HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

MSPR: Pentesting Dash Platform

Konzeption und Durchführung eines Pentestes für DP

Vincent Adobatti, Markus Dick, Ömer Gin, Marc Herschel, Sven Hundertmark, Marten Meinhardt, Jan Müller, Markus Neuber

6. Februar 2022

Abstract

Dash Platform ist ein Technologie-Stack für die Entwicklung dezentraler Anwendungen auf dem Dash-Netzwerk. Die beiden wichtigsten architektonischen Komponenten, Drive und DAPI, verwandeln das Dash P2P-Netzwerk in eine dokumentenbasierte Cloud, die Entwickler in ihre dezentralen Anwendungen integrieren können. Dash Platform befindet sich momentan noch in der Entwicklungsphase und ist noch nicht in einem Zustand, der einer ausgereiften Software entspricht. Hier setzt nun unser Masterprojekt an, in dem das gegebene Ziel daraus besteht, einen Penetrationstest von Dash Platform zu konzipieren und durchzuführen. Dieser von dem durchführenden Team verfasste Abschlussbericht gibt einen detaillierten Einblick in die beim Penetrationstest angewandte Methodik und den daraus entstandenen Ergebnissen. Um den Penetrationstest durchzuführen, wurden sowohl Methoden der statischen Source-Codenalyse gewählt als auch Techniken des Stresstests wie z. B. Denial of Service (DoS)-Angriffe versucht. Ebenfalls wurde API-Testing durch z. B. Fuzzing oder das Auswählen gezielter Payloads durchgeführt und Dash Platform auf spieltheoretische Ansätze untersucht. Es wurden im Rahmen dieses Projektes einige potenzielle Angriffsvektoren oder auch Unklarheiten in der Funktionalität von Dash Platform gefunden, die von dem Entwicklerteam bis zu dem erstmaligen post-beta Release adressiert werden sollten.

Schlüsselwörter: *Dash*, *Dash Platform*, *IT-Sicherheit*, *Pentesting*, *DLT-Technologie*

Inhaltsverzeichnis

Verzeichnisse

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

Dash Platform ist ein Technologie-Stack für die Entwicklung dezentraler Anwendungen auf dem Dash-Netzwerk. Dash ist eine Open Source Peer-to-Peer-Kryptowährung, die 2014 als Fork von Bitcoin entstanden ist und Bitcoin um zusätzliche Funktionen wie InstantSend, für sofortige Transaktionen, und CoinJoin für privatsphäre erhaltende Transaktionen erweitert. Diese zusätzlichen Funktionen werden mit Hilfe von Masternodes auf einem zweiten Layer realisiert. Dash Platform befindet sich zurzeit noch in der Entwicklung. Externe Sicherheitsüberprüfungen von Dash Platform haben bisher nicht stattgefunden, sind für die Aufdeckung von Schwachstellen und potenziellen Angriffen aber von hoher Bedeutung. Insbesondere auch aufgrund der hohen Komplexität und dem monetären Hintergrund von *Distributed-Ledger-Technologien* (DLTs) müssen potenzielle Angriffe aufgedeckt und Gegenmaßnahmen ergriffen werden. Das vorliegende Projekt beschäftigt sich daher mit einem Pentesting von Dash Platform, um einen Beitrag zur Weiterentwicklung von Dash Platform zu leisten.

1.1 Zielsetzung und Ablauf des Projektes

Die Ziele der vorliegenden Arbeit lassen sich in folgende Punkte gliedern:

- Einarbeitung in Dash Platform und Pentesting
- Einrichtung einer lokalen Testumgebung
- Konzeption, Entwurf und Durchführung eines Pentests
- Publikation der erstellten Artefakte auf GitHub
- Förderung der Fähigkeit zur Gruppenarbeit und Reflexion

Die beschriebenen Ziele wurden in verschiedenen Phasen erreicht und werden im Folgenden näher erläutert.

In der ersten Phase erfolgte die Einarbeitung in aktuelle DLTs, vorrangig Bitcoin, da es sich bei Dash um einen Software-Fork von Bitcoin handelt. Darauf aufbauend folgte die Einarbeitung in Dash Platform und das Dash Masternode-Netzwerk mit InstantSend, CoinJoin und ChainLocks. Hier konnten die Unterschiede und Ähnlichkeiten zwischen Bitcoin und Dash erkannt werden. Weiterführend wurde der aktuelle Entwicklungsstand von Dash Platform auf dem globalen Testnet mithilfe der offiziellen Dokumentation analysiert. Mit den hier gewonnen Erkenntnissen konnte ein Architekturdiagramm von Dash Platform erstellt werden, welches die Interaktionen zwischen relevanten Komponenten auf Layer 2, wie der DAPI, Drive, Envoy, Tenderdash und MongoDB widerspiegelt. Das Architekturdiagramm ist in Abbildung 1 dargestellt und dient als Grundlage zum Verständnis und der Planung des

weiteren Vorgehens. Im Anschluss erfolgte die Einarbeitung in die Grundlagen und Verfahren des Pentesting sowie übliche Angriffsvektoren gegen DLTs, wie beispielsweise 51% Attacken.

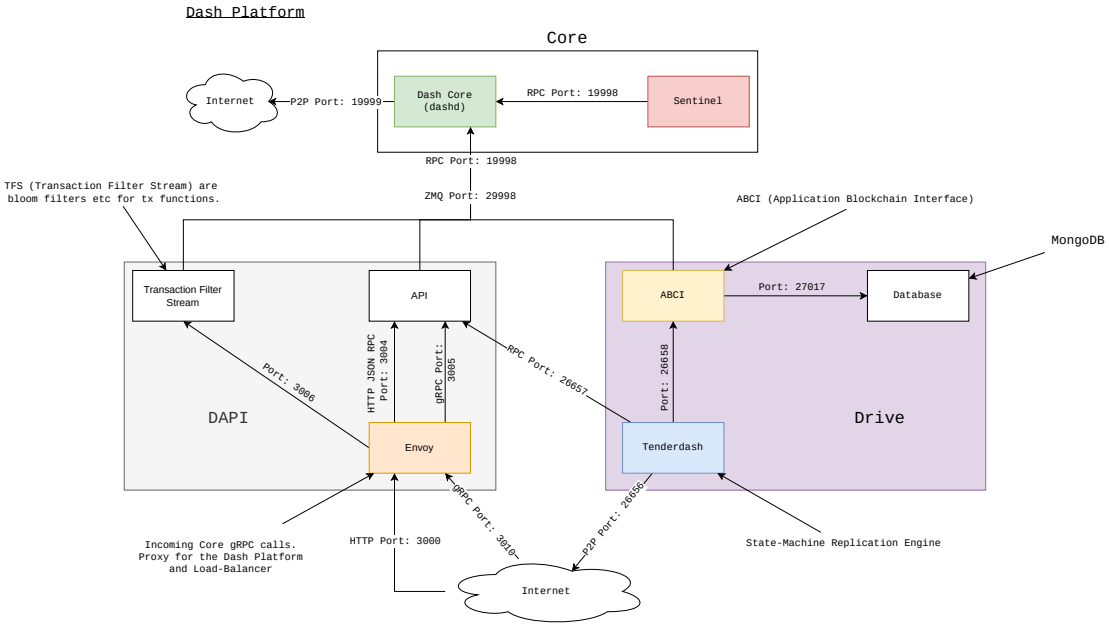

Abbildung 1: Architekturdiagramm Dash Platform

Basierend auf den erworbenen theoretischen Kenntnissen zu DLTs und Pentesting wurde ein Konzept für einen Pentest auf Dash Platform entworfen. Hierfür wurden die identifizierten Angriffe systematisch in Kategorien eingeteilt. Paralell dazu wurde ein lokales Dash Devnet eingerichtet. Die verschiedenen Angriffe auf Dash Platform wurden sowohl auf dem lokalen Devnet als auch, nach Absprache mit Dash Core Group, auf dem globalen Testnet ausgeführt. Die aufbereiteten Ergebnisse der Angriffe und die daraus gewonnnen Erkenntnisse, stellen den Beitrag dieser Arbeit dar.

Die relevanten Artefakte des Projektes werden als Beitrag zur OSS-Entwicklung und im Allgemeinen zur Weiterentwicklung von Dash Platform auf GitHub veröffentlicht. Zusätzlich wird ein prägnanter Bericht in englischer Sprache mit den wichtigsten Ergebnissen veröffentlicht und an die Dash Core Group übergeben.

Zum Abschluss des Projektes wurde innerhalb der Projektgruppe über den allgemeinen Projektverlauf, unsere Zusammenarbeit, die erreichten Ergebnisse, die Umsetzung der Projektziele sowie der dabei gelernten Lektionen diskutiert und reflektiert.

1.2 Aufbau der Arbeit und verwendete Methodik

Zur Durchführung des Pentestings von Dash-Platform wurden die in Abbildung 2 dargestellten Ansätze identifiziert. Die Arbeit gliedert sich nach einer Einführung in die verwendete Testumgebung in Abschnitt 2 entlang dieser Grafik. Es wird jeweils das Vorgehen sowie die Ergebnisse der Durchführung dieser Pentesting-Maßnahmen erläutert. In Abschnitt 3 wird die statische Source-Code-Analyse beschrieben. In Abschnitt 4 folgt die Untersuchung auf potenzielle Denial-of-Service-Vektoren auf Dash Core (Layer 1) und in Abschnitt 5 auf Dash Platform (Layer 2). Anschließend wird in Abschnitt 6 das Fuzzing der gRPC-Schnittstelle der DAPI vorgestellt. Nachfolgend werden Ansätze für Angriffe auf die Dashpay App in Abschnitt 7, Race Conditions auf Dash Platform in Abschnitt 8 sowie mögliche spieltheoretische Angriffe in Abschnitt 9 betrachtet. Abschließend erfolgt in Abschnitt 10 ein Fazit zum Projekt und eine Wertung zum aktuellen Stand von Dash Platform.

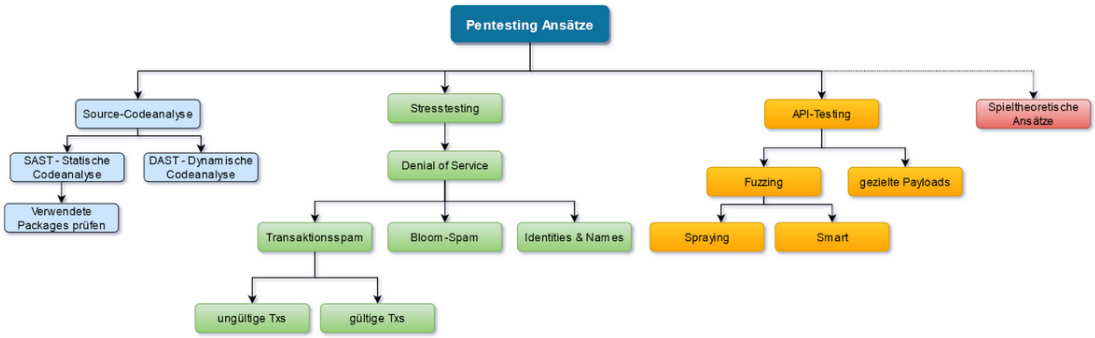

Abbildung 2: Verwendete Methodik und Pentesting-Ansätze

2 Infrastruktur

Im folgenden Kapitel wird unsere Testumgebung vorgestellt und die Konfiguration beschrieben, welche wir für den Großteil unserer Angriffe genutzt haben. Dabei werden die Komponenten von Dash Core und Platform erwähnt, sowie deren verwendeten Versionen und konfigurierten Parameter aufgelistet.

2.1 Testnet

Um unsere Skripte auf dem Testnet auszuführen haben wir eine eigene Virtuelle Maschine eingerichtet. Diese wurde auch als Masternode im Dash Testnet registriert, damit auch Auswirkungen auf Masternodes ausgewertet werden können. Zur Verfügung stand uns dafür eine virtuelle Maschine die auf dem Hochschul VM-Server lief.

Die virtuelle Maschine war mit folgender virtualisierter Hardware ausgestattet:

Hardware:

- CPU: Intel(R) Xeon(R) Gold 5115 CPU @ 1x2.40GHz
- RAM: 8GB
- Disk: 107 GB

Software:

- OS: Ubuntu 20.04, Kernel: 5.4.0
- Dash Core (dashd, sentinel)
  - coreRPC: 19998, coreP2P: 19999
  - v0.17.3
- Dash Platform:
  - Drive dev0.21
  - Dapi (+ TX-Filter) dev0.21
  - Tenderdash dev0.7.0 (installiert aber nicht funktionsfähig)
  - MongoDB v5.0.5

Die virtuelle Maschine wurde mit der, zu dem Zeitpunkt verfügbaren, neuesten Version aller Komponenten aufgesetzt. Durch einen Proxy-Server konnte die virtuelle Maschine auch von außen von anderen Knoten des Testnets erreicht werden und als vollständige Masternode verwendet werden.

2.2 Devnet

Um das Verhalten und unsere Angriffe testen zu können und mögliche Auswirkungen anhand der Log-Dateien zu erkennen, haben wir ein eigenes Devnet aufgebaut. Dafür stand uns das Netzlabor zu Verfügung, was wir uns mit dem Bachelorprojekt-Team geteilt haben. In unserem Netzwerk haben wir 12 Pool-Rechner verwendet, die in 5 eigene Sub-Netze aufgeteilt waren. Diese Konfiguration war von dem Netzlabor vorgegeben und an der Konfiguration oder dem Routing wurde nichts geändert.

Die einzelnen Rechner hatten die gleiche Hardware und Software installiert. Mit dem Unterschied das Dash Platform Dienste nur auf den Masternodes installiert wurde.

Hardware:

- CPU: Intel(R) Core(TM) i7-10700 CPU @ 8x2.90GHz
- RAM: 16GB DDR4 3200MHz (Single)
- Disk: PM9A1 NVMe Samsung 1024GB (read: 7 GB/s, write: 5.2 GB/s)

Software:

- OS: Ubuntu 20.04, Kernel: 5.11
- Dash Core (dashd, sentinel)
  - coreRPC: 19798, coreP2P: 19999
  - v0.17.3
- Dash Platform (nur auf Masternodes):
  - Drive dev0.21
  - Dapi (+ TX-Filter) dev0.21
  - Tenderdash dev0.7.0
  - MongoDB v5.0.5

Die Struktur und der Aufbau des Netzwerkes sieht man im folgender Grafik, dabei entsprechen die IP-Adressen der Standard-Konfiguration des Netzlabors.

Wie dem Netzwerk-Layout in Abbildung 3 zu entnehmen ist, haben wir 9 Rechner als Masternodes im Netzwerk eingerichtet und 3 Rechner als Fullnodes und "Miner". Unser eigenes Devnet lief unter der Bezeichnung “masterproject„ und alle verfügbaren Sporks wurden aktiviert. Der Long-Living-Masternode-Quorum-Type (LLMQ-Type) wurde auf das llmq-devnet gesetzt, damit sich bei der geringen Anzahl an Rechnern auch Quoren bilden. Dabei war in

unserer Konfiguration jeder Masternode Mitglied in jedem Quorum, da die Maximalanzahl des Devnet-Quorum 10 beträgt.

