Universität

Fakultät
Institut
Professur

Forschungsprojekt Anwendung zum Thema

Atomic Swaps auf öffentlichen Blockchains

noobWithAComputer
(Geboren irgendwann irgendwo)

Verantwortlicher Hochschullehrer:	ein Professor
Betreuer:	ein Betreuer

irgendwo, 21. September 2018

Inhaltsverzeichnis

Glossar

Locktime	Relativ oder absolut. Gibt an, wie lange oder bis wann ein UTXO gesperrt ist.
Miner	ist ein Knoten im Netzwerk, der versucht, einen neuen gültigen Block zu finden.
Offchain	Kommunikation, die nicht in der Blockchain gespeichert wird, also über ein anderes Kommunikationsmedium läuft.
Onchain	Transaktionen, die in der Blockchain ablaufen.
Proof of Work (PoW)	ist ein Konsensusalgorithmus, bei dem Rechenaufwand als Beweis genutzt wird.
Unspent Transaction Output (UTXO)	Ein UTXO ist ein Transaktionsoutput, der noch nicht für eine neue Transaktion ausgegeben wurde (unspent).

Abbildungsverzeichnis

Tabellenverzeichnis

1. Motivation und Ziel der Arbeit

Kryptowährungen haben über die vergangenen Jahre ein rasantes Wachstum sowohl in ihrer Marktkapitalisierung als auch in ihrer Medienpräsenz an den Tag gelegt. Voraussichtlich wird aber keine einzelne dieser blockchain-basierten Währungen den alleinigen Durchbruch schaffen. Daher werden immer mehr Protokolle und Anwendungen entwickelt, die die Interaktion verschiedener Blockchains untereinander ermöglichen sollen. Dabei hat sich ein Schema etabliert, um die Coins verschiedener Blockchains zwischen verschiedenen Parteien zu tauschen: Atomic Swaps. Sie sollen garantieren, dass ein solcher Tausch entweder ganz oder gar nicht stattfindet, wodurch unabhängig vom Ausgang des Tausches keine der beteiligten Parteien Verluste zu verzeichnen hat.

1.1. Zielsetzung

Im Rahmen dieser Arbeit soll nun untersucht werden, ob es möglich ist, Atomic Swaps zu erkennen und nachzuvollziehen. Zusätzlich wird eine Protokollanalyse durchgeführt, um die Sicherheit von Atomic Swaps zu beweisen.

Da die meisten Blockchain-Protokolle öffentlich einsehbar sind, können alle Transaktionen nachvollzogen werden. Allerdings verbinden Atomic Swaps mehrere Transaktionen auf mehreren Blockchains. Dieser fehlende Zusammenhang zwischen den unterschiedlichen Blockchains soll nun hergestellt werden. Dafür werden hier vier verschiedene Blockchains untersucht: Bitcoin (BTC) [1], Bitcoin Cash (BCH)[1], Litecoin (LTC)[2] und Decred (DCR)[3].

Diese vier Protokolle wurden ausgesucht, da sie auf der gleichen technologischen Grundlage basieren und gleichzeitig eine recht hohe Marktkapitalisierung aufweisen. So befinden sich Bitcoin, Bitcoin Cash und Litecoin in den Top 10 der Kryptowährungen; Decred liegt auf Platz 29[4].

[1] Siehe `https://www.bitcoincash.org/`
[2] Siehe `https://litecoin.org/`
[3] Siehe `https://decred.org/`
[4] Stand: 06.08.2018, siehe `https://www.coinmarketcap.com`

2. Blockchain und Bitcoin

Kryptowährungen stellen ein monetäres Transaktionssystem dar, welches Peer-to-Peer zwischen den Nutzern agiert. Dabei ist die Blockchain das Speichermedium, in dem alle Transaktionen festgehalten werden.

Die Transaktionen werden dabei in Blöcken gespeichert, die kryptographisch miteinander zu einer Kette verbunden werden. Dadurch können bereits angehängte Blöcke nicht mehr verändert werden. Diese kryptographische Sicherung kann über verschiedene Algorithmen erfolgen, welche Konsens-Algorithmen genannt werden [2]. Die meisten Blockchain-Protokolle bedienen sich des *Proof of Work* (POW). Dabei wird die Erstellung von neuen Blöcken als Mining (siehe Kapitel 2.1) bezeichnet.

2008 wurde von Satoshi Nakamoto die erste öffentlich nutzbare Blockchain entworfen, der Bitcoin [1]. Es stellte die erste Kryptowährung auf Basis einer Blockchain dar.

Allen gewählten Protokollen liegt die gleiche Codebasis zugrunde. Daher reicht es aus, eines dieser Protokolle zu verstehen. Da Bitcoin die älteste Kryptowährung ist und die anderen untersuchten Coins Kopien[5] von Bitcoin sind, liegt der Fokus auf Ersterem. Die anderen Protokolle weisen nur geringe Unterschiede auf. Der größte Unterschied ist wohl die Blockzeit, also die durchschnittliche Zeit bis ein neuer Block gefunden wurde.

2.1. Mining

Die Blöcke einer Blockchain bestehen aus einem Header, der die Metadaten enthält, und einem Body, der eine Liste der ausgeführten Transaktionen enthält. Die kryptographische Sicherung des Blockes erfolgt durch die Berechnung des Blockhashes[6]. Dieser muss aber eine besondere Eigenschaft erfüllen: der Hash muss mit einer vorgegebenen Anzahl an Null-Bits beginnen. Um eine Varianz für den gleichen Block zu erreichen, existiert ein Feld im Blockheader, der beliebig angepasst werden kann, die Nonce. Der Miner, also die Entität, die versucht, einen neuen gültigen Blockhash zu berechnen, variiert diese Nonce, um irgendwann einen Blockhash zu finden, der genug führende Nullen enthält.

Hat der Miner eine Nonce mit entsprechendem Blockhash gefunden, so sendet er diesen neuen Block an alle seine Nachbarn im Netzwerk und erhält dafür einen Mining-Reward, also eine Belohnung. Durch seine geleistete Arbeit wird sichergestellt, dass der Block eine hohe kryptographische Sicherheit besitzt (daher die Bezeichnung Proof of Work) [1].

2.2. Transaktionen

Wichtiger Bestandteil von Kryptowährungen ist die Übertragung von digitalen monetären Werten zwischen den Nutzern. Als Nutzer ist in dieser Betrachtung vor allem das Vorhandensein eines

[5] LTC und DCR verwenden angepasste Versionen des Bitcoin-Protokolls, verfügen aber über komplett losgelöste Blockchains. Bitcoin Cash hingegen ist ein Fork, also eine Abspaltung, aus der Bitcoin-Blockchain. Dadurch besitzen Bitcoin und Bitcoin Cash eine gemeinsame Vergangenheit, spalten sich dann aber am 1.8.2017.

[6] Ein Hash ist das Ergebnis einer deterministischen Einwegfunktion, die neben Kollisionsfreiheit auch einen Lawineneffekt beinhaltet. Sie bildet einen Input beliebiger Länge auf einen Output fester Länge ab [3].

kryptografischen Schlüsselpaars relevant. Transaktionen werden vom jeweiligen Sender signiert, bevor sie der Blockchain angehängt werden.

Für die Validierung von Transaktionen implementiert Bitcoin das System der *Unspent Transaction Outputs* (UTXO). Dabei benutzt jede Transaktion die Outputs vorheriger Transaktionen als Inputs, wodurch ein gerichteter Graph entsteht. Für eine neue Transaktion können nur noch nicht ausgegebene (englisch unspent) Outputs genutzt werden. Sobald ein Output als Input einer neuen Transaktion verwendet wird, wird dieser Output als „spent“ markiert. Dabei ist jeder Input und Output jeweils mit einem monetären Wert verbunden. Die Summe der Inputs muss dabei immer gleich oder größer der Summe der Outputs sein. Die Differenz der Outputs und Inputs geht dann als Gebühr an den Miner, der diese Transaktion mit in seinen Block aufnimmt.[7]

Sowohl die Inputs als auch die Outputs bestehen aus einem Skript. Die Verbindung dieser beiden Skripte bestimmt, ob eine Transaktion valide ist. Dabei existieren verschiedene Arten von Transaktionen:

1. P2PK: Pay to public key. Dies waren die ersten Formen von Transaktionen im Bitcoin-Netzwerk. Dabei besteht der Output aus dem Public-Key desjenigen, der diesen Output ausgeben darf. Dieser Nutzer muss eine valide Signatur mit seinem Private-Key erstellen, um den Output auszugeben.

2. P2PKH: Pay to public key hash. Um Angriffe über Rainbow-Tables zu erschweren, wurde eine Transaktionsart entwickelt, welche genau wie P2PK funktioniert, ein Nutzer dabei aber seinen Public-Key nicht preisgeben muss. Der Output besteht dabei aus dem Hash des Public-Keys. Aus Sicherheitsgründen sollte also auf P2PK-Transaktionen verzichtet werden.

3. P2SH: Pay to Script hash. Hierbei besteht der Output aus dem Hash eines Scriptes. Der Input, der diesen Output nutzen soll, muss ein Skript angeben, welches zu dem vorgegebenen Hash gehasht werden kann und mit frei wählbaren Input-Parametern True zurückgibt.

2.3. Script

Die Skriptsprache von Bitcoin heißt einfach nur Script und ist eine stack-basierte Sprache, die ähnlich zu Forth ist[8]. Es ist durch seine low-level Opcodes recht eingeschränkt und nicht turingvollständig und kann daher kaum mit höheren Sprachen wie Solidity (die Smart-Contract-Sprache von Ethereum [4, 5]) verglichen werden. Nichtsdestotrotz werden die Skripte auf Bitcoin oft als Smart Contracts bezeichnet.

Das Bitcoin-Skript stellt eine Reihe an Opcodes zur Verfügung, die von Kontrollstrukturen über Stackoperationen bis hin zu arithmetischen und kryptographischen Operationen reichen. Damit lassen sich viele verschiedene Skripte entwerfen[9].

[7] Siehe `https://en.bitcoin.it/wiki/` und [1].
[8] Siehe `https://en.bitcoin.it/wiki/Script`
[9] Siehe `https://en.bitcoin.it/wiki/Contracts`

3. Atomic Swaps

Atomic Swaps sind ein elegantes Protokoll zur Lösung eines weit verbreiteten ökonomischen Problems: Wie kann der Tausch von Werten zwischen zwei sich nicht vertrauenden Parteien über öffentlich einsehbare Blockchains hinweg realisiert werden? D.h. wie können Alice, die Bitcoin besitzt, und Bob, der Litecoin besitzt, diese gegenseitig sicher tauschen[10]?

Da sich die Parteien nicht vertrauen, muss der Tausch atomar geschehen. Weil beide Blockchains (BTC und LTC) keinen direkten Bezug haben, müssen die Transaktionen über Blockchains hinweg synchronisiert werden. Aufgrund der öffentlichen Einsehbarkeit der beiden Blockchains müssen die nötigen Transaktionen über Secrets gesichert werden.

Atomic Swaps setzen eine Reihe an Techniken wie Timelocks und Hashed-Timelock-Contracts voraus, die von der Skriptsprache der darunter liegenden Blockchain unterstützt werden müssen. Im Folgenden werden deshalb zuerst die Basistechniken vorgestellt und anschließend deren Nutzung in Atomic Swaps erläutert.

