinterlegung eines Programm Codes im Account State unter dieser Adresse [Woo17].
  "Typischerweise" deshalb, weil es durch die EVM möglich ist, Contract Creation Calls durchzuführen und am Ende der Durchführung des Calls durch eine *SELF DESTRUCT*-Instruktion den SC an der soeben erstellten Adresse wieder zu entfernen.

In Ethereum enthalten die Transaktionen neben einem definierten Ether-Betrag, den Empfänger, die Menge an Gas, die bereitgestellt wird, einen Nonce zur Verhinderung von Replay-Attacks[1] und gegebenenfalls Nutzdaten. Transaktionen werden nur nach einer erfolgreichen Validierung der kryptografischen Signatur und anderer Merkmale im Netzwerk verteilt. [Woo17]

2.1.3 Stack-basierte EVM

Die EVM basiert auf einer Stack-basierten Architektur, deren Programmcode von Read-only-Speichern gelesen werden (immutability). Die Wortlänge beträgt 256 bit und bietet volatile Speicher sowie nicht volatile Speicher an. Der volatile Speicher wird nach der Ausführung des Message Calls wieder freigegeben. Der nicht volatile Speicherplatz hingegen bleibt erhalten [Woo17].

Für diese Arbeit wurde ein simpler Interpreter für die Instruktionen dieser EVM aufgebaut.

2.1.4 SCs

SCs sind Programme, die auf einer Blockchain veröffentlicht wurden. In dieser Arbeit sind im Speziellen aber mit SCs jene Programme gemeint, die auf der Blockchain Ethereum veröffentlicht wurden. Diese Programme werden in Form von EVM-BC mithilfe einer speziellen Transaktion veröffentlicht (siehe Unterabschnitt 2.1.2). Später können Funktionen dieser SCs durch weitere externe Transaktionen oder interne Nachrichten aufgerufen werden. Werden SCs durch externe Akteure aufgerufen, sind sie in der Lage, zu einer quasi Turing-vollständigen Zustandsänderung zu führen.

Mit solc kompilierte SCs setzen sich typischerweise folgendermaßen zusammen:

- Konstruktor: Der Code, der ausschließlich beim Deployment des SC ausgeführt wird. Er führt beispielsweise initiale Speicherbelegungen durch, kann aber in sich auch andere SCs aufrufen. Zuletzt lädt er den zu veröffentlichenden Laufzeitcode in den volatilen Speicher und terminiert mit einer *RETURN*-Instruktion. Danach

[1] Das sind Angriffe die darauf abzielen eine bereits getätigte signierte Transaktion erneut abzusenden. Dadurch könnte beispielsweise der Transfer von Ether mehrfach ausgeführt werden, ohne das erneut eine Signierung notwendig wäre.

wird der Konstruktor verworfen, und lediglich der Runtimecode wird im Account State abgelegt.
- Laufzeitcode: Der Laufzeitcode liegt im Account State und wird bei jedem eingehenden Message Call aufgerufen. Er enthält typischerweise den Dispatchercode am Beginn.
- Dispatchercode: Wird gebildet aus den ersten Operation-Codes (OPCs) innerhalb eines SC. Er interpretiert den eingehenden Message Call und entscheidet anhand dessen die aufzurufende Funktion. Er kann in verschiedenen Formen ausgeprägt sein, als ein Baum oder aber auch als eine Art Switch Statement.
- Application-Binary-Interface (ABI): Das ABI enthält eine Beschreibung aller möglichen Einstiegspunkte in den SC und dessen Argumente. Es kann von verschiedenen Libraries benutzt werden, um gültige Message Calls aus einem Funktionsnamen und dessen Argumenten zu erzeugen. Die ABI selbst ist nicht Teil des Laufzeitcodes oder des Konstruktors. Es wird separat zum EVM-BC des SC ausgeliefert.

2.1.5 Dispatchercode

Wie bereits weiter oben angedeutet, handelt es sich um einen Code, der automatisch vom Solidity Compiler erzeugt wird. Er umfasst die ersten Instruktionen, die im Zuge des Aufrufs eines SC ausgeführt werden. Er entscheidet anhand der ersten vier bytes welche Solidity-Funktion aufgerufen werden soll und führt diese aus.

Aufgrund von Performance-Überlegungen kommt er in zwei unterschiedlichen Repräsentationen vor, entweder als Entscheidungsbaum oder in Form aufeinander folgender *JUMPI*-Instruktionen. Bei besonders vielen Solidity-Funktionen kommt der Entscheidungsbaum zum Einsatz.

2.1.6 Speicher-Variablen

Speicher-Variablen sind Werte, die in einem nicht volatilen Bereich der EVM abgespeichert werden. Dabei stellt ein einzelner Speicherslot 32 byte dar, der durch einen 32 byte langen Schlüssel referenziert werden kann. In einem solchen Slot können mehrere einzelne Datenpunkte (boolean, uint8, ...) kompaktiert hinterlegt sein.

Das Dekodieren eines solchen kompakten abgespeicherten Wertes in seine einzelnen Datenpunkte wird durch den Einsatz bestimmter Instruktionen (*DIV*-Instruktion, *EXP*-Instruktion, *AND*-Instruktion, ...) vom Programm selbst vorgenommen. Gleiches gilt für das Encoding.

2.1.7 Mapping

Mappings sind Solidity-Datenstrukturen, die ein Nachschlagen eines Wertes im nicht volatilen Speicherbereich aufgrund eines dynamischen Schlüssels ermöglichen. Der Schlüssel wird dabei immer gemeinsam mit der Position des Mappings im allgemeinen Speicherlayout (also die Position der Storage-Variable im SC) durch die *SHA3*-Instruktion gehasht.

Dadurch ist eine Kollision mit gleichen Schlüsseln in unterschiedlichen Mappings gewährleistet.

2.2 Grundlagen der Dekompilation

2.2.1 Basisblöcke

Ein Basisblock ist eine Sequenz aus Operationen ohne Sprüngen, ausgenommen an der letzten Stelle im Block. Sie können mit einer *JUMPDEST*-Instruktion beginnen, bzw als Folgeblock einer *JUMPI*-Instruktion auftreten. Zu jedem Programm gibt es genau einen Startblock, das ist der erste Block in einem Programm.

2.2.2 Control Flow Graphs (CFGs)

Der sogenannte CFG ist ein gerichteter Graph, der den Kontrollfluss eines Programms beschreibt. Knoten beschreiben dabei Basisblöcke und Kanten, potenzielle Sprünge von einem Basisblock zu einem anderen. Anders ausgedrückt, sie segmentieren EVM-BC in einzelne funktionale Blöcke die sich gegenseitig referenzieren.

In [CCCP21] werden vier grundlegende Probleme bei der Dekompilation von EVM-BC diskutiert:

1. Dynamische Sprungadressen: Sprungadressen werden von dem Stack gelesen. Sie sind nicht Teil des Payloads der *JUMP*-Instruktion oder der *JUMPI*-Instruktion und können daher dynamisch zur Laufzeit berechnet werden. Im Prinzip sind sie daher vor der konkreten Ausführung im Allgemeinen nicht zu errechnen. Durch verschiedene Heuristiken und der Verwendung von symbolischer Ausführung sind Sprünge relativ gut rekonstruierbar.

2. Sobald eine Funktion in Solidity aufgerufen wird, so wird die Sprungadresse dieser Funktion direkt vor der *JUMP*-Instruktion oder der *JUMPI*-Instruktion auf dem Stack abgelegt. Aufgrund der funktionsorientierten Natur von Solidity und dem fehlenden Konzept von Funktionen im EVM-BC muss die Return-Adresse der aufrufenden Funktion vor dem Aufruf der eigentlichen Funktion auf dem Stack abgelegt werden. Dieser Umstand macht es im Prinzip sehr schwierig, den EVM-BC in einen korrekten CFG umzuwandeln. Kurz gesagt, Sprungadressen am Stack sind nur bedingt korrelierbar zu der dazugehörigen *JUMP*-Instruktion oder *JUMPI*-Instruktion.

3. Der Einstieg in SC durch Message Calls wird über einen eigens kompilierten Dispatcher abgewickelt. Das ist der erste Code, der zur Ausführung gebracht wird. Er liest den Inhalt der Transaktion aus und entscheidet anhand der ersten vier bytes der Transaktion, welche Funktion im SC aufgerufen werden soll.

4. Der Konstruktor eines SC wird nur ein einziges Mal aufgerufen, danach wird er verworfen und ist somit nicht mehr Bestandteil des Account States des SC.

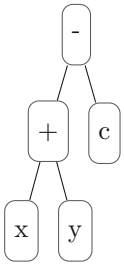
> Um an den Konstruktorcode zu kommen, muss der Umweg über die SC Creation Transaction gegangen werden. In der Regel ist das aufwendiger, da Transaktionsdaten mitunter nicht einfach vorliegen.

Um eine statische Code-Analyse des EVM-BC eines SC durchzuführen, bietet es sich an, einen CFG als Einstiegspunkt heranzuziehen [CCCP21]. Dieser kann dann dazu benutzt werden, mögliche Folgen von Basisblöcken eines Programms zu berechnen.

Allgemein ist es unmöglich einen vollständigen CFG zu einem gegebenen EVM-BC zu berechnen. Daher setzt man hier auf verschiedene Heuristiken. In [CCCP21] setzt man beispielsweise auf eine symbolische Ausführung, gepaart mit der Annahme, dass Sprünge zu bestimmten Blöcken genau dann erfolgen, wenn der Stack die Sprungadresse enthält. Insgesamt werden so Kanten im CFG ermittelt, die real nicht vorhanden sind. Es handelt sich daher um eine "over approximation" des eigentlichen CFG.

2.2.3 Abstract Syntax Trees

Ein Abstract Syntax Tree (AST) ist die Baumdarstellung einer Instruktion und all ihrer Input-Argumente in rekursiver Form. Dabei werden Operationen nicht in ihre Werte evaluiert. Im folgenden Baum sind die Instruktionen: a=x+y, b=a-c grafisch als AST dargestellt.



In dieser Arbeit fokussiere ich mich insbesondere auf diese AST-Darstellungen, da sie sich als guter Startpunkt in der Erkennung von EVM-BC-Muster durch XQuery-Abfragen erwiesen haben. Sie erlaubt, bis zum Ursprung aller vorangegangen Berechnung zurück zu gehen und macht es möglich, alle involvierten Operationen einer Berechnung darzustellen, auf deren Basis sich danach entsprechende Muster erkennen lassen.

2.2.4 Symbolische Ausführung

Bei der symbolischen Ausführung können anstelle von konkreten Werten, Symbole zum Einsatz kommen. Symbole stellen beispielsweise simple Variablen dar, aber auch Intervalle sind möglich. Dadurch können Programme auch kontextfrei, also ohne das Auflösen bestimmter Werte, wie zum Beispiel für die *BLOCKHASH*-Instruktion, ausgeführt werden.

In diesem Fall würde für den Wert blockhash am Stack stattdessen eine Variable abgelegt werden. Diese Variable steht zunächst für einen statischen Wert. Im Bedarfsfall können Symbole aber auch Intervalle abdecken. Das macht vor allem dann Sinn, wenn eine Bedingung den Wertebereich einer Variable einschränkt: if (10 < x && x < 100) ..., innerhalb des Blocks der Bedingung gilt, dass x einen Wert zwischen 10 und 100 darstellen kann.

Solche Einschränkungen ermöglichen in weiterer Folge verschiedene Deduktionen, die beim Erkennen von Sprüngen hilfreich sein können.

Für die Mustererkennung in dieser Arbeit selbst werden keine Intervalle herangezogen. In zukünftigen Arbeiten kann das die Spezifität des vorgeschlagenen PoC aber weiter erhöhen.

KAPITEL 3

Ausgewählte Programmmuster

In Kapitel 2 wurde näher auf die Funktionsweisen von SCs im Kontext von Ethereum und Solidity eingegangen. Im Fokus der Beschreibung standen Teile der Architektur der EVM und die damit einhergehenden Schwierigkeiten bei der Rekonstruktion eines CFG. Es wurde die Notwendigkeit eines CFG bei der statischen Analyse des EVM-BC begründet. In diesem Kapitel werden nun die einzelnen SC-Muster und die korrespondierenden EVM-BC-Muster beleuchtet. Im darauf folgenden Kapitel 4 wird genau beschrieben sein, wie die einzelnen SC-Muster in ihrer EVM-BC Repräsentation erkannt werden können.

Im Laufe der Zeit haben sich bestimmte Anwendungsfälle von SCs herauskristallisiert, und bestimmte Programmmuster haben sich entwickelt. Beispiele wie Tokenstandards, Guard Checks, SC-Stufen, Orakel, Zufallsabhängigkeiten, Aktualisierbarkeit, Notfallstops, sind nur ein paar wenige, die an dieser Stelle zu erwähnen sind[SHS22].

Diese Programmmuster finden in verschiedenen SCs ihre Verwendung. Das zeigen beispielsweise die Ähnlichkeiten der Funktionssignaturen [dAS20], die in weiterer Folge auch eine Ähnlichkeit der verwendeten SC-Muster erahnen lassen.

3.1 Zugangsbeschränkungen

SCs-Funktionen haben intrinsisch keinen Schutz gegen nicht autorisierte Akteure [WZ18a]. Um ausschließlich bestimmten Gruppen oder Adressen den Zugriff auf schützenswerte Funktionen zu erlauben und gleichzeitig allen anderen zu verbieten, haben sich bestimmte SC-Muster entwickelt. Im Folgenden werden zwei verschiedene Typen fokussiert:

3.1.1 Eigentümerschaft

Zentral in diesem SC-Muster steht eine Speicher-Variable, in der die Eigentümerschaft für eine Adresse gesetzt wird. Diese Speicher-Variable, im Folgenden owner genannt, wird

in der Regel im Konstruktor-Code gesetzt. Bei dem Aufruf einer Funktion, der in der Variable owner hinterlegten Adresse, wird owner mit msg.sender verglichen. Bei einem positiven Vergleich beider Werte wird die Ausführung fortgesetzt, anderenfalls erfolgt die Terminierung durch eine *STOP*-Instruktion.

OpenZeppelin bietet eine standardisierte Implementierung an [Opea]. Diese enthält neben den oben genannten Merkmalen noch folgende Funktionen:

- **Transfer der Eigentümerschaft**:
  Erlaubt das nachträgliche (also zur Laufzeit) Verändern der in der Variable owner gespeicherten Adresse, durch den Aufruf von transferOwnership. Der Aufruf der transferOwnership-Methode selbst ist der aktuell gesetzten Adresse vorbehalten.
- **Verzicht auf die Eigentümerschaft**:
  Durch den Aufruf der Funktion renounceOwnership wird die owner-Variable auf die Adresse 0 gesetzt. Da niemand den privaten Schlüssel zur Adresse 0 besitzt, ist somit niemand in der Lage, Funktionen aufzurufen, die durch dieses SC-Muster geschützt sind. Da auch der Transfer der Eigentümerschaft nur von der aktuellen Adresse ausgeführt werden kann, ist es somit auch in Zukunft nicht mehr möglich, die Eigentümerschaft zu ändern.
  Somit sind alle Funktionen, die durch dieses SC-Muster geschützt wurden, für immer blockiert.
- **Events zum Tracken von Eigentümer-Transfers**:
  Durch das Publizieren eines Events namens “OwnershipTransferred” wird die Nachvollziehbarkeit eines Wechsels der Eigentümerschaft gewährleistet.

In Auflistung 3.1 wird ein Basis SC dieses Musters dargestellt.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.2;
3  contract UsingOwnershipPattern {
4    address owner;
5
6    constructor() payable {
7      owner = msg.sender;
8    }
9
10   modifier only_owner() {
11     require(msg.sender == owner);
12     _;
13   }
14
15   function protected_function() public only_owner {
16     // some functions which should be execute only by owners
17   }
18 }
```

Auflistung 3.1: Zeigt einen SC, der das Eigentümerschaft-SC-Muster implementiert. Der SC wurde auf die relevanten Merkmale des Musters reduziert und kann so für die Mustererkennung im EVM-BC-Muster herangezogen werden.

3.1.2 Rollenbasierte Autorisierung

Möchte man das obere SC-Muster auf verschiedene Berechtigungsgruppen erweitern, bietet sich die Verwendung der rollenbasierten Autorisierung an. Dabei werden Adressen Gruppen zugeordnet, die dieselben Berechtigungen haben. Auch hierzu bietet wieder OpenZepplin [Opeb] eine standardisierte Implementierung an.

Eine Gruppe wird dabei durch einen kurzen Text (entspricht dem Namen einer Rolle) identifiziert. Die Liste der Adressen wird in einer eigenen Datenstruktur namens RoleData gespeichert. Neben den Adressen in der Gruppe wird in derselben Datenstruktur die Rolle angegeben, die die Adressen bearbeiten darf. Unter "bearbeiten" versteht man dabei das Hinzufügen oder Löschen von weiteren Adressen (entspricht SC Adressen).

Verschiedene Rollen werden durch ein Mapping des Rollennamens auf die zuvor erwähnte Datenstruktur (RoleData) abgebildet. Zusätzlich dazu bietet OpenZepplin folgende weitere Funktionen an:

- **Funktion "hasRole(role, account)"**:
  Gibt einen Boolean zurück, der angibt, ob eine Rolle einer bestimmten Adresse zugewiesen ist.
- **Funktion "getRoleAdmin(role)"**:
  Gibt den Namen der Rolle zurück, der für das Administrieren der angegeben Rolle berechtigt ist.
- **Funktion "grantRole(role, account)"**:
  Weist eine bestimmte Rolle einer Adresse zu. Diese Funktion kann ausschließlich von einer Adresse aufgerufen werden, die selbst Mitglied jener Rolle ist, die als administrative Rolle eingetragen wurde.
- **Funktion "revokeRole(role, account)"**:
  Entfernt eine Adresse von einer bestimmten Rolle. Diese Funktion kann ausschließlich von einer Adresse aufgerufen werden, die selbst Mitglied jener Rolle ist, die als administrative Rolle eingetragen wurde.
- **Funktion "renounceRole(role)"**:
  Entfernt die eigene Adresse aus einer Rolle. Setzt voraus, dass die eigene Adresse diese Rolle hat.
- **modifier onlyRole(role)**:
  Wird benutzt, um den Zugriff auf bestimmte Funktionen nur Adressen zu erlauben, diese Rolle haben.

In Auflistung 3.2 wird ein Basis-SC dieses Musters dargestellt.

3.2 Bedingte Einschränkungen

Im Vergleich zu Abschnitt 3.1 können Einschränkungen auch auf andere Umstände bezogen werden. Beispielsweise können SC ihre Einschränkungen auf Orakel beziehen. Denkbar sind hier viele verschiedene Ansätze. Im Folgenden möchte ich auf zwei konkrete Varianten eingehen.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.2;
contract UsingAccessControlPattern {
  struct RoleData {
    mapping(address => bool) members;
    bytes32 adminRole;
  }
  bytes32 public constant DEFAULT_ADMIN_ROLE = 0x00;
  mapping(bytes32 => RoleData) private _roles;

  constructor() payable {
    _roles["role1"].members[msg.sender] = true;
    _roles["role2"].members[msg.sender] = true;
  }

  modifier only_role(bytes32 role) {
    if (!_roles[role].members[msg.sender])
      revert();
    _;
  }

  function protected_function1() public only_role("role1") {
    // some functions which should be execute only by accoounts having role 1
  }
  function protected_function2() public only_role("role2") {
    // some functions which should be execute only by accoounts having role 2
  }
}
```

Auflistung 3.2: Zeigt einen SC, der das rollenbasierte Autorisierung-SC-Muster implementiert. Der SC wurde auf die relevanten Merkmale des Musters reduziert und kann so für die Mustererkennung im EVM-BC-Muster herangezogen werden.

3.2.1 Ablaufdatum

Eine mögliche Einschränkung kann die Gültigkeit eines SC über die Zeit sein. Dabei möchte man bestimmte Funktionen ab einem bestimmten Zeitpunkt aktivieren bzw. deaktivieren.

In diesen Fällen kommt ein SC-Muster zum Einsatz, das sogenannte “Automatic Deprecation Pattern”[WZ18a]. Grundsätzlich denkbar sind hier auch weitere Zeitpunkte, die durch verschiedene Modifier implementiert werden können.

Das Muster besteht hierbei in dem Vorkommnis eines Modifiers und eines Zeitpunkts, der im SC hinterlegt werden kann. Der Modifier selbst bezieht sich nun entweder auf die Blocknummer oder aber auf den Block-Zeitpunkt. Funktionen, die diesen Modifier tragen, werden so aktiviert bzw. deaktiviert.

In Auflistung 3.3 wird ein Basis-SC dieses Musters dargestellt.

```
contract UsingDeprecationPattern {
  uint public _deprecationTime = 0x00;

  constructor() payable {
    _deprecationTime = block.timestamp + 7 * 1 days;
  }

  modifier beforeDeprecation() {
    require(block.timestamp < _deprecationTime);
    _;
  }

  modifier afterDeprecation() {
    require(block.timestamp > _deprecationTime);
    _;
  }

  function protected_function1() public beforeDeprecation {
  }
  function protected_function2() public afterDeprecation {
  }
}
```

Auflistung 3.3: Zeigt einen SC, der das Automatische-Ablauf SC-Muster implementiert. Der SC wurde auf die relevanten Merkmale des Musters reduziert und kann so für die Mustererkennung im EVM-BC-Muster herangezogen werden.