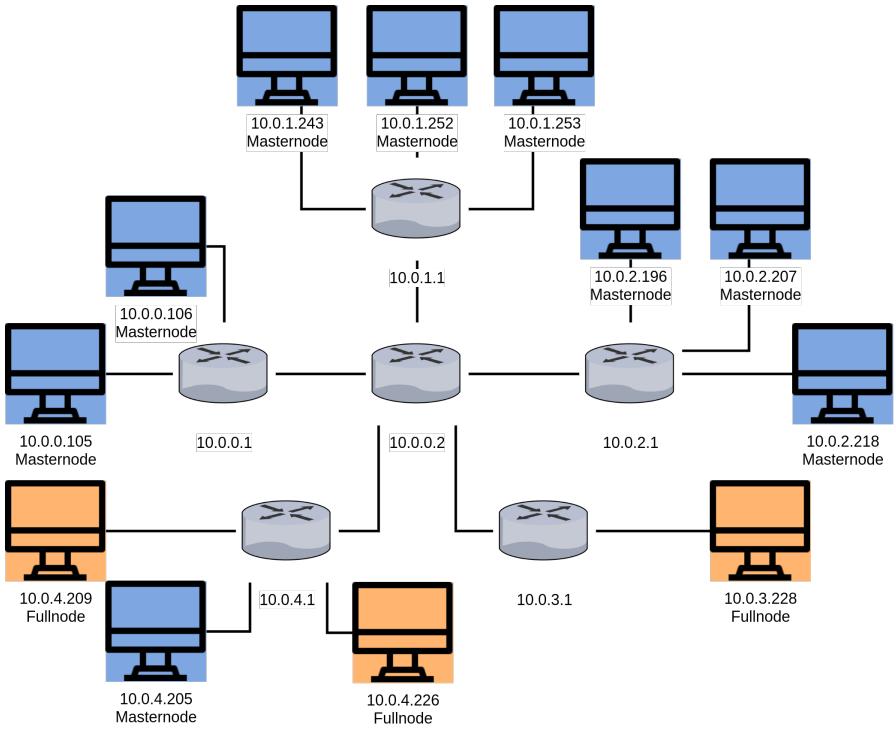

Abbildung 3: Devnets im Netzlabor

Die verwendete Dash-Core Konfiguration haben wir wie folgt festgelegt:

Listing 1: Dash-Core Konfiguration des lokalen Devnet

```
1   devnet=masterproject
2   #----
3   [devnet]
4   port=19999
5   rpcport=19798
6   llmqchainlocks=llmq_devnet
7   llmqinstantsend=llmq_devnet
8   minimumdifficultyblocks=1000000
9   addnode=10.0.0.105:19999
10  addnode=10.0.0.106:19999
11  addnode=10.0.3.216:19999
12  addnode=10.0.3.240:19999
13  addnode=10.0.3.228:19999
14  addnode=10.0.4.209:19999
```

```
15  addnode=10.0.1.238:19999
16  #----
17  listen=1
18  server=1
19  txindex=1
20  addressindex=1
21  spentindex=1
22  #----
23  rpcuser=dashrpc
24  rpcpassword=password
25  rpcallowip=127.0.0.1
26  #----
27  zmqpubrawtx=tcp://0.0.0.0:29998
28  zmqpubrawtxlock=tcp://0.0.0.0:29998
29  zmqpubhashblock=tcp://0.0.0.0:29998
30  zmqpubrawchainlock=tcp://0.0.0.0:29998
31  #----
32  allowprivatenet=1
33  masternodeblsprivkey=712352
        a8f32909c5b22c2f61d38821b7cb22701e6fc0e8be5e879e4afa8e0a7e
34  sporkkey=cNA8JoZh1zjArpb6mpDZKK51Mdvw1cBeNT1ocMM7sjzCgDsZie1P
35  sporkaddr=ySZoNXeV6YXQkg3ySCy1JSXkw9tHPdC3Lo
```

Bei Installation von Dash Core und Plattform wurde anhand der manuellen Installationsanweisung auf docs.dash.org[7] auf dem Stable Branch vorgegangen.

Die Konfiguration der Dash Platform Dienste orientierte sich größtenteils an den Standardeinstellungen und es wurde nur einzelne Parameter angepasst:

- Dashcore RPC Port: 19798
- Dashcore P2P Port: 19999
- Dashcore P2P Network: devnet-masterproject
- Network: devnet
- Initial Core Chainlocked Height: 6240

Die Ports der einzelnen Dienste für Dash Platform wurden auch von in der Standardkonfiguration verwendet:

- Drive (ABCI) (dev0.21)
  - Port: 26658
- DApi (dev0.21)
  - Port: 3004 (RPC), 3005 (GRPC)

- Tx-Filter-Stream (dev0.21)
  - Port: 3006 (TX-Filter)
- Tenderdash (dev0.7.0)
  - Port: 26656 (P2P), 26657 (RPC)
- MongoDB (v5.0.4)
  - Port: 27017

Bei der Tenderdash Konfiguration haben wir einen einzelnen Seed-Node verwendet, der dafür sorgt, dass sich alle Knoten finden und miteinander kommunizieren können. Für die erste Synchronisation haben wir einen einmaligen Genesis-Block erstellt der für alle Masternodes gleich war und eine eindeutige Chain-ID “dash-devnet-masterproject„ vergeben.

Listing 2: Genesis-Block der Tenderdash Blockchain

```
    {
  "genesis_time": "2021-11-16T12:32:34.537240692Z",
  "chain_id": "dash-devnet-masterproject",
  "initial_height": "0",
  "initial_core_chain_locked_height": 6240,
  "initial_proposal_core_chain_lock": null,
  "consensus_params": {
    "block": {
      "max_bytes": "22020096",
      "max_gas": "-1",
      "time_iota_ms": "1000"
    },
    "evidence": {
      "max_age_num_blocks": "100000",
      "max_age_duration": "172800000000000",
      "max_bytes": "1048576"
    },
    "validator": {
      "pub_key_types": [
        "bls12381"
      ]
    },
    "version": {}
  },
  "threshold_public_key": null,
  "quorum_type": "101",
  "quorum_hash": "",
  "app_hash": ""
}
```

Die einzelnen Services werden im System Betrieb von unterschiedlichen Prozessen gestartet, dabei wurde sich auch an den Vorgaben der Dash-Entwickler orientiert.

- über systemd:
  - dashd
  - Tenderdash
- über cron:
  - sentinel
- über forever (beim Neustart von cron gestartet):
  - verwendete node Version: v.16.13.1
  - drive
  - dapi
  - tx-filter

2.3 Probleme beim Einrichten

Beim Einrichten und Konfigurieren der einzelnen Kompenenten kam es öfter zu Problemen. Meistens lag es an der Dokumentation die entweder veraltete Links, nicht mehr gültige Schlüssel oder anzupassende Konfigurationen nicht enthielt. Außerdem erschwerten verschiedene Versionen der Dokumentation die Zuordnung der Schritte zu der neusten Version. Allerdings konnten die meisten Probleme in Absprache und Kommunikation über den Dash Developer Discord Server gelöst werden, was allerdings viel Zeit in Anspruch nahm, da natürlich nicht sofort geantwortet werden konnte und manche Fehlermeldungen auch nicht klar verständlich waren. Beim Einrichten des Testnet-Knoten traten bei der initialen Synchronisation der Tenderdash Blockchain Probleme auf, die leider aufgrund von Zeitmangel nicht behoben werden konnten. Außerdem konnte der Load-Balancer Envoy, welcher in der Architektur von Dash Platform verwendet wird, nicht genutzt werden, da die Entwickler in einem Update der Dienste einen für Envoy nötigen “Health Check„ deaktiviert hatten und die Konfiguration nicht einfach angepasst werden konnte. Eine Untersuchung unserer Angriffe in Kombination mit einem Load-Balancer hätte auch noch spannende Ergebnisse liefern können.

3 Statische Analyse

Die statische Analyse ist ein Testverfahren, bei dem der Quelltext einer Reihe formaler Prüfungen unterzogen wird. Hierbei werden bestimmte Sorten von Fehlern entdeckt, wie zum Beispiel eine mögliche Division durch null. Die statische Analyse ist dabei den White-Box-Testverfahren unterzuordnen, da man hier den Quellcode betrachtet. Anders als bei der dynamischen Analyse wird der Code nicht ausgeführt, sondern anhand von spezifizierten Regeln analysiert beziehungsweise untersucht. Desweiteren wird die Analyse in der Regel werkzeuggestützt durchgeführt. [35]

Innerhalb dieses Abschnittes wird die Anwendung der statischen Analyse beziehungsweise konkreter das Static Applcation Security Testing (SAST) auf relevante Dash-Repositories erläutert. Dies umfasst unter anderem die durchgeführten Arbeitsschritte innerhalb des SAST sowie die verwendeten Werkzeuge, Analyse und Ergebnisse.

3.1 Static Applcation Security Testing

Regeln, basierend auf Sichherheitsschwachstellen, bilden die Brücke zwischen der statischen Analyse und dem Static Application Security Testing. Das SAST wird dabei eingesetzt, um potenzielle Sicherheitslücken im Quellcode zu entdecken. [44]

Das SAST lässt sich dabei in 5 Phasen untergliedern, wie in Abbildung 4 dargestellt:

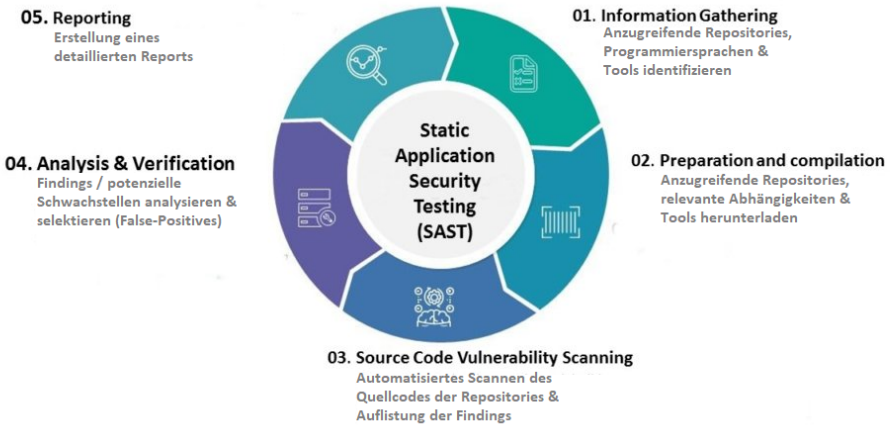

Abbildung 4: SAST Testing Phasen. Quelle: In Anlehnung an [33]

Innerhalb der ersten Phase des Information Gathering werden die anzugreifenden Repositiories, Programmiersprachen sowie Werkzeuge identifiziert. Darauffolgend werden in der zweiten Phase alle relevanten beziehungsweise identifizierten Repositories und Werkzeuge akquiriert und installiert. Anschließend werden in der dritten Phase automatisierte Werkzeuge genutzt um den Quellcode der jeweiligen Repositories zu untersuchen. In den letzten beiden Phasen werden die Funde untersucht und nach False-Positives selektiert und dokumentiert.

3.2 Identifizierte Repositories

Die DAPI stellt ein potenziell anzugreifende Repository dar, da es ein Ort der direkten Eingabe für User ist. Da die DAPI von weiteren Packages abhängig ist und jene auch untersucht werden sollen, werden jene auch identifiziert und ebenfalls im weiteren Verlauf heruntergeladen. Die Packages enthalten dabei größtenteils Clients, Utilities und Definition Files zu den RPC- und gRPC Schnittstellen.

Zu untersuchende Repositories:

Repository	Beschreibung
dapi	Decentralized API for the Dash network
dashd-rpc	Dash Client Library to connect to Dash Core (dashd) via RPC
js-grpc-common	Common JavaScript GRPC code and utils for platform projects
grpc-js	Pure JavaScript gRPC Client
dapi-grpc	Decentralized API GRPC definition files and generated clients

Tabelle 1: SAST - Identifizierte Repositories

3.3 Identifizierte Werkzeuge

Vor und während der Auswahl geeigneter Werkzeuge zur Untersuchung des Quellcodes haben sich unterschiedliche Anforderungen gebildet. Zum einen sollen die Werkzeuge Open Source und gebührenfrei sein. Desweiteren sollen Regelmengen vorhanden sein, um den Aufwand der zu entwickelnden Regeln zu minimieren. Die Werkzeuge sollen außerdem die in den Repositories vorhandenen Programmiersprachen wie JavaScript und TypeScript verstehen und unter GNU/Linux installierbar sein. Nach Durchsicht und vergleich unterschiedlicher Werkzeuge haben sich im Verlauf die Werkzeuge Semgrep und npm-audit hervorgetan.

Semgrep

wurde im Jahr 2020 [19] erstveröffentlicht und ist ein Werkzeug zur statischen Code Analyse. Der Name ist eine Kombination aus semantic und grep, was sich darauf bezieht, dass semgrep

ein Befehlszeilen-Dienstprogramm für die Textsuche ist, dass die Semantik des Quellcodes kennt. Das Werkzeug kann beispielsweise als integrierter SAST Analyzer in GitLab verwendet werden [24].

Innerhalb von Semgrep verwenden wir die Rulesets nodejs und nodejsscan. In den Rulesets sind Regeln enthalten, die beispielsweise evals für User-Eingaben, hardcoded credentials und veraltete Algorithmen erkennen.

npm-audit

ist eine in Node Package Manager (npm) eingebaute Sicherheitsfunktion, die auf Sicherheitslücken untersucht und einen Bewertungsbericht erstellt. Audit prüft zudem rekursiv die aktuelle Version der installierten Pakete innerhalb eines Projektes auf bekannte Sicherheitslücken, die in der öffentlichen npm-Registry gemeldet werden. Wenn es ein Sicherheitsproblem entdeckt, wird es gemeldet [38]. Wir nutzen npm-audit vorallem zur Ergänzung von Semgrep, um die Pakete zu prüfen, die in Abhängigkeit zu den zu untersuchenden Repositories stehen.

Installations - und Nutzungshinweise sind innerhalb des internen Wikis dokumentiert [46].

3.4 Ergebnisse

Nach Anwendung der Werkzeuge erhielten wir vorallem Findings wie object inejections, inefficient regular expressions, insecure randomness oder insecure cryptographic algorithms.

Das Validieren der Findings hat viel Zeit in Anspruch genommen, da neben offensichtlichen False Positives auch einige dabei waren, bei denen man sich intensiv in den Quellcode einlesen musste. Letztlich hat sich gezeigt, dass alle Findings False Positives sind. Daran zeigt sich, dass die bisherige Entwicklung mit ESLint zuverlässig ist. Da der hohe Aufwand weiterer statischer Analyse sich nicht die Waage mit dem Nutzen hält haben wir an dieser Stelle die statische Analyse abgeschlossen und alle relevanten oder eher interessanten Findings dokumentiert.