3.1. Timelocks

```
 1  04  OP_DATA_4                # pushe 4 Bytes auf den Stack
 2    <locktime 4bytes>          # das zu pushende Element (Locktime)
 3  b1  OP_CHECKLOCKTIMEVERIFY   # überprüft, ob die Locktime eingetreten ist,
                                   andernfalls brich ab
 4  75  OP_DROP                  # entferne das oberste Stack-Element (Locktime)
 5  76  OP_DUP                   # dupliziere das oberste Stack-Element
                                   (Public-Key)
 6  a9  OP_HASH160               # hashe das oberste Stack-Element mit Hash160
                                   (das Pre-Image wird entfernt)
 7  14  OP_DATA_20                # pushe 20 Bytes auf den Stack
 8    <pubkey_hash 20bytes>      # das zu pushende Element (Public-Key-Hash)
 9  88  OP_EQUALVERIFY           # vergleiche die beiden obersten Stack-Elemente
                                   und entferne diese. Wenn sie ungleich sind,
                                   brich ab
10  ac  OP_CHECKSIG              # überprüfe, ob der Hash des Transaktions-
                                   Skriptes mit dem ersten Element auf dem Stack
                                   als Public-Key das zweite Element (die
                                   Signatur) ergibt.
```

Abbildung 3.1.: Ein einfaches Timelock. Der Output kann nur nach Ablauf der Locktime und nur von dem Nutzer mit dem passenden Public-Key genutzt werden.

Ein Timelock ist eine Transaktion, die erst nach einer vorher festgelegten Zeit ausgegeben werden kann[11]. Im Bitcoin Script erfolgt diese Überprüfung über den Opcode `OP_CHECKLOCKTIMEVERIFY` oder über `OP_CHECKSEQENCEVERIFY`. Ersterer überprüft, ob entweder der festgelegte Zeitpunkt eingetreten ist[12] oder eine bestimmte Blockhöhe erreicht wurde. Dadurch wird eine absolute Zeit

[10] Siehe `https://en.bitcoin.it/wiki/Atomic_cross-chain_trading`
[11] Siehe `https://en.bitcoin.it/wiki/Timelock`
[12] Die Blockzeit wird mit der angegebenen Locktime verglichen

vorgegeben. Die Seqencenumber, die bei `OP_CHECKSEQENCEVERIFY` angegeben wird, besagt, nach wie vielen Blöcken ein UTXO wieder als Input genutzt werden darf. Dies implementiert eine relative Zeitsperre, in der die entsprechenden Coins nicht bewegt werden können.

3.2. Hashlocks

Ein Hashlock sperrt genau wie ein Timelock den erzeugten UTXO[13]. Allerdings erfolgt diese Sperre nicht über eine absolute oder relative Zeit, sondern über den Hash (Secrethash) eines für Außenstehende unbekannten Geheimnisses (Secret). Dadurch kann dieser UTXO nur durch Angabe des Secrets als Input für eine Folge-Transaktion genutzt werden. Im Unterschied zu klassischen private Keys, die zum Erzeugen von Signaturen dienen, sind Secret und Secrethash nicht an Adressen gebunden, sondern werden direkt in den Skripten verwendet. Jedes Secret sollte nur einmal benutzt werden, da es nach seiner Benutzung öffentlich in der Blockchain einsehbar sind.

```
1  a8  OP_SHA256                # hashe das oberste Stack-Element mit SHA256
                                  (Secret, das Pre-Image wird entfernt)
2  20  OP_DATA_32                # pushe 32 Bytes auf den Stack
3    <secret_hash 32bytes>      # das zu pushende Element (Secrethash)
4  87  OP_EQUAL                 # vergleiche die beiden obersten Stack-Elemente
                                  und entferne diese. True oder False
                                  verbleiben auf dem Stack
```

Abbildung 3.2.: Ein einfaches Hashlock mit SHA256-Secret. Der Output kann von jedem ausgegeben werden, der das Secret kennt.

3.3. Hashed-Timelock Contract

Ein Hashed-Timelock Contract (HTLC) verbindet nun ein Timelock mit einem Hashlock. Ein mit HTLC gesperrter UTXO kann nun entweder mittels Secret oder erst nach Ablauf des Timelocks weitergenutzt werden Zusätzlich können in HTLCs weitere Bedingungen festgelegt werden, die z.B. das Ausgeben eines UTXO nur auf bestimmte vorher bekannte Ziel-Adressen zulassen.

Somit erschafft ein HTLC ein Zeitfenster, in dem mit Hilfe eines Secrets die Transaktion geclaimt werden kann. Geschieht dies nicht, kann der hinterlegte Betrag vom Sender normal weiterverwendet werden[14].

3.4. Atomic Swap

Verbindet man nun zwei gegenläufige HTLCs, erhält man ein Geld-Transfer-Konstrukt, welches die zu überweisenden Geldmengen beider Parteien für eine bestimmte Zeit einsperrt, es sei denn, das Secret wird vorgelegt. Da die Transaktionen im Bitcoin-Netzwerk öffentlich einsehbar sind, kann jeder, der von der entsprechenden Transaktion weiß, das Secret entdecken und auslesen. Wird nun bei beiden HTLCs der gleiche Secrethash verwendet[15], so kann die erste Partei die

[13] Siehe `https://en.bitcoin.it/wiki/Hashlock`
[14] Siehe `https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts`
[15] Die zweite Partei kennt das Secret nicht, sondern sieht nur den Secrethash.

```
 1  63  OP_IF                    # wenn oberstes Stack-Element 1 (wird entfernt)
 2  a8  OP_SHA256                # hashe das oberste Stack-Element mit SHA256
                                   (Secret, das Pre-Image wird entfernt)
 3  20  OP_DATA_32               # pushe 32 Bytes auf den Stack
 4    <secret_hash 32bytes>      # das zu pushende Element (Secrethash)
 5  88  OP_EQUALVERIFY           # vergleiche die beiden obersten Stack-Elemente
                                   und entferne diese, wenn sie ungleich sind,
                                   brich ab
 6  76  OP_DUP                   # dupliziere das oberste Stack-Element
                                   (Public-Key)
 7  a9  OP_HASH160               # hashe das oberste Stack-Element mit Hash160
                                   (das Pre-Image wird entfernt)
 8  14  OP_DATA_20               # pushe 20 Bytes auf den Stack
 9    <pubkey_hash1 20bytes>     # das zu pushende Element (Public-Key-Hash)
10  88  OP_EQUALVERIFY           # vergleiche die beiden obersten Stack-Elemente
                                   und entferne diese, wenn sie ungleich sind,
                                   brich ab
11  ac  OP_CHECKSIG              # überprüfe, ob der Hash des Transaktions-
                                   Skriptes mit dem ersten Element auf dem Stack
                                   als Public-Key das zweite Element (die
                                   Signatur) ergibt.
12  67  OP_ELSE                  # wenn das oberste Stack-Element nicht 1
13  04  OP_DATA_4                # pushe 4 Bytes auf den Stack
14    <locktime 4bytes>          # das zu pushende Element (Locktime)
15  b1  OP_CHECKLOCKTIMEVERIFY   # überprüft, ob die Locktime eingetreten ist,
                                   andernfalls brich ab
16  75  OP_DROP                  # entferne das oberste Stack-Element (Locktime)
17  76  OP_DUP                   # dupliziere das oberste Stack-Element
                                   (Public-Key)
18  a9  OP_HASH160               # hashe das oberste Stack-Element mit Hash160
                                   (das Pre-Image wird entfernt)
19  14  OP_DATA_20               # pushe 20 Bytes auf den Stack
20    <pubkey_hash2 20bytes>     # das zu pushende Element (Public-Key-Hash)
21  88  OP_EQUALVERIFY           # vergleiche die beiden obersten Stack-Elemente
                                   und entferne diese, wenn sie ungleich sind,
                                   brich ab
22  ac  OP_CHECKSIG              # überprüfe, ob der Hash des Transaktions-
                                   Skriptes mit dem ersten Element auf dem Stack
                                   als Public-Key das zweite Element (die
                                   Signatur) ergibt.
23  68  OP_ENDIF                 # Ende der If-Verzweigung
```

Abbildung 3.3.: Ein HTLC mit SHA256-Secret.

ihr zustehenden Coins claimen, die andere Partei erfährt das Secret und kann nun selbst eine „Redeem“-Transaktion an den anderen HTLC senden.[16]

Um den korrekten Ablauf zu gewährleisten, sollten zusätzlich zu den Timelocks und Hashlocks auch die Public-Keys oder deren Hashes in den HTLCs hinterlegt werden. Dabei wird jeweils einer der Keys einem der beiden möglichen Ausführungspfade zugeordnet. Dadurch ist es nicht möglich, dass die Partei, die initial das Secret kennt, beide HTLCs für sich beansprucht. Diese Partei kann (und soll) den ersten HTLC claimen, müsste dann aber warten, bis die Locktime abgelaufen ist, bevor sie den anderen HTLC claimen kann. In diesem Zeitfenster muss die andere Partei das Secret extrahieren und ihre Transaktion losschicken. Kommt diese Transaktion nicht zustande, kann die erste Partei tatsächlich beide HTLCs für sich beanspruchen. Daher sollte das

[16] Vgl. https://github.com/decred/atomicswap

Zeitfenster des ersten HTLC (der, der zuletzt geclaimt wird) deutlich länger sein, als das des zweiten HTLC.

Dieses Konstrukt wird als Atomic Swap bezeichnet. Atomar in diesem Fall bedeutet, dass der Austausch der Coins entweder in beide Richtungen ausgeführt wird oder nicht stattfindet.

Dadurch können unterschiedliche Krypto-Währungen gegeneinander getauscht werden, ohne dass eine zentrale Exchange oder eine andere dritte Partei involviert sein muss, um die Vertrauenslücke zu schließen.

3.5. Der Ablauf im Detail

Die Durchführung eines Atomic Swaps kann in drei Phasen eingeteilt werden.

3.5.1. Festlegen der Parameter

Um die HTLCs zu erstellen, müssen noch ein paar Parameter zwischen den Teilnehmern vereinbart werden. Diese Kommunikation erfolgt offchain.

So sollte schon im Vorfeld geklärt werden, welche Blockchains genutzt werden, wie lang die Locktimes gesetzt werden, wie viele Coins jeweils ausgetauscht werden sollen, welcher Hashing-Algorithmus verwendet wird und welche Public-Key-Hashes[17] den Parteien auf den verschiedenen Blockchains gehören. Diese Parameter werden offchain, also abseits einer der Blockchains, ausgetauscht.

Zusätzlich zu diesen „gemeinsamen“ Parametern erstellt Alice (der Initiator) ein Secret und berechnet den dazugehörigen Hash.

3.5.2. Erstellen der HTLCs

Nun erstellt zuerst Alice einen HTLC mit den entsprechenden Parametern (*Create HTLC A* in Abbildung 3.4). Dabei wird der eigene Public-Key-Hash dem Timelock-Ausführungspfad zugeordnet und der Public-Key-Hash von Bob dem Ausführungspfad des Hashlocks.

Von diesem HTLC berechnet Alice nun den Script-Hash (*Create Scripthash A*) und erstellt eine Funding-Transaktion auf Blockchain α (*Create Funding Tx*), die diesen Script-Hash als Output besitzt[18]. Zusätzlich schickt Alice den HTLC offchain an Bob (*Send HTLC A*).