3.2.2 Notfall Stops

SCs, die einmal in Ethereum veröffentlicht wurden, können von allen Akteuren innerhalb des Netzwerks aufgerufen und ausgeführt werden. Es gibt keine im System verankerten Mechanismen, um den Aufruf solcher SC bei Bekanntwerden eines Fehlers zu unterbinden.

Es bietet sich daher an, eine Möglichkeit im Code des SC zu implementieren, um in genau solch einem Fall die weitere Ausführung zu unterbinden, bis der Fehlerzustand behoben wurde oder aber sogar für immer gestoppt werden muss.

Dieses SC-Muster erlaubt es im Notfall, die Ausführung bestimmter Funktionen gänzlich zu stoppen [WZ18b].

Das Muster besteht hierbei in dem Vorkommnis eines Modifiers und einer boolean Speicher-Variable, die im SC gesetzt werden kann. Der Modifier selbst bezieht sich auf diese Speicher-Variable und stellt sicher, dass sie nicht gesetzt wurde. Funktionen, die diesen Modifier tragen, können so im Zweifelsfall deaktiviert werden.

Denkbar ist auch ein weiterer Modifier, der die Ausführung im Notfall nicht beschränkt, sondern sie in diesem Fall gestattet. Denkbar wäre zum Beispiel eine withdraw-Funktion, die im Fehlerfall ausgeführt werden kann. So kann weiterer finanzieller Schaden abgewendet werden.

In Auflistung 3.4 wird ein Basis-SC dieses Musters dargestellt.

```
contract UsingEmergencyStop {
  bool public _isStopped = false;


  modifier stoppedInEmergency {
    require(!_isStopped);
    _;
  }

  modifier onlyAuthorized() {
    // some pattern to ensure proper authorization
    _;
  }

  function stopContract() public onlyAuthorized {
    _isStopped = true;
  }

  function resumeContract() public onlyAuthorized {
    _isStopped = false;
  }

  function protected_function1() public stoppedInEmergency {
  }
}
```

Auflistung 3.4: Zeigt einen SC, der das Notfall-Stop SC-Muster implementiert. Der SC wurde auf die relevanten Merkmale des Musters reduziert und kann so für die Mustererkennung im EVM-BC-Muster herangezogen werden.

3.3 Aktualisierbarkeit

Der Code eines unter einer Adresse veröffentlichten SCs ist unveränderlich. Mit einer *SELFDESTRUCT*-Instruktion lässt er sich gänzlich löschen, aber Änderungen sind nicht möglich. Eine ähnliche Problematik wie in Unterabschnitt 3.2.2 führt zu dem Bedarf an SCs, dessen Verhalten sich zu einem späteren Zeitpunkt ändern lässt.

Anstelle eines änderbaren SC könnte man auch einen gänzlich neuen SC und den Zustand des alten Klons veröffentlichen. Das hat mehrere Nachteile:

- Events sind adressbezogen. Eine neue Instanz eines SC würde diese Events nicht erben.
- Die Adresse des bestehenden SC kann nicht wiederverwendet werden. Sofern die Adresse einer größeren Personengruppe bekannt war, ist die nachträgliche Änderung und Sicherstellung, dass alle Akteure darüber im Klaren sind, nicht einfach durchzuführen.

Hier kann ein SC-Muster die gewünschte Funktionalität liefern, das sogenannte “Contract Relay”-SC-Muster [WZ18a]. Es erlaubt jegliche Interaktion mit dem SC (A) an einen anderen zu bestimmenden SC (B) weiter zu delegieren.

Dabei kann dann SC B direkt auf den Speicher des aufrufenden SC (A) zugreifen und diesen manipulieren. Damit bleibt sowohl der Speicher über ein Update, als auch die Adresse des SC erhalten. Bei einem Update ändert sich daher nur die Zieladresse von (A) auf den neu erstellten SC (C).

Das Muster besteht hierbei in dem Vorkommnis einer Speicher-Variable, welche die Zieladresse in A trägt, und der Verwendung der *DELEGATECALL*-Instruktion. Verschiedene Arten dieser SCs sind denkbar, hier soll nun auf eine besondere Ausprägung eingangen werden, welche die Transaktionsdaten direkt an B übermittelt. Dadurch werden alle eingehenden Aufrufe von A an B delegiert.

Typischerweise enthält A weitere Funktionen, um die Zieladresse zu ändern und um unbefugten Zugriff auf diese Funktion zu verhindern.

In Auflistung 3.5 wird ein Basis SC dieses Musters dargestellt.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.2;
3  contract UsingRelayPattern {
4    address private destination;
5
6    constructor(address initDestination) payable {
7      destination = initDestination;
8    }
9
10   fallback()  external {
11     (bool success, bytes memory data) = destination.delegatecall(msg.data);
12     require (success);
13   }
14 }
```

Auflistung 3.5: Zeigt einen SC, der das Aktualisierbarkeit SC-Muster implementiert. Der SC wurde auf die relevanten Merkmale des Musters reduziert und kann so für die Mustererkennung im EVM-BC-Muster herangezogen werden.

KAPITEL 4

Automatische Erkennung der Programmmuster

Dieser Abschnitt ist der Mustererkennung der in Kapitel 3 vorgestellten SC-Muster gewidmet. Wir konzentrieren uns auf den Prozess, der die Ermittlung solcher SC-Muster anhand von EVM-BC-Muster möglich macht, und beschreiben das konkrete EVM-BC-Muster für jedes SC-Muster. Abweichungen und Variationen zu den in Kapitel 3 angeführten SC-Mustern werden diskutiert und in den Kontext der Abfragesprache gestellt.

Gezeigt werden die vom PoC getätigten wesentlichen Schritte sowie die verschiedenen externen Tools, die für die Generierung des CFGs und der XML-Dokumente verwendet wurden. Die Module und der modulare Aufbau des PoC werden beschrieben und in Beziehung gesetzt, indem auf die wesentlichen Programmabschnitte sowie auf die symbolische Ausführung des EVM-BC eingegangen wird.

Anhand von Beispielen werden die entwickelten Transformationsregeln von ASTs in deren XML Repräsentationen dargestellt. Daran anschließend werden die so erzeugten XML-Dokumente eingesetzt, um anhand von XQuery-Abfragen zu zeigen, wie Muster erkennbar werden und durch welche Abfragen sie beschreibbar sind.

4.1 Im PoC angewandte Konzepte

In diesem Abschnitt soll kurz auf die angewendeten Methoden und deren Ergebnisse eingegangen werden, die während der Prozessierung der Daten verwendet werden bzw. anfallen. Das soll im darauffolgenden Abschnitt helfen, den Überblick über den Gesamtprozess zu bewahren.

4.1.1 Instruktionen, Basisblöcke, CFG und Symbolische Ausführung

In dieser Arbeit wurde eine symbolische Ausführungseinheit entwickelt, die gegebenen EVM-BC interpretiert und symbolisch ausführt. Zunächst wird der EVM-BC in seine Instruktionen dekodiert. Die Instruktionen werden danach in Blöcke und diese wiederum anhand bestimmter Heuristiken in Programme gruppiert.

Zu den so erhaltenen Programmen wird der jeweilige EVM-BC von einem externen Tool zu einem CFG verarbeitet. Dieser CFG stellt die Blöcke des jeweiligen Programms in Beziehung.

Im Anschluss werden die einzelnen Programme erneut durch den eigens entwickelten Interpreter symbolisch ausgeführt. Dabei wird der zuvor erhaltene CFG verwendet und adaptiert, um so auf möglichst alle unterschiedlichen Programmdurchläufe zu kommen. Diese Programmdurchläufe stellen jeweils eine Folge von aneinander geketteten Basisblöcken dar. Diese Abfolgen werden zunächst auf Plausibilität geprüft, bevor sie für die Generierung der XML-Dokumente herangezogen werden.

4.1.2 XML

Die einzelnen Programmdurchläufe werden nach definierten Regeln in XML-Dokumente transformiert. Die Dokumente können nun über bereits bestehende effiziente XML-Prozessoren verarbeitet werden.

XML ist eine strukturierte bzw. semistrukturierte Möglichkeit, Daten darzustellen. XML hat den Vorteil seiner bereits weit zurückreichenden Geschichte und bietet daher viele frei verfügbare Tools, die stark optimiert auf XML-Dokumenten operieren können. XML ist außerdem weit verbreitet, einfach zu lernen, und die leichte Lesbarkeit der Ausdrücke in XML-Pfadsprache (XPath) macht die Kombination von XML, XQuery und XPath zu einer interessanten Option.

4.1.3 XQueries

XML-Dokumente können über eine geeignete Abfragesprache wie XQuery verarbeitet werden [BCF+02]. Deshalb wird in dieser Arbeit XQuery als Abfragesprache zur Mustererkennung nach der symbolischen Ausführung von EVM-BC herangezogen.

Im Konkreten beschränkt sich diese Arbeit derzeit ausschließlich auf die Entwicklung von XPath-Ausdrücken zur Mustererkennung. Wie sich herausstellte, sind sie bereits aussagekräftig genug, um die in dieser Arbeit vorgestellten Muster zu erkennen. XQuery-Ausdrücke selbst erlauben noch einmal deutlich komplexere Ausdrücke zur Mustererkennung, die auf den in dieser Arbeit generierten XML-Dokumenten aufsetzen können.

4.2 Modularer Programmaufbau

Dieser Abschnitt enthält einen groben Überblick über die einzelnen Prozessschritte und die dafür herangezogenen Tools und implementierten Module. Der entwickelte PoC

verwendet die in Abschnitt 4.1 angeführten Konzepte.

Weiters wird das generierte XML-Dokument und dessen Aufbau beschrieben. In den darauf folgenden Abschnitten werden anhand der verschiedenen SC-Muster die Details des erzeugten XML-Dokuments beschrieben und aufgezeigt, wie bestimmte EVM-BC-Muster durch Abfragen in XQuery erkannt werden können.

In Abbildung 4.1 ist der Aufbau des Programms in seinen einzelnen Modulen gezeigt.

4.2.1 Compiler

Dieser Abschnitt fasst die zwei Komponenten "Compiler Selection" und "Compiler Service" zusammen. Im eigentlichen Sinn sind diese beiden Komponenten nicht Teil des Klassifizierungsmechanismus. Sie erfüllen aber einen wertvollen Bestandteil dieser Arbeit. Im Compiler Service findet sich jene Logik, die zu einem gegebenen SC-Code den neuesten möglichen Compiler auswählt.

Das Compiler Service lädt beim Applikationsstart alle aktuell verfügbaren Compiler von Github herunter.

Für die Entwicklung von möglichst robusten Ausdrücken in XQuery zur Erkennung von EVM-BC-Muster, ist es notwendig den Output unterschiedlicher Compilerversionen zu vergleichen, um so Asudrücke mit einer hohen Robustheit zu erzeugen.

In dieser Arbeit wurden die verschiedenen Compiler-Versionen zur Evaluierung der Robustheit der Abfragen in XQuery gegenüber unterschiedlichen Compiler-Versionen benutzt.

4.2.2 Instruction-Reader

Der "Instruction-Reader" dekodiert den rohen EVM-BC in seine einzelnen Instruktionen. Eine Instruktion entspricht dabei einem OPC und seinem Payload. Typischerweise ist der Payload leer. *PUSH*-Instruktionen können bis zu 32 Byte Payload enthalten. Alle möglichen OPCs finden sich im Yellow-Paper [Woo17] aufgelistet.

Der Instruktionen-Reader kennzeichnet dabei jede einzelne Instruktion, sodass deren exakte Position im EVM-BC über alle Module erhalten bleibt. Diese Position wird im Folgenden "global-position" genannt. Sie steht im Kontrast zur "Program-Position" und "Block-Position", die weiter unten erläutert werden. Die Position setzt sich zusammen aus dem Offset in Bytes sowie dem Index im Stream der Instruktionen des EVM-BC. Jede Instruktion erhält so gleichzeitig eine globale ID, die im generierten XML-Dokument als Attribut zu jedem Vorkommnis der Instruktion ausgegeben wird.

Die konkrete Implementierung für das Resultat beruht dabei auf der Implementierung eines IEnumerables. Das erlaubt das speichereffiziente Iterieren aller notwendigen Instruktionen, ohne das Zwischenspeichern als Liste.

Als Beispiel sei der Auszug der EVM-BC Sequenz des Runtime-Codes des SC in Auflistung 4.1 gegeben:

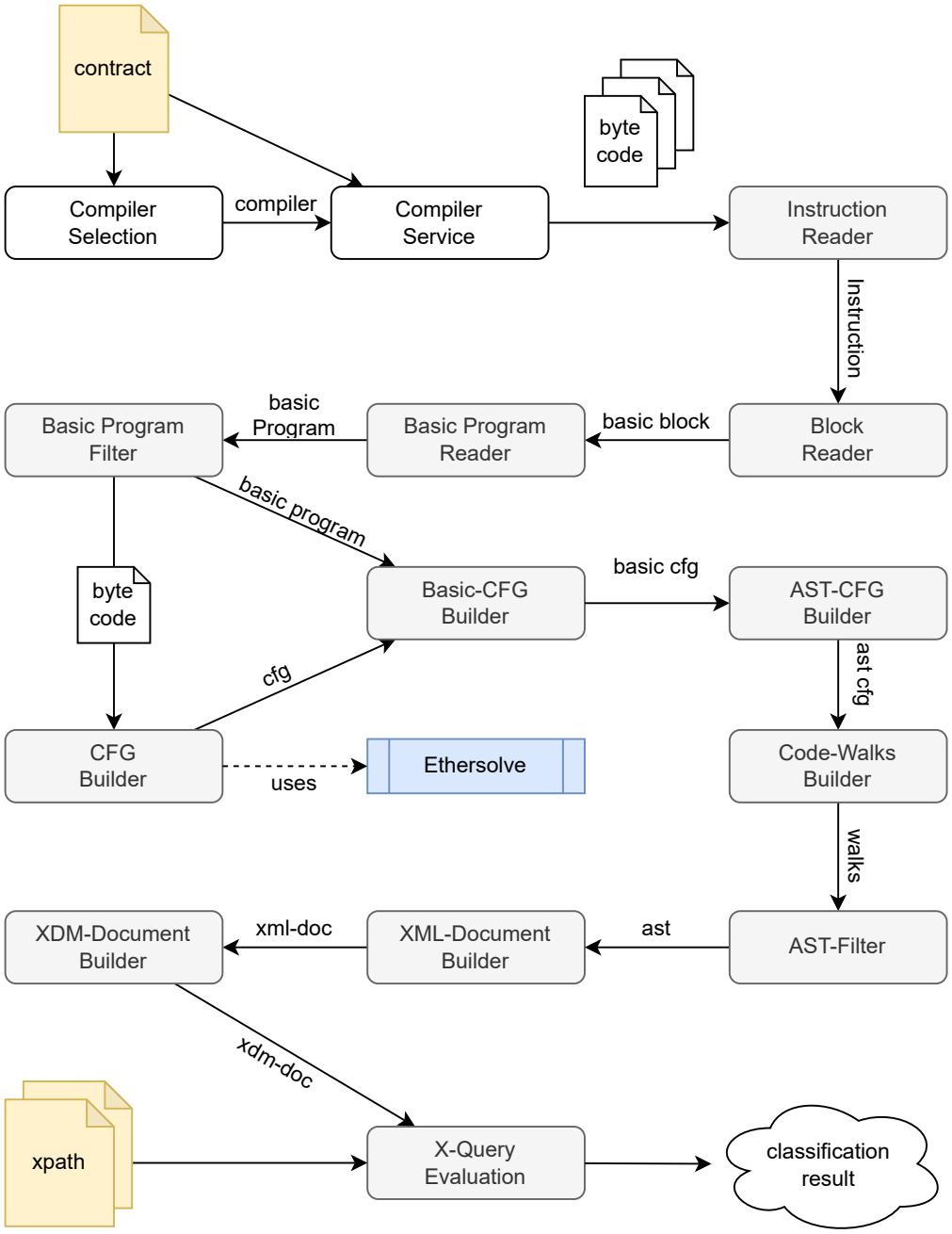


Abbildung 4.1: Zeigt den modularen Aufbau des entwickelten Prototypen zur Klassifizierung von EVM-ByteCode. Gezeigt ist der Data-Flow und der Datentyp nach der Ausführung der einzelnen Module.

```
pragma solidity 0.8.15;
contract Sample {
    function run() public pure returns (uint) {
        uint result = 0;
        for (uint i = 0; i < 10; i++) {
            result += i;
        }
        return result;
    }
}
```

Auflistung 4.1: Zeigt einen einfachen SC Source Code. Er enthält eine for-Schleife die die Summe über die Zahlen 0-9 bildet.

```
0x5b60008060009050600 5b600a8110156100825780826100 6d91906100ed56
```

Auflistung 4.2: Auszug der EVM-BC Sequenz des Runtime-Codes in Auflistung 4.1

Dekodiert in seine Instruktionen ergibt sich nachfolgende Instruktionssequenz. Dabei wird der Payload von *PUSH*-Instruktionen in runden Klammern gesetzt:

```
JUMPDEST PUSH1(0x00) DUP1 PUSH1(0x00) SWAP1 POP PUSH1(0x00) JUMPDEST
PUSH1(0xA) DUP2 LT ISZERO PUSH2(0x0082) JUMPI DUP1 DUP3 PUSH2(0x006D)
SWAP2 SWAP1 PUSH2(0x00ED) JUMP
```

Auflistung 4.3: Dekodierte Instruktionen des EVM-BC in Auflistung 4.2.

4.2.3 Block-Reader

Der Block-Reader nimmt ein IEnumerable von Instruktionen als Input. Dieses Enumerable wird nun schrittweise abgearbeitet, indem die Instruktionen in Basisblöcke gruppiert werden. Die übergebenen Instruktionen müssen daher in Leserichtung der EVM-BC-Sequenz sortiert übergeben werden.

Der Start eines Blocks ist durch eine der folgenden Bedingungen definiert:

- Aktuelle Instruktion ist eine Sprungadresse (*JUMPDEST*-Instruktion), und die Menge der zuvor gefundenen Instruktionen ist nicht leer. Die Menge darf nicht leer sein, um zu verhindern, dass ein neuer Block generiert wird, wenn auf eine *JUMPI*-Instruktion eine *JUMPDEST*-Instruktion folgt. In diesem Fall würde ansonsten durch die *JUMPI*-Instruktion ein neuer Block geöffnet werden und durch die *JUMPDEST*-Instruktion ein weiterer.
- Die vorherige Instruktion war ein Sprung (*JUMPI*-Instruktion oder *JUMP*-Instruktion). In jedem Fall führen Sprünge zu einem Abschluss eines Blocks, daher muss die darauffolgende Instruktion bereits Teil eines neuen Blocks sein.
- Die vorherige Instruktion war eine programmterminierende Instruktion (*STOP*-Instruktion, *RETURN*-Instruktion, *REVERT*-Instruktion, *INVALID*-Instruktion oder *SELFDESTRUCT*-Instruktion). Nach der Terminierung werden keine weiteren

Instruktionen mehr ausgeführt. Nachfolgende Instruktionen können also nicht mehr Teil des Blocks sein.
- Es handelt sich um die erste Instruktion des EVM-BC. Die erste Instruktion markiert immer den Beginn eines Blocks. Jedes Programm besitzt genau einen Startblock. Das ist der Block an der Programmposition 0.

Instruktionen, die nicht den Start eines Blocks einleiten, werden bis zum Ende eines Blocks in einer sortierten Datenstruktur, einer Liste, aufgehoben. Dabei wird jede Instruktion wiederum mit dessen relativer Position im Block getaggt. Die Position setzt sich zusammen aus dem Offset in Bytes, sowie aus dem Index der Instruktionen eines Blocks.

Sobald eine blockterminierende Instruktion oder der Start eines neuen Blocks gefunden wird, wird der aktuell bearbeitete Block geschlossen und zurückgegeben. Die konkrete Implementierung beruht hier ebenfalls auf der Implementierung eines IEnumerables, das ähnliche Performance gewinne wie im Instruction-Reader ermöglicht.

Instruktionen, die einen Block terminieren lassen sind:

- *JUMP*-Instruktionen: Sie springen direkt an den Beginn eines anderen Blocks, daraus folgt, dass der aktuelle Block enden muss.
- *JUMPI*-Instruktionen: Ist die Kondition der *JUMPI*-Instruktion true, verhält sie sich ident zur *JUMP*-Instruktion. Anderenfalls erfolgt die Ausführung der nachfolgenden Instruktionen. Da dies aber nicht immer der Fall ist, muss der Block an dieser Stelle enden.
- *STOP*-Instruktion, *RETURN*-Instruktion, *REVERT*-Instruktion, *INVALID*-Instruktion und *SELFDESTRUCT*-Instruktion: führen zur Terminierung einer Programmausführung. Damit ist das Blockende natürlicherweise gegeben.

Blöcke, die ein Programm terminieren lassen werden, im folgenden Exit-Blöcke genannt [CCCP21]. Exit-Blöcke, die eine Zustandsänderung zulassen (also all jene Blöcke, die mit einer *RETURN*-Instruktion, *SELFDESTRUCT*-Instruktion oder einer *STOP*-Instruktion enden), werden als Final-Blöcke bezeichnet. Exit-Blöcke, die keine Zustandsänderung zulassen (Blöcke die mit einer *REVERT*-Instruktion oder einer *INVALID*-Instruktion enden), werden Revert-Blöcke genannt.

Für diese Arbeit sind sowohl die Erkennung von Programmausführungen relevant, die zu einer Zustandsänderung führen, wie auch jene, die zu keiner Zustandsänderung führen. Beide haben in der Mustererkennung ihren Stellwert. Mehr dazu ist weiter unten dargestellt.

4.2.4 Basic-Program-Reader

Die zuvor dekodierten Blöcke beschreiben ein oder mehrere Programme, die in weiterer Abfolge von Interesse sein können. Der Basic-Program-Reader übernimmt daher das Enumerable von Blöcken als Input und gruppiert diese Blöcke in einzelne Programme.

Analog zum Block-Reader müssen auch hier die einzelnen Blöcke in Leserichtung übergeben werden. Der Basic-Program-Reader setzt voraus, dass Programme aus Regionen von