Auf Abbildung 5 ist ein Beispielauzug (Screenshot) aus den dokumentierten Findings zu sehen.

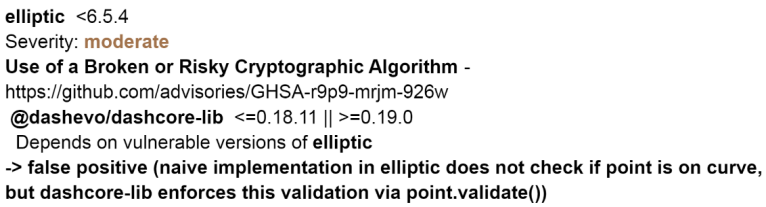

Abbildung 5: Screenshot aus den detailliert dokumentierten Findings: Broken or Risky Cryptographic Algorithm. Quelle: Internes Dokument [45]

4 DoS auf Layer 1

Folgender Abschnitt befasst sich mit dem Angriffsvektor bei dem ein *Denial of Service* (DoS) auf Layer 1, also dem Dash Core Layer, herbeigeführt wird. Die Funktionalitäten, die von Dash Platform bereitgestellt werden, erfordern einen funktionierenden unterliegenden Core Layer. Als Beispiel seien hier die *Asset Lock Transactions* genannt, die beim Erstellen einer Transaktion ausgeführt und verarbeitet werden müssen [40]. Etwaige Einschränkungen der Verfügbarkeit des Core Layers würde also nicht nur die Dienste des Core Layers unverfügbar machen, sondern auch die Dienste des Platform Layers.

4.1 Ansatz

Der von uns gewählte Ansatz, einen DoS auf Layer 1 zu provozieren, zielt darauf ab, möglichst viel CPU-Zeit und Speicherverbrauch auf einem Masternode bzw. im gesamten Masternode-Netzwerk mit unehrlichen Transaktionen zu vergeuden. Dadurch, dass der Masternode mit der Verifizierung und Abarbeitung von unehrlichen Transaktionen beschäftigt wird, werden Transaktionen mit ehrlicher Absicht verzögert und später in die Blockchain aufgenommen.

Listing 3: Datenschema einer Transaktion in Dash

```
{
    "txid" : "c80b343d2ce2b5d829c2de9854c7c8d423c0e33bda264c4013\
              8d834aab4c0638",
    "hash" : "
       c80b343d2ce2b5d829c2de9854c7c8d423c0e33bda264c40138d834aab4c0638",
    "size" : 85,
    "vsize" : 85,
    "version" : 1,
    "locktime" : 0,
    "vin" : [
         List of all referenced inputs including signatures for each...
    ],
    "vout" : [
         List of all outputs...
    ]
}
```

In Listing 3 ist das Datenschema einer standardmäßigen Transaktion in Dash dargestellt. Wenn bei einem Node eine Transaktion über das Netzwerk eingeht, muss der Node zwei Tätigkeiten durchführen, um die Transaktion zu validieren. Er muss überprüfen, ob jeder der als Input referenzierten Outputs als *Unspent Transaction Output* (UTXO) im UTXO-Set vorliegt. Anschließend muss er zu jedem referenzierten UTXO prüfen, ob sein *Locking Script* mit dem in der Transaktion vorliegenden *Unlocking Script* gelöst werden kann. Der zuletztgenannte Schritt beinhaltet meist das Validieren einer digitalen Signatur mit dem

Elliptic Curve Signature Algorithm (ECDSA) [1]. Zusätlzich belegt jede neue Transaktion Platz im Mempool sowie auf der Blockchain, was letztendlich die pro Transaktion anfallenden Gebühren steigen lässt.
Jede der oben genannten Mechanismen sorgt dafür, dass Ressourcen auf dem Node gebunden werden und könnten womöglich einen DoS herbeiführen.

4.2 DoS durch invalide Transaktionen

Bei diesem Ansatz, wird versucht CPU-Zeit auf dem Knoten zu verbrauchen, indem invalide Transaktionen an den Knoten versendet werden und dieser diese überprüfen muss.

4.2.1 Versuchsaufbau

Der Angriff wird durch ein selbstimplementiertes Python-Skript ausgeführt. Dieses baut parallel randomisierte Transaktionen auf und sendet sie über einen Shell-Aufruf an das Dash-CLI über den Befehl `sendrawtransaction`. Um die Transaktionen zu erstellen wird die Bibliothek *pycoin* [34] verwendet. Durch den parallelen Aufbau des Skript konnten innerhalb der Testumgebung über 5000 Transaktionen pro Sekunde erstellt und versendet werden.

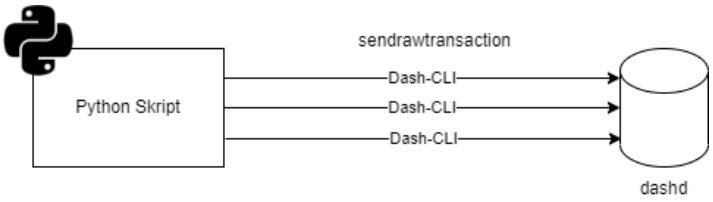

Abbildung 6: Schematischer Aufbau des DoS auf Layer 1. Quelle: Eigene Darstellung

Eine schematische Darstellung des Aufbaus kann Abbildung 6 entnommen werden.

4.2.2 Variante 1: Transaktion referenziert keinen gültigen UTXO

Bei dieser Variante wird kein valider UTXO innerhalb der gesendeten Transaktion referenziert. Ziel ist es, die Last auf dem Node zu erhöhen, da der Node für jede Transaktion einen Zugriff auf sein intern gespeichertes UTXO-Set ausführen muss.
Ergebnis des Versuches war, dass der Angriff keinerlei Auswirkungen auf den Knoten hatte. Eingaben auf den Knoten, sowohl Nutzereingaben als auch aus dem Netzwerk, wurden ohne Verzögerung bearbeitet. Das liegt vermutlich daran, dass der verwendete Datenspeicher für

das UTXO-Set eine *LevelDB*-Instanz [31] ist. Diese ist ein optimierter *Key-Value-Store* und erlaubt einen Zugriff in konstantem Aufwand *O(1)*.

4.2.3 Variante 2: Transaktionen referenziert gültigen UTXO

Bei dieser Variante wird ein valider UTXO aus dem UTXO-Set in der Transaktion referenziert und eine Signatur zu diesem gefälscht. Der Knoten verschwendet also CPU-Zeit, indem er die gefälschten Signaturen überprüft. Hierfür musste ein zweites Programm implementiert werden, welches das UTXO-Set ausliest und in einer geeigneten Datenstruktur zur Verfügung stellt. Die eigentlichen UTXOs in der LevelDB sind lediglich Byte-Arrays und durch eine Konvertierung in JSON oder CSV Dateien wird die Weiterverarbeitung erleichtert. Das für das Auslesen angewandte Verfahren kann [18] entnommen werden.

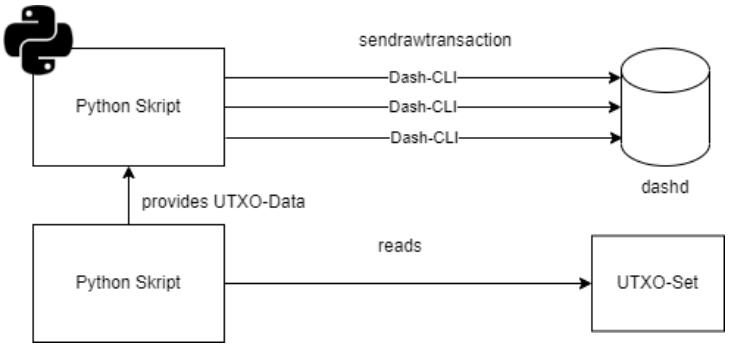

Abbildung 7: Schematischer Aufbau des DoS auf Layer 1 unter Verwendung des UTXO-Sets. Quelle: Eigene Darstellung

In Abbildung 7 ist der schematische Aufbau des Angriffs dargestellt. Ein Python-Skript wird für die Auslesung und Datenumwandlung des UTXO-Sets verwendet. Die ausgelesenen Daten werden als CSV-Datei abgespeichert, sodass das ursprüngliche Skript auf diese zugreifen kann und für die Erstellung der Transaktionen nutzen kann.

Bei der Durchführung des Angriffs konnte ebenso kein Effekt auf dem Dash-Knoten beobachtet werden. Eingaben wurden wiederholt ohne Verzögerung abgearbeitet. Vermutlich liegt das daran, dass die für Bitcoin Core und Dash Core verwendete ECDSA-Implementierung *libsecp256k1* [4] sehr effizient und optimiert ist. Bei Durchführung eines Benchmarks auf einem handelsüblichen CPU vom Typ *AMD Ryzen 5 2600* [20] mit 12 logischen Kernen, konnten im Durchschnitt über 19 000 Signaturverifikationen pro Sekunde durchgeführt werden.

4.3 DoS durch valide Transaktionen

Dieser Ansatz verwendet valide Transaktionen um den Mempool zu füllen und so das Netzwerk zu überlasten. Je größer der Mempool ist, desto länger müssen ehrliche Transaktionen auf eine Aufnahme in die Blockchain und eine Bestätigung warten. Außerdem ist das Netzwerk mit der Verifizierung der unehrlichen Transaktionen beschäftigt und verbraucht so CPU-Zeit und Speicher.

4.3.1 Versuchsaufbau und Durchführung

Um den Angriff durchzuführen, wurde das in Unterunterabschnitt 4.2.1 genannte Python-Skript angepasst, sodass valide Transaktionen erstellt werden. Außerdem wurde ein weiteres Programm implementiert, das im 10-Sekunden-Takt den Mempool mithilfe des Dash-CLI Befehls `getmempoolinfo` ausliest und die Daten in einer CSV-Datei abspeichert.

Für die Durchführung wurde das von uns eingerichtete lokale Devnet verwendet.

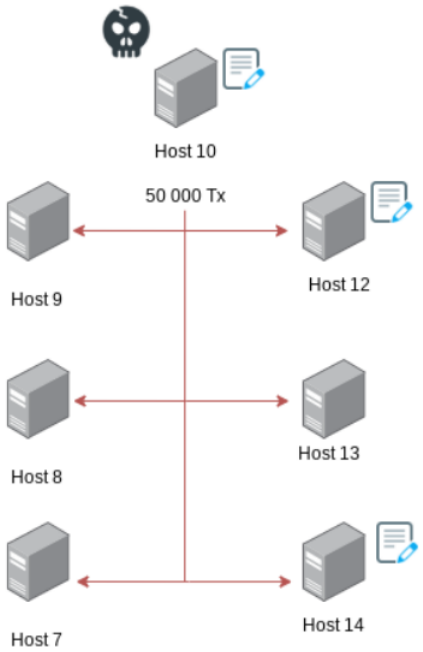

Abbildung 8: Versuchsaufbau des durchgeführten Angriffs im lokalen Devnet

In Abbildung 8 ist der Aufbau des Angriffs dargestellt. Der Angriff geht von Host 10 aus, dargestellt durch das Totenkopfsymbol. Host 10, 12 und 14 zeichnen jeweils ihre Mempool-

Statistiken auf. Außerdem wird der Netzwerk-Verkehr mithilfe von *Wireshark* [29] überwacht. Insgesamt läuft der Angriff über einem Zeitraum von 2 Stunden und es werden 50 000 Transaktionen erstellt und versendet.

4.3.2 Ergebnis

Insgesamt war bei allen im Netzwerk beteiligten Knoten eine Eingabeverzögerung zu beobachten. Außerdem wurden testweise getätigte ehrliche Transaktionen erst am Ende des Angriffs, also nach zwei Stunden in die Blockchain aufgenommen und bestätigt.

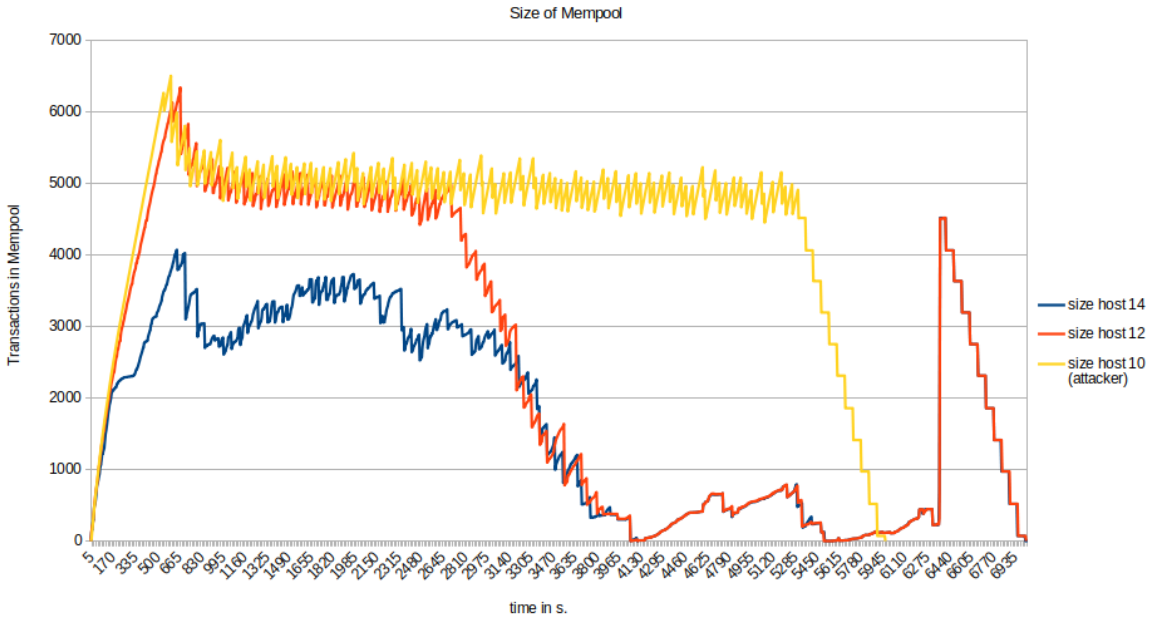

Abbildung 9: Diagramm zur Mempool-Größe der beteiligten Knoten

In Abbildung 9 ist die Entwicklung der Größe des Mempools über den Angriffszeitraum zu sehen. Gut zusehen ist hierbei an der generellen Zickzack-Bewegung der Kurven, dass die einkommenden Transaktionen nach einer gewissen Zeit in neue Blöcke aufgenommen werden und der Mempool so schwindet. Eine weitere interessante Beobachtung ist die anfängliche Diskrepanz der Kurven von Host 12 und Host 14. Es scheint, dass sämtliche Transaktionen Host 12 erreichen, Host 14 allerdings nicht. Außerdem lässt sich ab ungefähr der Hälfte der Zeit beobachten, dass die Kurven von Host 12 und 14 beinahe deckungsgleich verlaufen, und bei ca. Sekunde 6440 steil nach oben verlaufen, der Mempool also schlagartig anwächst.