Bob prüft nun im HTLC, den er von Alice erhalten hat, ob die Parameter den Vereinbarungen entsprechen (*Audit HTLC A*). Weiterhin prüft Bob, ob der Script-Hash der Transaktion von Alice auch dem Hash des erhaltenen HTLC entspricht (*Create Scripthash A* und *Audit Scripthash A*).

Ist dies alles korrekt, erstellt Bob nun selbst einen HTLC (*Create HTLC B*). Dieser enthält die abgesprochenen Parameter und die entsprechenden Public-Key-Hashes. Die Locktime wird hier deutlich kürzer gesetzt, damit Bob in der zweiten Phase die nötige Zeit hat, um das Secret auszulesen und seine Redeem-Transaktion zu senden. Der eigene Public-Key-Hash wird wieder dem Timelock zugeordnet; der Public-Key-Hash von Alice dem Hashlock.

Genau wie vorher auch, hasht nun Bob den HTLC (*Create Scripthash B*) und erstellt eine Funding-Transaktion; diesmal jedoch auf Blockchain β (*Create Funding Tx*). Auch dieser HTLC wird offchain an die andere Partei geschickt (*Send HTLC B*).

Somit kann nun auch Alice überprüfen, ob alle Parameter wie besprochen gesetzt sind (Audit HTLC B). Wie zuvor Bob prüft Alice zusätzlich, ob der Scripthash in Blockchain β auch dem Hash des HTLC von Bob entspricht (*Create Scripthash B* und *Audit Scripthash B*).

[17] Es könnten auch die Public-Keys selbst ausgetauscht werden, es reicht aber auch, nur die Hashes zu benutzen.
[18] Also eine P2SH-Transaktion, die nur mit dem HTLC als Input-Script ausgegeben werden kann.

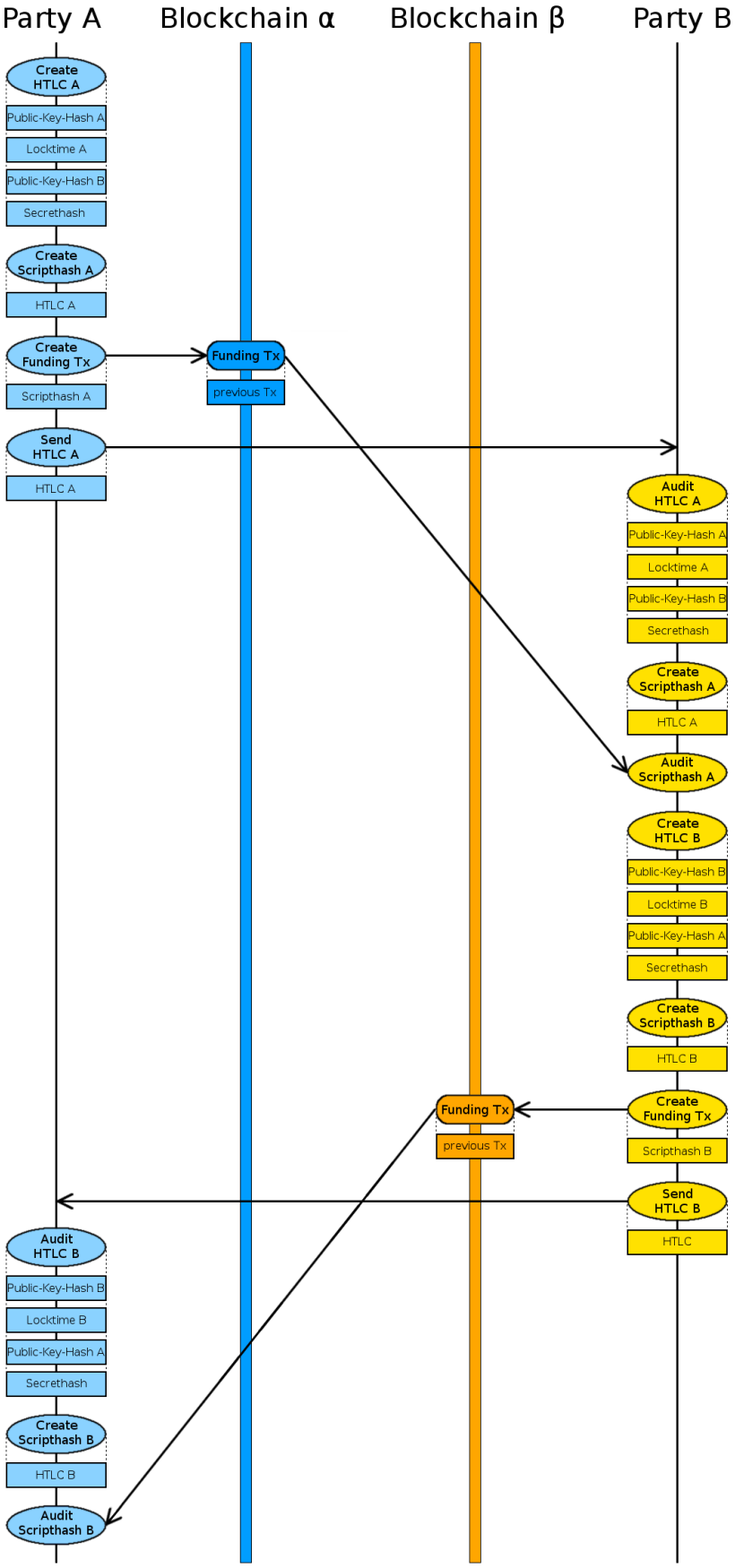

Abbildung 3.4.: Ablauf der zweiten Phase eines Atomic Swaps.

3.5.3. Claimen der HTLCs

Nur, wenn beide Parteien bisher zufrieden sind, wird dieser Punkt erreicht. Ist eine Partei nicht mit dem HTLC der anderen Partei einverstanden oder will den Swap aus einem anderen Grund nicht durchführen, könnte ein Abbruch von Seite der ersten Partei noch bis zu diesem Punkt stattfinden. Bob hätte schon in beim *Audit HTLC A* abbrechen müssen. Unter Abbruch versteht man, dass keine weiteren Protokollnachrichten oder Blockchain-Transaktionen gesendet werden. Nach Ablauf der Locktimes ist dieser Protokollzustand auch final.

Nun kann der tatsächliche Austausch der Coins beginnen. Dazu sendet Alice den HTLC, den sie von Bob erhalten hatte, als Input an die Funding-Transaktion von Bob auf Blockchain β (*Create Redeem Tx* in Abbildung 3.5). Als Input-Parameter werden eine 1 oder 0[19], das Secret, der eigene Public-Key und eine dazugehörige Signatur angegeben. Der HTLC überprüft die Eingaben und schickt dann die Coins an eine von Alice vorgegebene Adresse. Somit ist der von Bob erstellte HTLC bereits ausgegeben worden.

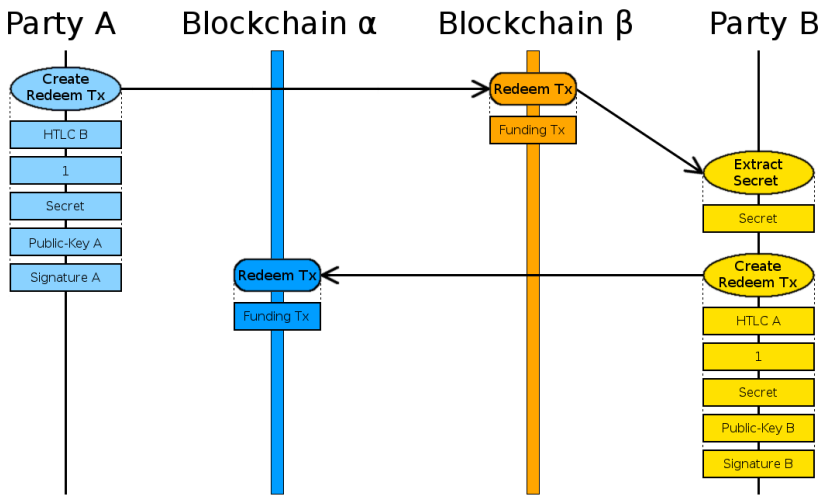

Abbildung 3.5.: Ablauf der dritten Phase eines Atomic Swaps.

Also bleibt noch der HTLC von Alice auf Blockchain α. Alice kann ihn aber zu diesem Zeitpunkt nicht claimen, obwohl sie das Secret kennt. Die Zuordnung der Public-Key-Hashes verhindert dies. Alice müsste warten, bis die Locktime abgelaufen ist. Auch ein Angreifer kann den HTLC von Bob nicht claimen, da das Hashlock zusätzlich mit Bobs Public-Key gesperrt ist.

Dieses Zeitfenster nutzt Bob nun, um z.B. in einem Blockexplorer die Blockchain β nach der Transaktion von Alice zu durchsuchen. Dort sind auch die Input-Parameter einsehbar, wodurch Bob das Secret auslesen kann (*Extract Secret*). Mit diesem Secret kann Bob den HTLC auf Blockchain α für sich claimen. Dazu sendet er eine Transaktion mit der Funding-Transaktion von Alice auf Blockchain α als Input und den entsprechenden Parametern einschließlich des Secrets (*Create Redeem Tx*).

Insgesamt wurden also zwei Nachrichten[20] offchain gesendet und vier Transaktionen onchain, um den gesamten Atomic Swap zu vollziehen.

[19] Je nachdem, welcher Pfad des HTLC ausgeführt werden soll.

[20] Plus den Parameteraustausch, welcher abhängig von eventuellen Unstimmigkeiten mehrere Nachrichten umfassen kann.

3.6. Protokollanalyse

Um die Sicherheit und Korrektheit von Atomic Swaps zu untersuchen, wird in diesem Kapitel eine Protokollanalyse durchgeführt. Zusätzlich wird dann auf weitere mögliche Angriffsvektoren eingegangen.

3.6.1. Burrows-Abadi-Needham-Logik

Die Burrows-Abadi-Needham-Logik ist eine Modallogik zur Analyse von kryptographischen Protokollen [6]. Sie formalisiert den Protokollablauf, wodurch fehlerhafte Annahmen einfach erkannt werden können.

In dieser Protokollanalyse wird ein Subset der von Burrows et al. postulierten Konstrukte verwendet:

1. $\xrightarrow{K} P$: P verwendet K als öffentlichen Schlüssel.
2. $P \mid\equiv X$: P glaubt X.
3. $B \triangleleft X$: B sieht die Nachricht X.
4. $\#(X)$: X ist im Zuge des aktuellen Protokolldurchlaufes noch nicht gesendet worden.

Aus diesen Konstrukten lassen sich Axiome der Form $\frac{A_1,\ldots,A_n}{F}$ erstellen, wobei $A_1, \ldots, A_n$ Prämissen darstellen und F die Schlussfolgerung ist.

Durchführung

Durch den Parameteraustausch wissen beide Parteien, welcher Public-Key-Hash der anderen Partei gehört.

$$A \mid\equiv \xrightarrow{PKH_B} B \tag{3.1}$$

$$B \mid\equiv \xrightarrow{PKH_A} A \tag{3.2}$$

Alice erstellt den HTLC A mit den vereinbarten Parametern und berechnet daraus den Scripthash A. Diesen benutzt sie für ihre Funding-Transaktion, die sie auf Blockchain α veröffentlicht.

$$A \rightarrow \alpha : \{Funding - Tx\} \tag{3.3}$$

Diese Transaktion sieht Bob in einem neu erstellten Block auf Blockchain α.

$$\frac{B \triangleleft Funding - Tx, B \mid\equiv \xrightarrow{PKH_A} A, B \triangleleft \#(Block_x)}{B \mid\equiv Funding - Tx} \tag{3.4}$$

Durch die Signatur der Funding-Transaktion kann Bob überprüfen, dass die Transaktion tatsächlich von Alice kommt. Dadurch ist die Funding-Transaktion für Bob glaubhaft und zusätzlich kennt er nun den Scripthash A.