```
JUMPDEST
PUSH 0
DUP1
PUSH 0
SWAP1
POP
PUSH 0
```

```
JUMPDEST
PUSH A
DUP2
LT
ISZERO
PUSH 0x0082
JUMPI
```

```
DUP1
DUP3
PUSH 0x006D
SWAP2
SWAP1
PUSH 0x00ED
JUMP
```

Abbildung 4.2: Zeigt die Instruktionen aus Auflistung 4.3 separiert in einzelnen Blöcke.

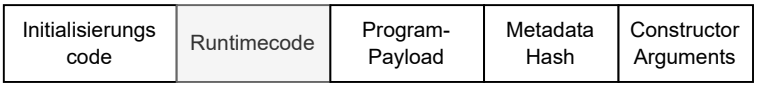


Abbildung 4.3: Zeigt den typischen Aufbau eines mit solc kompilierten Solidity SC.

zusammenhängenden Blöcken bestehen. Diese Regionen werden nicht durch andere Programme unterbrochen, hingegen sind Nutzdaten (Konstruktor-Argumente, verschachtelte Programme) des Programms, die außerhalb der Region liegen, zulässig.

Typischerweise setzen sich Programme, die durch solc kompiliert wurden, aus verschiedenen semantischen Bereichen zusammen, siehe Abbildung 4.3:

- Initialisierungscode: ein Programm, das Runtimecode auf der Ethereum Blockchain persistiert. In Solidity entspricht es dem Funktionskörper des Konstruktors. Er selbst wird dabei nicht auf der Blockchain in Form von Code an der Adresse abgespeichert. Stattdessen wird er nach der Ausführung der Transaktion verworfen. Er bleibt aber in den Transaktionsdaten vorhanden. Im Fall einer *CREATE*-Instruktion oder einer *CREATE2*-Instruktion bleibt der Konstruktor im ausführenden SC erhalten.
- Runtimecode: das Programm, das auf der Ethereum Blockchain persistiert wird und zu einem späteren Zeitpunkt durch Transaktionen aufgerufen werden kann. Er enthält typischerweise den zu klassifizierenden Programmcode.
- Programm Payload: Programme, die beim Ausführen einer *CREATE*-Instruktion oder *CREATE2*-Instruktion auf der Blockchain durch den Runtimecode oder den Initialisierungscode erstellt werden können. Es handelt sich um verschachtelte Programme, die durch ein Programm veröffentlicht werden können.
- Metadata Hash: ein Hash, der den Inhalt des Programms beschreibt. Typischerweise wird auch die verwendete Compilerversion im Metadaten Bereich mit angeführt.
- Konstruktor-Argumente: Input für die Argumente des Konstruktors.

Diese Programme zu separieren ist durch die Anwendung einer Heuristik möglich. Wie bereits oben angeführt, werden beim Aufbau des Kompilats des typischerweise verwendeten Compilers (solc) Programme hintereinander angeordnet. Diese Programme selbst bestehen aus mehreren Blöcken, die wiederum aus Instruktionen bestehen. Das Ende eines Programms und damit der Beginn eines neuen lässt sich folgendermaßen definieren:

- Der Block hat keine Sprungadresse (startet nicht mit einer *JUMPDEST*-Instruktion).
- Der Block ist nicht der Folgeblock eines Blocks, der mit einer *JUMPI*-Instruktion endet.
- Der Block ist nicht der erste Block, der untersucht wurde.

Sofern die ersten zwei Möglichkeiten ausgeschlossen werden können, handelt es sich um den Start eines Programmes. Die nachfolgenden Blöcke werden also zu einem neuen Programm gruppiert.

Der Basic-Program-Reader kennzeichnet dabei jeden einzelnen Block und jede einzelne Instruktion darin, sodass die exakte Position jedes Blocks und jeder Instruktion im Kontext eines Programms ermittelt werden kann. Diese Position wird im folgenden Programm-Position genannt. Die Position setzt sich zusammen aus dem Offset in Bytes sowie dem Index der Liste der Blöcke des Programms.

Die Blöcke werden so lange zu einem Programm gruppiert, bis das Programm endet. Ein Programm endet typischerweise:

- mit einer *INVALID*-Instruktion nach dem letzten Block,
- seinem Payload bzw. dem Metadatenhash
- oder dem Ende des EVM-BC.

Die Metadaten am Ende entsprechen beispielsweise einem Swarm hash, er besteht aus dem gehashten Metadatenfile und hat je nach verwendetem Compiler einen unterschiedlichen Inhalt, ist jedoch stets CBOR[1] kodiert.

Beispiele für die möglichen unterschiedlichen Inhalte des Metadatenhashes sind:

- in v0.8.17 kann er beispielsweise folgendes codiertes JSON enthalter (weitere Felder sind möglich):
  `{"ipfs": <IPFS hash>, "solc": <compiler version>}` [sola]
- in v0.4.26 hingegen
  `{"bzzr0": <Swarm hash>}` [sola]

Der Metadaten-Abschnitt wird in dieser Arbeit nicht weiter betrachtet. Er kann in zukünftigen Version für eine verbesserte versionsabhängige Mustererkennung herangezogenen werden. Damit könnte basierend auf den Daten entschieden werden, welche XQuery zur Erkennung von SC-Muster herangezogen werden soll.

An dieser Stelle soll außerdem darauf hingewiesen werden, dass solch ein Hash nicht zwangsweise vorhanden sein muss, um die Compilerversion bestimmbar zu machen. Typischerweise ändert sich die Zusammensetzung von bestimmten Operation bei der Übersetzung der höher leveligen Sprache Solidity unter Verwendung unterschiedlicher Compilerversionen. Dadurch könnten bestimmte EVM-BC-Muster selbst Aufschluss über die verwendete Compilerversion geben. Aus dieser Sicht könnte auch hier wieder eine

[1] Ist eine binäre Datenkodierungssprache. CBOR und JSON dienen dazu, beliebige Daten zu serialisieren.

Erkennung von EVM-BC-Muster durch die in dieser Arbeit vorgeschlagene Methode verwendet werden.

4.2.5 Basic-Program-Filter

Dieses Modul wurde sehr einfach gehalten, es überspringt das erste Programm und nimmt dann das darauf folgende Programm. Alle weiteren Programme werden übersprungen. Damit wird sichergestellt, das ausschließlich der Runtime-Code betrachtet wird.

Verschiedene Anpassungen sind denkbar, bei trivialen Fragestellungen, die sich auf den Konstruktor-Code beziehen, müsste dieser Filter entsprechend angepasst werden. Die Einschränkung auf Konstruktor bzw. Runtime-Code kann aber bereits vor dem Instruction-Reader durch die Übergabe des spezifischeren EVM-BC bewerkstelligt werden. Hier bietet der Basic-Program-Filter keinen zusätzlichen Vorteil.

Anwendungsorientierter sind daher Fragestellungen, die sich auf einen verschachtelten Programmcode beziehen oder aber diesen ausschließen möchten. Hier könnte der Basic-Program-Filter an Bedeutung gewinnen.

4.2.6 CFG-Builder

Die CFG-Builder Komponente wandelt einen gegebenen EVM-BC in seinen CFG um. Der CFG selbst wird dabei durch ein externes Tool konstruiert. Der PoC integriert sowohl EtherSolve [CCCP21] als auch den EVM-CFG-Builder, der die Basis für Manticore bildet [MMH+19].

Der CFG-Builder normalisiert den Output des gewählten externen Tools in eine einheitliche Struktur. Die externen Tools geben in der Regel zu einer gegebenen Blockadresse alle seine möglichen Nachfolgeblöcke in Form der Adressen zurück. Diese Struktur wird als 2-Tupel mit dem zuvor dekodierten Programm dem nächsten Modul übergeben.

Der CFG, der durch EtherSolve erzeugt wird, entspricht einer sogenannten “over approximation”, der durch die Anwendung einer “symbolic stack execution” ermittelt wurde [CCCP21]. In der Regel führt das dazu, dass mehr Knoten und Kanten im CFG gefunden werden als tatsächlich existieren.

Solch eine “over approximation” stellt sich in dieser Arbeit als sinnvoll heraus, da spätestens beim Codewalk genau solche Knoten und Kanten in der Regel gefunden und eliminiert werden.

In der Abbildung 4.4 wird der Output des externen Tools beispielhaft dargestellt.

4.2.7 Basic-CFG-Builder

Die Basic-CFG-Builder Komponente wandelt den Output des CFG-Builders (Output von EtherSolve bzw. EVM-CFG-Builder) von seiner adressbasierten Darstellung, also dem Mapping von Blockadressen zu Blockadressen, in seine Objektreferenzen um. Die Objektreferenzen werden dabei vom Basic-Program-Filter übergeben.

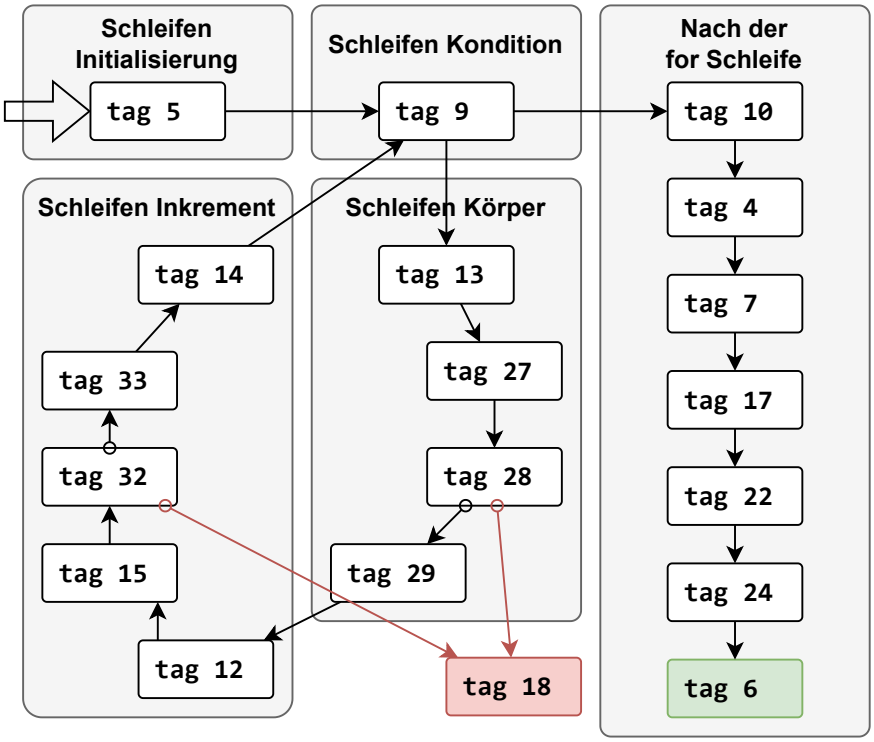


Abbildung 4.4: Vereinfachter CFG des Programms in Auflistung 4.1. Der Block “tag 18” zeigt eine Fehlersituation aufgrund eines Over- oder Underflows beim Berechnen der Laufvariable und dem Inkrementieren der Ergebnisvariable. Der Block “tag 6” zeigt hingegen den Return-Block am Ende der Funktion. Der “Funktionen-Dispatcher” des SC ist hier nicht dargestellt.

Dadurch entsteht eine bidirektionale Datenstruktur mit dem Informationsgehalt des CFG und des Outputs des Program-Builders. Diese kombinierte Datenstruktur erlaubt schließlich das Zusammenstellen von Codewalks in den nachfolgenden Schritten.

In Abbildung 4.5 und Abbildung 4.6 wird die Datenstruktur auf Grundlage des Programms in Auflistung 4.1 dargestellt. Dabei werden Kanten zu Block “tag 18” einfachheitshalber ausgeblendet. Bei Block “tag 18” handelt sich um den Exit-Block, der bei Überläufen in der Integer-Arithmetik das Programm zur Terminierung bringt. Durch eine rote Färbung wurden die *JUMP*-Instruktionen kenntlich gemacht, die diesen Block adressieren.

Block “tag 16” nimmt das Stack-Element mit Index 1 und tauscht es gegen das Stack-Element mit Index 0 und springt dann zu dem Element mit Index 1. Das hat in diesem Programm keinen semantischen Effekt. Daher wurde bei jedem Sprung auf “tag 16” stattdessen direkt der Sprung auf das Element am Stack mit Index 1 dargestellt. Semantisch ist so der korrekte Programmfluss gegeben. Um das Programm korrekt auszuführen, muss aber der Umweg über Block “tag 16” gegangen werden.

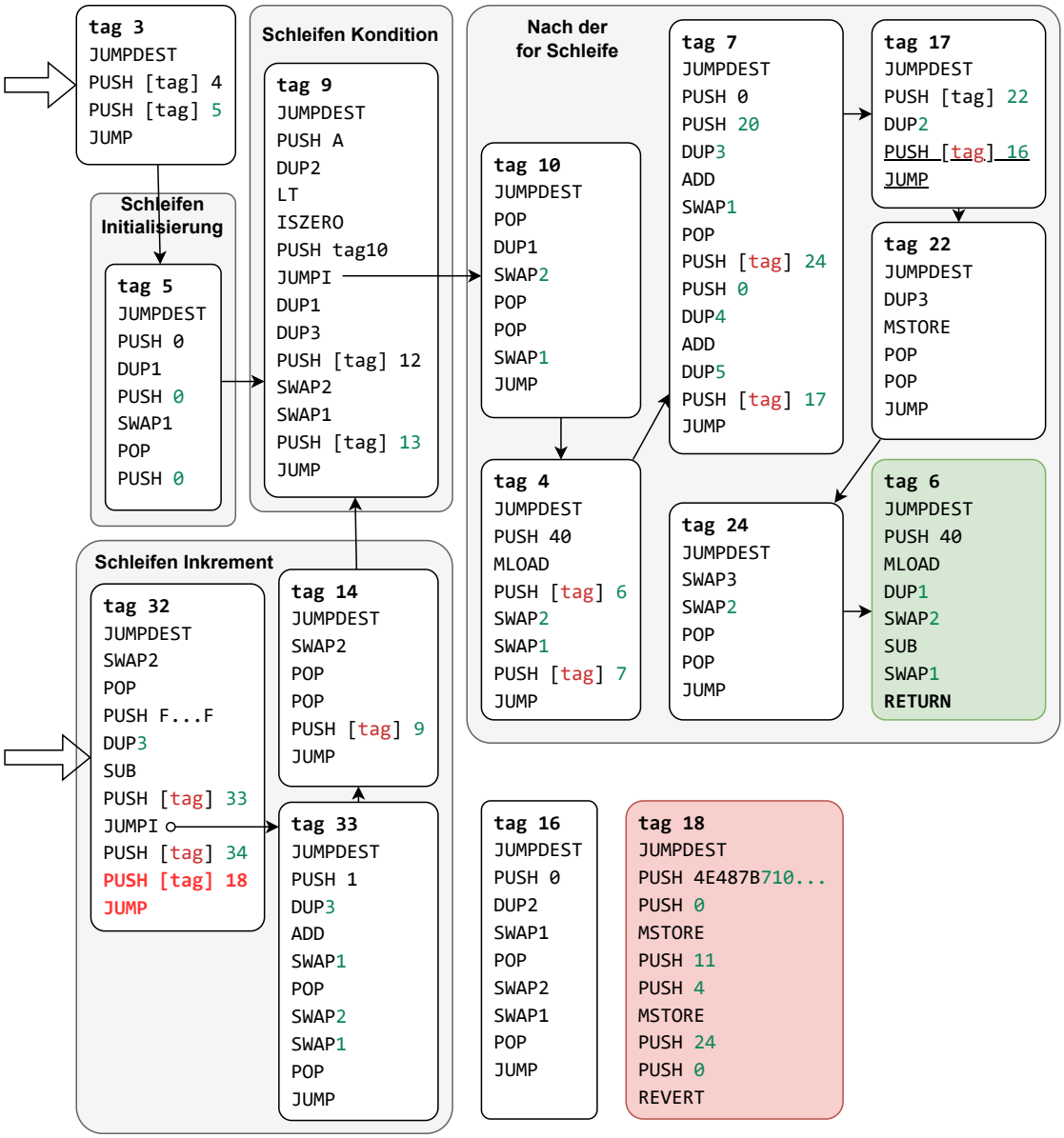


Abbildung 4.5: Zeigt das Disassembly des SC in Auflistung 4.1. Hierbei handelt es sich nicht ausschließlich um Basic-Blocks. Folgeblöcke von *JUMPI*-Instruktionen sind nicht separat als einzelne Blöcke aufgeführt. Sprungadressen sind durch Tags realisiert. Diese Abbildung zeigt nicht den Schleifen-Körper, dieser findet sich in Abbildung 4.6

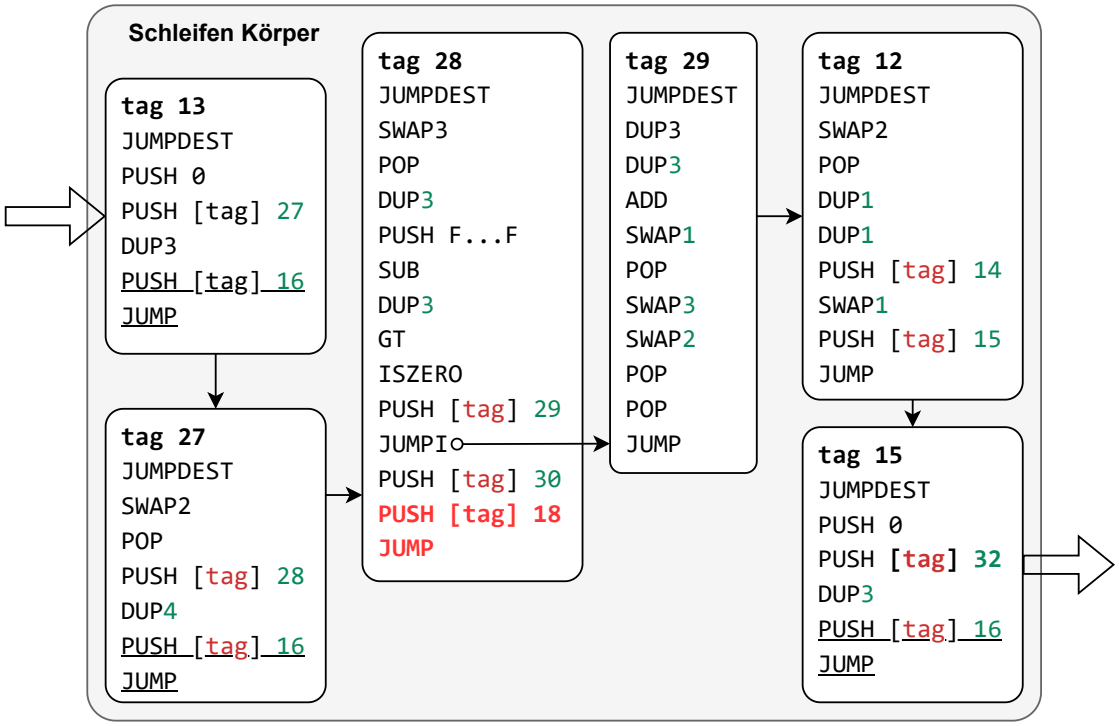


Abbildung 4.6: Zeigt das Disassembly des SC in Auflistung 4.1. Zu Veranschaulichungszwecken handelt es sich nicht um eine Basic-Block Representation, Continuation Blocks sind nicht seperat aufgeführt. Sprungadressen sind durch Tags markiert. Diese Abbildung zeigt ausschließlich den Schleifen Körper, der Rest findet sich in Abbildung 4.5

4.2.8 AST-CFG-Builder

Der AST-CFG-Builder wandelt die Instruktionen eines Blocks in dessen AST Repräsentation um. Dabei wird jede einzelne Instruktion in ihre AST-Darstellung konvertiert. Operationen ohne Stack-Argumente, wie beispielsweise PUSH und *CALLER*-Instruktionen, bilden dabei die Blattknoten für Operationen mit Stack-Argumenten wie ADD und *SHA3*-Instruktionen.

Für jeden Block, der mehr als eine eingehende Kante besitzen könnte, muss ein neuer Stack aufgebaut werden. Dieser Stack ist zunächst leer. Neue Elemente können dabei auf dem Stack abgelegt und wieder behoben werden.

Blöcke, die maximal eine eingehende Kante haben, sind Blöcke, die auf eine *JUMPI*-Instruktion folgen und nicht mit einer *JUMPDEST*-Instruktion starten. Sie sind lediglich über den vorangegangenen Block erreichbar.

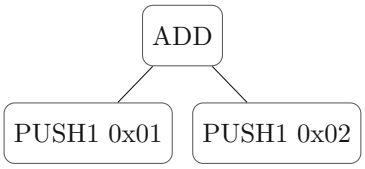
Am Ende eines Blocks können Elemente am Stack verbleiben, diese Elemente werden im folgenden Outputargumente eines Blocks genannt. Sie sind das Ergebnis, das nach der Ausführung eines Blocks am Stack verbleibt. Ein Block lässt sich somit beschreiben durch folgende Parameter:

- die Anzahl und Position der Inputargumente, die am Beginn des Blocks bereits am Stack für eine gültige Ausführung des Blocks liegen müssen.
- die Anzahl an Outputargumenten, dargestellt als ASTs, die am Ende der Ausführung am Stack verbleiben.
- die Menge aller AST-Elemente, die den Block ausmachen (alle umgewandelten Instruktionen eines Blocks).

Zur Erstellung der AST-Elemente wird ein speziell implementierter Stack verwendet, siehe Unterabschnitt 4.2.9.

4.2.9 AST Stack

Bei der symbolischen Ausführung der einzelnen Instruktionen wird zunächst ausschließlich in Form von AST-Elementen gearbeitet. Das steht im Gegensatz zu den aus den Operationen berechenbaren konkreten Werten. Beispielhaft sei hier die Addition der numerischen Werte 1 und 2 durch zwei *PUSH*-Instruktionen und eine *ADD*-Instruktion gezeigt, die vom Ergebnis, dem numerischen Wert 3, zu unterscheiden ist.



Dadurch liegt zu jedem Zeitpunkt eine genaue Darstellung der Ausgangswerte, Operationen und Zwischenergebnisse vor.

Bei Instruktionen, die Elemente auf den Stack legen, werden daher anstelle der Elemente deren AST-Repräsentationen abgelegt. Jedes Element auf dem Stack entspricht selbst also wiederum einem AST. Instruktionen werden durch das Abheben von AST-Argumenten vom Stack und deren Verknüpfung zu einem neuen AST abgearbeitet.

Bei der Behebung von Elementen von einem leeren Stack werden Platzhalter erstellt, die später durch Werte ersetzt werden können. Sie entsprechen den Inputargumenten des Blocks.

4.2.10 Codewalk-Builder

Mit einem Codewalk ist die Ausführung eines Programmstranges gemeint. Dieser Abschnitt beschäftigt sich mit der Frage, wie alle relevanten Zustände eines Programms errechnet werden können, um darauf aufbauend die Mustererkennung durchführen zu können.

Im Folgenden werden die verschiedenen Kernaspekte betrachtet, welche die Entwicklung des Algorithmus möglich gemacht haben. Dabei soll zuerst ein einheitliches Bild auf die verwendeten Begriffe und Konzepte gegeben werden, bevor der Algorithmus im Detail diskutiert wird.

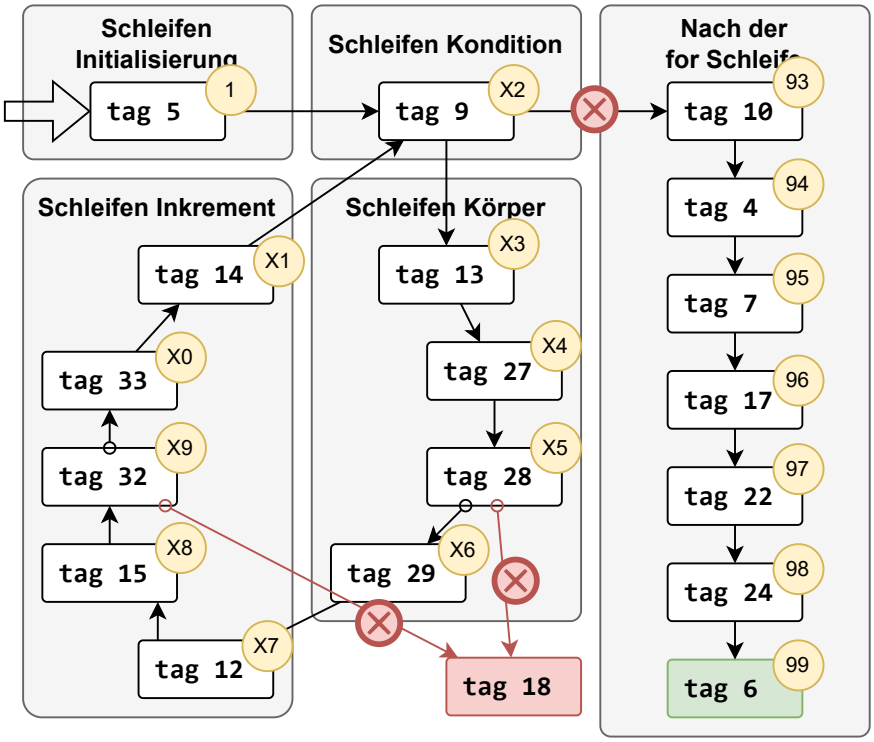


Abbildung 4.7: Zeigt einen Codewalk über den CFG des Programmes in Auflistung 4.1. Das X steht dabei für eine Zahl zwischen 0-9. tag 18 wird nie erreicht da es nie zu einem under oder overflow kommt. In diesem Fall kann bei jeder *JUMPI*-Instruktion dessen Kondition berechnet werden. Ein CFG ist für die Ausführung des Codewalks daher nicht notwendig.

Eine Programmausführung (Codewalk) entspricht einem möglichen Weg durch den CFG* [2]. Eine derartige Programmausführung beginnt immer bei einem Start-Block und endet entweder in einem Final-Block oder wird verworfen. Alle Instruktionen der verwendeten Blöcke werden symbolisch unter Verwendung des AST-Stacks ausgeführt.

Anders ausgedrückt entspricht eine Programmausführung der Sequenz aller Instruktionen in Form der AST-Repräsentation, die zwischen einem Start-Block und dem Erreichen eines Final-Blocks ausgeführt wurden. Für das Programm in Auflistung 4.1 und den CFG in Abbildung 4.4 wird so ein Codewalk schematisch in Abbildung 4.7 dargestellt.

Die Menge der Kanten des CFG* entspricht dabei nicht zwangsläufig der Menge der Kanten des zuvor ermittelten CFG. Sobald eine *JUMP*-Instruktion oder eine *JUMPI*-Instruktion erreicht wird, wird versucht, die Sprungadresse durch die symbolische Ausführung des AST-Elements zu berechnen. Sofern eine Berechnung gelingt (alle In-

[2] Beschreibt einen intermediären CFG der infolge des Codewalks immer wieder erweitert und bearbeitet wird.

puts und deren rekursiv verarbeiteten Inputs sind berechenbar), wird diese konkrete Sprungadresse für die folgende Abarbeitung herangezogen. Das geschieht unabhängig von den Daten des CFG. Ist keine eindeutige Sprungadresse ermittelbar, wird der zuvor berechnete CFG zur Bestimmung der Folgeblöcke herangezogen.

Kann kein Nachfolgeblock bestimmt werden, so werden alle Blöcke in Betracht gezogen.

Somit ist der genutzte CFG lediglich ein Hilfsmittel, um die Abarbeitung zu beschleunigen, denn in vielen Fällen kann eine Sprungadresse nicht berechnet werden.

Bei dem Anhängen eines neuen Blocks werden dessen Input-Argumente mit den Output-Argumenten der bisherigen Ausführung ersetzt. So ergibt sich eine fortlaufende symbolische Berechnung aller Werte bis zum aktuellen Ende der Kette von Blöcken.

Zur Traversierung aller möglichen Wege durch den CFG wurde eine Breitensuche implementiert. Dadurch werden iterativ immer weitere mögliche Ausführungen ermittelt. Da ein Programm aus vielen Verzweigungen (if), aber auch Kreisen (Schleifen und Funktionsaufrufen) besteht (siehe auch Abbildung 4.8), ergibt sich ein Vorteil gegenüber einer Tiefensuche, worauf im Folgenden näher eingegangen wird.

Durch die Verwendung der Breitensuche kann eine gültige Ausführung nach der anderen errechnet werden bei Priorisierung möglichst kurzer, dafür aber alternativer Routen durch den CFG. Beispielhaft sei hier die *JUMPI*-Instruktion. Sie liefert zwei mögliche Folgeblöcke, die in der Breitensuche gleichwertig und gleichzeitig behandelt werden können. Bei der Tiefensuche hingegen dringt die Ausführung tief und erschöpfend in eine Richtung vor.

Unabhängig von der Suchvariante gestaltet sich die Suche dennoch unlimitiert und damit potenziell auf unbestimmte Zeit. Das ist der Tatsache geschuldet, dass CFGs Kreise enthalten können, die nicht zwangsläufig zu einer Abbruchbedingung führen müssen. Daher muss die Abarbeitung an bestimmte Abbruchbedingungen gebunden und dadurch zur Terminierung gezwungen werden.

Es werden alle möglichen Durchläufe betrachtet, die potenziell zu einer Zustandsänderung oder aber auch zu einem Abbruch führen. Durchläufe, die nicht zu einer Zustandsänderung führen, sind zum Beispiel Ausführungen, die mit einem Revert-Block enden oder während des Durchlaufs keine Zustandsänderungen bewirkt haben. Solche Durchläufe zu betrachten, erlaubt es, sie später ebenfalls zu klassifizieren.

Im PoC implementierte Abbruchbedingungen sind:

- Eine vordefinierte Tiefe wurde erreicht (Maximalanzahl an traversierter Knoten innerhalb eines Durchlaufs wurde erreicht).
- Aufgrund eines Timeouts wird die weitere Bearbeitung abgebrochen.
- Erreichen eines Revert-Blocks
- Erreichen eines Final-Blocks
- Erreichen eines ungültigen Zustands: Beispielsweise verarbeitet die *RETURN*-Instruktion zwei Elemente vom Stack. Ist die Anzahl der Stack-Elemente nun

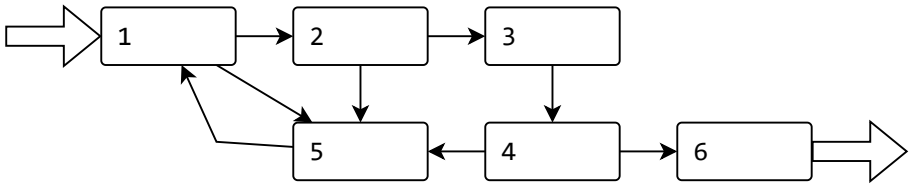


Abbildung 4.8: Zeigt einen komplexen CFG. Durch diesen Graphen sind triviale Wege wie beispielsweise 1,2,3,4,6 denkbar. Ein komplexerer Weg wäre: 1,2,5,1,2,5,1,5,1,2,3,4,6

kleiner als 2, so kann es sich nicht um eine gültige Ausführung handeln. In diesem Fall wird die Ausführung verworfen und nicht weiter betrachtet.

- Unauflösbare JUMP-Adresse am Ende des Blocks: Findet sich am Ende eines Blocks eine berechenbare *JUMP*- oder *JUMPI*-Instruktion, so wird die Sprungadresse durch die Berechnung ermittelt. Führt diese Sprungadresse nun zu einer *JUMPDEST*-Instruktion, so wird an dieser Stelle die Ausführung fortgesetzt. Führt die Adresse hingegen nicht zu einer *JUMPDEST*-Instruktion, so wird die Ausführung unterbrochen und die aktuelle Programmausführung verworfen. Eine berechenbare Sprungadresse ist eine Sprungadresse, die keine Variablen enthält (*calleraddress*, *blocknumber*, ...) und dessen Wert auf eine Position innerhalb des Programms zeigt, die eine *JUMPDEST*-Instruktion hält.
- Erreichen eines ungültigen Zustands: Es wurde versucht, auf das Element mit dem Index -1 im AST Stack zuzugreifen.

Im Gegensatz zu Abbruchbedingungen stehen beispielsweise *JUMPI*-Instruktionen, die zu zwei verschiedenen Ausführungssträngen führen (siehe auch Abbildung 4.8). Dabei bleibt der Satz der bisher traversierten Blöcke ident, allerdings haben beide Stränge jeweils eine unterschiedliche Fortsetzung. In diesem Fall werden beide möglichen Ausführungen einer FIFO Queue hinzugefügt.

4.2.11 AST Filter

Der AST-Filter reduziert die Anzahl der AST-Elemente, die dem nächsten Modul übergeben werden. Beispielsweise haben *PUSH*-Instruktionen oder *DUP*-Instruktionen keine Relevanz im generierten XML-Dokument. In zukünftigen Versionen könnten solche primitiven Operationen aber Aufschluss über den verwendeten Compiler geben und gegebenenfalls inkludiert werden.

Eine *PUSH*-Instruktion für sich genommen hat noch keine Auswirkung und ändert sich über den Verlauf einer Ausführung nicht. Das gilt gleichermaßen für arithmetische Operation. Eine Ausnahme stellt beispielsweise die *SHA3*-Instruktion dar. Sie liest aus dem zu diesem Zustand geltenden Speicherbereich. Daher muss eine solche Instruktion mit dem jeweiligen Zeitpunkt im XML erfasst werden.

Anderenfalls würde bei dem folgenden Beispiel nicht mehr ersichtlich sein, dass es zu einer Änderung kam: MSTORE(0, x) SHA3(0) MSTORE(0, y) SHA3(0). Die zwei

SHA3-Operationen wären nicht direkt voneinander unterscheidbar.

Relevant für den Output sind alle AST-Elemente,

- die in den Speicher oder Memory schreiben,
- die aus dem Speicher oder Memory lesen,
- die die Ausführung eines Programms beenden,
- die einer *create*- oder *CREATE2*-Instruktion entsprechen,
- die einer *jump*- oder *JUMPI*-Instruktion entsprechen,
- die einen anderen SCs aufrufen.

4.2.12 XML-Document-Builder

Der XML-Document-Builder transformiert die zuvor gefilterten AST-Operationen in deren XML-Repräsentation. Dabei handelt es sich um eine rekursive Übersetzung, in der jeder AST vollständig in sein XML-Äquivalent transformiert wird. Die Struktur enthält neben dem OPC-Namen andere Informationen, wie eine eindeutige ID, die verschiedenen Positionen und alle Input-Argumente. Damit ist sichergestellt, dass nachfolgende XQuery-Abfragen eine umfassende Datenbasis als Grundlage haben. Sie ermöglicht Abfragen über mehrere semantisch verbundene AST-Elemente.

Weiters wird dort, wo es möglich ist, der berechenbare Wert zusätzlich als Attribut im XML-Dokument angefügt.

Zusammenfassend lassen sich also folgende Anforderungen definieren:

- Jede Operation und deren Inputs sollen bis in die Tiefe dargestellt werden.
- Die Struktur eines XML-Knotens repräsentiert ein AST-Element.
- Zwischenergebnisse werden, sofern sie berechenbar sind, im XML aufgelöst.

In Abbildung 4.9 wird der AST dargestellt, der dem XML-Dokument in Auflistung 4.4 zugrunde liegt.

Die so erzeugten XML-Dokumente können somit potenziell in einer Datenbank hinterlegt und für spätere Klassifikationen genutzt werden.

4.2.13 XDM-Document-Builder

Hier wird die lineare textuelle Repräsentation des XML-Dokument in eine interne XQuery fähige Darstellung konvertiert. Datentechnisch ändert sich hierdurch nichts.

4.2.14 X-Query Evaluation

Der letzte Schritt prüft die angegebenen XQuery-Abfragen gegen die zuvor generierten XML-Dokumente. Dabei werden alle positiven Übereinstimmungen gesammelt.

Als XQuery-Engine kommt SaxonCS der Fa. Saxonica zum Einsatz, die ihr Tool dankenswerterweise zur Verfügung gestellt haben. Damit sind XQuery-Abfragen im Standard 3.1 möglich.

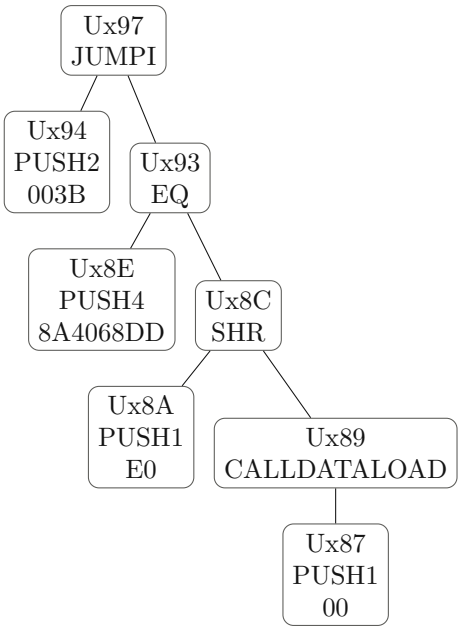


Abbildung 4.9: Stellt einen AST dar, der in dieser Form typischerweise im Function-Dispatcher eines Solidity SC zu finden ist, der mit solc als Compiler kompiliert wurde.