Eine mögliche Erklärung dieses Verhalten wäre in der Netzwerkbeschaffenheit oder in der Eingabeverzögerung in Folge des Angriffes zu finden.

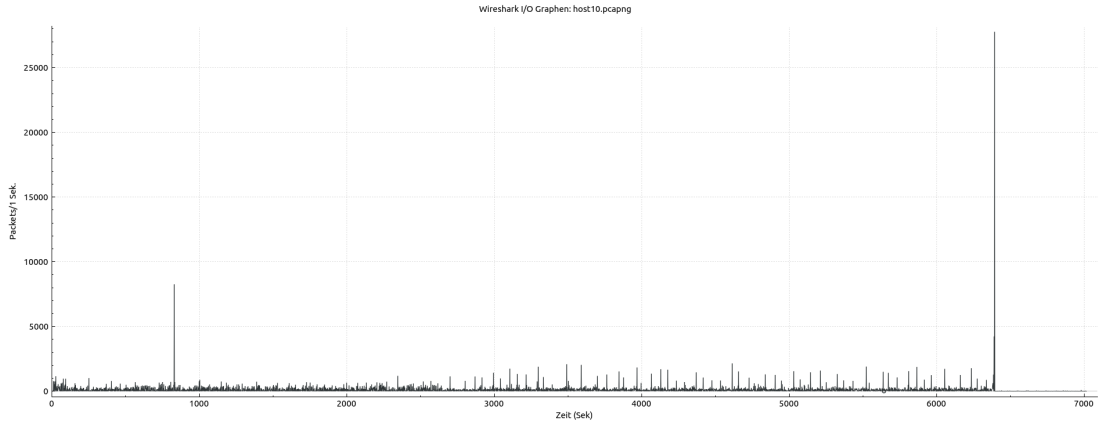

Abbildung 10: Diagramm zur Größe des Netzwerk-Traffics

In Abbildung 10 sind die Menge der ein- und ausgehenden IP-Pakete an Host 10 als Diagramm dargestellt. Hier lässt sich beobachten, dass der Netzwerk-Traffic zweimal steil nach oben geht. Dies deckt sich zeitlich mit der in Abbildung 9 vorhandenen Kurvenanstiegs ab Sekunde 6440 von Host 12 und 14 ab.

4.4 Fazit und Ausblick

Es lässt sich festhalten, dass Dash Core gut gegen jegliche Arten von DoS-Attacken geschützt ist. Dies liegt vermutlich daran, dass Dash Core ein Software-Fork von Bitcoin Core ist und somit schon eine gewisse Software-Reife und Ausfallsicherheit mitbringt. Mit einer großen Anzahl von Transaktionen lässt sich zwar das Netzwerk zu einem gewissen Grad verlangsamen, allerdings nicht zum kompletten Stillstand bringen. Außerdem würde in einem realen Mainnet ein solcher Angriff hohe Kosten durch steigende Gebühren verursachen, was einen möglichen Angriff ebenso einschränkt.

Weitere Folgeprojekte könnten versuchen, den Angriff verteilt durchzuführen, also von mehreren Angreiferknoten aus. Außerdem könnten testweise andere Transaktionstypen als die hier verwendeten *Pay-to-Public-Key-Hash* (P2PKH) ausprobiert werden.

5 DoS und Flooding auf Layer 2

Folgender Abschnitt befasst sich mit dem Angriffsvektor, einen *Denial of Service* (DoS) auf Layer 2, also der Dash Platfrom Layer herbeizuführen. Dies ist ein attraktives Angriffsziel, da insbesondere die leichtgewichtigen Clients, also die ohne lokale Kopie der Blockchain, sowie erweiterte Funktionalität, wie Data Contracts, darauf bauen. Im Folgenden zeigen wir drei konkrete Ansätze, die wir dabei verfolgt haben. Dazu gehören Angriffe über den Transaction Filter Stream, die Abfrage von Identities und den Dash Platform Name Service (DPNS).

5.1 Transaction Filter Stream

Bei der Analyse der verfügbaren Schnittstellen der DAPI sind wir im Core gRPC Endpoint auf die Methode `subscribeToTransactionsWithProofs` gestoßen. Diese Methode ermöglicht es Clients Transaktionen, an denen sie interessiert sind, abzufragen sowie über entsprechende zukünftige Transaktionen informiert zu werden. Hierdurch wird Clients auf Layer 2 der Zugriff auf eine gewünschte Untermenge der Transaktionen auf Layer 1 ermöglicht, ohne dass diese eine lokale Kopie der Blockchain vorhalten müssen. Dabei handelt es sich insbesondere um Clients, die eine Simplified Payment Verification (SPV) durchführen möchten.

Intern wird diese Methode durch den Transaction Filter Stream (`tx-filter-stream.js`) implementiert. Dieser Dienst wird unabhängig vom Rest der DAPI als eigenständiger Prozess ausgeführt. Ein Proxy (Envoy) leitet die Aufrufe entsprechend an den Transaction Filter Stream oder die restliche DAPI weiter.

Die Methode erwartet beim Aufruf einen Bloom-Filter vom Client. Dieser wird vom Client selbst erzeugt und genutzt, um die für den Client relevanten Transaktionen herauszufiltern.

Der erste Ansatz, den wir für einen Angriff in Erwägung zogen, war den Dienst mit speziellen Bloom-Filtern anzugreifen. Hierfür betrachteten wir zunächst die Funktionsweise von Bloom-Filtern.

5.1.1 Bloom-Filter

Um einen Bloom-Filter zu konstruieren, wird zunächst ein Array mit m Bits mit Nullen gefüllt. Für jedes Element, welches vom Filter erfasst werden soll, werden mittels einer parametriesierten Hashfunktion k unterschiedliche Abbildungen auf den Wertebereich von 0 bis $m - 1$ erzeugt. Jedes Bit, auf welches eine Abbildung verweist, wird auf 1 gesetzt. Sollte ein Bit bereits auf 1 gesetzt sein, bleibt es so – es findet also eine Oder-Verknüpfung statt. Abbildung 11 verdeutlicht die Konstkution an einem Beispiel.

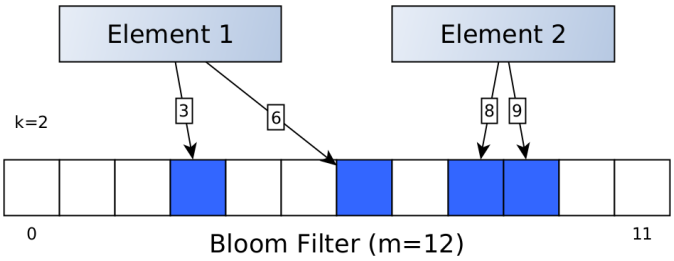

Abbildung 11: Beispiel für die Konstruktion eines Bloom-Filters: Dem Filter der Größe $m = 12$ werden zwei Elemente hinzugefügt. Für jedes dieser Elemente werden $k = 2$ Abbildungen erzeugt, welche auf Bits im Filter-Array verweisen. Durch die Abbildungen von Element 1 werden die Bits 3 und 6 auf 1 gesetzt und analog durch Element 2 die Bits 8 und 9.

Bei der Anwendung des Bloom-Filters wird für das zu filternde Element eine Bitmaske erzeugt. Dies erfolgt analog zur Konstruktion des Bloom-Filters, jedoch mit nur diesem einen Element. Anschließend wird geprüft, ob jedes Bit, welches in der Bitmaske des Elements auf 1 gesetzt ist, auch im Filter auf 1 gesetzt ist. Eine effiziente Implementierung kann die Abbildungen nacheinander berechnen und beim ersten Bit, welches im Filter nicht auch auf 1 gesetzt ist, abbrechen. Abbildung 12 visualisiert einen solchen Abgleich mit dem Bloom-Filter aus dem vorherigen Beispiel.

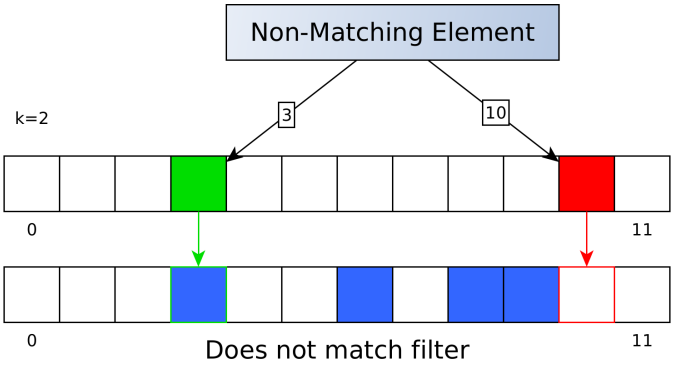

Abbildung 12: Beispiel für den Abgleich eines Bloom-Filters mit einem Element, welches nicht vom Filter erfasst wird: Es werden für das Element $k = 2$ Abbildungen berechnet. Eine Abbildung verweist auf das Bit 3, welches im Filter ebenfalls auf 1 gesetzt ist, die andere Abbildung verweist auf das Bit 10, welches im Filter jedoch nicht auf 1 gesetzt ist. Damit wird dieses Element nicht vom Filter erfasst und zurückgewiesen.

Da es sich beim Bloom-Filter nur um ein Bit-Array handelt, lassen sich nur wenige Rück-

schlüsse auf die gesuchten Elemente ziehen. Anhand des Filters lässt sich nicht bestimmen, welche der k Abbildungen von welchem Element dazu geführt hat, dass dieses Bit auf 1 gesetzt ist. Es lässt sich lediglich durch die Anzahl, der auf 1 gesetzten, Bits und die Anzahl der Abbildungen k, eine Untergrenze für die Anzahl der gesuchten Elemente ermitteln. Die dadurch entstehende Mehrdeutigkeit macht den Bloom-Filter probabilistisch: Zwar werden garantiert die gesuchten Elemente gefunden, wenn sie existieren, jedoch auch weitere, die nur zufällig zum Filter passen – dabei handelt es sich dann um sogenannte False Positives. Abbildung 13 zeigt einen False Positive für den in Abbildung 11 konstruierten Bloom-Filter. Diese Eigenschaften ermöglichen es dem Client etwas Privatssphäre gegenüber dem Dienst zu wahren, da dieser selbst bei einem Treffer nur mit einer gewissen Wahscheinlichkeit sagen kann, ob der Client tatsächlich nach diesem Element gesucht hat.

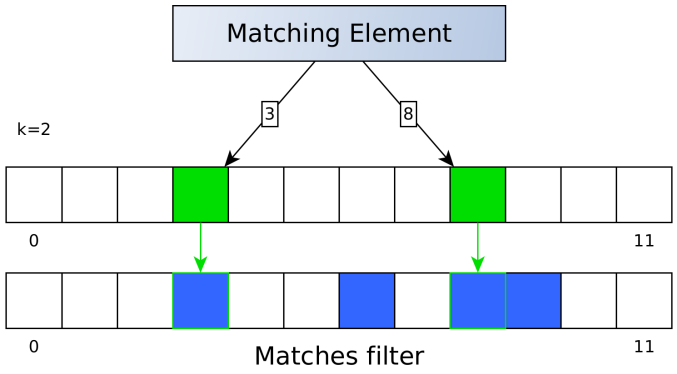

Abbildung 13: Beispiel für ein False Positive bei der Anwendung eines Bloom-Filters: Die Abbildungen für das Element verweisen zwar auf Bits, die im Bloom-Filter auf 1 gesetzt sind, womit dieses vom Filter als Treffer erkannt wird, jedoch wurde der Filter nicht konstruiert, um nach diesem Element zu suchen (vgl. Abbildung 11). Somit handelt es sich bei diesem Treffer um einen False Positive.

5.1.2 Angriff

Grundsätzlich sind Bloom-Filter recht effizient, da nur simple Hashfunktionen berechnet, und einfache Vergleiche durchgeführt werden müssen. Eine Möglichkeit den Bloom-Filter selbst ineffizient zu konstruieren wäre, den Parameter k sehr groß zu wählen, wodurch der Server entsprechend viele Hashfunktionen berechnen muss. Wenn der Filter zudem viele (oder ausschließlich) Bits auf 1 gesetzt hat, wird ein früher Abbruch unwahrscheinlich (bzw. ausgeschlossen). In der Praxis wird k von der DAPI jedoch auf 50 begrenzt, und ist somit viele Größenordnungen kleiner als notwendig, um signifikante Last bei der Anwendung auf ein einzelnes Element zu verursachen.

Daher haben wir unseren Ansatz dahingehend adaptiert, dass wir den Server dazu zwingen, möglichst viele Elemente zu verarbeiten – schließlich muss der Server für jedes dieser Elemente (bis zu) k Hashfunktionen berechnen. Hierfür nutzen wir aus, dass die Methode zusätzliche Parameter entgegennimmt, welche steuern, ab welchem Block in der Blockchain gesucht werden soll, sowie wie viele Transaktionen zurückgegeben werden sollen. Damit lassen sich Transaktionen bereits ab Blockhöhe 0 (Beginn der Blockchain) suchen. Zudem sorgt der spezielle Wert 0 für die Anzahl dafür, dass die Suche nach passenden Transaktionen nicht nach einer bestimmten Anzahl abbricht und sogar zukünftige Transaktionen umfasst.

Um die Verwundbarkeit auf diesen Angriff zu testen haben wir, unter Nutzung des offiziellen und in JavaScript geschriebeben DAPI-Clients mit Node.js, ein Skript geschrieben. Dieses Skript sendet mehrere solcher Anfragen (also Bloom-Filter mit allen Bits auf 1, ab Block 0, unbegrenzte Anzahl) parallel an den Server. Somit rufen wir effektiv mehrfach die gesamte Blockchain über eine Methode ab, die dafür eigentlich nicht vorgesehen ist und dabei zusätzliche Last verursacht. Das Ergebnis war, dass wir den Prozess mit ausreichend vielen Anfragen zuverlässig zum Absturz bringen konnten. Die Ursache dafür war, dass dem Prozess der Heap-Speicher ausging.

Da wir mit 16.000 Anfragen relativ wenig Aufwand benötigten, um den Dienst bei einem Masternode auszuschalten, dürfte dieser Angriff mit einem kleinen Botnet auf das gesamte Netzwerk skalieren. Zwar ist nur der Transaction Filter Stream davon betroffen, wodurch Full Clients, welche die gesamte Blockchain vorhalten weiterhin funktionsfähig bleiben, jedoch wären andere Clients nicht mehr in der Lage Transaktionen auf diese weise Abzufragen. Daher würden wir diese Schwachstelle als kritisch einstufen.

5.2 Identities

Die mit DIP0011 eingeführten *Identities* erweitern das Konzept der Blockchain User, welche wiederum in DIP005 eingeführt wurden [40]. Identities agieren als dezentraler und verifizierbarer Identifikator auf Dash Platform, u.a. für Personen, Organisationen, Anwendungen oder Datenmodelle [11].