Danach sendet Alice Bob den HTLC A im Klartext.

$$A \rightarrow B : \{HTLC_A\} \tag{3.5}$$

Bob kann nun den Hash des HTLC A berechnen und mit dem Scripthash A aus der Funding-Transaktion vergleichen.

$$\frac{B \triangleleft HTLC_A, B \mid\equiv Scripthash_A}{B \mid\equiv HTLC_A} \tag{3.6}$$

Stimmen die beiden Hashes überein, kann Bob auf die Korrektheit des HTLC A vertrauen. Dadurch kann Bob aus diesem die verwendeten Parameter auslesen und überprüfen. Somit erhält er auch Kenntnis über den Secrethash.

Danach findet der bisherige Ablauf erneut mit vertauschten Parteien statt. Zuletzt erfolgt das Claimen der HTLCs.

Dazu sendet Alice nun eine Transaktion an Blockchain α, welche öffentlich einsehbar das Secret enthält.

$$A \rightarrow \beta : \{Redeem - Tx\} \tag{3.7}$$

Wieder wird diese Transaktion von Bob auf der Blockchain gesehen.

$$\frac{B \triangleleft Redeem - Tx, B \mid\equiv \xrightarrow{PKH_A} A, B \triangleleft \#(Block_y)}{B \mid\equiv Redeem - Tx} \tag{3.8}$$

Da die Transaktion sich erneut in einem neuen Block befindet und Bob wieder Alice' Signatur überprüfen kann, ist auch diese Transaktion für ihn glaubhaft. Aufgrund der Validität der Transaktion[21] kann Bob sicher sein, dass auch das Secret korrekt ist. Dieses kann er nun selbst verwenden, um eine Redeem-Transaktion zu erstellen.

Da Alice bereits die ihr zustehenden Coins erhalten hat, ist die Redeem-Transaktion von Bob für die korrekte Ausführung des Protokolls uninteressant. Dadurch ist hiermit das Protokoll abgeschlossen.

3.6.2. Schwachstellen und mögliche Angriffsvektoren

Nachfolgend werden einige Angriffsmöglichkeiten auf ihre Durchführbarkeit und Schwachstellen auf ihr Potenzial analysiert. Bei allen Angriffen wird davon ausgegangen, dass der Angreifer (Mallory) rational handelt.

Impersonation-Angriff

Bei einem Impersonation-Angriff auf einen Atomic Swap zwischen Alice und Bob versucht Mallory, sich als Bob auszugeben, ohne dass Alice es merkt, und damit Alice oder Bob Schaden zuzufügen. Diese Impersonation kann dabei an verschiedenen Stellen im Protokoll unternommen werden.

1. Gleich zu Beginn des Protokolls: Mallory gibt sich von Anfang an als Bob aus und führt den Parameteraustausch mit Alice durch. Dabei muss sie ihre Public-Key-Hashes angeben. Da der Werteaustausch zwischen den zugehörigen Public-Keys stattfindet, wäre es nicht rational sinnvoll, andere Public-Key-Hashes anzugeben. Aufgrund der vom Protokoll geforderten Transaktionen auf den gewählten Blockchains, ist Mallory gezwungen, solche Transaktionen von ihrem Account zu versenden. Tut sie dies nicht, fällt das sofort durch das Fehlen der zweiten Funding-Transaktion auf und Alice kann an der Stelle abbrechen, indem sie dem Protokoll nicht weiter folgt und nach der Locktime ihr Geld zurückholt.

[21] Sonst wäre die Transaktion nicht in einen Block aufgenommen worden

Folgt Mallory dem Protokoll weiterhin, muss sie die Funding-Transaktion erstellen, wofür sie den HTLC B erstellen muss. Da das Protokoll auch das Senden des HTLCs an Alice erfordert, muss Mallory den der Transaktion zugrunde liegenden HTLC B ihr zuschicken. Erhält Alice den HTLC B nicht, bricht sie wie im obigen Fall ab. Wenn Alice den HTLC B erhalten hat, kann sie die Parameter überprüfen. Selbst wenn Mallory über die Parameter lügen würde (z.B. ein falscher Public-Key-Hash), fällt Alice dies auf, da sie den Scripthash des HTLC selbst berechnen kann und mit der Funding-Transaktion in der Blockchain vergleichen kann. Sind die Angaben von Mallory nicht korrekt, bricht Alice wie oben ab. Ist jedoch alles in Ordnung, erstellt Alice ihre Redeem-Transaktion und erhält dadurch das ihr zustehende Geld. Da Bob nie involviert war, wurde keiner Partei geschadet. Nur hat Alice ihre Coins nicht mit Bob sondern mit Mallory getauscht. Da Atomic Swaps aber Grundsätzlich für den Werteaustausch zwischen zwei sich nicht vertrauenden Parteien bestimmt sind, macht dies für Alice keinen Unterschied.

2. Später: Sobald Alice und Bob den Parameteraustausch durchgeführt haben, müsste Mallory für einen Impersonation-Angriff weiterhin Bobs Public-Key-Hashes verwenden. Dadurch wäre es ihr nicht möglich, an das Geld von Alice zu kommen. Daher ist ein späterer Angriff nicht sinnvoll.

Somit ist ein Impersonation-Angriff auf einen Atomic Swap nicht sinnvoll möglich.

Abspaltung vom Netzwerk

Da Blockchains dezentral über ein Peer-2-Peer-Netzwerk agieren, kommt es gelegentlich vor, dass Teile des Netzwerkes die Verbindung zu den restlichen Knoten verlieren. So können sich kurzzeitg kleine Inseln bilden, die ihre eigene Blockchain besitzen[22].

Dies ist eine Schwachstelle des Blockchain-Protokolls, die auch bei Atomic Swaps zum Tragen kommen kann. Verliert Bob die Verbindung, nachdem er seinen HTLC B erstellt hat und bevor er seine Redeem-Transaktion senden konnt, kann Alice diesen HTLC für sich beanspruchen und nach Ablauf der Locktime des HTLC A das dort gesperrte Geld zurückholen. Fällt Bobs Verbindung vorher aus, kommt es nicht zum Austausch; fällt sie danach aus, macht Bob keinen Verlust.

Alice ist von potentiellen Abspaltungen nicht bedroht. Sie kann keine Coins verlieren.

Netzwerk-Congestion

Der Begriff „Netzwerk-Congestion“ beschreibt die Überlastung der Blockchain durch zu viele Transaktionen. Abhängig von der Blockgröße der betrachteten Blockchain kann nur eine bestimmte Anzahl an Transaktionen in einem Block verarbeitet werden. Werden aber mehr Transaktionen gesendet, dauert es immer länger, bis eine Transaktion verarbeitet wurde. Durch erhöhte Transaktionsgebühren kann man die Bearbeitung eigener Transaktionen priorisieren, allerdings kann das zum einen jeder Teilnehmer im Netzwerk und zum anderen werden Transaktionen dadurch natürlich teurer.

Sollte das Netzwerk also überlastet sein, kann es dazu kommen, dass eine Redeem-Transaktion zu viel Zeit benötigt und dadurch erst nach Ablauf der Locktime verarbeitet wird. Dadurch ist sie ungültig und der andere Teilnehmer kann die Coins zurückholen.

Diese Gefahr bedroht sowohl Alice als auch Bob. Sollte Alice ihre Redeem-Transaktion losgeschickt haben, während das Netzwerk überlastet ist, wird diese Transaktion nicht verarbeitet.

[22] Natürlich bleibt die Historie gleich, doch die neuesten Blocks sind nicht die gleichen wie in der Mainchain. Natürlich gilt dies nur, falls die Insel genug Hashing-Power aufbringen kann, um neue Blöcke zu minen.

Da sich die Transaktion aber im öffentlichen Mempool[23] befindet, kann Bob trotzdem das Secret auslesen. Schafft Bob es nun (beispielsweise durch erhöhte Transaktionsgebühren), seine Redeem-Transaktion verarbeiten zu lassen, erhält er das Geld aus dem HTLC A und kann nach Ablauf der Locktime auch die Coins aus HTLC B zurückholen. Damit hat Alice ihre Coins verloren.

Ist Alice' Transaktion erfolgreich und das Netzwerk wird erst danach überlastet, so wird Bobs Redeem-Transaktion nicht verarbeitet. Nachdem die Locktime des HTLC A abgelaufen ist, kann Alice also auch aus diesem die Coins rausziehen. Somit hat Bob sein Geld verloren.

Denial of Service

Durch Denial-of-Service-Attacken (DoS-Attacken) kann das Gleiche erreicht werden wie durch Netzwerk-Congestion. Wenn Bob einen eigenen Netzwerk-Knoten betreibt, der auch von außen Anfragen entgegennimmt, oder Alice den Remote-Knoten kennt, über den Bob seine Transaktionen verarbeiten lässt, könnte Alice den entsprechenden Knoten zum richtigen Zeitpunkt mit einer DoS-Attacke angreifen. Dadurch befindet sich Bob in der gleichen Situation wie bei einer Netzwerk-Congestion[24]. Ebenso kann auch die Blockchain selbst durch einen DoS-Angriff überlastet werden.

Diese Angriffsmöglichkeit bietet sich genauso auch für Bob gegen Alice an.

Doppelverwendung des Secrets

Hat Alice bereits einen Atomic Swap mit Bob durchgeführt und möchte nun Geld mit Charlie tauschen, sollte sie ein neues Secret verwenden. Tut sie dies nicht, kann Charlie einfach eine Redeem-Transaktion an den HTLC A schicken, noch bevor er seinen HTLC C überhaupt erstellt hat.

Somit hat Alice ihr Geld verloren und Charlie musste sogar nur eine statt zwei Transaktionen schicken.

3.7. Rahmenbedingungen

Eine grundlegende Bedingung für einen Atomic Swap ist das Wissen über die Erstellung und Verwendung von HTLCs. Ob dies über eine Software oder per Hand geschieht, ist nicht relevant. Beiden Parteien muss es darüber hinaus auch möglich sein, den HTLC der anderen Partei zu überprüfen. Weiterhin muss Partei B in der Lage sein, Transaktionen auf der Blockchain β zu sehen, um das Secret aus der Transaktion von A auslesen zu können.

Um zu verhindern, dass eine Partei versucht, die andere zu hintergehen, müssen die Locktimes richtig gesetzt werden. Die Locktime des ersten HTLCs muss dabei über die Locktime des zweiten HTLCs deutlich hinausgehen. Dies wird so gehandhabt, damit B genug Zeit hat, nach der Redeem-Transaktion von A ihre eigene Redeem-Transaktion zu senden. Da nicht jeder Bitcoin-Nutzer immer online ist und die Bestätigung von Transaktionen unter Umständen recht lange dauern kann, sollte die Locktime also mehrere Stunden umfassen.

Hinzu kommt, dass Abspaltungen vom Netzwerk und Netzwerk-Congestion (siehe Kapitel 3.6.2) die Verarbeitungsgeschwindigkeit von Transaktionen zusätzlich verringen können. Um also beiden Parteien eine angemessene Zeit einzuräumen, sollte die Locktime des ersten HTLCs auf mindestens 48 Stunden gesetzt werden und die des zweiten HTLCs auf 24 Stunden.