```
<term op-name="JUMPI" op-code="87" block-pos="10" program-pos="2A"
    global-pos="2A" id="Ux97">
  <term val="59" op-name="PUSH2" op-payload="003B" op-code="97"
      block-pos="0D" program-pos="27" global-pos="27" id="Ux94"/>
  <term op-name="EQ" op-code="20" block-pos="0C" program-pos="26"
      global-pos="26" id="Ux93">
    <term val="2319476957" op-name="PUSH4" op-payload="8A4068DD"
        op-code="99" block-pos="07" program-pos="21"
        global-pos="21" id="Ux8E"/>
    <term op-name="SHR" op-code="28" block-pos="05"
        program-pos="1F" global-pos="1F" id="Ux8C">
      <term val="224" op-name="PUSH1" op-payload="E0" op-code="96"
          block-pos="03" program-pos="1D" global-pos="1D"
          id="Ux8A"/>
      <term op-name="CALLDATALOAD" op-code="53" block-pos="02"
          program-pos="1C" global-pos="1C" id="Ux89">
        <term val="0" op-name="PUSH1" op-payload="00"
            op-code="96" block-pos="00" program-pos="1A"
            global-pos="1A" id="Ux87"/>
      </term>
    </term>
  </term>
</term>
```

Auflistung 4.4: Zeigt den in Abbildung 4.9 dargestellten AST nach XML konvertiert.

```
modifier only_owner() {
  require(msg.sender == owner);
  _;
}
```

Auflistung 4.5: Zeigt einen typischen Code, der für die Implementierung des Eigentümerschaft-SC-Muster herangezogen wird.

4.3 Mustererkennung und Musterdefinition in XQuery

4.3.1 Erkennung der Eigentümerschaft

Als Basis wird der Code in Auflistung 3.1 herangezogen. Er entspricht in den wesentlichen Teilen dem Code, der sich auch in [Opea] finden lässt. Es ist naheliegend, dass genau dieser Code in vielen weiteren SCs Anwendung findet. Eine genauere Erklärung der Anwendungsgebiete findet sich in Unterabschnitt 3.1.1.

Nach dem Kompilieren des Codes in Auflistung 3.1 erhält man die Ausgangsdaten, die zur Definition von Mustern im EVM-BC notwendig sind. Zuerst werden jene Sequenzen verworfen, die nicht mit dem SC-Muster in Verbindung stehen. Darunter sind der Function-Dispatcher, der Deployment-Code und die Beispielfunktionen.

Anschließend werden Sequenzen verworfen, die nicht ausschlaggebend für das SC-Muster sind, dabei handelt es sich um Zusatzfunktionen, die mit dem SC-Muster in Verbindung stehen, aber nicht notwendigerweise vorhanden sein müssen.

Zuletzt bleibt lediglich der EVM-BC übrig, der den Modifier in Auflistung 4.5 implementiert.

Die konkrete Sequenz an Instruktionen ist in Auflistung 4.6 aufgeführt. Diese Sequenz lässt sich nun zur einfacheren Untersuchung als AST darstellen. Dazu werden die einzelnen OPCs statisch ausgeführt (also am Stack hinterlegt und abgehoben) und so miteinander verknüpft. Der erhaltene AST ist in Abbildung 4.10 dargestellt.

Der nächste Schritt ist die Definition eines geeigneten EVM-BC-Muster. Es sollte möglichst spezifisch für das gewählte SC-Muster sein und möglichst unspezifisch für die Art der Implementierung des SC-Muster. Dabei führt eine spezifischere Definition zu weniger falsch-positiven Ergebnissen bei gleichzeitig sinkender Sensitivität. Umgekehrt führt eine unspezifischere Definition zu einer höheren Falsch-Positivrate bei steigender Sensitivität.

Bei der Wahl und Definition des EVM-BC-Muster sollte auf grundlegende Eigenschaften des SC-Muster eingegangen werden. In diesem Fall wird auf die Tatsachen eingegangen, dass in einem require die msg.sender-Adresse mit einer Speichervariable verglichen wird. Solidity-require Statements werden in *JUMPI*-Instruktionen umgewandelt, so kommt man zusammenfassend auf folgende Eigenschaften:

- Die Kondition der *JUMPI*-Instruktion muss eine *EQ*-Instruktion sein.
- Eine *CALLER*-Instruktion muss als ein Argument der *EQ*-Instruktion vorkommen.