Vor der Erstellung einer Identity muss eine, in Kapitel 4 bereits erwähnte, *Asset Lock Transaction* auf Layer 1 durchgeführt werden. Diese spezielle Transaktion transformiert Dash, nach einer fixen Umwandlungsrate, in *Platform Credits* auf Layer 2. Diese Credits dienen als Zahlungsmittel für Aktivitäten auf Dash Platform, den sogenannten *State Transitions*, welche in der Platform Chain persistiert werden. Ein Beispiel für eine solche State Transition ist die *Identity Create Transition*, welche Identities erstellt und den Output der zuvor ausgeführten Asset Lock Transaction referenziert. Der Output der Asset Lock Transaction muss einen *Public Key Hash* enthalten, um zu verifizieren, dass der Ersteller der Identity autorisiert ist den Output der Asset Lock Transaction auszugeben. Zur Verifizierung muss die Identity

Create Transition eine digitale Signatur enthalten, basierend auf dem Key, welcher für den Public Key Hash in der Asset Lock Transaction genutzt wurde. Bei der Erstellung der Identity werden die Credits auf diese übertragen und sie wird mit einem, vom Nutzer angegebenen, *Identity Public Key* assoziert. Credits und weitere Public Keys können über eine *Identity Topup Transition* [16] bzw. über eine *Identity Update Transition* [11] hinzugefügt werden. Verschiedene Keys können für unterschiedliche Anwendungen oder Datenmodelle genutzt werden. Der Identity Erstellungsprozess ist in Abbildung 14 dargestellt.

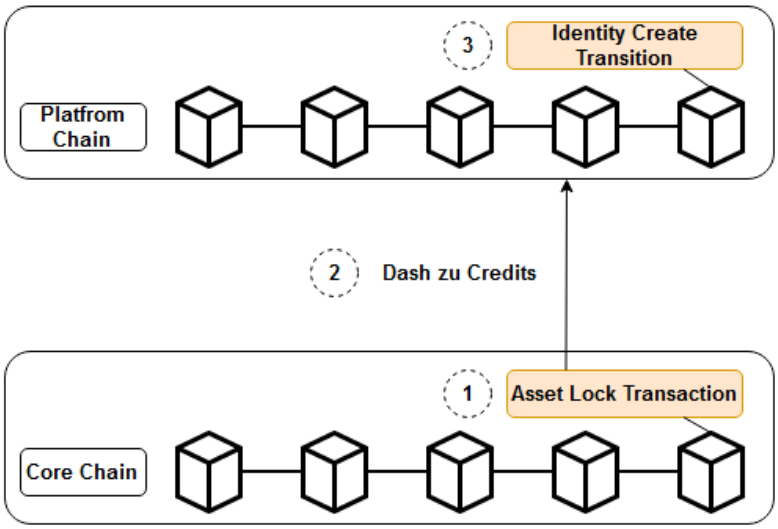

Abbildung 14: Identity Erstellungsprozess. Quelle: In Anlehnung an [40]

Identities werden auf Dash Platform für eine Vielzahl von Anwendungsfällen genutzt. Hierbei handelt es sich um State Transitions, u.a. zum Registrieren von Namen für eine Identity, zum Registrieren von *Data Contracts* oder zur Aktualisierung von Dokumenten, die innerhalb eines Data Contracts definiert wurden [40]. Die Identity signiert die State Transitions und agiert als Identifikator für Namen, Data Contracts oder Dokumente. Unterkapitel 5.3 behandelt Namen und den verwandten *Dash Platform Name Service* (DPNS) im Detail, während Kapitel 8 Data Contracts im Kontext von Race Conditions behandelt.

5.2.1 Ansatz

Aufgrund der zentralen Bedeutung der Identities für diverse Dienste auf Dash Platform wurde die Suche nach ihnen auf potenzielle Angriffsvektoren untersucht. Für die Suche nach Identities bietet die DAPI den Platform gRPC Endpunkt `getIdentitiesByPublicKeyHashes` an

[6]. Dieser Endpunkt erlaubt es, Identities über ein Array von Public Key Hashes, der assozierten Identity Public Keys, abzufragen. In der Dash Dokumentation ist keine Beschränkung für die Anzahl der übergebaren Public Key Hashes angegeben. Aufgrund dieser Eigenschaft können mehrere Identities gleichzeitig über ihren jeweiligen Public Key Hash abgefragt werden. Hierbei ist zu klären was geschieht, wenn eine Abfrage sehr viele Identites liefert. Es ist anzumerken, dass, durch die begrenzte Größe von gRPC-Payloads von 4 MB [37], nicht beliebig viele Identities übertragen werden können. Des Weiteren könnte eine solche Suchanfrage Last auf einem Masternode erzeugen. Bei ausreichender Last könnte ein *Denial-of-Service* (DoS) herbeigeführt werden.

Der Ausfall oder die eingeschränkte Verfügbarkeit einer oder mehrerer Masternodes würde in eine eingeschränkte Verfügbarkeit der Dienste auf Dash Platform resultieren.

5.2.2 Durchführung

Zu Untersuchung der in Unterkapitel 5.3 beschriebenen Ansätze haben wir, unter Verwendung der *Dash SDK* [5], ein Skript erstellt, welches nebenläufig mehrere Identities auf einer einzigen Wallet anlegt. Bei der Ausführung des Skriptes auf dem loaklen Devnet meldete die DAPI einen internen Serverfehler. Der Grund hierfür war, dass der gRPC Healthcheck von Envoy (Loadbalancer) zur DAPI durch die Dash Entwickler ausgeschaltet wurde [41]. Jedoch setzt die Nutzung der Dash SDK, zum Anlegen von Identities, einen funktionsfähigen gRPC Healthcheck voraus. Nachdem die Versuche, den gRPC Healthcheck selbst zu implementieren oder in der DAPI auszuschalten, scheiterten, konkatierten wir die Dash Entwickler und schilderten unsere Problemstellung. Es stellte sich heraus, dass die Entwickler vergessen haben den gRPC Healthcheck in der DAPI auszuschalten. Somit gab es keine unmittelbare Lösung für das Problem auf dem loaklen Devnet.

Als Ausweichlösung entschieden wir uns, das Skript auf dem globalen Testnet zu erproben. Bei der Ausführung des Skriptes wurde durch die nebenläufige Erstellung von Identities schnell das *IP-Rate-Limit* erreicht, sodass keine weiteren Anfragen von der selben IP-Adresse durch die Masternode angenommen wurden. Alternativ hätten die Anfragen über mehrere Proxies geleitet werden können, jedoch standen im Projekt keine Proxies zur Verfügung. Aufgrund der fortgeschrittenen Zeit konnten keine weiteren Ansätze getestet werden. Die Weiterentwicklung des Ansatzes könnte durch anschließende Projekte erfolgen.

Während der Ausführung des Skriptes auf dem globalen Testnet konnten folgende Verhaltensweisen in der DashPay Android App beobachtet werden. Die App meldete die Detektion eines versuchten *Double Spends*. Zusätzlich wurden mehrmals drei bis fünf Testdash auf die verwendete Wallet überwiesen. Die Dash Entwickler wurden über dieses Verhalten informiert.

5.3 DPNS

Der mit DIP0012 eingeführte Dash Platform Name Service, kurz DPNS, stellt eine Erweiterung des Konzeptes von Nutzer-Identities dar [32]. DPNS wird durch spezifische DPNS-Contracts auf Layer 2 realisiert und ermöglicht den Nutzern einen eigenen Namen festzulegen, der mit ihrer Identity verknüpft ist. Diese Namen werden unter anderem von DashPay verwendet, um den Nutzern benutzerfreundliche Transaktionen und übersichtliche Kontaktlisten bieten zu können, ohne dass sich der Benutzer lange Public Key Hashes merken, diese eingeben oder vergleichen muss. Das Prinzip ist vergleichbar mit der Namensauflösung von DNS, bei der der Nutzer keine IP-Adresse, sondern lediglich den DNS-Namen einer Webseite eingibt. Die zugehörige IP-Adresse für den Verbindungsaufbau wird dann im Hintergrund ermittelt, ohne das der Nutzer hiervon etwas bemerkt.

Diese DPNS-Namen sind sehr komfortabel für den Benutzer, bergen aber potenzielle Gefahren. Ein Angreifer könnte probieren, Geld von seinem Opfer zu erbeuten, indem er sich einen dem gewollten Empfänger-Namen ähnlichen Namen erstellt und das Opfer dazu verleitet an diesen zu zahlen. Dies könnte zum Beispiel durch die Verwendung von UTF-Zeichen erreicht werden, die bei bestimmten Schriftarten ein ähnliches Erscheinungsbild haben oder durch die überlange UTF-8-Kodierung eines Zeichens, sofern der UTF-8-Decoder auch längere, nicht kanonische Kodierungen zulässt [21]. Ein solcher Angriff ist nicht unrealistisch, was auch durch einen ähnlichen Angriff im Play Store von Google im Jahr 2017 deutlich wird [39]. Es muss daher sichergestellt werden, dass die gewählten Namen vor der Registrierung ausreichend validiert werden.

5.3.1 Validierung von Namen

Um der Frage nachzugehen, ob eine Validierung der Namen durchgeführt und welche Namen vom Netzwerk akzeptiert werden, wurde mittels Dash SDK (Node.js) probiert einen Namen für eine Identity zu registrieren, der einen UTF-Charakter in Form eines Emojis beinhaltet. Dieser Versuch wurde bereits vor dem Absenden durch die clientseitige Validierung des Dash SDKs verhindert. Schaltet man die clientseitige Validierung aus, erhält man als Antwort auf seinen Registrierungsversuch eine Konsens-Fehlermeldung vom Masternode zurück. Das gleiche gilt für überlange Namen. Es findet also sowohl eine client- als auch eine serverseitige Validierung statt. Die Validierung erfolgt mittels des regulären Ausdruckes `^[a-zA-Z0-9][a-zA-Z0-9-]{0,61}[a-zA-Z0-9]$` und bietet so keine Möglichkeit des Angriffes durch bestimmte Sonderzeichen. Lediglich die Verwendung von ähnlichen Zeichen wie zum Beispiel das kleine „L “ und das große „i “ zur Täuschung eines Nutzers wären hier möglich.

5.3.2 Angriffe durch die Suche nach DPNS-Namen

Neben dem Anlegen von ähnlichen Namen, was als Angriff auf einen Benutzer abzielt, muss auch die Suche nach DPNS-Namen auf potenzielle Angriffsvektoren untersucht werden. Grundsätzlich existieren zwei Möglichkeiten zur Suche nach DPNS-Namen: Die exakte Suche nach einem bestimmten Namen und die Wildcard-Suche nach einem beliebigen Suffix [26]. Die exakte Suche bietet kein Angriffspotenzial, da nur genau ein Name gesucht und übertragen werden kann. Die Wildcard-Suche bietet hingegen die Möglichkeit den Beginn eines Namens festzulegen und nach allen DPNS-Namen zu suchen, die mit diesem beginnen. Zu klären ist hierbei was geschieht, wenn eine Wildcard-Suche sehr viele Namen liefert. Durch die begrenzte Größe von gRPC-Payloads von 4 MB [37] können nicht unbegrenzt viele DPNS-Namen übertragen werden. Zur Überprüfung des Verhaltens in diesem Fall wurden über 1.000 DPNS-Namen angelegt, die einen einheitlichen Präfix besitzen und eine Wildcard-Suche nach diesem Präfix durchgeführt. Dabei hat sich herausgestellt, dass die Wildcard-Suche auf 100 Ergebnisse begrenzt ist. Ein Ausreizen der maximalen gRPC-Payload-Größe ist damit selbst bei Namen maximaler Länge nicht möglich.

Ein weiterer potenzieller Angriffsvektor ist die Möglichkeit, Last auf einem Masternode durch das Stellen von Wildcard-Anfragen zu erzeugen, was bei ausreichender Last zu einem Denial-of-Service führen könnte. Die Wildcard-Suche hat eine Laufzeit von O(n), da jedes Zeichen des gesuchten Präfixes mit jedem Präfixzeichen jedes Namens auf Gleichheit geprüft werden muss, bis ein abweichendes Zeichen gefunden worden ist. Der Suchaufwand kann durch geschickt gewählte DPNS-Namen wie folgt maximiert werden. Man erstellt dazu möglichst viele Namen die einen möglichst langen identischen Präfix haben. Da die Wildcard-Suche wie zuvor beschrieben maximal 100 Ergebnisse liefert, darf bei maximal 99 Namen dasselbe Zeichen auf das gemeinsamen Präfix folgen. Fragt man nun mittels Wildcard-Suche das Präfix gefolgt von dem Zeichen ab, das bei maximal 99 Namen auf das Präfix folgt, so ist der Masternode gezwungen alle Namen durchzugehen und muss Zeichen für Zeichen auf Gleichheit prüfen. Er wird dabei bei den erstellten Namen jeweils erst beim letzten Zeichen bemerken, dass der Name nicht zum gesuchten Muster passt. Da die erzeugte Last mit der Anzahl der erzeugten Namen skaliert, müssen sehr viele DPNS-Namen angelegt werden. Dies stellt aber kein Hindernis dar, da das System zurzeit nur eine sehr geringe Sybilresistenz bezüglich des Anlegens von DPNS-Namen aufweist. Dies ist damit zu begründen, dass das Anlegen eines Namens nur wenige Credits kostet. Bei dem aktuellen Kurs von etwa 100$ pro Dash [3] und der festen Umwandlungsrate von einem Duff (= 0,00000001 Dash) zu 1000 Credits [28], ist der monetäre Aufwand zum Anlegen großer Mengen an Namen zu vernachlässigen. Zum Test dieses Szenarios wurde ein Tool in Javascript zur Registrierung von DPNS-Namen unter Verwendung des Dash SDKs nach dem in Tabelle 2 dargestellten Schema erstellt. Alle Namen beginnen mit einem identischen Präfix in Form von 56 Mal dem Buchstaben „A “, worauf ein Infix folgt, der bei der Suche später das letzte Zeichen darstellt. Im Anschluss folgt ein Zeichen als Thread-Index, um die Namensregistrierung parallelisieren

Name	Präfix	Infix	Thread Index	Counter
1	AAAAAAAAAAAAAAAAAAAAAAAAAA...A	**B**	Z	00000
...				
99	AAAAAAAAAAAAAAAAAAAAAAAAAA...A	**B**	Z	0002w
100	AAAAAAAAAAAAAAAAAAAAAAAAAA...A	**C**	Z	00000

Tabelle 2: Muster für DPNS-Namen für einen Angriff basierend auf der Wildcard-Suche

zu können. Zum Schluss folgt ein Bereich von fünf Zeichen zum Hochzählen (wobei die Zahlen von 0-9 und die Buchstaben a-z zum Hochzählen möglich sind, wodurch pro Thread 34^5 verschiedene Namen angelegt werden können).