Als letzter Punkt sollte ein gutes Secret gewählt werden. So sollte es zufällig sein (am Besten pseudo-random) und eine gewisse Länge aufweisen.

[23] Jeder Miner speichert sich alle noch nicht verarbeiteten Transaktionen im sog. Mempool. Für die Berechnung eines neuen Blocks sucht sich der Miner beliebige Transaktionen aus dem Mempool und verarbeitet sie in dem

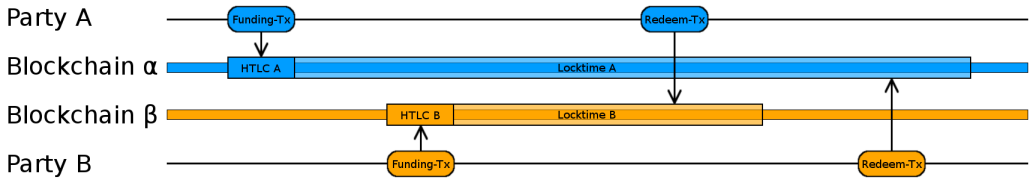

Abbildung 3.6.: Zeitlicher Ablauf eines Atomic Swaps.

Wie in Kapitel 3.6.2 gezeigt, sollte jedes Secret auch nur einmal verwendet werden.

3.8. Skripte für HTLCs

Im Rahmen dieser Arbeit wurden mehrere Blockchains nach HTLCs durchsucht. Dabei wurden mehrere verschiedene Skripte gefunden, die die Funktionalität eines HTLC umsetzen. Diese verschiedenen Typen sind im Anhang A zu finden.

Sie folgen alle dem gleichen Schema: Eine If-Verzweigung, welche zwischen dem Hashlock und dem Timelock unterscheidet und die entsprechenden Werte überprüft.

In manchen der HTLCs sind die Public-Key-Hashes hinterlegt. Daher werden die mit der Transaktion mitgelieferten Public-Keys gehasht und dann mit den vordefinierten Public-Keys-Hashes verglichen.

Andere HTLCs benutzen nur die Public-Keys ohne diese zu hashen. Dies hat allerdings keine Auswirkung auf die Ausführung[25].

Zuletzt wird überprüft, ob eine gültige Signatur zu diesem Public-Key vorhanden ist.

Der größte Unterschied zwischen einigen der gefundenen HTLC-Typen ist der benutzte Hashing-Algorithmus. Das Bitcoin-Skript bietet dafür SHA256, Ripemd160 und Hash160. Decred benutzt zusätzlich noch Blake256. Die nachgestellte Zahl gibt jeweils die Länge des Outputs in Bits an. Hash160 ist eine Bitcoin-eigene Erfindung, welche einfach die Kombination aus SHA256 gefolgt von Ripemd160 darstellt. Sie wird meist für das Hashen von Public-Keys verwendet.

Davon abgesehen sind die verschiedenen Typen sehr ähnlich. Durch unterschiedliche Reihenfolgen in den Opcodes sind manche Typen etwas kürzer und manche etwas länger. So variieren alle Typen von 77 bis 116 Bytes bei einfachem Secret. Dies hat einen geringen Einfluss auf die Transaktionskosten und den Ausführungsaufwand.

Einige abweichende HTLCs gibt es allerdings doch. Zwei der gefundenen Typen (Type 8a und 8b) verwenden statt nur einem Secret gleich 15 Secrets. Bei den Typen 2, 6a und 6b wird das Secret nicht bloß auf einer Seite benötigt, sondern unabhängig von der claimenden Fraktion angefragt. Typ 18 benutzt hingegen für beide Ausführungspfade zwei verschiedene Secrets. Die Typen 19a und b und der Typ 20 implementieren neben der HTLC-Funktionalität noch zusätzlich ein Multi-Signature-Wallet[26].

neuen Block. Als Lohn erhält er die Transaktionsgebühren.

[24] Es ist zwar keine Netzwerk-Congestion, da die Blockchain normal weiterläuft, aber für Bob hat es die gleichen Folgen.

[25] Wie in Kapitel 2.2 beschrieben ist, bieten Transaktionen mit Public-Keys mehr Angriffsvektoren als die Verwendung von Public-Keys-Hashes.

[26] Ein Multi-Signature-Wallet stellt einen Smart Contract dar, der eine festgelegte Anzahl der vorher bestimmten Public-Keys erfordert, um verwendet zu werden. So können beispielsweise drei von fünf Keys gefordert sein, um eine Transaktion auslösen zu können.

Vermutlich sind die verschiedenen Typen der Dezentralität der Communities geschuldet. Durch fehlende Abstimmung unter den Entwicklern entstanden unterschiedlichen Implementierungen.

3.9. Finden und Zusammenfügen von HTLCs

Da Atomic Swaps aus mehreren Transaktionen bestehen, ist es nicht möglich, in der Blockchain direkt nach einem Atomic Swap zu suchen. Vielmehr muss nach HTLCs gesucht werden. Dafür wurde ein Programm in Golang geschrieben, welches eine Blockchain nach Input-Skripten durchsucht. Dabei wurde nach allen Skripten gesucht, die ein Timelock (`OP_CHECKLOCKTIMEVERIFY` oder `OP_CHECKSEQENCEVERIFY` und eine Hash-Funktion beinhalten. Diese wurden dann per Hand analysiert, welche Funktionalität sie tatsächlich implementieren. Über ein weiteres Go-Programm wurden gleiche Skripte als HTLCs eines Typen gruppiert und diese Typen dokumentiert. Mit Hilfe der identifizierten Struktur dieser Typen können dann alle vorher gefundenen Skripte typisiert werden. Dabei werden zu einem passenden Skript die folgenden Metadaten gespeichert:

1. Die Blockchain
2. Die Blocknummer
3. Der Zeitstempel des Blockes
4. Der Transaktionshash des HTLCs
5. Der Transaktionshash der vorherigen Transaktion.[27]
6. Die Anzahl der transferierten Coins
7. Der HTLC-Typ
8. Das Timelock
9. Die Public-Key-Hashes
10. Das/die Secret(s)
11. Der/die Secrethash(es)

Dann kann versucht werden, ein Matching zwischen zwei HTLCs zu finden. Um ein Matching zu ermöglichen, müssen die Parameter der HTLCs verglichen werden. Dabei ist der wichtigste Parameter das Secret bzw. der Secrethash. Da das Secret für jeden Atomic Swap neu gewählt werden und möglichst zufällig sein sollte, ist es unwahrscheinlich, dass es zwei Atomic Swaps mit dem gleichen Secret gibt. Um diese Wahrscheinlichkeit weiter zu senken, kann zusätzlich zum Secret auch der Zeitstempel der Redeem-Transaktionen verwendet werden. Da das Zeitfenster von HTLCs meist nur einige Stunden beträgt, ist es noch unwahrscheinlicher, dass in diesem Zeitfenster mehrere Atomic Swaps durchgeführt wurden, die das gleiche Secret verwenden. Dazu wurde wiederum ein Go-Programm geschrieben, welches jeweils für zwei der gewählten Blockchains über die dort gefundenen HTLCs iteriert und diese paarweise vergleicht. Passen die Secrets, Secrethashes und die Zeitstempel der Transaktionen (mit einer Varianz von einem Tag) überein, wird dies als Atomic Swap gewertet und abgespeichert.

[27] Die Transaktion, dessen Output durch den HTLC benutzt wurde.

4. Ergebnisse

Mit Hilfe der in Kapitel 3.9 beschriebenen Programme wurden im Rahmen dieser Arbeit alle gewählten Blockchains nach HTLCs durchsucht. Dabei wurde bei den aktuellsten Blöcken angefangen und sich dann rückwärts durch die Blockchain gearbeitet. Da Atomic Swaps erst im Jahr 2017 aufgekommen sind, wurden die Blockchains bis Anfang 2017 durchsucht. BCH wurde nur bis August 2017 gecrawlt, da es sich an diesem Zeitpunkt von Bitcoin abspaltete, wodurch alles davor identisch zu der BTC-Blockchain ist. Dabei wurden insgesamt 2200 HTLCs gefunden.

Typ	BTC	BCH	LTC	DCR
Typ 1a	1	0	0	0
Typ 1b	1	0	0	0
Typ 2	0	23	6	0
Typ 3a	2	2	9	3
Typ 3b	7	0	0	0
Typ 3c	87	3	223	0
Typ 4	1	1	0	0
Typ 5a	10	6	18	12
Typ 5b	7	3	0	0
Typ 6a	2	0	0	0
Typ 6b	22	0	0	0
Typ 7	0	2	0	0
Typ 8a	35	0	0	0
Typ 8b	18	0	0	0
Typ 9a	211	152	85	0
Typ 9b	79	16	19	0
Typ 10a	4	0	31	0
Typ 10b	0	0	32	0
Typ 11	2	0	0	0
Typ 12	18	0	200	0
Typ 13	0	1	0	0
Typ 14	5	10	295	0
Typ 15	12	0	1	0
Typ 16	0	3	0	0
Typ 17	2	0	0	0
Typ 18	91	28	27	0
Typ 19a	0	1	0	0
Typ 19b	0	370	0	0
Typ 19c	0	1	0	0
Typ 20	0	1	0	0
Gesamt	617	622	946	15

Tabelle 4.1.: Die gefundenen HTLCs gestaffelt nach Blockchain und Typ.

Diese sind in Tabelle 4.1 aufgeführt, aufgeteilt in die verschiedenen gefundenen Typen und die

Blockchain, auf der sie genutzt wurden. Hier werden mehrere Dinge ersichtlich. Zum einen sind auf der Bitcoin-Blockchain die meisten verschiedenen Typen zu finden. Das liegt wahrscheinlich an ihrer Vorreiterrolle als größte und bekannteste Kryptowährung. Hinzu kommt, dass die Bitcoin-Core-Entwickler sehr interessiert an neuen und vielversprechenden Entwicklungen sind. So werden aktuell im Rahmen des Lightning-Networks[28] Payment-Channels entwickelt, welche ebenfalls HTLCs nutzen.

Als nächstes scheint BCH neben dem auch bei den anderen Chains verbreiteten Typ 9a hauptsächlich auf HTLCs des Typs 19b zu setzen. Im Vergleich zu den anderen HTLCs stellt der Typ 19 (a, b, c) neben der HTLC-Funktionalität auch noch ein Multi-Signature-Wallet bereit.

Des Weiteren sind HTLCs auf LTC offensichtlich sehr beliebt. Dies liegt vermutlich an der kürzeren Blockzeit im Vergleich zu BTC und BCH, was das Entwickeln und Testen von HTLCs angenehmer gestaltet. Ebenfalls könnte der Preis eine Rolle spielen. Da der LTC-Preis deutlich unter dem des BTC liegt, sind auch die Transaktionsgebühren auf LTC niedriger als bei Bitcoin.

Als letztes ist festzustellen, dass auf DCR nur zwei Typen benutzt werden. Da diese Typen keine Besonderheiten gegenüber den anderen Typen aufweisen, ist nicht ersichtlich, warum nur diese Typen verwendet werden.

Das letzte entwickelte Programm versuchte dann, diese HTLCs miteinander zu verbinden. Dies erfolgte über das verwendete Secret und die zeitliche Nähe der beiden HTLCs. Dadurch wurden 102 verschiedene Atomic Swaps gefunden. Die gefundenen Atomic Swaps sind in der nachfolgenden Tabelle aufgeführt.