```
1  PUSH 0;
2  DUP1;
3  SLOAD; // load owner from storage index 0
4
5  SWAP1;
6  PUSH 100;
7  EXP;
8  SWAP1;
9  DIV;
10
11 PUSH FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // cast owner to address
12 AND;
13 PUSH FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
14 AND;
15
16
17 CALLER; // load msg.sender and cast to address
18 PUSH FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
19 AND;
20
21
22 EQ; // compare msg.sender == owner
23 PUSH [tag] 7; // continuation address
24
25
26 JUMPI; // if msg.sender == owner resume @ continuation address
27
28
29 PUSH 0;
30 DUP1;
31 REVERT; // else revert
```

Auflistung 4.6: Das in 4.5 dargestellte kompilierte Resultat. Zeigt die Sequenz, die für den Vergleich `msg.sender == owner` erstellt wird.

- Eine *SLOAD*-Instruktion muss als Argument der *EQ*-Instruktion vorkommen.
- Das Argument der *SLOAD*-Instruktion muss eine *PUSH*-Instruktion sein.

Als nächster Schritt wird der AST in Abbildung 4.10 in ein XML-Dokument umgeformt. Das Ergebnis ist in Auflistung 4.7 angeführt. Anhand des transformierten AST können die oben genannten Eigenschaften durch eine XQuery-Abfrage abgefragt werden.

Die in Auflistung 4.8 angeführte XQuery-Abfrage sucht nach einer *JUMPI*-Instruktion, die als zweiten Subterm eine *EQ*-Instruktion hat. Diese *EQ*-Instruktion muss nun selbst wiederum in der Liste der Kindterme[3] eine *CALLER*-Instruktion und eine *SLOAD*-Instruktion haben. Die *SLOAD*-Instruktion muss als Zieladresse eine statische Adresse aufweisen, gegeben durch eine *PUSH*-Instruktion.

[3]Hier sind die Kinder der *EQ*-Instruktion und rekursive alle darunter liegenden Kinder gemeint.

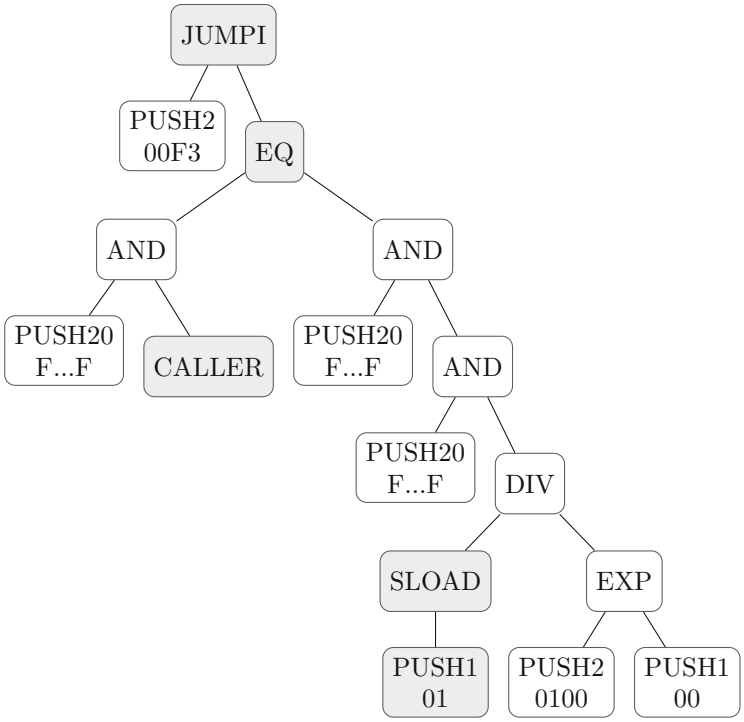


Abbildung 4.10: Stellt die in Auflistung 4.6 angeführte Sequenz als AST dar. Dabei sind die für dieses EVM-BC-Muster entscheidenden Knoten grau eingefärbt.

```
<term op-name="JUMPI">
   <term op-name="PUSH2"/>
   <term op-name="EQ">
    <term op-name="AND">
     <term op-name="PUSH20"/>
     <term op-name="CALLER"/>
    </term>
    <term op-name="AND">
     <term op-name="PUSH20"/>
     <term op-name="AND">
      <term op-name="PUSH20"/>
      <term op-name="DIV">
         <term op-name="SLOAD">
          <term op-name="PUSH1"/>
         </term>
         <term op-name="EXP">
          <term op-name="PUSH2"/>
          <term op-name="PUSH1"/>
         </term>
       </term>
      </term>
     </term>
    </term>
</term>
```

Auflistung 4.7: Ein in XML umgewandelter AST, der markant für das Eigentümer-EVM-BC-Muster ist (siehe 4.5).

```
//term
  [@op-name="JUMPI"]
  [./term[2]
    [@op-name="EQ"]
    [.//term[@op-name="CALLER"]]
    [.//term[@op-name="SLOAD"][./term[1][contains(@op-name, "PUSH")]]]
  ]
```

Auflistung 4.8: Zeigt eine Abfrage, die SC erkennt, die das Eigentümer-EVM-BC-Muster implementiert haben (siehe Auflistung 4.5)

4.3.2 Rollenbasierte Autorisierung

Als Basis wird der Code in Auflistung 3.2 herangezogen. Er entspricht in den wesentlichen Teilen dem Code, der sich auch in [Opea] finden lässt. Es ist naheliegend, dass genau dieser Code in vielen weiteren SCs Anwendung findet. Eine genauere Erklärung der Anwendungsgebiete findet sich in Unterabschnitt 3.1.2.

Wie zuvor wird zuerst der entsprechende EVM-BC herangezogen und auf die wesentlichen Inhalte reduziert. Am Ende bleibt lediglich der EVM-BC für den Code in Auflistung 4.9 übrig.

```
modifier only_role(bytes32 role) {
    if (!_roles[role].members[msg.sender])
        revert();
    _;
}
```

Auflistung 4.9: Zeigt das typische Muster für rollenbasierte Autorisierung.

Bei näherer Betrachtung des AST in Abbildung 4.11 fällt auf, dass die *CALLER*-Instruktion fehlt. Da im entsprechenden SC-Muster allerdings der Solidity-Code `msg.sender` vorkommt und dieser per Definition von Solidity in die *CALLER*-Instruktion umgewandelt wird, kann es sich bei der angegeben *JUMPI*-Instruktion nicht um den vollständigen Code handeln.

Die tatsächlich involvierte EVM-BC Sequenz fällt nun komplexer aus und ein einzelner AST reicht für die Ermittlung eines spezifischen Musters nicht mehr aus.

Tatsächlich erstreckt sich die Adressierung des entsprechenden Speicherbereichs der *SLOAD*-Instruktion über drei verschiedene *MSTORE*-Instruktionen. Das ergibt sich aus der Art und Weise, wie der Solidity-Compiler seine Speicher-Adressierung implementiert.

In unserem konkreten Fall handelt es sich um ein `mapping(bytes32, struct)` an der Position `0` im SC.

Um für die Adresse `msg.sender` die Rolle "`role1`" festzustellen, wird die Speicheradresse folgendermaßen ermittelt [solb]:

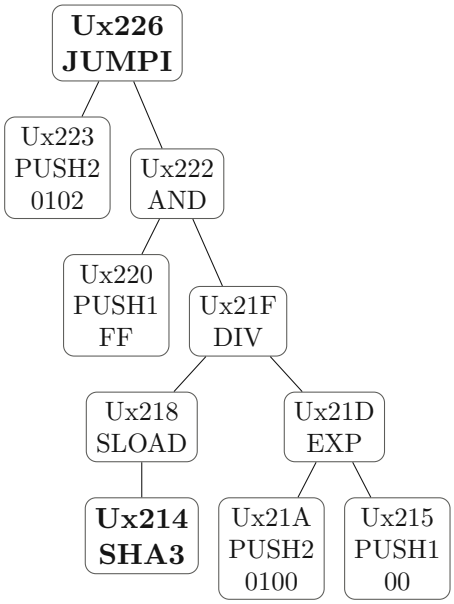


Abbildung 4.11: Zeigt die AST-Repräsentation der `require` Bedingung, des rollenbasierten Autorisierung-SC-Muster. Auffällig ist das Fehlen der *CALLER*-Instruktion.

- Konkatenieren der `bytes32("role1")` mit `uint256(0)` (aufgrund der Position der Speichervariable an der Position `0`)
  `byte32("role1"). uint256(0)`
- Hashen der konkatenierten Daten siehe auch Abbildung 4.12:
  `keccak256(byte32("role1"). uint256(0))`
- Addition von `0x0` da sich das Feld members innerhalb des Structs an Position `0` befindet:
  `keccak256(byte32("role1"). uint256(0))+ 0`
- Konkatenieren der msg.sender Adresse mit dem zuvor erhaltenen Hash:
  `uint256(msg.sender). (keccak256(byte32("role1"). uint256(0))+ 0)`
- Hashen der konkatenierten Daten:
  `keccak256(uint256(msg.sender). (keccak256(byte32("role1"). uint256(0))+ 0) )`

In Auflistung 4.10 ist das zugehörige XML-Dokument zu den ASTs in Abbildung 4.12 und Abbildung 4.13 angeführt.

Die in Auflistung 4.11 angeführte XQuery-Abfrage sucht nach einer *JUMPI*-Instruktion, die als zweiten Subterm eine *AND*-Instruktion aufweist. Die *AND*-Instruktion kommt dabei von der auftretenden Speicherdekompaktierung und dem Cast auf einen Boolean, da der ausgelesene Wert im Speicher auf mehrere Variablen zurückführbar sein könnte. Die *AND*-Instruktion muss einen Term beinhalten, dessen Id in einer *MSTORE*-Instruktion der vorangegangen Terme vorkommt. Diese *MSTORE*-Instruktion muss das Ergebnis der *CALLER*-Instruktion im volatilen Speicher ablegen.

```
<term op-name="MSTORE" id="Ux29C">
  <term val="0" op-name="PUSH1" op-payload="00" id="Ux26C"/>
  <term op-name="AND" id="Ux29A">
    <term op-name="PUSH20" op-payload="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" id="
        Ux285"/>
    <term op-name="AND" id="Ux284">
      <term op-name="PUSH20" op-payload="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" id
          ="Ux26F"/>
      <term op-name="CALLER" id="Ux26E"/>
    </term>
  </term>
</term>

<term op-name="MSTORE" id="Ux2A2">
  <term val="32" op-name="ADD" id="Ux29F">
    <term val="32" op-name="PUSH1" op-payload="20" id="Ux29D"/>
    <term val="0" op-name="PUSH1" op-payload="00" id="Ux26C"/>
  </term>
  <term op-name="ADD" id="Ux26B">
    <term val="0" op-name="PUSH1" op-payload="00" id="Ux269"/>
    <term op-name="SHA3" id="Ux268">
      <term val="0" op-name="PUSH1" op-payload="00" id="Ux266"/>
      <term val="64" op-name="ADD" id="Ux265">
        <term val="32" op-name="PUSH1" op-payload="20" id="Ux263"/>
        <term val="32" op-name="ADD" id="Ux25F">
          <term val="32" op-name="PUSH1" op-payload="20" id="Ux25D"/>
          <term val="0" op-name="PUSH1" op-payload="00" id="Ux257"/>
        </term>
      </term>
    </term>
  </term>
</term>

<term op-name="JUMPI" id="Ux2BA">
  <term val="406" op-name="PUSH2" op-payload="0196" id="Ux2B7"/>
  <term op-name="AND" id="Ux2B6">
    <term val="255" op-name="PUSH1" op-payload="FF" id="Ux2B4"/>
    <term op-name="DIV" id="Ux2B3">
      <term op-name="SLOAD" id="Ux2AC">
        <term op-name="SHA3" id="Ux2A8">
          <term val="0" op-name="PUSH1" op-payload="00" id="Ux2A6"/>
          <term val="64" op-name="ADD" id="Ux2A5">
            <term val="32" op-name="PUSH1" op-payload="20" id="Ux2A3"/>
            <term val="32" op-name="ADD" id="Ux29F">
              <term val="32" op-name="PUSH1" op-payload="20" id="Ux29D"/>
              <term val="0" op-name="PUSH1" op-payload="00" id="Ux26C"/>
            </term>
          </term>
        </term>
      </term>
      <term val="1" op-name="EXP" id="Ux2B1">
        <term val="256" op-name="PUSH2" op-payload="0100" id="Ux2AE"/>
        <term val="0" op-name="PUSH1" op-payload="00" id="Ux2A9"/>
      </term>
    </term>
  </term>
</term>
```

Auflistung 4.10: Zeigt für das rollenbasierte Autorisierungs-SC-Muster den relevanten transformierten XML-Code zur Mustererkennung.

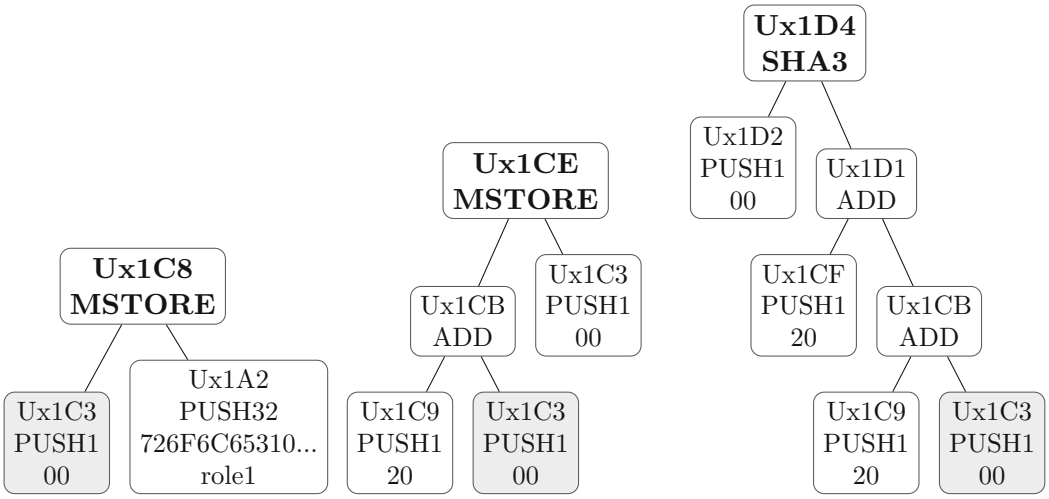


Abbildung 4.12: Zeigt den Aufbau der Formel `keccak256(byte32("role1"). uint256(0))` nach der Kompilierung in EVM-BC und der anschließenden Umwandlung in die AST Repräsentation. Drei wesentliche Instruktionen werden dargestellt. Die ersten zwei *MSTORE*-Instruktionen, die den Speicherbereich entsprechend aufbereiten, und das anschließende Hashen über den zuvor präparierten Bereich.

```
1  //term
2    [@op-name="JUMPI"]
3    [./term[2]
4      [@op-name="AND"]
5      [.//term
6        [@id = //term
7          [.//ancestor::term
8            [@op-name="MSTORE"]
9            [.//term[@op-name="CALLER"]]
10         ]/@id
11       ]
12     ]
13   ]
```

Auflistung 4.11: Zeigt eine Abfrage, die SC erkennt, die das Rollenbasierte-Authorisierungs-SC-Muster implementiert haben (siehe Auflistung 4.9).

Durch diese XQuery-Abfrage wird über Terme hinweg die Verbindung des konkreten volatilen-Speicher-Wertes mit der *JUMPI*-Instruktion aufgebaut und abgefragt.

4.3.3 Ablaufdatum

Als Basis wird der Code in Auflistung 3.3 herangezogen. Eine genauere Erklärung der Anwendungsgebiete findet sich in Unterabschnitt 3.2.1.

Wie zuvor wird zuerst der entsprechende EVM-BC herangezogen und auf die wesentlichen

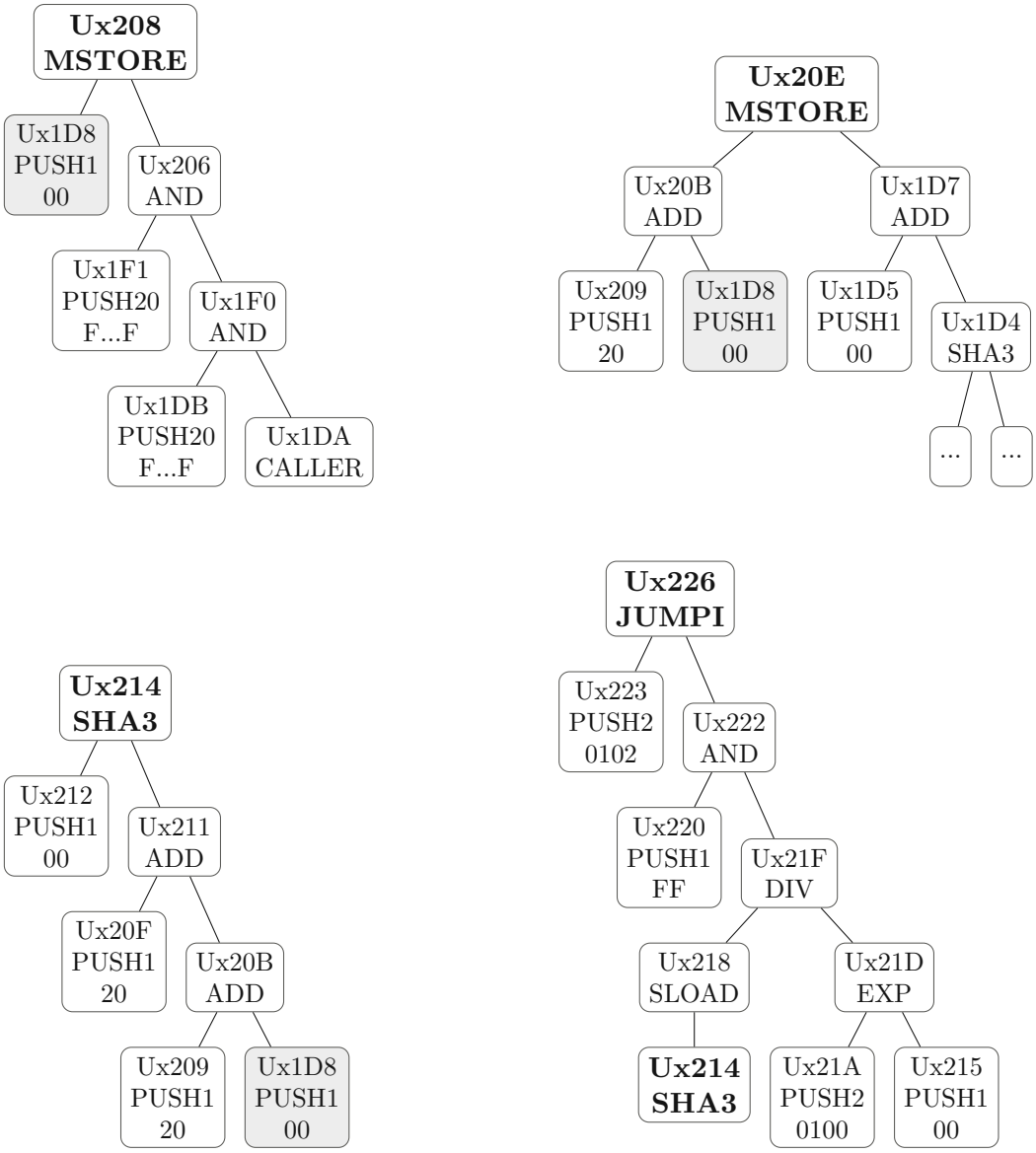


Abbildung 4.13: Zeigt den Aufbau der Formel `keccak256(uint256(msg.sender). (keccak256(byte32("role1"). uint256(0))+ 0))` nach der Kompilierung in EVM-BC und der anschließenden Umwandlung in die AST Repräsentation. Vier wesentliche Instruktionen werden dargestellt. Die ersten zwei *MSTORE*-Instruktionen, die den Speicherbereich entsprechend aufbereiten, und das anschließende Hashen über den zuvor präparierten Bereich. Und zuletzt eine *JUMPI*-Instruktion, die als der Hauptbestandteil des Musters für die rollenbasierte Autorisierung herangezogen wird.
Die grau unterlegte Instruktion verknüpft dabei das Vorkommen der *CALLER*-Instruktion und der *JUMPI*-Instruktion.

```
modifier beforeDeprecation() {
  require(block.timestamp < _deprecationTime);
  _;
}

modifier afterDeprecation() {
  require(block.timestamp > _deprecationTime);
  _;
}
```

Auflistung 4.12: Zeigt einen typischen Code, der für die Implementierung des Ablaufdatum-SC-Muster herangezogen wird.

Inhalte reduziert. Am Ende bleibt lediglich der EVM-BC für den Code in Auflistung 4.12 übrig.

An dieser Stelle bleibt dem Leser die Abfolge der OPCs erspart und wir widmen uns direkt dem entsprechenden AST, der den Modifier in Auflistung 4.12 ausmacht.

Die Detektion gestaltet sich analog zu Unterabschnitt 4.3.1. Die in Auflistung 4.12 gezeigten Implementierungen lassen sich wiederum in ASTs umwandeln, siehe Abbildung 4.14. Das entsprechende XML-Dokument findet sich in Auflistung 4.13.