Bei der Ausführung des Tools wurde durch die schnelle Registrierung von DPNS-Namen schnell das IP-Rate-Limit erreicht, bei dem keine weitere Anfragen von der selben IP-Adresse durch den Masternode angenommen werden. Das Aufteilen der Anfragen auf unterschiedliche Masternodes ist im Dash SDK nicht ohne Weiteres möglich. Um dieses Problem zu umgehen, hätte das Tool so angepasst werden müssen, dass es die Anfragen über verschiedene Proxys leitet. Da keine Proxys zur Verfügung standen, wurde die Erstellung von Namen wegen des IP-Limits stark ausgebremst. Neben der zu geringen Anzahl an angelegten Namen wird zur Überprüfung der erzeugten Last ein Masternode im Testnet benötigt, der volle Layer 2-Funktionalität bietet und dessen Ressourcenverbrauch während dem Stellen einer Wildcard-Suchanfrage nach dem Muster `search(AAAAAAAAAAAAAAAAAAAAAAAAAA...AB*)` beobachtet werden kann. Da diese aufgrund von Problemen mit Tenderdash ebenfalls nicht zur Verfügung stand, bleibt die Fortführung dieses Ansatzes eine Möglichkeit für anschließende Projekte. Ebenfalls konnte bei der schnellen Erstellung vieler Namen die in Listing 4 gezeigte Fehlermeldung beobachtet werden. Zu beachten ist hierbei, dass es sich um eine interne IP-Adresse auf dem Port der Kommunikation zwischen Tenderdash und der DAPI handelt. Tenderdash ist also nicht mehr per TCP-Socket für die DAPI erreichbar. Wie dieser Fehler zustande kommt und welche Auswirkungen er für den Masternode hat, benötigt ebenfalls weitere Untersuchungen auf einem Masternode mit Layer 2-Funktionalität und Zugriff auf die Log-Dateien.

Listing 4: Fehlermeldung bei der schnellen Erstellung vieler DPNS-Namen

```
MaxRetriesReachedError
  cause: [REMOTE STACK] Error: connect ECONNREFUSED 192.168.128.4:26657
  at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1146:16) {
  name: 'InternalServerError',
  code: 13,
  data: {
    stack: 'Error: connect ECONNREFUSED 192.168.128.4:26657\n' +
    'at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1146:16)'
},
```

6 Fuzzing

Fuzzing ist das Beschießen von Schnittstellen mit mehr oder weniger zufälligen bzw. unerwarteten Daten. Das Ziel ist ungewolltes Verhalten zu provozieren und zu überprüfen, wie robust die Anwendung hinter der Schnittstelle auf die gesendeten Daten reagiert. Erkannte Probleme bei der Verarbeitung der gesendeten Anfragen dienen als Anhaltspunkt für potenzielle Schwachstellen oder Angriffsvektoren und müssen daher im Anschluss weiter analysiert werden.

Die DAPI bietet zwei Schnittstellen als Entrypoints zu den Masternodes. Zum einen die gRPC- und zum anderen die JSON-RPC-Schnittstelle. Die JSON-RPC-Schnittstelle besitzt nur drei Endpunkte, die gemäß Dokumentation in Zukunft auf die gRPC-Schnittstelle migriert werden sollen [17, JSON-RPC Endpoints - gRPC Migration]. Aus diesem Grund wird in dieser Arbeit nur eine Möglichkeit zum Fuzzing der gRPC-Schnittstelle gesucht. Die gesuchte Lösung soll um zusätzliche Endpunkte erweitert werden können, sodass der Fuzzer nach der Migration der JSON-RPC-Endpunkte auch für diese genutzt werden kann.

Aus der Wahl der gRPC-Schnittstelle für das Fuzzing resultiert, dass die gesuchte Lösung mit den beim gRPC-Protokoll genutzten Protokoll Buffern umgehen können muss. Das gRPC-Protokoll verwendet eine Interface-Sprache zur Beschreibung der Struktur der übertragenen Daten in so genannten proto-Definitions-Dateien. Diese ermöglichen eine effiziente Umwandlung der gesendeten Daten in einen Bytestrom. Der Fuzzer muss die Daten daher mit der definierten Struktur erstellen und mittels gRPC in einen Bytestrom umwandeln, um diese an die DAPI senden zu können. Die dafür gefundenen möglichen Lösungen werden im folgenden Abschnitt beschrieben.

6.1 Betrachtete Ansätze

Ein beliebter Standard-Fuzzer ist American Fuzzy Lop (kurz AFL) [25]. Es handelt sich um einen Smart Fuzzer, der das zu testende Programm mit einem speziellen Compiler kompiliert und dabei Hilfsfunktionen zum Verfolgen des Kontrollflusses in das Programm einbaut. Dadurch kann AFL das Verhalten des Programms auf bestimmte Eingaben nachvollziehen und seine Anfragen intelligent daran anpassen, um eine möglichst hohe Code-Abdeckung durch die Anfragen zu erreichen. Der Compiler funktioniert allerdings nur für C++-Programme. Eine einfache Lösung wäre es, die NodeJS-DAPI mit einem C++-Wrapper zu ummanteln, wodurch AFL zwar anwendbar wäre, aber auch seine Intelligenz verlieren würde, da die Hilfsfunktionen nur im Wrapper vorhanden und damit nutzlos wären.

Sowohl Fuzzino [22] als auch proto-fuzzer [43] bieten die Möglichkeit Payload-Daten anhand von Vorlagen zu generieren, welche dann durch einen selbst entwickelten Client an die DAPI gesendet werden könnten. Die Vorlagen zur Generierung der Payload-Daten müssen aber für

jeden Endpunkt einzeln erstellt werden, was sehr aufwändig geworden wäre. Zudem waren die erzeugten Payloads bei einem ersten Test wenig erfolgversprechend.

Die folgenden beiden Ansätze verwenden die proto-Dateien, wodurch automatisch passende Daten für die Endpunkte generiert werden können. Mayhem for API [23] kann als Reverse-Proxy eingesetzt werden und bietet eine REST-Schnittstelle, die mit einem beliebigen Rest-Fuzzer gefuzzt werden kann. Mayhem übersetzt dabei die REST-Anfragen nach gRPC. Dafür müssen zu Beginn einmal die proto-Dateien konvertiert werden. Das Problem hierbei ist, dass es sich um eine kommerzielle Software handelt und diese zusätzlich zur Zeit noch experimentell ist. ProtoFuzz [42] liest ebenfalls die proto-Dateien ein und generiert für diese passende Daten aus einer Datenbank mit potenziell gefährlichen Payloads (FuzzDB [36]). Die generierten Daten können im Anschluss durch einen selbst entwickelten Client an die DAPI übertragen werden.

Zur Umsetzung wird ProtoFuzz eingesetzt, da es den besten Kompromiss aus Qualität der generierten Payloads und zeitlichem Aufwand zur Umsetzung bietet. Es werden bei diesem Ansatz keine zufälligen Daten generiert, sondern es werden gezielt Daten erzeugt, die bereits in anderen Anwendungen zu Fehlern geführt haben. Damit wird überprüft, ob die DAPI robust gegenüber diesen bekannten Schwachstellen ist.

6.2 Implementierung

Mit Hilfe von ProtoFuzz werden die proto-Dateien eingelesen und Daten für die möglichen Parameter generiert. Aufgrund der Menge an Endpunkten wird Multithreading verwendet. Während der Entwicklung hat sich herausgestellt, dass die Masternodes nicht mehr als zwei Anfragen von einer IP-Adresse innerhalb einer Sekunde akzeptieren. Aus diesem Grund werden maximal zwei Anfragen pro Sekunde an denselben Masternode gesendet. Um dies zu realisieren wird pro Thread ein anderer Masternode angesprochen und die Zeit zwischen Anfrage und Antwort gemessen. Liegt diese unter 500 Millisekunden, wird der Thread pausiert, bis die 500 Millisekunden erreicht sind. Dies stellt sicher, dass alle Anfragen verarbeitet werden und gleichzeitig durch die parallele Verwendung mehrerer Masternodes der zeitliche Aufwand nicht zu stark ansteigt. Um den zeitlichen Aufwand weiter zu senken, wurde die maximale Anzahl von Anfragen pro Endpunkt auf eine Million begrenzt, die nur bei einem Endpunkt (getDocuments) aufgrund der Menge von Parametern überschritten worden wäre.

Für jeden Endpunkt wird in einer Protokoll-Datei jede Anfrage mit der zugehörigen Antwort gespeichert, um diese im Nachgang analysieren zu können.

6.3 Auswertung

6.3.1 Ansatz

Durch das Speichern jeder Anfrage mit jeder Antwort ist eine enorme Datenmenge entstanden, die nicht mehr händisch ausgewertet werden kann. Aus diesem Grund wurde ein Skript entwickelt, welches für jedes Paar von Anfrage und Antwort aus einer Protokoll-Datei eines Endpunktes den Status-Code der Antwort ausliest und daraus einen Bericht erstellt. Dieser Bericht beinhaltet eine Übersicht über die Verteilung der aufgetretenen Status-Codes, zeigt aufgetretene Fehlermeldungen aggregiert an und führt für diese jeweils auf, welche Eingaben zu dieser Meldung geführt haben. Dadurch können schnell die relevanten Anfragen ermittelt und weiter analysiert werden. Ein Beispiel eines solchen Berichtes wird in Listing 5 gezeigt.

Besonders interessant für die weitere Analyse sind die Fehlermeldungen mit dem Status-Code INTERNAL, da diese auf einen internen Fehler bei der Verarbeitung der Anfrage hindeuten. Aber auch die Status-Codes UNAVAILABLE, RESCOURCE_EXHAUSTED und DEADLINE_EXCEEDED sind wichtig, da sie auf einen Ausfall des Masternodes hindeuten können, der durch die gesendete Anfrage ausgelöst worden sein könnte.

Listing 5: Gekürztes Beispiel eines Berichtes zur Auswertung des Endpunktes „GetTransaction“ (gekürzte Stellen sind mit „...“ gekennzeichnet)

```
{
 "ID": "GetTransactionRequest",
 "TESTNET": true,
 "REQUESTS": 12417,
 "SUCCESS": 0,
 "EXCEPTION": 12417,
 "ERROR_CODES_RAW_DISTRIBUTION": {
   "INTERNAL": 12414,
   "RESOURCE_EXHAUSTED": 3
 },
 "ERROR_CODES_QUOTA_DISTRIBUTION": {
   "INTERNAL": 0.9997583957477651,
   "RESOURCE_EXHAUSTED": 0.00024160425223483932
 },
 "OCCURED_ERROR_MESSAGES": {
  "0": {
   "error_returned": "... StatusCode.INTERNAL\n\tdetails = \"parameter 1 must be
       hexadecimal string not the provided input ...}",
   "count": 12239,
   "occuredByInput": [
    "id: \"C:/boot.ini\"\n",
    "id: \"/0x2e0x2e\\\\0x2e0x2e\\\\0x2e0x2e\\\\{FILE}\"\n",
    "id: \"/%25c0%25ae%25c0%25ae\\\\%25c0%25ae%25c0%25ae\\\\{FILE}\"\n",
    "id: \"/\\\\\\\\\\\\\\\\%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%5c{FILE}\"\n",
    "id: \"/AAAAAAAA...A/../../../../../../../../{FILE}\"\n",
    "id: \"//..\\\\\/..\\\\\/..\\\\\/..\\\\\{FILE}\"\n",
    "id: \"<!--#echo var=\\\"document_name\\\" -->\"\n",
    "id: \"{\\\"0\\\":[\\\"1\\\",\\\"2\\\"]}\"\n",
    "id: \"{\\\"toJSON\\\":\\\"while(1);\\\"}\"\n",
```

```
29      "id: \"<iframe src=\\\"data:text/html,%3C%73%63%70%74%3E\\\"></iframe>\"\n",
30      "id: \"<\"\n",
31      "id: \"%25253C\"\n",
32      "id: \"&lt;\"\n",
33      "id: \"dir+c:\\\\\\\"\n",
34      "id: \"& ping -i 30 127.0.0.1 &\"\n",
35      "id: \"/etc/hostname.bge\"\n",
36      "id: \"%20s\"\n",
37      "id: \"%d%d%d%d%d%d%d%d%d%...d%d%d%d%d%d%d%d%d%d%d%d\"\n",
38      "id: \"////////////////////////////...////////////////\"\n"
39      ...
40    ]
41  },
42  "1": {
43    "error_returned": "... StatusCode.INTERNAL\n\tdetails = \"Parse error at /usr/
          src/app/lib/externalApis/dashcore/rpc.js:151:14\" ... ",
44    "count": 118,
45    "occuredByInput": [
46      "id: \"\\303\\274\\303\\261\\303\\270\\303\\260\\303\\251@address.com\"\n",
47      "id: \"address@\\303\\274\\303\\261\\303\\260\\303\\251.com\"\n",
48      "id: \"\\320\\201\\320\\202\\320\\203\\320\\204\\320\\...\\217\"\n",
49          ...
50    ]
51  },
52  "2": {
53    "error_returned": "... StatusCode.RESOURCE_EXHAUSTED\n\tdetails = \"received
          trailing metadata size exceeds limit ...",
54    "count": 3,
55    "occuredByInput": [
56      "id: \"[\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",...,\\\"1\\\"]\"\n",
57      "id: \"[1,1,...,1,1]\"\n",
58      "id: \"-99999999999999999999999999999999...99999999999999999\"\n"
59    ]
60  },
61  "3": {
62    "error_returned": "... StatusCode.INTERNAL\n\tdetails = \"parameter 1 must be of
          length 64 (not X [replaced]) ...",
63    "count": 16,
64    "occuredByInput": [
65      "id: \"00\"\n",
66      "id: \"fc\"\n",
67      "id: \"0000\"\n"
68      "id: \"4294967295\"\n",
69      "id: \"00e183908585139435661145466033750546974 5\"\n",
70          ...
71    ]
72  }
73 }
```

6.3.2 Analyse der Ergebnisse

In diesem Abschnitt werden die Ergebnisse aus der Ausführung des Fuzzers auf dem Testnet beschrieben. Angewendet wurde der Fuzzer sowohl im Devnet als auch im Testnet. Da sich die

Ergebnisse im Devnet nur marginal von den Ergebnissen im Testnet unterscheiden und diese zu denselben Erkenntnissen führen, werden die Devnet-Ergebnisse hier nicht gesondert aufgeführt. Es wurden sowohl alle Platform- als auch die Core-Endpunkte getestet, da diese Teil der DAPI sind. Die einzige Ausnahme ist der Core-Endpunkt `getStatus`, da dieser keine Parameter besitzt. Eine Liste der vorhandenen Endpunkte, ihrer Funktion und ihrer Parameter kann der Dokumentation [17] entnommen werden. Neben den dort genannten Endpunkten konnte in den proto-Dateien noch der Core-Endpunkt `subscribeToBlockHeadersWithChainLocks`, der aber auf den Masternodes nicht implementiert ist und der Platform-Endpunkt `getConsensusParams` identifiziert werden.

Über den Core-Endpunkt `broadcastTransaction` kann eine Layer 1-Transaktion über einen Masternode verbreitet werden. An diesen Endpunkt wurden 49.668 Anfragen gesendet, von denen alle wegen ungültiger Argumente abgelehnt worden sind. Dies ist darauf zurückzuführen, dass keine gültigen Transaktionen gesendet wurden und ist damit ein positives Ergebnis.