Typ	BTC-LTC	BTC-BCH	BTC-DCR	LTC-BCH	LTC-DCR
Typ 3a	0	0	0	0	1
Typ 3c	75	0	0	0	0
Typ 4	0	1	0	0	0
Typ 5a	7	0	3	0	2
Typ 5b	0	6	0	0	0
Typ 12	3	0	0	0	0
Typ 14	0	0	0	3	0
Typ 15	1	0	0	0	0
Gesamt	86	7	3	3	3

Tabelle 4.2.: Die gefundenen Atomic Swaps aufgeteilt nach Blockchain-Paaren und Typ.

Die meisten Atomic Swaps fanden in Verbindung mit BTC statt und ein Großteil ist vom Typ 3c. Abseits von BTC fanden nur sehr wenige der gefundenen Atomic Swaps statt.

Da für einen Atomic Swap immer zwei HTLCs benötigt werden, ergibt sich eine Differenz von 1996 nicht gematchten HTLCs. Diese können entweder in Verbindung mit nicht untersuchten Blockchains stattgefunden haben, Teile von Payment-Channels[29] sein oder in ihrer Ausführung gescheitert sein.

Es fällt auf, dass viele der gefundenen HTLC-Typen nicht bei den gefundenen Atomic Swaps vertreten sind. So kommen nur acht verschiedene Typen bei den Atomic Swaps vor, obwohl 30 HTLC-Typen identifiziert wurden. Möglicherweise wurden manche der Typen für bestimmte Blockchain-Kombinationen entworfen, die in dieser Arbeit jedoch nicht untersucht wurden. Oder diese HTLCs wurden nicht für Atomic Swaps genutzt.

Neben den untersuchten Blockchains existieren noch weitere kompatible Protokolle. Die meisten Bitcoin-Forks und -Kopien können die vorgestellten HTLCs ebenfalls nutzen. Für andere

[28] Siehe `https://en.bitcoin.it/wiki/Lightning_Network`
[29] Siehe `https://en.bitcoin.it/wiki/Payment_channels`

Blockchain-Protokolle mit Smart-Contract-Funktionalität sähe der Ablauf eines Atomic Swaps zwar gleich aus, aber die technische Umsetzung der HTLCs müsste angepasst werden.

Umfangreichere Smart-Contract-Sprachen erlauben deutlich vielfältigere HTLCs, wodurch die Suche nach solchen HTLCs erschwert wird. Bei der Programmiersprache Solidity, welche für Ethereum-Smart-Contracts verwendet wird, könnten so hunderte verschiedene HTLCs entwickelt werden, die jeweils leicht unterschiedliche Implementierungen darstellen. Somit wäre die Suche nach HTLCs bei Ethereum sehr viel aufwändiger als bei BTC-ähnlichen Blockchains.

Interessant ist allerdings, dass die ersten bekannten Atomic Swaps am 19. September 2017 zwischen LTC und DCR[30] und am 22. September 2017 zwischen BTC und LTC[31] stattfanden. Im Rahmen dieser Arbeit wurden allerdings mehrere Atomic Swaps aus dem April 2017 gefunden. Der älteste ist in Abbildung 4.1 dargestellt.

[30] Siehe `https://blog.altcoin.io/the-evolution-of-atomic-swaps-e33ad3af8818?gi=e59f9e73505b`

[31] Siehe `http://www.cryptovibes.com/crypto-news/charlie-lees-atomic-swap-between-litecoin-and-bitcoin-was-a-success/`

```
{
  "HTLC1": {
    "chain": "btc",
    "block": 462707,
    "timestamp": "2017-04-20 13:35:05 +0000 UTC",
    "transaction": "
       f2260dbf59ffad0c58682317041d95ccb162b1785637ca531949a9c66d4c6804",
    "input_tx": "506
       c4c157a0b8f6a860937fa0318896449cbe1342e89c223c6ea961358028d0a",
    "input_value": 0.0056,
    "type": "Type15",
    "timelock": "4",
    "pub_key_hashes1": [
      "9ba8acbfee73df9344d23e62c6a2f538a1077270"
    ],
    "pub_key_hash2": "4e186b45c42c61a04b1c99c306a6929ef4e7268d",
    "secrets": [
      "0226e65bd1c79f3445c08e3586d5c235cd785c595cb0b743c4c3b389f861bc4a5e"
    ],
    "secret_hashes": [
      "99caa9d9e3ba9945c0190692c9376a76ac22504f"
    ]
  },
  "HTLC2": {
    "chain": "ltc",
    "block": 1189306,
    "timestamp": "2017-04-20 13:41:06 +0000 UTC",
    "transaction": "006
       e3b75e608662fb747b2f44eadefb52ade0859e3cb629b46679995812f2183",
    "input_tx": "
       d63eca6bc6f8bb76393cde0f1ad0c0ae0a1ad3382ba53f7ca666ea31112da354",
    "input_value": 0.0706,
    "type": "Type15",
    "timelock": "4",
    "pub_key_hashes1": [
      "4e186b45c42c61a04b1c99c306a6929ef4e7268d"
    ],
    "pub_key_hash2": "9ba8acbfee73df9344d23e62c6a2f538a1077270",
    "secrets": [
      "0226e65bd1c79f3445c08e3586d5c235cd785c595cb0b743c4c3b389f861bc4a5e"
    ],
    "secret_hashes": [
      "99caa9d9e3ba9945c0190692c9376a76ac22504f"
    ]
  }
}
```

Abbildung 4.1.: Der älteste gefundene Atomic Swap.

5. Zusammenfassung

In Kapitel 2 wurden alle nötigen Grundlagen zu Bitcoin und der Blockchain-Technologie erklärt. Dann wurde in Kapitel 3 zuerst die Funktionsweise von verschiedenen Bitcoin-Skripten erläutert und deren Zusammenspiel in einem Atomic Swap dargelegt. So erlauben Atomic Swaps den atomaren Austauch von monetären Werten unterschiedlicher Blockchains, ohne dass sich die beteiligten Parteien gegenseitig vertrauen müssen.

Anschließend wurde in Kapitel 3.6 eine Protokollanalyse mit Hilfe der BAN-Logik durchgeführt. Zusätzlich wurden Atomic Swaps auf weitere Angriffsvektoren untersucht. Im Rahmen dieser Untersuchung wurde festgestellt, dass Atomic Swaps aufgrund von Schwachstellen der zugrundeliegenden Blockchain-Technologie angreifbar sind. So können Angriffe auf die Konnektivität des Opfers dazu führen, dass es nicht mehr in der Lage ist, dem Protokoll zu folgen und dadurch seine monetären Werte verliert. Angriffe, die dabei auf das Protokoll der Atomic Swaps abzielen, haben sich jedoch als nicht effektiv herausgestellt, solange das Protokoll von allen anderen Parteien weiterhin ordnungsgemäß durchgeführt wird. Weiterhin wurden allgemeine Rahmenbedingungen abgeleitet, die eingehalten werden sollten, damit die strukturelle Sicherheit eines Atomic Swaps nicht gefährdet wird.

Anschließend wurde in Kapitel 3.9 aufgezeigt, wie HTLCs gefunden und daraufhin miteinander zu Atomic Swaps gematcht werden können. Diese Suche wurde dann auf vier ausgewählten Blockchains durchgeführt: Bitcoin, Bitcoin Cash, Litecoin und Decred. In Kapitel 4 schließlich wurden die Ergebnisse der durchgeführten Suche dargestellt.

Hier wurde deutlich, dass es sehr viele verschiedene Arten gibt, einen HTLC zu implementieren. So wurden 30 verschiedene Implementierungen entdeckt. Zusätzlich konnte gezeigt werden, dass sich HTLCs miteinander matchen lassen, um somit stattgefundene Atomic Swaps zu erkennen. Auf die entdeckten HTLC-Typen verteilen sich 2200 gefundene HTLCs, welche zu 102 Atomic Swaps gematcht werden konnten. Somit konnte ein grober Überblick über Atomic Swaps zwischen Bitcoin-ähnlichen Protokollen geschaffen werden.

Da es neben den gewählten Blockchains noch viele weitere direkt oder indirekt kompatible Blockchain-Protokolle gibt, kann eine breitere Analyse Aufschluss über viele der ungematchen HTLCs geben. Des Weiteren ermögliche auch die Untersuchung höherer Smart-Contract-Programmiersprachen wie beispielsweise Solidity (Ethereum) einen erweiterten Überblick über den aktuellen Stand von Atomic Swaps.

A. Anhang

```
1  63  OP_IF
2  a8  OP_SHA256
3  20  OP_DATA_32
4    <secret_hash 32bytes>
5  88  OP_EQUALVERIFY
6  21  OP_DATA_33
7    <pubkey1 33bytes>
8  67  OP_ELSE
9  54  OP_4
10   <locktime op>
11 b1  OP_CHECKLOCKTIMEVERIFY
12 75  OP_DROP
13 21  OP_DATA_33
14   <pubkey2 33bytes>
15 68  OP_ENDIF
16 ac  OP_CHECKSIG
```

Abbildung A.1.: Typ 1a mit SHA256-Secret, Public-Keys, einem Opcode für die Locktime, 12 Opcodes und einer Länge von 110 Bytes.

```
1  63  OP_IF
2  a8  OP_SHA256
3  20  OP_DATA_32
4    <secret_hash 32bytes>
5  87  OP_EQUAL
6  69  OP_VERIFY
7  21  OP_DATA_33
8    <pubkey1 33bytes>
9  ac  OP_CHECKSIG
10 67  OP_ELSE
11 04  OP_DATA_4
12   <locktime 4bytes>
13 b1  OP_CHECKLOCKTIMEVERIFY
14 75  OP_DROP
15 21  OP_DATA_33
16   <pubkey2 33bytes>
17 ac  OP_CHECKSIG
18 68  OP_ENDIF
```

Abbildung A.2.: Typ 1b mit SHA256-Secret, Public-Keys, 14 Opcodes und einer Länge von 116 Bytes.

```
 1  76  OP_DUP
 2  a8  OP_SHA256
 3  20  OP_DATA_32
 4    <secret_hash 32bytes>
 5  87  OP_EQUAL
 6  63  OP_IF
 7  75  OP_DROP
 8  21  OP_DATA_33
 9    <pubkey1 33bytes>
10  67  OP_ELSE
11  03  OP_DATA_3
12    <locktime 3bytes>
13  b1  OP_CHECKLOCKTIMEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkeyhash2 20bytes>
19  88  OP_EQUALVERIFY
20  68  OP_ENDIF
21  ac  OP_CHECKSIG
```

Abbildung A.3.: Typ 2 mit SHA256-Secret, Public-Key im Hashlock und Public-Key-Hash im Timelock, einem gemeinsamen Secret für beide Pfade, 3-Byte-Locktime, 17 Opcodes und einer Länge von 105 Bytes.

```
 1  63  OP_IF
 2  a8  OP_SHA256
 3  20  OP_DATA_32
 4    <secret_hash 32bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  67  OP_ELSE
11  04  OP_DATA_4
12    <locktime 4bytes>
13  b1  OP_CHECKLOCKTIMEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkey_hash2 20bytes>
19  68  OP_ENDIF
20  88  OP_EQUALVERIFY
21  ac  OP_CHECKSIG
```