```
<term op-name="JUMPI" op-code="87" block-pos="08" program-pos="82" global-pos="82" id="
    Ux141">
  <term val="135" op-name="PUSH1" op-payload="87" op-code="96" block-pos="06" program-
      pos="80" global-pos="80" id="Ux13F"/>
  <term op-name="GT" op-code="17" block-pos="05" program-pos="7F" global-pos="7F" id="
      Ux13E">
    <term op-name="TIMESTAMP" op-code="66" block-pos="04" program-pos="7E" global-pos="
        7E" id="Ux13D"/>
    <term op-name="SLOAD" op-code="84" block-pos="03" program-pos="7D" global-pos="7D"
        id="Ux13C">
      <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="01" program
          -pos="7B" global-pos="7B" id="Ux13A"/>
    </term>
  </term>
</term>
```

Auflistung 4.13: In XML umgewandelter AST, der markant für das Ablaufdatum-SC-Muster ist (siehe 4.12).

Relevant für die XQuery-Abfrage sind der Sprung, die Vergleichsoperation, das Vorkommen der *TIMESTAMP*-Instruktion und eine *SLOAD*-Instruktion, die von einem statischen Speicherbereich liest.

Anders als das Eigentümerschaft-SC-Muster ist dieses SC-Muster allerdings nicht immer in derselben Form implementiert, was die Detektion erschwert. Mit der vorgeschlagenen Abfragesprache XQuery ist es möglich, die verschiedenen Varianten zu beschreiben.

Hierzu werden verschiedene Ausprägungen des Patterns vorgestellt. Die Variationen sind in Auflistung 4.15 angeführt:

- Variationen in der Bedingung wie beispielsweise die Verwendung der Solidity-Operatoren <= und >= bzw. < und >.

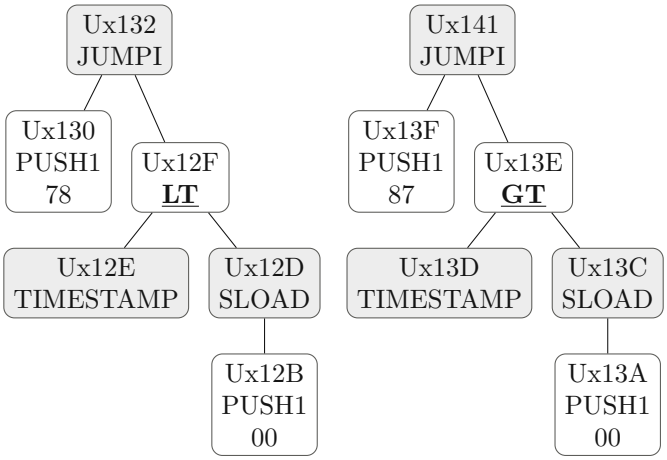


Abbildung 4.14: Stellt die in Auflistung 4.12 relevanten ASTs, der Modifier, dar. Dabei sind die für dieses EVM-BC-Muster entscheidenden Knoten grau eingefärbt.

```
1  //term
2    [@op-name="JUMPI"]
3    [./term[2]
4      [(@op-name="LT") or (@op-name="GT")]
5      [./term[@op-name="TIMESTAMP"]]
6      [./term[@op-name="SLOAD"][./term[1][contains(@op-name, "PUSH")]]]
7    ]
```

Auflistung 4.14: Zeigt eine Abfrage, die SCs matched, die das Ablaufdatum-SC-Muster implementiert haben (siehe Auflistung 4.12).

- Variation in den Argumenten der Bedingung. Denkbar sind hier Subtraktionen, Additionen, Modulo-Operationen, zusammengefasst arithmetische Operationen im Allgemeinen.

```
1  modifier beforeDeprecation() {
2    require(block.timestamp <= _deprecationTime);
3    _;
4  }
5  modifier beforeDeprecation2() {
6    require(block.timestamp < _deprecationTime + 1 days);
7    _;
8  }
```

Auflistung 4.15: Zeigt einen adaptierten Code des Ablaufdatum-SC-Muster. Er unterscheidet sich in Details vom Code in Auflistung 4.12. Im ersten Modifier ist die Vergleichsoperation von einem < auf ein <= abgeändert. Im zweiten Modifier wird zur storage Variable _deprecationTime ein weiterer Tag addiert.

In Abbildung 4.15 sind die entsprechenden ASTs grafisch dargestellt. Die XQuery-Abfrage aus Auflistung 4.14 lässt sich nun folgendermaßen anpassen:

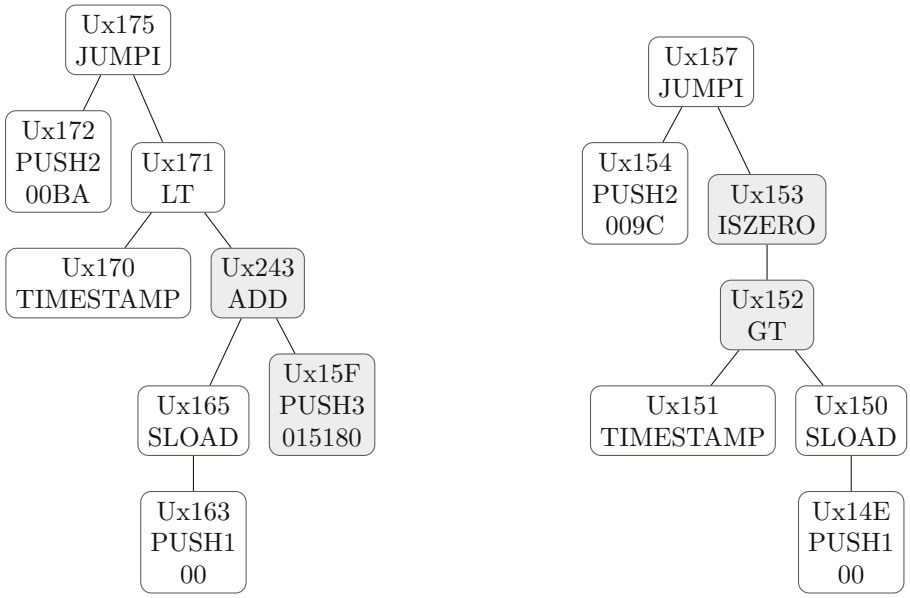


Abbildung 4.15: Im Gegensatz zu Abbildung 4.14 werden komplexere Formen der möglichen ASTs gezeigt. Sie unterscheiden sich minimal vom EVM-BC-Muster, machen aber eine relaxierte Form der Abfrage notwendig. Die linke Abbildung zeigt dabei das require Statement: `block.timestamp < _deprecationTime + 1 days`.
Die Rechte: `block.timestamp <= _deprecationTime`

```
1 //term
2   [@op-name="JUMPI"]
3   [./term[2]
4     [(@op-name="LT") or (@op-name="GT") or (@op-name="EQ") or (@op-name="
          ISZERO")]
5     [.//term[@op-name="TIMESTAMP"]]
6     [.//term[@op-name="SLOAD"][./term[1][contains(@op-name, "PUSH")]]]
7   ]
```

Auflistung 4.16: Zeigt eine Abfrage, die SCs matched, die das Ablaufdatum-SC-Muster implementiert haben (siehe Auflistung 4.12).

- Die Vergleichsoperation muss auch *EQ*-Instruktionen sowie *ISZERO*-Instruktionen erlauben.
- Anstelle des Vergleichs der direkten Kinder müssen alle Nachkommen gegenüber der *TIMESTAMP*-Instruktion und *SLOAD*-Instruktion verglichen werden.

Daraus folgt die verbesserte XQuery-Abfrage in Auflistung 4.16.

```
1 modifier stoppedInEmergency {
2   require(!isStopped);
3   _;
4 }
```

Auflistung 4.17: Zeigt einen typischen Code, der für die Implementierung des Notfall-Stop-SC-Muster herangezogen wird.

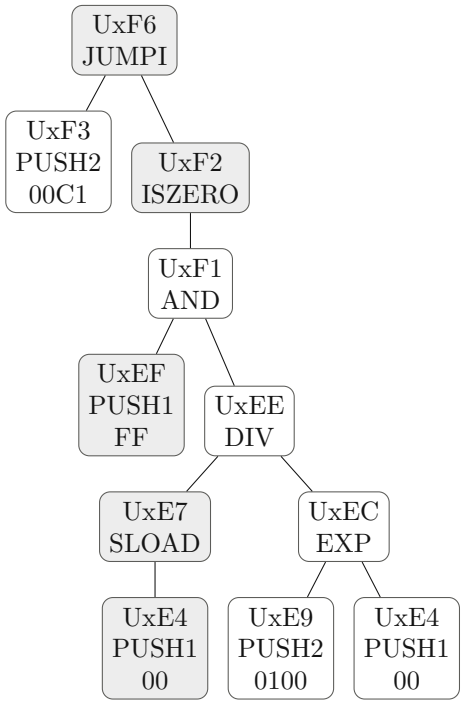


Abbildung 4.16: Stellt die, in Auflistung 4.17 angeführte, Sequenz als AST dar. Dabei sind die als für dieses EVM-BC-Muster entscheidenden Knoten grau eingefärbt.

4.3.4 Notfallstop

Als Basis wird der Code in Auflistung 3.4 herangezogen. Eine genauere Erklärung der Anwendungsgebiete findet sich in Unterabschnitt 3.2.2.

Wie zuvor wird zuerst der entsprechende EVM-BC herangezogen und auf die wesentlichen Inhalte reduziert. Am Ende bleibt lediglich der EVM-BC für den Code in Auflistung 4.17 übrig.

Auch hier gestaltet sich die Definition des EVM-BC-Muster in der Abfragesprache analog zu jener aus Unterabschnitt 4.3.1. Die in Auflistung 4.17 gezeigte Implementierung lässt sich wiederum in AST-Repräsentationen umwandeln (siehe Abbildung 4.16). Das dazugehörige XML-Dokument findet sich in Auflistung 4.18.

Die wesentlichen Merkmale des AST sind:

```
<term op-name="JUMPI" op-code="87" block-pos="13" program-pos="BC" global-pos="BC" id="
    UxF6">
<term val="193" op-name="PUSH2" op-payload="00C1" op-code="97" block-pos="10" program-
    pos="B9" global-pos="B9" id="UxF3"/>
<term op-name="ISZERO" op-code="21" block-pos="0F" program-pos="B8" global-pos="B8" id=
    "UxF2">
   <term op-name="AND" op-code="22" block-pos="0E" program-pos="B7" global-pos="B7" id=
       "UxF1">
    <term val="255" op-name="PUSH1" op-payload="FF" op-code="96" block-pos="0C" program
        -pos="B5" global-pos="B5" id="UxEF"/>
    <term op-name="DIV" op-code="4" block-pos="0B" program-pos="B4" global-pos="B4" id=
        "UxEE">
     <term op-name="SLOAD" op-code="84" block-pos="04" program-pos="AD" global-pos="AD"
         id="UxE7">
      <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="01" program
          -pos="AA" global-pos="AA" id="UxE4"/>
     </term>
     <term val="1" op-name="EXP" op-code="10" block-pos="09" program-pos="B2" global-
         pos="B2" id="UxEC">
      <term val="256" op-name="PUSH2" op-payload="0100" op-code="97" block-pos="06"
          program-pos="AF" global-pos="AF" id="UxE9"/>
      <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="01" program
          -pos="AA" global-pos="AA" id="UxE4"/>
     </term>
    </term>
   </term>
</term>
</term>
<term op-name="JUMP" op-code="86" block-pos="01" program-pos="C2" global-pos="C2" id="
    UxFC">
<term val="99" op-name="PUSH2" op-payload="0063" op-code="97" block-pos="01" program-
    pos="5C" global-pos="5C" id="Ux96"/>
</term>
<term op-name="STOP" op-code="0" block-pos="01" program-pos="64" global-pos="64" id="
    Ux9E"/>
```

Auflistung 4.18: Ein in XML umgewandelter AST, der markant für das Notfall-Stop-EVM-BC-Muster ist (siehe 4.17).

- Das Auslesen einer einfachen Speichervariable sowie das Dekompaktieren und der Cast auf einen Boolean. Nicht betrachtet wird hierbei die *DIV*-Instruktion. Sie übernimmt im Falle von angewendeter Speicherkompaktierung das Verschieben der relevanten Bits an den Beginn des Wortes. Die *AND*-Instruktion übernimmt das auf 0 setzen aller nicht relevanten Bits.
- Das Vorkommnis einer *ISZERO*-Instruktion ist dabei nicht zwangsläufig notwendig.
- Das Terminieren mit einer *STOP*-Instruktion, sofern die require Bedingung in Auflistung 4.17 nicht erfüllt ist.

4.3.5 Aktualisierbarkeit

Als Basis wird der Code in Auflistung 3.5 herangezogen. Eine genauere Erklärung der Anwendungsgebiete findet sich in Abschnitt 3.3.

Wie zuvor wird zuerst der entsprechende EVM-BC herangezogen und auf die wesentlichen Inhalte reduziert. Am Ende bleibt lediglich der EVM-BC für den Code in Auflistung 4.20 übrig.

Die Erkennung gestaltet sich analog zur Unterabschnitt 4.3.1. Die in Auflistung 4.20

```
1  //term
2    [@op-name="JUMPI"]
3    [./term[2]
4      [@op-name="ISZERO"]
5      [./term[1]
6        [@op-name="AND"]
7        [./term[1][@op-name="PUSH1" and @op-payload="FF"]]
8      ]
9      [./term[1]/term[2]//term[@op-name="SLOAD"]
10       [./term[1][contains(@op-name, "PUSH")]]
11     ]
12   ]
```

Auflistung 4.19: Zeigt eine Abfrage, die SC erkennt, die das Notfallstop-EVM-BC-Muster implementiert haben (siehe Auflistung 4.17).

```
1 fallback()  external {
2   (bool success, bytes memory data) = destination.delegatecall(msg.data);
3   require (success);
4 }
```

Auflistung 4.20: Zeigt einen typischen SC-Code, der für die Implementierung des ProxyDelegate-SC-Muster herangezogen wird.

gezeigte Implementierung lässt sich wiederum in seine AST-Repräsentation umwandeln, siehe Abbildung 4.17. Das entsprechende XML-Dokument findet sich in Auflistung 4.21.

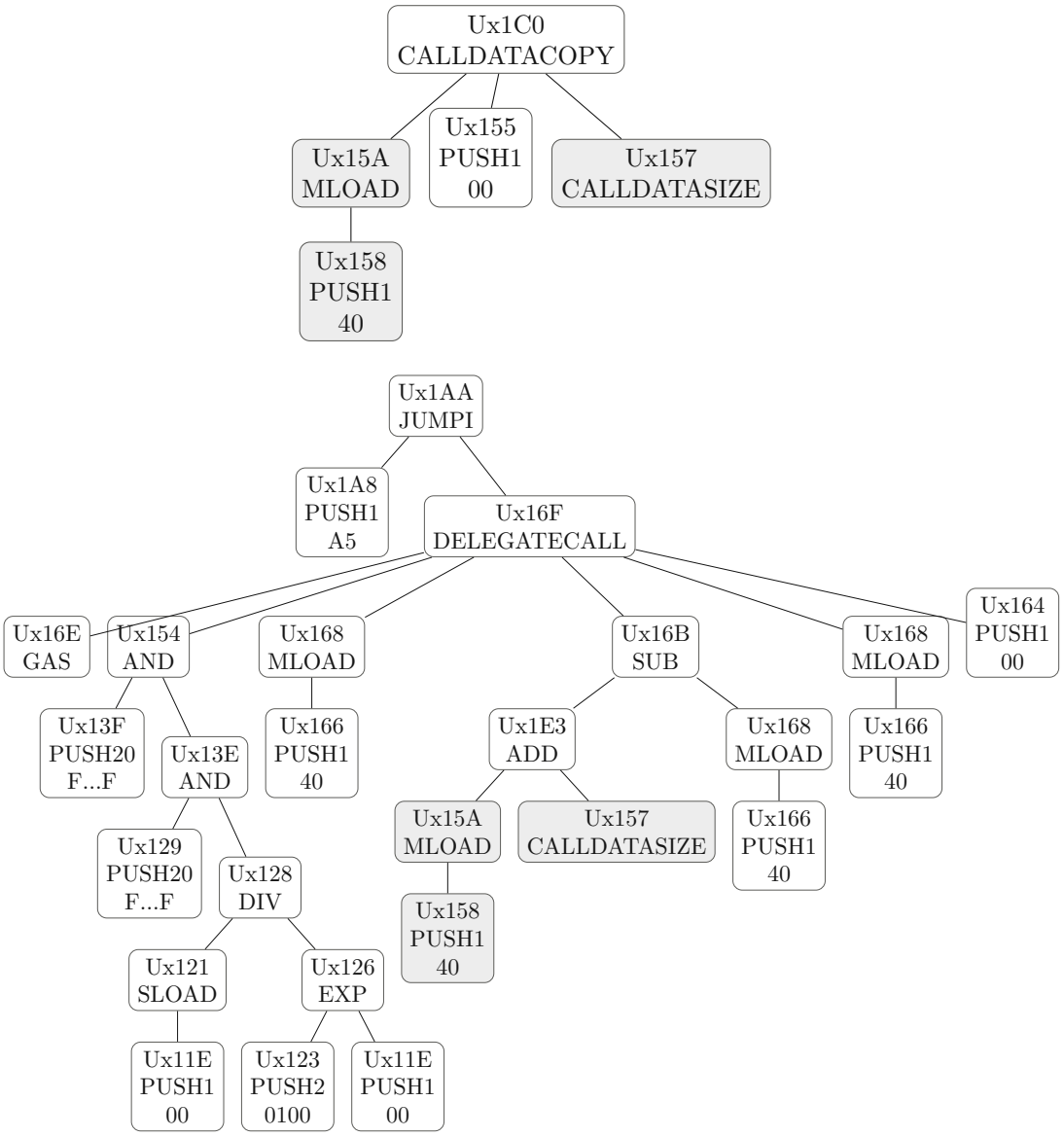


Abbildung 4.17: Stellt den relevanten AST des Codes in Auflistung 4.20 dar. Hervorgehoben sind die Instruktionen, die eine *CALLDATACOPY*-Instruktion mit der darauf folgenden *DELEGATECALL*-Instruktion verbindet. Diese Verbindung ist bedeutsam in der Erkennung des Proxy EVM-BC-Muster. Die XQuery-Abfrage zur Erkennung des EVM-BC-Muster findet sich in Auflistung 4.22

```
<terms>
  <term op-name="CALLDATACOPY" op-code="55" block-pos="04" program-pos="B6" global-pos="B6" id="Ux1C0">
    <term op-name="MLOAD" op-code="81" block-pos="41" program-pos="50" global-pos="50" id="Ux15A">
      <term val="64" op-name="PUSH1" op-payload="40" op-code="96" block-pos="3F" program-pos="4E" global-pos="4E" id="Ux158"/>
    </term>
    <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="3C" program-pos="4B" global-pos="4B" id="Ux155"/>
    <term op-name="CALLDATASIZE" op-code="54" block-pos="3E" program-pos="4D" global-pos="4D" id="Ux157"/>
  </term>
  <term op-name="MSTORE" op-code="82" block-pos="0A" program-pos="BC" global-pos="BC" id="Ux1C6">
    <term op-name="ADD" op-code="1" block-pos="09" program-pos="BB" global-pos="BB" id="Ux1C5">
      <term op-name="MLOAD" op-code="81" block-pos="41" program-pos="50" global-pos="50" id="Ux15A">
        <term val="64" op-name="PUSH1" op-payload="40" op-code="96" block-pos="3F" program-pos="4E" global-pos="4E" id="Ux158"/>
      </term>
      <term op-name="CALLDATASIZE" op-code="54" block-pos="3E" program-pos="4D" global-pos="4D" id="Ux157"/>
    </term>
    <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="05" program-pos="B7" global-pos="B7" id="Ux1C1"/>
  </term>
  <term op-name="JUMPI" op-code="87" block-pos="09" program-pos="A0" global-pos="A0" id="Ux1AA">
    <term val="165" op-name="PUSH1" op-payload="A5" op-code="96" block-pos="07" program-pos="9E" global-pos="9E" id="Ux1A8"/>
    <term op-name="DELEGATECALL" op-code="244" block-pos="0C" program-pos="65" global-pos="65" id="Ux16F">
      <term op-name="GAS" op-code="90" block-pos="0B" program-pos="64" global-pos="64" id="Ux16E"/>
      <term op-name="AND" op-code="22" block-pos="3B" program-pos="4A" global-pos="4A" id="Ux154">
        <term val="1461501637330902918203684832716283019655932542975" op-name="PUSH20" op-payload="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" op-code="115" block-pos="26" program-pos="35" global-pos="35" id="Ux13F"/>
        <term op-name="AND" op-code="22" block-pos="25" program-pos="34" global-pos="34" id="Ux13E">
          <term val="1461501637330902918203684832716283019655932542975" op-name="PUSH20" op-payload="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" op-code="115" block-pos="10" program-pos="1F" global-pos="1F" id="Ux129"/>
          <term op-name="DIV" op-code="4" block-pos="0F" program-pos="1E" global-pos="1E" id="Ux128">
            <term op-name="SLOAD" op-code="84" block-pos="08" program-pos="17" global-pos="17" id="Ux121">
              <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="05" program-pos="14" global-pos="14" id="Ux11E"/>
            </term>
            <term val="1" op-name="EXP" op-code="10" block-pos="0D" program-pos="1C" global-pos="1C" id="Ux126">
              <term val="256" op-name="PUSH2" op-payload="0100" op-code="97" block-pos="0A" program-pos="19" global-pos="19" id="Ux123"/>
              <term val="0" op-name="PUSH1" op-payload="00" op-code="96" block-pos="05" program-pos="14" global-pos="14" id="Ux11E"/>
            </term>
          </term>
          </term>
        </term>
        <term op-name="MLOAD" op-code="81" block-pos="05" program-pos="5E" global-pos="5E" id="Ux168">
          <term val="64" op-name="PUSH1" op-payload="40" op-code="96" block-pos="03" program-pos="5C" global-pos="5C" id="Ux166"/>
        </term>
        <term op-name="SUB" op-code="3" block-pos="08" program-pos="61" global-pos="61" id="Ux16B">
          <term op-name="ADD" op-code="1" block-pos="03" program-pos="D9" global-pos="D9" id="Ux1E3">
            <term op-name="MLOAD" op-code="81" block-pos="41" program-pos="50" global-pos
```