Bei der Abfrage eines Layer 1-Blockes mittels `getBlock` wurden 136.582 der 136.587 Anfragen wegen ungültiger Argumente abgewiesen. Das liegt daran, dass keine gültigen Hashes oder Blockhöhen abgefragt worden sind. Die restlichen fünf Anfragen erhielten keine Antwort innerhalb des Timeouts von fünf Sekunden. Bei der manuellen Übertragung dieser Anfragen konnte dies nicht reproduziert werden, auch dann nicht wenn einige der zuvor gesendeten Anfragen erneut gesendet wurden. Diese manuellen Anfragen führten ebenfalls zur Meldung ungültiger Argumente. Es ist daher nicht davon auszugehen, dass dieser Fehler aufgrund der gesendeten Anfragen auftrat.

Bei dem Core-Endpunkt `getTransaction` führten 12.414 Anfragen zu einem internen Fehler. Diese unterteilen sich in die folgenden drei Fehlermeldungen: Es wurde ein hexadezimaler String erwartet, aber keiner geliefert. Es trat ein Parsing Fehler auf und Anfragen wurden aufgrund falscher Parameterlängen abgewiesen. All diese Fehler sind äquivalent zu ungültigen Argumenten und damit nicht kritisch. Zusätzlich gab es drei Anfragen, die zu dem Status-Code RESOURCE_EXHAUSTED geführt haben. Dieses Verhalten ist mit manuellen Anfragen reproduzierbar, allerdings entsteht dabei keine Last oder Ähnliches auf dem Masternode. Der Fehler resultiert aus zu langen Metadaten.

Bei dem letzten Core-Endpunkt `subscribeToTransactionsWithProofs`, der zur Abfrage von Transaktionen anhand eines Bloom-Filters dient, erhielt keine der 577.293 Anfragen eine Antwort innerhalb des Timeouts von fünf Sekunden. Der Grund liegt darin, dass kein Bloom-Filter übermittelt wurde, der einer gültigen Transaktion entspricht.

Der erste Platform-Endpunkt, `broadcastStateTransition`, dient der Übertragung einer State Transition. Hier wurden von 12.417 Anfragen 9.447 wegen ungültiger Argumente abgelehnt. Weitere 2.888 Anfragen ergaben, dass die übermittelte State Transition bereits existiert. Die restlichen 82 Anfragen führten zu einem internen Serverfehler, da an einer

erwarteten Stelle keine Protokollversion gefunden werden konnte, was auf die ungültigen Eingaben zurückzuführen ist.

Zur Abfrage, ob ein bestimmter State Transition-Hash in einem Block bestätigt wurde, wird der Endpunkt `waitForStateTransitionResult` verwendet. Hier erhielt keine der 24.834 Anfragen eine Antwort innerhalb des Timeouts. Dies macht Sinn, da keine State Transitions mit den übertragenen Hash-Werten erzeugt wurden.

Sowohl der Platform-Endpunkt `getIdentity`, zur Abfrage von Identities, als auch der Endpunkt `getContract` für die Abfrage von Data Contracts, führten zu identischen Ergebnissen. Jeweils 50% der 24.834 Anfragen waren erfolgreich und die anderen 50% führten zu der Meldung, dass die Identity bzw. der Data Contract nicht gefunden werden konnte. Die nicht gefundenen Identities und Data Contracts sind ein zu erwartendes Ergebnis, da keine gültigen IDs übertragen wurden. Die „erfolgreichen“ Anfragen sind auf die bei Dash Platform verwendeten Platform Proof zurückzuführen [27]. Es handelt sich dabei um einen Boolean-Parameter in der Anfrage, der angibt, dass die Antwort einen Beweis enthalten soll, der die Korrektheit der Antwort sicherstellt. Dieser Parameter war bei 50% der Anfragen gesetzt, wodurch die Antwort hier eine leere Identity bzw. einen leeren Data Contract mit dem zugehörigen Beweis enthielt.

Bei der Abfrage von Identites bzw. IdentityIDs anhand von Public Key Hashes mittels `get-IdentitiesByPublicKeyHashes` bzw. `getIdentityIdsByPublicKeyHashes` waren jeweils alle 24.834 Anfragen erfolgreich. Dies ist damit zu begründen, dass die Endpunkte für nicht gefundene Public Key Hashes eine leere Liste von Identites bzw. IdentityIDs übertragen.

Der Platform-Endpunkt zur Abfrage von Documents, `getDocuments`, besitzt sehr viele Parameter, weswegen dieser Endpunkt auf eine Million Anfragen begrenzt wurde. Von diesen wurden 712.258 korrekterweise wegen ungültiger Argumente abgewiesen. Weitere 227.740 Anfragen führten aufgrund eines Parsing-Fehlers zu einem internen Serverfehler ohne Auswirkungen. Bei 60.000 Anfragen konnte keine Verbindung mit dem Masternode aufgebaut werden. Da sich diese Anzahl exakt durch die verwendete Größe zur Aufteilung der Anfragen von 10.000 Anfragen pro verwendeten Thread (und damit pro Masternode) teilen lässt, liegt die Vermutung nahe, dass einer der Masternodes aus der Liste statischer IP-Adressen während des Fuzzing-Vorganges nicht erreichbar war. Manuelle Tests im Nachhinein bestätigten dies. Aufgrund der bereits großen Menge durchgeführter Anfragen wurden die 60.000 Anfragen nicht erneut getestet, da kein anderes Ergebnis als bei den anderen Anfragen zu erwarten ist. Zwei weitere Anfragen führten ebenfalls zum Status-Code UNAVAILABLE einmal aufgrund der Schließung des Sockets und einmal mit der Meldung „Documents temporary unavailable“. Diese konnten aber nicht reproduziert werden.

Der letzte Platform-Endpunkt `getConsensusParams` liefert Informationen über die Konsensparameter, die bei einem bestimmten Block gültig waren, wie zum Beispiel die maximale Blockgröße. Hier waren von 24 Anfragen zwölf aufgrund des gesetzten „prove“-Parameters

ungültig, da für diesen Endpunkt die Platform-Proofs noch nicht implementiert wurden. Die restlichen zwölf Anfragen teilen sich in drei erfolgreiche Anfragen, da gültige Blockhöhen übermittelt wurden und neun interne Fehler aufgrund der Übertragung ungültiger Blockhöhen.

6.3.3 Fazit

Insgesamt wurden knapp zwei Millionen Anfragen an verschiedene Masternodes gesendet. Bei keinem dieser Anfragen konnten Systemabstürze, ungewolltes Verhalten oder ähnliches beobachtet werden. Dieses positive Ergebnis kann entweder aus einer sicheren Implementierung oder aus schlechten Payloads resultieren, die zu wenige Fälle abdecken. Da die Payloads aus der FuzzDB generiert werden, in der viele Angriffsmuster für gängige Schwachstellen enthalten sind, sind schlechte Payloads als Ursache auszuschließen. Das Ergebnis zeigt daher, dass die DAPI robust gegenüber diesen Eingaben ist, was auf eine gute und sichere Implementierung schließen lässt.

7 DashPay Android-App

Um eine benutzerfreundliche Schnittstelle für den Data Contract der dezentralen Anwendung *DashPay* [8] zu garantieren, existiert eine sich noch in der Entwicklung befindende Android-App[1], die von Dash zur Verfügung gestellt wird.

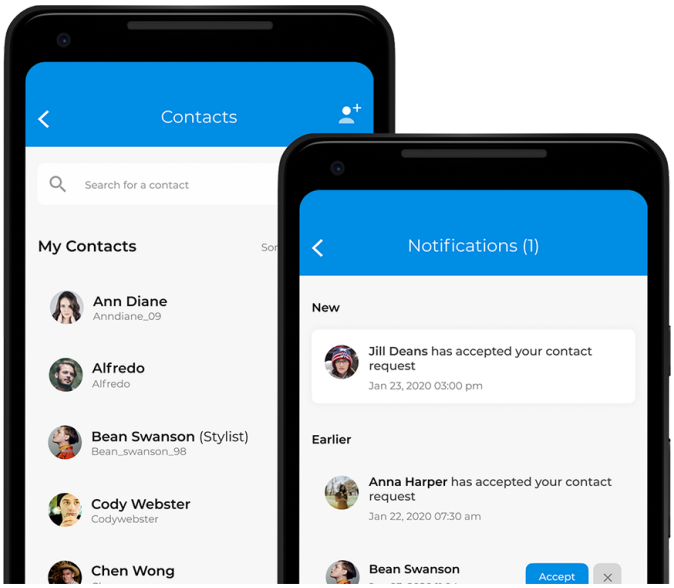

Abbildung 15: Illustration der Benutzeroberfläche von DashPay [8]

Das Ziel von *DashPay* ist es, die nutzerunfreundliche Verwendung von Kryptoadressen für den Zahlungsverkehr durch ein kontaktbasiertes System zu ersetzen. Nutzer können Profile erstellen, die einer Identity zugeordnet sind und sich dann gegenseitig über einen freiwählbaren und eindeutigen DPNS-Eintrag als Profilname suchen und vernetzen. Über diese Profile (welche mithilfe von Dash Platform auf Layer 2 persistiert werden), können die Nutzer dann untereinander Layer 1 Zahlungskanäle durch das ableiten von passenden Kryptoadressen erstellen und sich somit bidirektional Dash über Layer 1 senden. Der nutzerunfreundliche Teil, welcher das direkte Handhaben von Kryptoadressen enthält, wird durch die Benutzerschnittstelle von *DashPay* wegabstrahiert und bleibt dem Nutzer somit verborgen. Der Nutzer muss sich lediglich darum kümmern, Kontakte mit den zukünftigen Zahlungspartnern aufzubauen. *DashPay* bietet somit also eine Benutzererfahrung die vergleichbar mit der von PayPal[2] ist, aber auf einem dezentralisierten Zahlungssystem basiert.

[1] https://github.com/dashevo/dash-wallet/tree/dashpay
[2] https://www.paypal.com/de/home

7.1 Ansatz

Dadurch, dass die Android-App eine direkte Benutzerschnittstelle zu *DashPay* darstellt, kommt ihr eine gewisse Relevanz zu. Es reicht in unseren Augen nicht aus, uns nur auf Layer 1 und 2 zu fokussieren, sondern zumindest auch die außerhalb von Dash und Dash Platform liegenden Benutzerschnittstellen zu beäugen. Unsere Idee für das Testen auf Schwachstellen und Fehler in der Android-App basiert darauf, dass wir die App auf unser lokales Testnetzwerk umleiten und dann schauen, ob z. B. Namen angelegt werden können, die nicht den jeweiligen Validierungsstandards [12] entsprechen. Unser Ziel ist es dann zu beobachten, ob Anomalien in der Funktion der Android-App auftreten wie z. B. undefiniertes Verhalten, fehlerhafte Zustände oder sogar Abstürze/Crashs.

7.2 Durchführung

Um die in Unterabschnitt 7.1 beschriebene Vorgehensweise umzusetzen, ist es notwendig, die Android-App insoweit zu verändern, das eine Verbindung zu unserem lokalen Testnetzwerk aufgebaut werden kann. Der erste Versuch bestand daraus einen der verfügbaren APK-Decompiler[3] zu verwenden um anschließend die IP-Adressen der Masternodes zu ändern und aus dem Quellcode anschließend wieder eine APK (Android Package) zusammenzubauen. Bei dieser Vorgehensweise haben wir allerdings recht schnell gemerkt, dass unser vorhandenes Tooling ohne extensive Android-Erfahrung unzureichend ist, um wieder bei einer installierbaren APK zu landen. Daher entschieden wir, uns den Dash-Entwickler *hashengineering*[4], der die Android-App betreut zu kontaktieren und mit ihm die Notwendigkeiten abzusprechen.

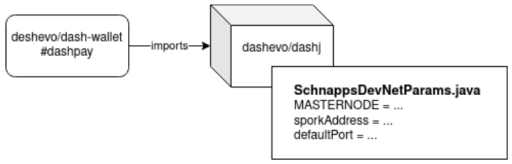

Abbildung 16: Notwendige Änderungen an der *DashPay* Android-App, um sich mit einem eigenen Netzwerk zu verbinden

Wie in Abbildung 16 zu sehen ist es dafür notwendig eine Version der Android-App mit einer modifizierten *DashJ*[5]-Abhängigkeit (einem Fork von *BitcoinJ*[6]) zu bauen und in dieser die Parameter in der Java-Klasse *SchnappsDevNetParams.java* auf die von unserem lokalen

[3] http://www.javadecompilers.com/apk
[4] https://github.com/HashEngineering
[5] https://github.com/dashevo/dashj
[6] https://bitcoinj.org/

Testnetzwerk anzupassen. Diesmal ist es uns auch gelungen, die Android-App erfolgreich zu bauen und eine APK als Artefakt zu generieren. Eine direkte Installation dieser war allerdings noch nicht möglich, da Android-Apps zumindest selbstsigniert sein müssen, um vom System akzeptiert zu werden.

Abbildung 17: Wirkungskette zum Build-Prozess der *DashPay* Android-App

Wie in Abbildung 16 zu erkennen ist, hat auch das selbstsignieren mithilfe des Werkzeuges *jarsigner*[7] keine Abhilfe bezüglich der Gültigkeit der aus dem Build-Prozess entstandenen APK geschaffen. Es war nicht möglich, die APK auf einem lokalen Gerät oder einem Emulator zu installieren. Auch Methoden wie die Installation über *ADB*[8] lieferten hier keinen Erfolg. Die Installation endete immer in einer Fehlermeldung, die auf ein invalides Paket hinweist. Es war uns aufgrund der fortgeschrittenen Zeit und mangelnder Android-Erfahrung im Team nicht mehr möglich, dieses Problem zu beheben, sodass wir unseren in Unterabschnitt 7.1 definierten schlussendlich Ansatz nicht durchführen und verifizieren konnten.

[7] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html
[8] https://developer.android.com/studio/command-line/adb

8 Race Conditions auf Dash Platform

In diesem Abschnitt wird auf die gegebene Möglichkeit von Race Conditions eingegangen, welche zu Problemen bei der Konsistenz von den Daten dezentraler Anwendungen führen können. Zuerst werden die notwendigen Hintergrundinformationen bezüglich Dash Platform erläutert, um anschließend zwei Szenarien von Race Conditions vorzustellen und das Kapitel mit einem *educated guess* abzuschließen, da wir in der aufgrund der fortgeschrittenen Zeit im Projekt nicht mehr die Möglichkeit hatten, die Vermutungen durch praktische Tests zu überprüfen.

8.1 Hintergrund: Funktionsweise von Dash Platform

Wie in Abbildung 18 zu erkennen ist, greifen Klienten mithilfe der DAPI [9] auf die Layer 2 Serviceleistungen von Dash Platform zu. *State transitions*, welche Anweisungen zum Modifizieren des *platform states*, sprich den auf Dash Platform persitierten Daten enthalten werden über die DAPI von den Nutzern dezentraler Layer 2 Anwendungen an einem beliebiges (nämlich dem momentan Ausgewählten) Masternode eingereicht.