Abbildung A.4.: Typ 3a mit SHA256-Secret, Public-Key-Hashes, 17 Opcodes und einer Länge von 93 Bytes.

```
 1  63  OP_IF
 2  a8  OP_SHA256
 3  20  OP_DATA_32
 4    <secret_hash 32bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  88  OP_EQUALVERIFY
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  04  OP_DATA_4
14    <locktime 4bytes>
15  b1  OP_CHECKLOCKTIMEVERIFY
16  75  OP_DROP
17  76  OP_DUP
18  a9  OP_HASH160
19  14  OP_DATA_20
20    <pubkey_hash2 20bytes>
21  88  OP_EQUALVERIFY
22  ac  OP_CHECKSIG
23  68  OP_ENDIF
```

Abbildung A.5.: Typ 3b mit Sha256-Secret, Public-Key-Hashes, 19 Opcodes und einer Länge von 95 Bytes.

```
 1  63  OP_IF
 2  82  OP_SIZE
 3  01  OP_DATA_1
 4    <secret_length 1byte>
 5  88  OP_EQUALVERIFY
 6  a8  OP_SHA256
 7  20  OP_DATA_32
 8    <secret_hash 32bytes>
 9  88  OP_EQUALVERIFY
10  76  OP_DUP
11  a9  OP_HASH160
12  14  OP_DATA_20
13    <pubkey_hash1 20bytes>
14  67  OP_ELSE
15  04  OP_DATA_4
16    <locktime 4bytes>
17  b1  OP_CHECKLOCKTIMEVERIFY
18  75  OP_DROP
19  76  OP_DUP
20  a9  OP_HASH160
21  14  OP_DATA_20
22    <pubkey_hash2 20bytes>
23  68  OP_ENDIF
24  88  OP_EQUALVERIFY
25  ac  OP_CHECKSIG
```

Abbildung A.6.: Typ 3c mit Sha256-Secret, Public-Key-Hashes, Abfrage der Secret-Size, 20 Opcodes und einer Länge von 96 Bytes.

```
 1  74  OP_DEPTH
 2  52  OP_2
 3  87  OP_EQUAL
 4  63  OP_IF
 5  a6  OP_RIPEMD160
 6  14  OP_DATA_20
 7    <secret_hash 20bytes>
 8  88  OP_EQUALVERIFY
 9  21  OP_DATA_33
10    <pubkey_hash1 33bytes>
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  04  OP_DATA_4
14    <timelock 4bytes>
15  b1  OP_CHECKLOCKTIMEVERIFY
16  75  OP_DROP
17  21  OP_DATA_33
18    <pubkey_hash2 33bytes>
19  ac  OP_CHECKSIG
20  68  OP_ENDIF
```

Abbildung A.7.: Typ 4 mit Ripemd160-Secret, Public-Keys, Abfrage der Stackgröße, 16 Opcodes und einer Länge von 106 Bytes.

```
 1  63  OP_IF
 2  a6  OP_RIPEMD160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  67  OP_ELSE
11  04  OP_DATA_4
12    <timelock 4bytes>
13  b1  OP_CHECKLOCKTIMEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkey_hash2 20bytes>
19  68  OP_ENDIF
20  88  OP_EQUALVERIFY
21  ac  OP_CHECKSIG
```

Abbildung A.8.: Typ 5a mit Ripemd160-Secret, Public-Key-Hashes, 17 Opcodes und einer Länge von 81 Bytes.

```
 1  63  OP_IF
 2  a6  OP_RIPEMD160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  88  OP_EQUALVERIFY
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  03  OP_DATA_3
14    <timelock 3bytes>
15  b2  OP_CHECKSEQUENCEVERIFY
16  75  OP_DROP
17  76  OP_DUP
18  a9  OP_HASH160
19  14  OP_DATA_20
20    <pubkey_hash2 20bytes>
21  88  OP_EQUALVERIFY
22  ac  OP_CHECKSIG
23  68  OP_ENDIF
```

Abbildung A.9.: Typ 5b mit Ripemd160-Secret, Public-Key-Hashes, 3-Byte-Locktime, 19 Opcodes und einer Länge von 82 Bytes.

```
 1  a6  OP_RIPEMD160
 2  14  OP_DATA_20
 3    <secret_hash 20bytes>
 4  87  OP_EQUAL
 5  63  OP_IF
 6  21  OP_DATA_33
 7    <pubkey1 33bytes>
 8  67  OP_ELSE
 9  04  OP_DATA_4
10    <timelock 4bytes>
11  b1  OP_CHECKLOCKTIMEVERIFY
12  75  OP_DROP
13  21  OP_DATA_33
14    <pubkey2 33bytes>
15  68  OP_ENDIF
16  ac  OP_CHECKSIG
```

Abbildung A.10.: Typ 6a mit Ripemd160-Secret, Public-Keys, einem gemeinsamen Secret für beide Pfade, 12 Opcodes und einer Länge von 102 Bytes.

```
 1  a6  OP_RIPEMD160
 2  14  OP_DATA_20
 3    <secret_hash 20bytes>
 4  88  OP_EQUALVERIFY
 5  21  OP_DATA_33
 6    <pubkey1 33bytes>
 7  87  OP_EQUAL
 8  63  OP_IF
 9  21  OP_DATA_33
10    <pubkey1 33bytes>
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  04  OP_DATA_4
14    <timelock 4bytes>
15  b1  OP_CHECKLOCKTIMEVERIFY
16  75  OP_DROP
17  21  OP_DATA_33
18    <pubkey2 33bytes>
19  ac  OP_CHECKSIG
20  68  OP_ENDIF
```

Abbildung A.11.: Typ 6b mit Ripemd160-Secret, Public-Keys, einem gemeinsamen Secret für beide Pfade und einem Abgleich des ersten Public-Keys, 15 Opcodes und einer Länge von 138 Bytes.

```
 1  63  OP_IF
 2  a6  OP_RIPEMD160
 3  a6  OP_RIPEMD160
 4  a6  OP_RIPEMD160
 5  14  OP_DATA_20
 6    <secret_hash 20bytes>
 7  88  OP_EQUALVERIFY
 8  76  OP_DUP
 9  a9  OP_HASH160
10  14  OP_DATA_20
11    <pubkey_hash1 20bytes>
12  88  OP_EQUALVERIFY
13  ac  OP_CHECKSIG
14  67  OP_ELSE
15  03  OP_DATA_3
16    <timelock 3bytes>
17  b2  OP_CHECKSEQUENCEVERIFY
18  75  OP_DROP
19  76  OP_DUP
20  a9  OP_HASH160
21  14  OP_DATA_20
22    <pubkey_hash2 20bytes>
23  88  OP_EQUALVERIFY
24  ac  OP_CHECKSIG
25  68  OP_ENDIF
```

Abbildung A.12.: Typ 7 mit Ripemd160-Secret, welches dreimal gehasht wird, Public-Key-Hashes, 3-Byte-Locktime, 21 Opcodes und einer Länge von 84Bytes.

```
 1  63  OP_IF
 2  a6  OP_RIPEMD160
 3  14  OP_DATA_20
 4    <secret_hash1 20bytes>
 5  88  OP_EQUALVERIFY
 6  a6  OP_RIPEMD160
 7  14  OP_DATA_20
 8    <secret_hash2 20bytes>
 9  88  OP_EQUALVERIFY
10
11    ...
12
13  a6  OP_RIPEMD160
14  14  OP_DATA_20
15    <secret_hash15 20bytes>
16  88  OP_EQUALVERIFY
17  21  OP_DATA_33
18    <signature1 33bytes>
19  ac  OP_CHECKSIG
20  67  OP_ELSE
21  04  OP_DATA_4
22    <timelock 4bytes>
23  b1  OP_CHECKLOCKTIMEVERIFY
24  75  OP_DROP
25  21  OP_DATA_33
26    <signature2 3bytes>
27  ac  OP_CHECKSIG
28  68  OP_ENDIF
```

Abbildung A.13.: Typ 8a mit 15 Ripemd160-Secrets, Public-Keys, 55 Opcodes und einer Länge von 425 Bytes.

```
 1  74  OP_DEPTH
 2  60  OP_16
 3  87  OP_EQUAL
 4  63  OP_IF
 5  a6  OP_RIPEMD160
 6  14  OP_DATA_20
 7    <secret_hash1 20bytes>
 8  88  OP_EQUALVERIFY
 9  a6  OP_RIPEMD160
10  14  OP_DATA_20
11    <secret_hash2 20bytes>
12  88  OP_EQUALVERIFY
13
14    ...
15
16  a6  OP_RIPEMD160
17  14  OP_DATA_20
18    <secret_hash15 20bytes>
19  88  OP_EQUALVERIFY
20  21  OP_DATA_33
21    <signature1 33bytes>
22  67  OP_ELSE
23  03  OP_DATA_3
24    <timelock 3bytes>
25  b1  OP_CHECKLOCKTIMEVERIFY
26  75  OP_DROP
27  21  OP_DATA_33
28    <signature2 33bytes>
29  68  OP_ENDIF
30  ac  OP_CHECKSIG
```

Abbildung A.14.: Typ 8b mit 15 Ripemd160-Secrets, Public-Keys, Abfrage der Stackgröße, 3-Byte-Locktime, 57 Opcodes und einer Länge von 426 Bytes.

```
 1  63  OP_IF
 2  04  OP_DATA_4
 3    <timelock 4bytes>
 4  b1  OP_CHECKLOCKTIMEVERIFY
 5  75  OP_DROP
 6  21  OP_DATA_33
 7    <pubkey2 33bytes>
 8  ac  OP_CHECKSIG
 9  67  OP_ELSE
10  a9  OP_HASH160
11  14  OP_DATA_20
12    <secret_hash 20bytes>
13  88  OP_EQUALVERIFY
14  21  OP_DATA_33
15    <pubkey1 33bytes>
16  ac  OP_CHECKSIG
17  68  OP_ENDIF
```

Abbildung A.15.: Typ 9a mit Hash160-Secret, Public-Keys, 13 Opcodes und einer Länge von 103 Bytes.

```
 1  63  OP_IF
 2  04  OP_DATA_4
 3    <timelock 4bytes>
 4  b1  OP_CHECKLOCKTIMEVERIFY
 5  75  OP_DROP
 6  21  OP_DATA_33
 7    <pubkey2 33bytes>
 8  ac  OP_CHECKSIG
 9  67  OP_ELSE
10  82  OP_SIZE
11  01  OP_DATA_1
12    <secret_length 1byte>
13  88  OP_EQUALVERIFY
14  a9  OP_HASH160
15  14  OP_DATA_20
16    <secret_hash 20bytes>
17  88  OP_EQUALVERIFY
18  21  OP_DATA_33
19    <pubkey1 33bytes>
20  ac  OP_CHECKSIG
21  68  OP_ENDIF
```

Abbildung A.16.: Typ 9b mit Hash160-Secret, Public-Keys, Abfrage der Secret-Size, 16 Opcodes und einer Länge von 106 Bytes.

```
 1  63  OP_IF
 2  21  OP_DATA_33
 3    <pubkey2 33bytes>
 4  ac  OP_CHECKSIG
 5  54  OP_4
 6    <timelock 1byte>
 7  b2  OP_CHECKSEQUENCEVERIFY
 8  75  OP_DROP
 9  67  OP_ELSE
10  21  OP_DATA_33
11    <pubkey1 33bytes>
12  ad  OP_CHECKSIGVERIFY
13  a9  OP_HASH160
14  14  OP_DATA_20
15    <secret_hash 20bytes>
16  87  OP_EQUAL
17  68  OP_ENDIF
```

Abbildung A.17.: Typ 10a mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, 13 Opcodes und einer Länge von 99 Bytes.

```
 1  63  OP_IF
 2  21  OP_DATA_33
 3    <pubkey1 33bytes>
 4  ad  OP_CHECKSIGVERIFY
 5  a9  OP_HASH160
 6  14  OP_DATA_20
 7    <secret_hash 20bytes>
 8  87  OP_EQUAL
 9  67  OP_ELSE
10  21  OP_DATA_33
11    <pubkey2 33bytes>
12  ac  OP_CHECKSIG
13  54  OP_4
14    <timelock 1byte>
15  b2  OP_CHECKSEQUENCEVERIFY
16  75  OP_DROP
17  68  OP_ENDIF
```

Abbildung A.18.: Typ 10b mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, 13 Opcodes und einer Länge von 99 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  67  OP_ELSE
11  04  OP_DATA_4
12    <timelock 4bytes>
13  b1  OP_CHECKSLOCKTIMEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkey_hash2 20bytes>
19  68  OP_ENDIF
20  88  OP_EQUALVERIFY
21  ac  OP_CHECKSIG
```

Abbildung A.19.: Typ 11 mit Hash160-Secret, Public-Key-Hashes, 17 Opcodes und einer Länge von 81 Bytes.