```
                =\"50\" id=\"Ux15A\">
43              <term val=\"64\" op-name=\"PUSH1\" op-payload=\"40\" op-code=\"96\" block-pos=\"3F\"
                    program-pos=\"4E\" global-pos=\"4E\" id=\"Ux158\"/>
44            </term>
45            <term op-name=\"CALLDATASIZE\" op-code=\"54\" block-pos=\"3E\" program-pos=\"4D\"
                  global-pos=\"4D\" id=\"Ux157\"/>
46            </term>
47            <term op-name=\"MLOAD\" op-code=\"81\" block-pos=\"05\" program-pos=\"5E\" global-pos
                  =\"5E\" id=\"Ux168\">
48            <term val=\"64\" op-name=\"PUSH1\" op-payload=\"40\" op-code=\"96\" block-pos=\"03\"
                  program-pos=\"5C\" global-pos=\"5C\" id=\"Ux166\"/>
49            </term>
50          </term>
51          <term op-name=\"MLOAD\" op-code=\"81\" block-pos=\"05\" program-pos=\"5E\" global-pos=\"
                5E\" id=\"Ux168\">
52            <term val=\"64\" op-name=\"PUSH1\" op-payload=\"40\" op-code=\"96\" block-pos=\"03\"
                  program-pos=\"5C\" global-pos=\"5C\" id=\"Ux166\"/>
53          </term>
54          <term val=\"0\" op-name=\"PUSH1\" op-payload=\"00\" op-code=\"96\" block-pos=\"01\"
                program-pos=\"5A\" global-pos=\"5A\" id=\"Ux164\"/>
55        </term>
56      </term>
57    </terms>
```

Auflistung 4.21: Ein in XML umgewandelter AST, der markant für das Proxy-EVM-BC-Muster ist (siehe 4.20).

Die wesentlichen Merkmale des AST für die EVM-BC-Muster Erkennung sind:

- das Vorkommen einer *DELEGATECALL*-Instruktion,
- das Übergeben der Zieladresse für die *DELEGATECALL*-Instruktion aus dem Speicher,
- der Adressparameter der *DELEGATECALL*-Instruktion muss von einer Speicheradresse geladen werden (mittels *SLOAD*-Instruktion und einer einfachen *PUSH*-Instruktion),
- der Input der *DELEGATECALL*-Instruktion, also die Stackparameter 2 und 3, müssen den Output einer zuvor erfolgten *CALLDATACOPY*-Instruktion lesen.

Die Verknüpfung der zuvor erfolgten *CALLDATACOPY*-Instruktion kann über die Id erfolgen. Dabei wird die *MLOAD*-Instruktion mit der ID Ux15A, die in der *CALLDATACOPY*-Instruktion und in der *DELEGATECALL*-Instruktion aufscheint, über die ID in Verbindung gebracht.

Ein einfacheres Muster könnte die Tatsache nützen, dass die Inputlänge der *CALLDATACOPY*-Instruktion dem Ergebnis der *CALLDATASIZE*-Instruktion entsprechen muss. Im Grunde genommen scheint es ausreichend zu sein, darauf zu prüfen. Nichtsdestotrotz möchte ich die komplexere Abfrage wählen, um dessen Umsetzbarkeit in der vorgeschlagenen Abfragesprache aufzuzeigen.

```
1 //term
2   [@op-name="DELEGATECALL"]
3   [./term[2]//term[@op-name="SLOAD"][./term[1][contains(@op-name, "PUSH")]]]
4   [./term[3]/@id = ./term[4]/term[2]/@id]
5   [./term[4][@op-name="SUB"]/term[1][@op-name="ADD"]/term[1][@op-name="MLOAD"
        ]/@id = //term[@op-name="CALLDATACOPY"]/term[1]/@id]
6   [./term[4][@op-name="SUB"]/term[1][@op-name="ADD"]/term[2][@op-name="
        CALLDATASIZE"]/@id = //term[@op-name="CALLDATACOPY"]/term[3]/@id]
```

Auflistung 4.22: Zeigt eine Abfrage, die all jene SC erkennt, die das ProxyDelegate-EVM-BC-Muster implementiert haben (siehe Auflistung 4.20).

KAPITEL 5

Evaluierung

5.1 Robustheit

In diesem Kapitel werde ich die Robustheit der in Kapitel 4 gezeigten XQuery-Abfragen gegenüber unterschiedlichen Compilerversionen zeigen. Für die Auswertung wurden die in Kapitel 3 gezeigten SCs mithilfe aller zum heutigen Tag (17.04.2023) verfügbaren Compiler in EVM-BC umgewandelt.

Der ausgeführte Befehl zur Kompilierung des entsprechenden Source-Code files ist: `solc --bin-runtime "temp.sol"`. Dabei wurden zunächst für jedes SC-Muster versionsabhängige Solidity Files erzeugt.

Diese versionsabhängigen, leicht angepassten Solidity Files wurden anschließend mit folgenden Compilern in deren EVM-BCs umgewandelt:

- 0.4.0 - 0.4.26 (27 Versionen)
- 0.5.0 - 0.5.15 (16 Versionen)
- 0.6.0 - 0.6.12 (13 Versionen)
- 0.7.0 - 0.7. 6 (7 Versionen)
- 0.8.0 - 0.8.18 (19 Versionen)

So wurden insgesamt 82 verschiedene EVM-BCs pro SC-Muster erhalten. Durch den Einsatz des PoC wurde für jede Version und jedes SC-Muster ein XML-Dokument erzeugt. Diese 510 XML-Dokumente enthalten jeweils alle Codewalks. Sie wurden nun mithilfe aller in Abschnitt 4.3 definierten XQuery-Abfragen ausgewertet. Die so erhaltenen Ergebnisse wurden in Form einer Tabelle dargestellt, siehe Tabelle 5.1.

	solc 0.4.0 - solc 0.4.9	solc 0.4.10 - solc 0.5.5	solc 0.5.6 - solc 0.8.18
Ablaufdatum	✗	✗	✓
Ablaufdatum - mit Variation	✓	✓	✓
Aktualisierbarkeit	✓	✓	✓
Eigentümerschaft	✗	✗	✓
Notfallstop	✓	✗	✓
Rollenbasierte Autorisierung	✗	✗	✓

Tabelle 5.1: Zeigt die in Abschnitt 4.3 angeführten XQuery-Abfragen und SCs gegenüber unterschiedlicher Compilerversionen ausgewertet. Ein "✓" zeigt eine erfolgreiche Erkennung des EVM-BC-Muster. Ein "✗" zeigt eine fehlgeschlagene Erkennung. Die in der Kopfzeile angeführten Bereiche sind die gruppierten Ergebnisse aller einzelnen Ergebnisse. Beispielsweise umfasst `solc 0.4.0 - solc 0.4.9` neun einzelne Compilerversionen.

5.1.1 Interpretation

Die Gründe für die in Tabelle 5.1 angeführten fehlgeschlagenen Erkennungen werden im Folgenden diskutiert. Um die in der Tabelle 5.1 angeführten Änderungen zu diskutieren, wurden die jeweils aneinandergrenzenden EVM-BCs-Versionen miteinander verglichen. Dabei wurde zunächst für beide das XML-Dokument herangezogen und mittels eines sogenannten Diff[1]-Werkzeugs verglichen.

- Ablaufdatum - `v0.4.0 - 0.5.5` vs. `0.5.6 - 0.8.18`:
  Im Vergleich zeigt sich in den älteren Versionen das Vorkommen zweier *ISZERO*-Instruktionen, die ineinander geschachtelt wurden. Der entsprechende AST ist in Abbildung 5.1 dargestellt. Die beiden *ISZERO*-Instruktionen heben gemeinsam ihre Wirkung auf. Semantisch besteht also kein Unterschied.
- Eigentümerschaft - `v0.4.0 - 0.5.5` vs. `0.5.6 - 0.8.18`:
  Analog zum Ablaufdatum zeigt sich dasselbe Verhalten. In den älteren Versionen sind zwei ineinander verschachtelte *ISZERO*-Instruktionen vorhanden.
- Notfallstop - `v0.4.0 - 0.4.9` vs. `0.4.10 - 0.5.5` vs. `0.5.6 - 0.8.18`:
  Analog zum Ablaufdatum zeigt sich dasselbe Verhalten.

[1]Sind Programme die Unterschiede in Textdokumenten hervorheben und so einfach erkennbar machen.

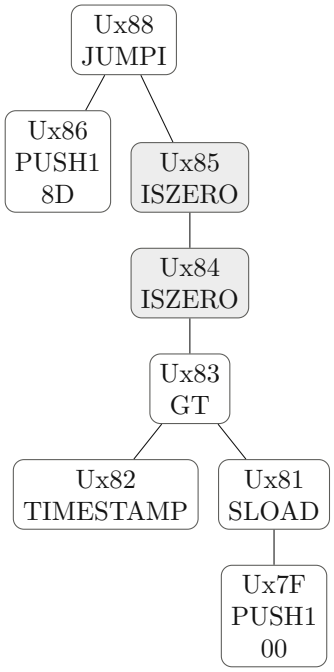


Abbildung 5.1: Stellt den in Auflistung 4.12 relevanten AST dar. Der gewählte Compiler hat die Version 0.5.5. Hervorgehoben sind die beiden *ISZERO*-Instruktionen, die in neueren Compilerversionen entfernt wurden. Semantisch ist dieser AST gleichbedeutend mit einem AST, der die zwei markierten Instruktionen nicht beinhaltet. Eine *ISZERO*-Instruktion kann im Kontext einer *JUMPI*-Instruktion wie eine NOT-Operation interpretiert werden. Die doppelte Negation hebt sich daher auf.

5.2 Feldtest

Aus den Daten von [dADFS23] wurden 1023 SCs gegen die in Kapitel 3 definierten Muster geprüft. Im Folgenden beschreibe ich zunächst die Erhebung der Daten durch [dADFS23], die weitere Filtrierung in dieser Arbeit, die Auswertung und diskutiere anschließend die erhaltenen Ergebnisse.

5.2.1 Datenerhebung 1

Die Daten von [dADFS23] wurden folgendermaßen erhoben:

- Zunächst wurden die EVM-BCs aller SC, die auf der Ethereum-Main-Chain bis zum Block 14.000.000 erfolgreich[2] veröffentlicht wurden, heruntergeladen.
- Die EVM-BCs wurden anschließend auf deren funktionale Bestandteile reduziert. Metadaten und Konstruktorargumenten werden entfernt. Die Payloads von *PUSH*-Instruktionen werden auf 0 gesetzt.

[2]eine erfolgreiche Veröffentlichung kann durch die Ausführung einer *CREATE*-Instruktion, einer *CREATE2*-Instruktion oder durch eine erfolgreiche Transaktion erfolgen.

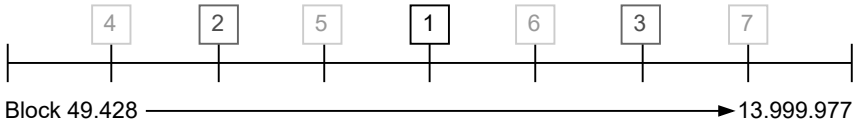


Abbildung 5.2: Zeigt das Auswahlschema relevanter SCs für die Evaluierung des PoC. Ganz links im Strahl steht dabei der Block des ersten SC der betrachtet wird. Ganz rechts der letzte. Zunächst wird ein SC gewählt der möglichst genau in der Mitte steht. Danach werden zwei SCs gewählt, die etwa bei 25 % und 75 % vorkommen. Usw.

- EVM-BCs, die nach der Reduktion aus Nullen bestehen oder deren Länge 0 ist, werden verworfen. Das entspricht der Entfernung aller SCs, die keinen Runtimecode haben.
- Die übrig gebliebenen EVM-BCs werden gruppiert. Aus jeder Gruppe wird ein SC aufgrund bestimmter Kriterien selektiert, siehe [dADFS23].

Auf dieser Basis werden in dieser Arbeit jene SCs selektiert, zu denen ein verifizierter Source Code vorliegt. So erhält man 89.293 SCs. Sie liegen im Bereich der Blöcke 49.428 bis 13.999.977. Im nächsten Schritt werden aus dieser Menge 1023 SCs herangezogen (1 + 2 + 4 + ... + 256 + 512 = 1023). Diese SCs wurden nach folgendem Schema selektiert:

- die SCs werden nach ihrem zugeordneten Block sortiert.
- der abgedeckte Bereiche an Blöcken [49.428, 13.999.977] wird in immer kleiner werdende Intervalle zerlegt. Die Grenzen dieser Intervalle stellen dabei heranzuziehende Blöcke dar. Da nicht zu jedem Block ein SC vorliegt, wird durch eine binäre Suche der erste SC ermittelt, der am nächsten zum Zielblock liegt. Beispielsweise ist dies in der ersten Iteration der Fall: `((49.428 + 13.999.977)/ 2)= 7024702`, siehe auch Abbildung 5.2.

Die so gefundenen SCs werden nun gegen die XQuery-Abfragen validiert. Die 1023 so validierten SCs segmentieren den angeführten Block-Bereich in etwa 13.600 Segmente, an deren Grenze jeweils ein SC ausgewertet wurde.

Aus den 1023 SCs, die bei mindestens einer XQuery-Abfrage eine positive Übereinstimmung haben, werden für jedes SC-Muster Negativbeispiele herangezogen um zu zeigen, dass die definierten XQuery-Abfragen richtigerweise keine positiven Ergebnisse erzeugen. So wird der Arbeitsumfang reduziert, da positive Beispiele von XQuery-Abfragen gleichzeitig Negativbeispiele für andere XQuery-Abfragen darstellen können.

Weiters werden für jedes SC-Muster zufällig zehn SCs herangezogen, die von einer XQuery-Abfrage erkannt wurden. Der originale Source Code wird anschließend ausgewertet und das entsprechende SC-Muster gesucht. Die Ergebnisse werden verglichen, um so auf die Güte der gewählten XQuery-Abfrage zu schließen.

5.2.2 Datenerhebung 2

Die in Unterabschnitt 5.2.1 angeführte Datenerhebung führt zu sehr wenigen positiven Übereinstimmung beim Aktualisierbarkeits-SC-Muster. Verschiedene Gründe kommen hier unter anderem infrage:

- Der in Unterabschnitt 5.2.1 ausgewertete Datensatz beinhaltet nur 22 SCs, die positiv auf den regulären Ausdruck "`\.delegatecall\(.*msg.data.*\)`"reagieren. Nach manueller Inspektion beinhalten 20 davon das entsprechende SC-Muster. 19 von diesen 20 Contracts sind in der Verwendung der *DELEGATECALL*-Instruktion ident. Ein Beispiel-SC findet sich unter `0xbadcf640bf54d28b15235dcec5817756f247cced`.
- SCs, die dieses SC-Muster beinhalten, implementieren typischerweise keine andere Funktionalität. Durch die in Unterabschnitt 5.2.1 angeführte Vorselektion, ist daher davon auszugehen, dass viele Instanzen in einer einzelnen Gruppe zusammengefasst wurden und daher eher nicht gefunden werden.
- Die gewählte XQuery-Abfrage ist nicht optimal gewählt.

Aus diesen Gründen wurde für die Evaluierung dieses SC-Musters, vorab 1437 SCs ausgewählt, die eine *DELEGATECALL*-Instruktion beinhalten und diese dann gegen die XQuery-Abfrage evaluiert.

SCs, die eine *DELEGATECALL*-Instruktion beinhalten, wurden durch die folgenden zwei Bedingungen erkannt:

- Muss eine *DELEGATECALL*-Instruktion im Runtimecode beinhalten.
- Muss einen delegatecall Aufruf im SC beinhalten. (Überprüfung über einen regulären Ausdruck "`\.\s*delegatecall\(`")

Anschließend wurde auf einem Teil der so erhaltenen SC die XQuery-Abfrage zur Erkennung des Aktualisierbarkeits-SC-Muster angewendet, solange bis insgesamt zehn SC gefunden wurden.

5.3 Auswertung

Von 1023 SCs wurden insgesamt 577 mit zumindest einer positiven XQuery-Abfrage gefunden. Elf SCs haben dabei auf vier verschiedene Abfragen positiv reagiert. 48 lieferten auf 3 Abfragen ein positives Ergebnis, 152 auf 2 Abfragen und 366 auf eine.

Von diesen ausgewerteten SCs wurden randomisiert zehn ausgewählt. Es wurde ein C#-Random mit dem Seed 1 initialisiert und zehn eindeutige Indexes durch das Iterieren der Random-Funktion selektiert. Die Indexe bewegen sich dabei im Bereich 0 bis zur Anzahl aller gefundener SCs eines SC-Musters. Die SCs an den Indexen (alphanumerisch sortiert) wurden dann für die manuelle Auswertung herangezogen.

Die ASTs, die durch die XQuery-Abfragen gefunden wurden, wurden mit dem in Kapitel 3 beschriebenen AST verglichen. Dadurch wird sichergestellt das auch die XQuery-Abfragen die beschriebenen Muster erkennen, und keine unerwarteten ASTs als positive Übereinstimmungen dedektieren. Im folgenden verwende ich den Begriff EVM-BC-Muster synonym

für ein Muster das nachdem Kompilieren und dem generieren der XML-Dokumente auftreten kann.

5.3.1 Ablaufdatum

Folgende SCs, die durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x8071e0602a9e5d50c5bdda6ce15bd4e3683a7258` - im Block 10989273
1 `0x8e98ef0adb8a7e8dcd089e2d961d5bb7a72005e0` - im Block 11030449
2 `0xed61372660aeb0776d5385df2c5f99a462de0245` - im Block 12147331
3 `0x0da25a3fbc8e988ae73f2a8c4504b1e242e02308` - im Block 12373822
4 `0xe13c0ec78d283eb8cca72edd018a9c13ad0e002a` - im Block 12788890
5 `0xb115371645354b0d4c231005b064f88ba53b2404` - im Block 12924554
6 `0xf2555451d1ed412d20369300bb15f30339b38bd9` - im Block 13942979
7 `0x1f36f2249ec89266ad4ee4351538148aaebc72e8` - im Block 1653270
8 `0xe9c417eae5e9be4a4837b844795f67f9681df69f` - im Block 4014533
9 `0xd96e3d7b96b7894dee1311fb5e234481089780ab` - im Block 7161250

Jeder einzelne SC wurde nachfolgend manuell gegenüber dem Ablaufdatum-SC-Muster ausgewertet. Die ausgewählten SCs implementieren Spiele, Tokens, und Wettprogramme: WarlordCamp (Idx 0), AxiaVault (Idx 1), RibbonCoveredCall (Idx 2), PopMarketplace (Idx 3), MLP - Token staking (Idx 4), eT - ERC 20 (Idx 5), MasterChefMod (Idx 6), Wettspiel für den Eurocup (Idx: 7), Crowdsale (Idx: 8), generelles Wettspiel (Idx 9).

Sie alle verwenden entweder `block.timestap` oder `now`, um den dadurch erhaltenen Wert innerhalb einer `require` oder `if` Abfrage gegenüber einer Speichervariable zu vergleichen. Nicht immer handelt es sich dabei um die Implementierung des Ablaufdatum-SC-Muster. In jedem einzelnen Fall wurde aber das entsprechende EVM-BC-Muster erkannt. In sieben von zehn Fällen implementiert der SC das SC-Muster in derselben oder einer komplexeren Form.

Folgende SCs, die **nicht** durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x144bf2bbc61a5ae867d7e1fd603015df7f6d4fb6` - im Block 1249695
1 `0x1d3b2638a7cc9f2cb3d298a3da7a90b67e5506ed` - im Block 1516094
2 `0xc5912c5aa88d7c947c9510b3c8d76543c72ed98c` - im Block 1522625
3 `0x3474627d4f63a678266bc17171d87f8570936622` - im Block 1970363
4 `0x9b27a23006b6612b2dfc840d0a25f4e347121d21` - im Block 2353746
5 `0x4c2c4511ba8f5b5f356adf7eaebce8dd2ea9a9c6` - im Block 2651739
6 `0x3bafb3af16203c817ee9208c6b8a748398dae689` - im Block 4749581
7 `0x8cc926f5bd83a420e127d80ee2874c22aa368b14` - im Block 5430764
8 `0x2634baad203cba4aa4114c132b2e50a3a6027ff9` - im Block 5499184
9 `0x30b5e711162a0b41e45c93389ed8e4f891766e02` - im Block 8905533

BCFSafe (Idx 2), ZperMainSale (Idx 7), BIRTHDAY_GIFT_1_ETH (Idx 8) und LockETH (Idx 9) implementieren das EVM-BC-Muster, wurden aber aufgrund der gewählten Compilerversionen (kleiner als 0.5.5) nicht erkannt, siehe auch Abschnitt 5.1.

Alle anderen oben genannten Beispiele erzeugen das EVM-BC-Muster nicht. Sie wurden daher von der XQuery-Abfrage auch nicht detektiert.

5.3.2 Notfallstop

Folgende SCs, die durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x1a9c1c1914a20fe9ae67b25913ffb8227c5cb617` - in Block 4123737
1 `0xb2b528413288a2ae024907fc654793ebdb3df537` - in Block 8973279
2 `0xbb128bc208c45b3dd277e001f88e1c6648060c64` - in Block 12146643
3 `0xf1e3af9152ad1f93be2ab2f8766d41744fad823a` - in Block 12196585
4 `0xbe6e3669464e7db1e1528212f0bff5039461cb82` - in Block 13020344
5 `0x0fbfcfb6d08f2fdfd480dd9f0399e011841fbf15` - in Block 13338511
6 `0x624425eb51c276cef390f4b337bd73267376f460` - in Block 13520536
7 `0x9a8643e17d93dff9a364c0046b8cc57ad14235b8` - in Block 13605366
8 `0xabdcebcbc44174a4579c22afe820146e3ef268c9` - in Block 13900248
9 `0x5a44ff097652acaa29d0ab52dd25343f213326c6` - in Block 13910180

Crowdsale (Idx 0), HashToken (Idx 1), FeederPool (Idx 2), CryptoGogos (Idx 3), MetaTunes (Idx 7), FlippinWhapeClub (Idx 8) und Donuts (Idx 9) implementieren das EVM-BC-Muster über ein sehr ähnliches SC-Muster. Sie alle implementieren ein SC-Muster namens `Pausable`. Dies sind SCs, welche die Funktionalität über einen unbestimmten Zeitraum deaktivieren können. Sie implementieren das SC-Muster über eine Speichervariable vom Typ `bool`.

PumpkinHedzOGBadge (Idx 5) und DissectedToadz (Idx 6) haben hingegen semantisch gesehen ein anderes SC-Muster implementiert. Es handelt sich um ein Muster, das in Mintable-Tokens auftritt. Die Phase des Minting kann dabei über eine Variable vom Typ `bool` auf unbestimmte Zeit aktiviert und deaktiviert werden.

Folgende SCs, die **nicht** durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x144bf2bbc61a5ae867d7e1fd603015df7f6d4fb6` - im Block 1249695
1 `0x407113f6b520f3ac72386cfb0eee9ed3930512c0` - im Block 1323986
2 `0x1d3b2638a7cc9f2cb3d298a3da7a90b67e5506ed` - im Block 1516094
3 `0xc5912c5aa88d7c947c9510b3c8d76543c72ed98c` - im Block 1522625
4 `0x3474627d4f63a678266bc17171d87f8570936622` - im Block 1970363
5 `0x9b27a23006b6612b2dfc840d0a25f4e347121d21` - im Block 2353746
6 `0x4c2c4511ba8f5b5f356adf7eaebce8dd2ea9a9c6` - im Block 2651739
7 `0x3bafb3af16203c817ee9208c6b8a748398dae689` - im Block 4749581
8 `0x8cc926f5bd83a420e127d80ee2874c22aa368b14` - im Block 5430764
9 `0x30b5e711162a0b41e45c93389ed8e4f891766e02` - im Block 8905533

Alle oben genannten Beispiele erzeugen das EVM-BC-Muster nicht. Sie wurden daher von der XQuery-Abfrage nicht detektiert.

5.3.3 Aktualisierbarkeit

Folgende SCs, die durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x6f539a9456a5bcb6334a1a41207c3788f5825207` - im Block 5226469
1 `0x3e460895ddeea50dd67f098f1bceffb03f36a04c` - im Block 11820759
2 `0xbadcf640bf54d28b15235dcec5817756f247cced` - im Block 11997610
3 `0x5b2733c3d3acd4abaf86719365fd683408b2c0a8` - im Block 11997741
4 `0x8ee4f840d0f8d6734022eefe4da8e5c6681b9724` - im Block 11997779
5 `0x913b6c7cdc4aaee92cd1efbdb6560dd04da98813` - im Block 11997806
6 `0x4d22af37cdbab2fe1b4f21340bab51ef3c12ad2d` - im Block 12037964
7 `0xdf83c19a6958ce28b1cb9e1751ed05b060505cfd` - im Block 12102368
8 `0xf0e71caf62cf7f3dba2793cdc2bba1712cea93aa` - im Block 12970468
9 `0x036cf3821ea246e935feef1c29ed8ddffac1dbb0` - im Block 13009221

Alle oben genannten SCs haben in ihrem Source Code den Aufruf `<storagevariable>.delegatecall(msg.data)`. Ohni (Idx 0) und MinePoolProxy (Idx 8) implementieren das SC-Muster bedingt. Ausschließlich Funktionsupdates sind möglich, das Hinzufügen von Funktionen ist durch die Art der Implementierung nicht möglich.

UniFarm (Idx 1), MVTUniswapMiningProxy (Idx 6) und BuyoutProposalsDelegator (Idx 7) implementieren das SC-Muster.

BErc20USDCDelegator (idx 2), BErc20Delegator(idx 3), BErc20WBTCDelegator(idx 4), BErc20ZRXDelegator (idx 5) und CErc20Delegator (idx 9) implementieren das SC-Muster nur unter der Bedingung, das `msg.data` übergeben wird. Vor der `delegatecall` Instruktion wird sichergestellt, dass `msg.value == 0` ist, was die Aktualisierbarkeit der Funktionen oder nicht-vorhandenen Funktionen einschränkt.

Folgende SCs, die **nicht** durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 `0x1d3b2638a7cc9f2cb3d298a3da7a90b67e5506ed` - im Block 1516094
1 `0xc5912c5aa88d7c947c9510b3c8d76543c72ed98c` - im Block 1522625
2 `0x3474627d4f63a678266bc17171d87f8570936622` - im Block 1970363
3 `0x9b27a23006b6612b2dfc840d0a25f4e347121d21` - im Block 2353746
4 `0x4c2c4511ba8f5b5f356adf7eaebce8dd2ea9a9c6` - im Block 2651739
5 `0x3bafb3af16203c817ee9208c6b8a748398dae689` - im Block 4749581
6 `0x8cc926f5bd83a420e127d80ee2874c22aa368b14` - im Block 5430764
7 `0x2634baad203cba4aa4114c132b2e50a3a6027ff9` - im Block 5499184
8 `0x30b5e711162a0b41e45c93389ed8e4f891766e02` - im Block 8905533
9 `0x58efca0e52f254b76aec95dbd8d8e687e1bb70b1` - im Block 12675815

Alle oben genannten Beispiele erzeugen das EVM-BC-Muster nicht. Sie wurden von der XQuery-Abfrage daher nicht detektiert.

5.3.4 Eigentümerschaft

Folgende SCs, die durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 0xaf396dce15cb9e834ff9187f34f589db22c849a8 - im Block 1331031
1 0x9b27a23006b6612b2dfc840d0a25f4e347121d21 - im Block 2353746
2 0xf766f9763443bb07e9b5974f566efb79dd8b705f - im Block 8292062
3 0x57e9a39ae8ec404c08f88740a9e6e306f50c937f - im Block 11915659
4 0x6999d62c9997ad1449415aa1d0373346941457bb - im Block 11997485
5 0x58efca0e52f254b76aec95dbd8d8e687e1bb70b1 - im Block 12675815
6 0x251bd8138eb61a4c2981568c7c5e9876db86bbad - im Block 13044618
7 0x506bac140906af4f85ff9bef70c8e42de6e5d45c - im Block 13226961
8 0x5f001a2a283d58192e9af7181ad10110fbbeee19 - im Block 13754755
9 0xdbd30abbe7c65e608b2ead6027aa1234de2707fe - im Block 13782015

LastIsMe (Idx 0), BlocklogyCertificate (Idx 2), ERC721(Polka City Asset - Idx 3), SimplePriceOracle (Idx 4), NFTXFeeDistributor(Idx 5), UBXTDistribution(Idx 6), TwoHundredKeys(Idx 7), nft (Idx 8) und RefiGoose (idx 9) implementieren das SC-Muster.

Escrow (Idx 1) implementiert das SC-Muster nicht. Anstelle dessen kommt ein identerscheinendes SC-Muster zum Einsatz, das in ein identes EVM-BC-Muster kompiliert wird. Eine Unterscheidung kann nur aufgrund semantischer Metadaten im Source Code getroffen werden.

Folgende SCs, die **nicht** durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 0x144bf2bbc61a5ae867d7e1fd603015df7f6d4fb6 - im Block 1249695
1 0x1d3b2638a7cc9f2cb3d298a3da7a90b67e5506ed - im Block 1516094
2 0xc5912c5aa88d7c947c9510b3c8d76543c72ed98c - im Block 1522625
3 0x3474627d4f63a678266bc17171d87f8570936622 - im Block 1970363
4 0x9b27a23006b6612b2dfc840d0a25f4e347121d21 - im Block 2353746
5 0x4c2c4511ba8f5b5f356adf7eaebce8dd2ea9a9c6 - im Block 2651739
6 0x3bafb3af16203c817ee9208c6b8a748398dae689 - im Block 4749581
7 0x8cc926f5bd83a420e127d80ee2874c22aa368b14 - im Block 5430764
8 0x2634baad203cba4aa4114c132b2e50a3a6027ff9 - im Block 5499184
9 0x30b5e711162a0b41e45c93389ed8e4f891766e02 - im Block 8905533

Nur eines der oben genannten Beispiele erzeugte das EVM-BC-Muster nicht. Es wurde von der XQuery-Abfrage nicht detektiert. Die Ursache dafür liegt in den gewählten Compilerversionen (kleiner als 0.5.5). Lediglich eine wurde korrekterweise nicht erkannt.

5.3.5 Rollenbasierte Autorisierung

Folgende SCs, die durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 0x0ccbee9c238e7b165aa5dbe652360b64ef23ad29 - im Block 8373550
1 0x56c707414bc4c9d3fd71a19d3c266295cf7f7312 - im Block 12678524
2 0x114ef3df5820d55d32045afaa15d8cc7ba2156d7 - im Block 12689750
3 0x7a4508a1457c8ad4e38159df3f544bee5c044ac5 - im Block 13120389
4 0xd6a4e574a3cfd9d50143085d84b9176fe4b80c94 - im Block 13195970
5 0x0b9e8415f47da353ccc7ad58234321f50d49063a - im Block 13260342

6 0x66bd3d6121a67b0caee80ff1267dc7f4464f9d5a - im Block 13290185
7 0xa96058225591284 76f6ebb85154c60e1f7c5e432 - im Block 13518695
8 0x931acd5c6766a94a43bd237b83aa8da923a80533 - im Block 13524270
9 0x14e94ab7303bb51a713af965438dd35f255c32bb - im Block 13754255

Hier zeigt sich, dass kein einziger SCs das SC-Muster implementiert. Sie alle erzeugen das EVM-BC-Muster. Wie sich herausstellt, ist die XQuery-Abfrage so generell gefasst, dass bereits das Vorkommen eines `mapping (address => bool)` innerhalb eines `require` oder `if` Ausdruck zu einem positiven Ergebnis führt.

Die angeführten SCs verwenden solch ein Konstrukt, um beispielsweise die Menge aller erfolgten Zustimmungen zu einer Entscheidung zu speichern.

Folgende SCs, die **nicht** durch die XQuery-Abfrage erkannt wurden, werden herangezogen:

0 0x144bf2bbc61a5ae867d7e1fd603015df7f6d4fb6 - im Block 1249695
1 0x1d3b2638a7cc9f2cb3d298a3da7a90b67e5506ed - im Block 1516094
2 0xc5912c5aa88d7c947c9510b3c8d76543c72ed98c - im Block 1522625
3 0x3474627d4f63a678266bc17171d87f8570936622 - im Block 1970363
4 0x9b27a23006b6612b2dfc840d0a25f4e347121d21 - im Block 2353746
5 0x4c2c4511ba8f5b5f356adf7eaebce8dd2ea9a9c6 - im Block 2651739
6 0x3bafb3af16203c817ee9208c6b8a748398dae689 - im Block 4749581
7 0x8cc926f5bd83a420e127d80ee2874c22aa368b14 - im Block 5430764
8 0x2634baad203cba4aa4114c132b2e50a3a6027ff9 - im Block 5499184
9 0x30b5e711162a0b41e45c93389ed8e4f891766e02 - im Block 8905533

Alle oben genannten Beispiele erzeugen das EVM-BC-Muster nicht. Sie wurden von der XQuery-Abfrage daher nicht detektiert.

KAPITEL 6

Fazit

In Kapitel 3 habe ich zunächst übliche SC-Muster, die bei der Recherche aufgetreten sind und im Rahmen dieser Arbeit sinnvoll bearbeitet werden konnten, aufgeführt. Ich bin auf die Anwendungsgebiete und deren üblichen Implementierungen anhand von bekannten Standards eingegangen. Danach habe ich die SC-Muster auf ihre wesentlichen Implementierungsdetails reduziert, um im nächsten Schritt deren EVM-BC-Muster analysieren zu können.

In einem iterativen Prozess habe ich einen Prototyp, der das Kompilieren, Dekompilieren, die symbolische Ausführen und das Umwandeln in die entsprechenden ASTs und XML-Dokumente durchführen kann, entwickelt. Gleichzeitig habe ich mit den so erhaltenen Daten die einzelnen SC-Muster und EVM-BC-Muster untersucht. Die Ergebnisse habe ich anschließend in die Weiterentwicklung des Prototyps einfließen lassen.

Nachdem der Prototyp in der Lage war, zuverlässig die entsprechenden Rohdaten zu generieren, wurden erkennbare EVM-BC-Muster definiert. Der Prototyp wurde nachfolgend um die Abfragesprache XQuery erweitert, sodass nun die ursprüngliche Fragestellung bearbeitet wurde: “Inwiefern ist XQuery als allgemeine Klassifizierungssprachen fähig EVM-BC-Muster beschreibbar zu machen”.

Bei der Entwicklung der XQuery-Abfragen zur Klassifizierung stellten sich einige zu erwartende Probleme heraus. Prominent ist hier die Klasse der Probleme, die durch statische Programmausführung auftreten. Beispielsweise kann nicht immer statisch entschieden werden, wie der Programmablauf fortgesetzt werden soll, oder ob Probleme, die durch das Design der EVM selbst gegeben sind siehe [CCCP21].

Nichtsdestoweniger konnte zuerst in Kapitel 4 und später in Kapitel 5 anhand verschiedener EVM-BC-Muster gezeigt werden, das XQuery als Klassifikationssprache für SC infrage kommen kann. Dabei wurde ersichtlich, dass das Erkennen der EVM-BC-Muster funktioniert. Um nun Rückschlüsse, von den bisher detektierten EVM-BC-Muster, auf

Implementierungsdetails im Source Code durchzuführen, bedarf es aber weiterer XQuery-Abfragen. Einzelne XQuery-Abfragen scheinen aber nur in manchen Fällen ausreichend zur Detektion von SC-Muster zu sein. Hier sind tiefergehende Untersuchungen notwendig, die im Rahmen dieser Arbeit allerdings nicht mehr möglich waren.

Zukünftige Arbeiten

Das Tool würde zukünftig stark von der Implementierung verschiedener Speicher-Tracking-Strategien profitieren. Beispielsweise könnten SAT-Solver eingesetzt werden, um die Abhängigkeiten über volatile Speicheraufrufe hinweg aufzulösen. Verbesserungen, die auf die Menge der exportierten Metadaten im XML-Dokument betreffen sind denkbar. Dabei könnten Eigenschaften hinzugefügt werden, die über den weiteren Programmablauf Auskunft geben können. Das würde noch einfachere XQuery-Abfragen ermöglichen, die nur Ausführungsstränge betrachten, die global betrachtet bestimmte Muster erfüllen.

Durch die Kombination verschiedener XQuery-Abfragen könnten SC-Muster mit einer höheren Sicherheit detektiert werden. Hier empfiehlt es sich, bestimmte XQuery-Abfragen zu ergänzen und schlussendlich Schnittmengen dieser Abfragen heranzuziehen, um nicht nur das Vorkommen von EVM-BC-Muster, sondern auch Rückschlüsse auf SC-Muster zu ermöglichen.

Abbildungsverzeichnis

Akronyme

ABI Application-Binary-Interface. 11

AST Abstract Syntax Tree. 13, 23, 35, 36, 38–45, 47–55, 57, 60, 61, 63, 69, 71

CFG Control-Flow-Graph. 2, 3, 5, 12, 13, 15, 23, 24, 31, 32, 36–38, 71

DOS Denial-of-Service. 1

EVM Ethereum-Virtual-Machine. ix, 1, 2, 5, 9–11, 15, 69

EVM-BC EVM-Bytecode. ix, 2, 3, 5, 10–13, 15, 23–25, 27, 28, 30, 31, 41, 44, 47–49, 52, 53, 59–62

EVM-BC-Muster EVM-Bytecode-Muster. 2, 3, 13, 15, 16, 18–21, 23, 25, 30, 31, 41, 43, 44, 50–55, 57, 58, 60, 63–71

OPC Operation-Code. 11, 25, 39, 41, 49

P2P Peer to Peer. 1, 6, 7, 9

POA Proof-of-Authority. 7, 9

PoC Proof of Concept. xiii, 2, 3, 14, 23, 24, 31, 37, 59, 62, 71

PoS Proof-of-Stake. 6–9

PoW Proof-of-Work. 1, 6–9

SC Smart Contract. ix, 1–3, 7, 9–13, 15–21, 25, 27, 29, 32–34, 39–41, 44, 47, 50, 51, 54, 58–69, 71

SC-Muster Smart-Contract Muster. ix, 2, 3, 15–21, 23, 25, 30, 41, 44–47, 49–52, 54, 59, 62–71

XML Extensible Markup Language. 2, 3, 23–25, 38–40, 42, 43, 45, 46, 49, 52–54, 57, 59, 60, 64, 69, 70

Literaturverzeichnis

[BCF+02] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language. 2002. URL: https://immagic.com/eLibrary/ARCHIVES/SUPRSDED/W3C/W030430D.pdf.

[BP17] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. *Lecture Notes in Computer Science*, 03 2017. doi:10.1007/978-3-319-70278-0_31.

[But15] Vitalik Buterin. A next generation smart contract & decentralized application platform. 2015. URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[CCCP21] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 127–137, 2021. doi:10.1109/ICPC52881.2021.00021.

[Cho17] Usman Chohan. The double spending problem and cryptocurrencies. *SSRN Electronic Journal*, 01 2017. doi:10.2139/ssrn.3090174.

[dADFS23] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. Evolution of automated weakness detection in ethereum bytecode: a comprehensive study, 2023. arXiv:2303.10517.

[dAS20] Monika di Angelo and Gernot Salzer. Assessing the similarity of smart contracts by clustering their interfaces. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1910–1919, 2020. doi:10.1109/TrustCom50675.2020.00261.

[de 18] Alex de Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018. URL: https://www.sciencedirect.com/science/article/pii/S2542435118301776, doi:10.1016/j.joule.2018.04.016.

[GBSS19] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019. doi:10.1109/ICSE.2019.00120.

[HHY+21] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Transactions on Information Forensics and Security*, 16:2144–2156, 2021. doi:10.1109/TIFS.2021.3050051.

[HS91] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 437–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. doi:10.1007/3-540-38424-3_32.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982. URL: https://dl.acm.org/doi/pdf/10.1145/2402.322398.

[McD21] Kyle McDonald. Ethereum emissions: A bottom-up estimate. *CoRR*, abs/2112.01238, 2021. URL: https://arxiv.org/abs/2112.01238, arXiv:2112.01238.

[MMH+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019. doi:10.1109/ASE.2019.00133.

[MPJ18] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2018. doi:10.1109/ICCCNT.2018.8494045.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[NPS+17] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen. Automated labeling of unknown contracts in ethereum. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2017. doi:10.1109/ICCCN.2017.8038513.

[Opea] OpenZeppelin. Access control. Letzter Zugriff: 2023-04-17. URL: https://docs.openzeppelin.com/contracts/4.x/access-control.

[Opeb] OpenZeppelin. Openzeppelin smartcontracts. Letzter Zugriff: 2023-04-17. URL: https://docs.openzeppelin.com.

[Sax] Saxon. Saxoncs. Letzter Zugriff: 2023-04-17. URL: https://www.saxonica.com/html/documentation11/saxon-cs/.

[SHS22] Nicolas Six, Nicolas Herbaut, and Camille Salinesi. Blockchain software patterns for the design of decentralized applications: A systematic literature review. *Blockchain: Research and Applications*, 3(2):100061, 2022. URL: https://www.sciencedirect.com/science/article/pii/S209672092200001X, doi:10.1016/j.bcra.2022.100061.

[sola] soliditylang. Encoding of the metadata hash in the bytecode. Letzter Zugriff: 2023-04-17. URL: https://docs.soliditylang.org/en/v0.8.17/metadata.html#encoding-of-the-metadata-hash-in-the-bytecode.

[solb] soliditylang. Layout of state variables in storage. Letzter Zugriff: 2023-04-17. URL: https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html.

[SXY+22] Chaochen Shi, Yong Xiang, Jiangshan Yu, Longxiang Gao, Keshav Sood, and Robin Ram Mohan Doss. A bytecode-based approach for smart contract classification. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1046–1054, 2022. doi:10.1109/SANER53432.2022.00122.

[Woo17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2023-04-24. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[WZ18a] Maximilian Wöhrer and Uwe Zdun. Design patterns for smart contracts in the ethereum ecosystem. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1513–1520, 2018. doi:10.1109/Cybermatics_2018.2018.00255.

[WZ18b] Maximilian Wöhrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018. doi:10.1109/IWBOSE.2018.8327565.