```
 1  63  OP_IF
 2  03  OP_DATA_3
 3    <timelock 3bytes>
 4  b1  OP_CHECKLOCKTIMEVERIFY
 5  75  OP_DROP
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash2 20bytes>
10  88  OP_EQUALVERIFY
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  76  OP_DUP
14  a9  OP_HASH160
15  14  OP_DATA_20
16    <pubkey_hash1 20bytes>
17  88  OP_EQUALVERIFY
18  ad  OP_CHECKSIGVERIFY
19  a9  OP_HASH160
20  14  OP_DATA_20
21    <secret_hash 20bytes>
22  87  OP_EQUAL
23  68  OP_ENDIF
```

Abbildung A.20.: Typ 12 mit Hash160-Secret, Public-Key-Hashes, 3-Byte-Locktime, 19 Opcodes und einer Länge von 82 Bytes.

```
 1  63  OP_IF
 2  82  OP_SIZE
 3  55  OP_5
 4  88  OP_EQUALVERIFY
 5  a9  OP_HASH160
 6  14  OP_DATA_20
 7    <secret_hash 20bytes>
 8  88  OP_EQUALVERIFY
 9  76  OP_DUP
10  a9  OP_HASH160
11  14  OP_DATA_20
12    <pubkey_hash1 20bytes>
13  67  OP_ELSE
14  04  OP_DATA_4
15    <timelock 4bytes>
16  b1  OP_CHECKLOCKTIMEVERIFY
17  75  OP_DROP
18  76  OP_DUP
19  a9  OP_HASH160
20  14  OP_DATA_20
21    <pubkey_hash2 20bytes>
22  68  OP_ENDIF
23  88  OP_EQUALVERIFY
24  ac  OP_CHECKSIG
```

Abbildung A.21.: Typ 13 mit Hash160-Secret, Public-Key-Hashes, Abfrage der Secret-Size, 20 Opcodes und einer Länge von 84 Bytes.

```
 1  63  OP_IF
 2  03  OP_DATA_3
 3    <timelock 3bytes>
 4  b1  OP_CHECKLOCKTIMEVERIFY
 5  75  OP_DROP
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash2 20bytes>
10  88  OP_EQUALVERIFY
11  ac  OP_CHECKSIG
12  67  OP_ELSE
13  76  OP_DUP
14  a9  OP_HASH160
15  14  OP_DATA_20
16    <pubkey_hash1 20bytes>
17  88  OP_EQUALVERIFY
18  ad  OP_CHECKSIGVERIFY
19  82  OP_SIZE
20  01  OP_DATA_1
21    <secret_length 1byte>
22  88  OP_EQUALVERIFY
23  a9  OP_HASH160
24  14  OP_DATA_20
25    <secret_hash 20bytes>
26  87  OP_EQUAL
27  68  OP_ENDIF
```

Abbildung A.22.: Typ 14 mit Hash160-Secret, Public-Key-Hashes, Abfrage der Secret-Size, 22 Opcodes und einer Länge von 85 Bytes.

```
 1  63  OP_IF
 2  54  OP_4
 3  b1  OP_CHECKLOCKTIMEVERIFY
 4  75  OP_DROP
 5  76  OP_DUP
 6  a9  OP_HASH160
 7  14  OP_DATA_20
 8    <pubkey_hash2 20bytes>
 9  88  OP_EQUALVERIFY
10  ac  OP_CHECKSIG
11  67  OP_ELSE
12  76  OP_DUP
13  a9  OP_HASH160
14  14  OP_DATA_20
15    <pubkey_hash1 20bytes>
16  88  OP_EQUALVERIFY
17  ad  OP_CHECKSIGVERIFY
18  a9  OP_HASH160
19  14  OP_DATA_20
20    <secret_hash 20bytes>
21  87  OP_EQUAL
22  68  OP_ENDIF
```

Abbildung A.23.: Typ 15 mit Hash160-Secret, Public-Key-Hashes, einem Opcode für die Locktime, 19 Opcodes und einer Länge von 79 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  67  OP_ELSE
11  54  OP_4
12    <timelock 1byte>
13  b2  OP_CHECKSEQUENCEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkey_hash2 20bytes>
19  68  OP_ENDIF
20  88  OP_EQUALVERIFY
21  ac  OP_CHECKSIG
```

Abbildung A.24.: Typ 16 mit Hash160-Secret, Public-Key-Hashes, einem Opcode für die Locktime, 17 Opcodes und einer Länge von 77 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  76  OP_DUP
 7  a9  OP_HASH160
 8  14  OP_DATA_20
 9    <pubkey_hash1 20bytes>
10  67  OP_ELSE
11  01  OP_DATA_1
12    <timelock 1byte>
13  b2  OP_CHECKSEQUENCEVERIFY
14  75  OP_DROP
15  76  OP_DUP
16  a9  OP_HASH160
17  14  OP_DATA_20
18    <pubkey_hash2 20bytes>
19  68  OP_ENDIF
20  88  OP_EQUALVERIFY
21  ac  OP_CHECKSIG
```

Abbildung A.25.: Typ 17 mit Hash160-Secret, Public-Key-Hashes, 1-Byte-Locktime, 17 Opcodes und einer Länge von 88 Bytes.

```
 1  63  OP_IF
 2  04  OP_DATA_4
 3    <timelock 4bytes>
 4  b1  OP_CHECKLOCKTIMEVERIFY
 5  75  OP_DROP
 6  82  OP_SIZE
 7  01  OP_DATA_1
 8    <secret_length2 1byte>
 9  88  OP_EQUALVERIFY
10  a9  OP_HASH160
11  14  OP_DATA_20
12    <secret_hash2 20bytes>
13  88  OP_EQUALVERIFY
14  21  OP_DATA_33
15    <pubkey2 33bytes>
16  ac  OP_CHECKSIG
17  67  OP_ELSE
18  82  OP_SIZE
19  01  OP_DATA_1
20    <secret_length1 1byte>
21  88  OP_EQUALVERIFY
22  a9  OP_HASH160
23  14  OP_DATA_20
24    <secret_hash1 20bytes>
25  88  OP_EQUALVERIFY
26  21  OP_DATA_33
27    <pubkey1 33bytes>
28  ac  OP_CHECKSIG
29  68  OP_ENDIF
```

Abbildung A.26.: Typ 18 mit Hash160-Secret, Public-Keys, einem Secret in beiden Pfaden, Abfrage der Secret-Size, 22 Opcodes und einer Länge von 132 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  52  OP_2
 7  21  OP_DATA_33
 8    <pubkey1a 33bytes>
 9  21  OP_DATA_33
10    <pubkey1b 33bytes>
11  52  OP_2
12  ae  OP_CHECKMULTISIG
13  67  OP_ELSE
14  58  OP_8
15  b2  OP_CHECKSEQUENCEVERIFY
16  21  OP_DATA_33
17    <pubkey2 33bytes>
18  ac  OP_CHECKSIG
19  68  OP_ENDIF
```

Abbildung A.27.: Typ 19a mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, Multi-Signature-Wallet, 15 Opcodes und einer Länge von 134 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  52  OP_2
 7  21  OP_DATA_33
 8    <pubkey1a 33bytes>
 9  21  OP_DATA_33
10    <pubkey1b 33bytes>
11  52  OP_2
12  ae  OP_CHECKMULTISIG
13  67  OP_ELSE
14  58  OP_8
15  b2  OP_CHECKSEQUENCEVERIFY
16  75  OP_DROP
17  21  OP_DATA_33
18    <pubkey2 33bytes>
19  ac  OP_CHECKSIG
20  68  OP_ENDIF
```

Abbildung A.28.: Typ 19b mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, Multi-Signature-Wallet, 16 Opcodes und einer Länge von 135 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  52  OP_2
 7  21  OP_DATA_33
 8    <pubkey1a 33bytes>
 9  21  OP_DATA_33
10    <pubkey1b 33bytes>
11  52  OP_2
12  ae  OP_CHECKMULTISIG
13  67  OP_ELSE
14  58  OP_8
15  b2  OP_CHECKSEQUENCEVERIFY
16  75  OP_DROP
17  21  OP_DATA_33
18    <pubkey2 33bytes>
19  ad  OP_CHECKSIGVERIFY
20  68  OP_ENDIF
```

Abbildung A.29.: Typ 19c mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, Multi-Signature-Wallet, 16 Opcodes und einer Länge von 135 Bytes.

```
 1  63  OP_IF
 2  a9  OP_HASH160
 3  14  OP_DATA_20
 4    <secret_hash 20bytes>
 5  88  OP_EQUALVERIFY
 6  52  OP_2
 7  21  OP_DATA_33
 8    <pubkey1a 33bytes>
 9  4c  OP_PUSHDATA1
10    <length 1byte>
11    <pubkey1b 33bytes>
12  52  OP_2
13  ae  OP_CHECKMULTISIG
14  67  OP_ELSE
15  58  OP_8
16  b2  OP_CHECKSEQUENCEVERIFY
17  75  OP_DROP
18  21  OP_DATA_33
19    <pubkey2 33bytes>
20  ac  OP_CHECKSIG
21  68  OP_ENDIF
```

Abbildung A.30.: Typ 20 mit Hash160-Secret, Public-Keys, einem Opcode für die Locktime, Multi-Signature-Wallet, 16 Opcodes und einer Länge von 136 Bytes.

Literaturverzeichnis

[1] Nakamoto, Satoshi. "Bitcoin: A peer–to–peer electronic cash system, 2008." `http://www.bitcoin.org/bitcoin.pdf` (2012).

[2] Zheng, Zibin, et al. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends." *Big Data (BigData Congress), 2017 IEEE International Congress on.* IEEE, 2017.

[3] Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[4] Buterin, Vitalik. "Ethereum: A next–generation smart contract and decentralized application platform." `https://github.com/ethereum/wiki/wiki/White-Paper` (2014).

[5] Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." *Ethereum Project Yellow Paper 151* (2014).

[6] Burrows, Michael, Martin Abadi, and Roger Michael Needham. "A logic of authentication." *Proc. R. Soc. Lond. A 426.1871* (1989).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich den von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichten Großen Beleg zum Thema *Atomic Swaps auf öffentlichen Blockchains* vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 21. September 2018

noobWithAComputer