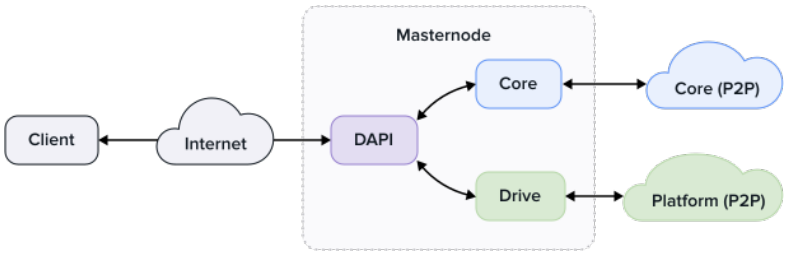

Abbildung 18: Schematische Darstellung der DAPI [9]

In der Abbildung 19 auf der nächsten Seite wird ersichtlich, wie die einzelnen Masternodes im Platform Chain Netzwerk den *platform state* persistieren. Über die sogenannte Platform State Machine werden die jeweiligen eingehenden *state transitions* auf den lokalen *platform state* des jeweiligen Knotens angewandt. Damit die Masternodes im gegenseitigen Austausch stehen und sich synchronisieren, werden *state transitions* jeweils beim über die DAPI eingehenden Knoten auf die Gültigkeit der darin enthaltenen Daten validiert. Anschließend werden die *state transitions* geordnet und in einem Block geschrieben, der im Netzwerk propagiert wird [10]. Über den Tenderdash-Konsensmechanismus [13] wird garantiert das nur *state transitions*, die aus gültigen Blöcken, also Blöcken, bei denen mehr als 2/3 der Teilnehmer diesen als valide bestätigt haben, stammen von den Masternodes in den global gültigen *platform state* übernommen werden. In Abbildung 20 auf der folgenden Seite ist dieser Prozess in

dem die Eingereichten *state transitions* das Dash Platform Netzwerk durchwandern noch mal visuell verdeutlicht.

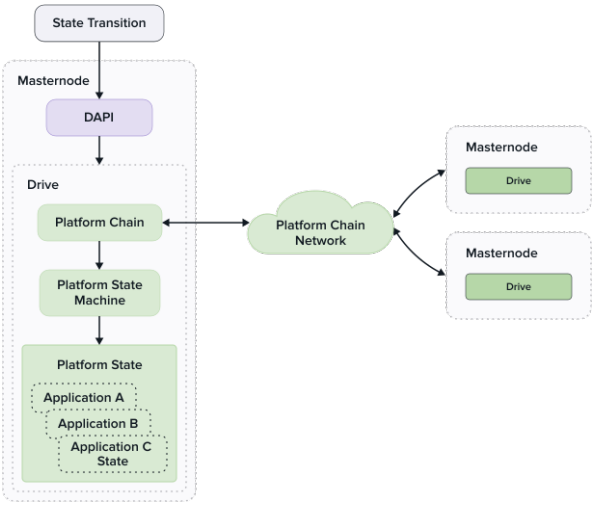

Abbildung 19: Persistierung des *Platform States* in Dash Platform [14]

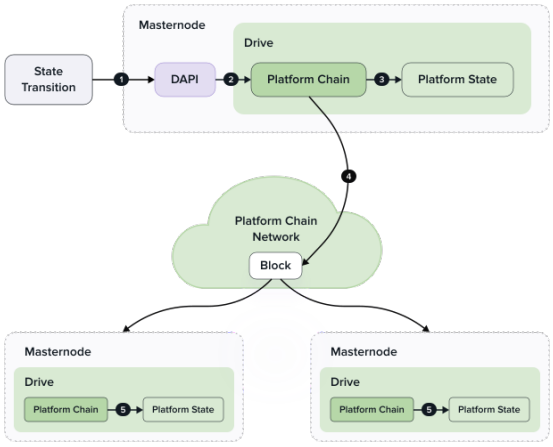

Abbildung 20: Propagation von *state transitions* über die DAPI in Dash Platform [10]

Mit dem groben Wissen über die Funktion von Drive, dem Datenspeicher von Dash Platform der die *state transitions* persistiert, stellt sich nun die Frage, inwiefern in diesem Prozess möglicherweise Race Conditions auftreten können und inwieweit diese für Komplikationen sorgen.

8.2 Ansatz: Mögliche Arten von Race Conditions

Um diesen Sachverhalt eindeutig zu veranschaulichen, wurden zwei Szenarien möglicher Race Conditions ausgearbeitet und zum besseren Verständnis durch eigens erstellte Abbildungen visualisiert.

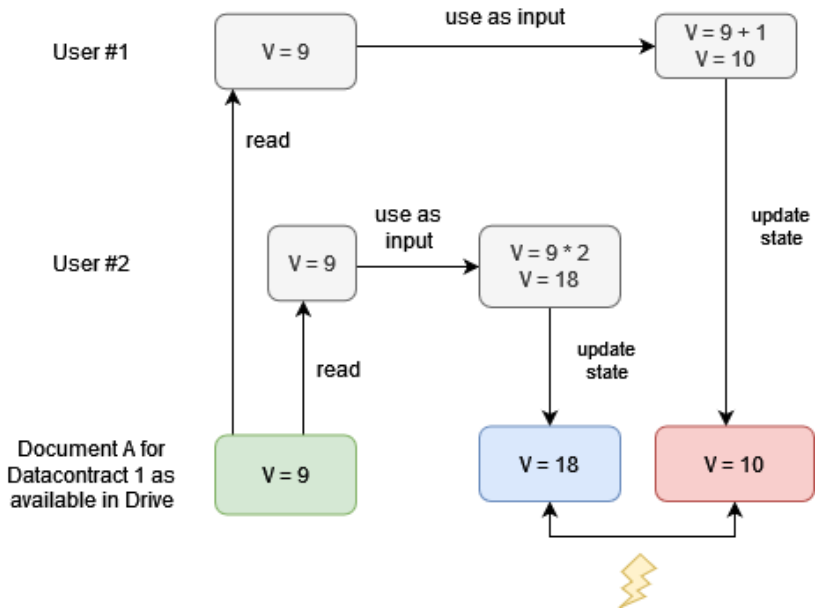

Abbildung 21: Szenario 1: Gleiches lesen, ungleiches schreiben

In dem ersten Szenario, das in der Abbildung 21 dargestellt ist greifen zwei Benutzer auf den gleichen Eingangszustand, nämlich $V = 9$ im *Document A* des Data Contracts *1* zu. Anschließend verwenden die jeweiligen Klienten der dezentralen Anwendung den Wert V als Eingabewert um lokale Berechnungen durchzuführen. Im Falle von Benutzer 1 wird $V = 9 + 1 = 10$ ausgeführt, während Benutzer 2 die Operation $V = 9 * 2 = 18$ durchführt. Beide Nutzer möchten nun zur gleichen Zeit ihre lokale Berechnung über eine *state transition* zum Persistieren im *platform state* von Dash Platform einreichen. Dabei entsteht ein Konflikt - in diesem Szenario haben beide Nutzer einen ***gleichen*** lesenden Zugriff, aber einen ***ungleichen*** schreibenden Zugriff.

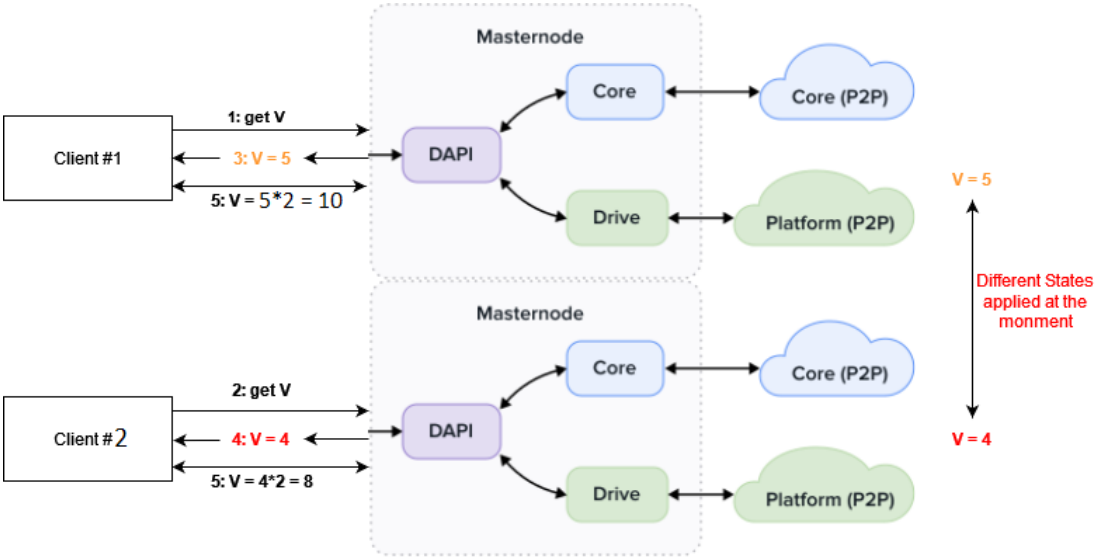

Abbildung 22: Szenario 2: Ungleiches lesen, ungleiches schreiben

In dem zweiten Szenario, das in der Abbildung 22 dargestellt ist greifen die zwei Benutzer nun aber auf zwei unterschiedliche Eingangszustände zu. Möglicherweise hat der Masternode vom zweiten Benutzer hier den aktuellen *platform state* noch nicht vollständig synchronisiert ($V = 4$), während der erste Benutzer schon die aktuellste Version mit $V = 5$ sieht. Diesmal führen beide Nutzer allerdings die gleiche Berechnung $V * 2$ durch. Da die Eingangswerte aber unterschiedlich kommen diesmal für Benutzer 1 ($V = 5$) und Benutzer 2 ($V = 4$) die Ergebnisse $V = 10$ und $V = 8$ heraus. Auch hier gibt es wieder eine Diskrepanz, sobald die beiden Nutzer ihren lokalen Ausgangswert über eine *state transition* im Datenspeicher von Dash Platform persistieren wollen.

8.3 Educated Guess: Folgen für Data Contracts

Nachdem der Sachverhalt nun detailliert geschildert wurde, stellt sich die Frage, inwiefern Dash Platform mit dem Problem der Race Conditions umgeht. In der Dokumentation [13] finden sich nur spärlich Details wie z. B.:

> *Tenderdash handles state transition ordering, block creation, and network communication related to the platform chain while Dash Platform Protocol ensures the consistency and integrity of the data itself.*

oder

> *Tendermint provides Byzantine Fault Tolerant (BFT) State Machine Replication via blocks containing transactions. Consensus is arrived at when more than 2/3 of the set of validator nodes vote for a proposed block and commit to it. Information regarding the voting is then included in the subsequent block to provide proof of the block commitment.*

aus den sich zwar schließen lässt, dass die Einkommenden *state transitions* zwar validiert, geordnet und in einem Block verewigt werden (welcher, dann von min. 2/3 der Netzwerkteilnehmer abgesegnet werden muss), nicht aber ob etwaige Race Conditions taktvoll einer Konfliktlösungsstrategie unterzogen werden oder überhaupt aufkommen können. Es scheint so, als ob durch Tendermint immer eine Version als die gültige und aktuelle *state transition* gewählt wird und durch das Ordnen ein, wer zuerst bei der Mehrheit ankommt, mahlt zuerst gilt, wodurch sich Race Conditions zwar vermeiden lassen (da, dann nur eine Wahrheit als die gültige akzeptiert wird), es dann aber mitunter Angriffsvektoren wie z. B. *front running* auftreten können.

Da wir aus zeitlichen Gründen keine praktischen Tests bezüglich des von uns vermuteten Sachverhaltes mehr durchführen konnten, bleibt offen, ob Race Conditions ein mögliches Problem auf Dash Platform darstellen. Es wäre in diesem Fall hilfreich, wenn einer der Dash-Entwickler zu diesem Sachverhalt eine Stellungnahme abgibt und aufklärt, was genau bei diesen zwei von uns konzipierten Sachverhalten geschieht und mit den daraus gewonnen Informationen die Dokumentation von Dash Platform ergänzt um diesen Sachverhalt auch für die Zukunft zu adressieren.

Sollten allerdings tatsächlich Race Conditions auf Dash Platform existieren, ist es geraten, keine Data Contracts zu deployen die in etwa accountbasierten Token wie z. B. auf dem Ethereum-Netzwerk entsprechen, um zu verhindern, das Werte durch Race Conditions oder mögliche *frontrunning*-Angriffe verloren gehen.

9 Spieltheoretische Angriffe

In unserem letzten Kapitel untersuchen wir, ob es potenzielle spieltheoretische Angriffe in Verbindung mit *Credits* [11], der Layer 2-Währung, welche auf Dash Platform zum Bezahlen von *state transitions* eingesetzt wird, die sowohl auf Layer 1 als auch auf Layer 2 für Probleme mit der Hauptwährung Dash sorgen können. Wie in Abbildung Abbildung 23 zu sehen ist werden *state transitions* dafür verwendet um die Zustände der auf Dash Platform persistierten und einem Data Contract zugehörigen Dokumente zu ändern. *State transitions* stellen somit einen integralen Teil der Funktionalität von Dash Platform dar.

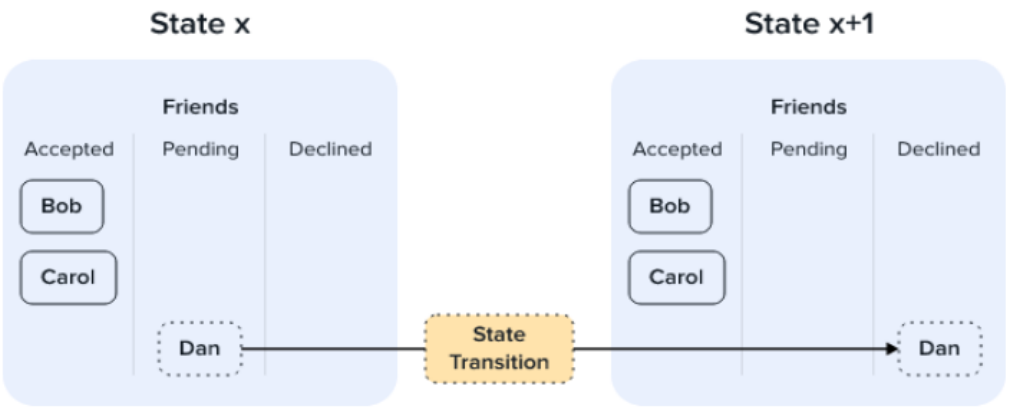

Abbildung 23: *State transitions* erlauben das Modifizieren von Zuständen in den auf Dash Platform persistierten Daten [15]

Credits können durch Sogenannte *asset lock transactions* [11] [16] auf Layer 1 beim Erstellen einer Identity erworben werden. In dem Prozess werden Dash auf Layer 1 gespeert und nach einer fixen Umwandlungsrate in *Credits* umgewandelt, die anschließend für das Einreichen von *state transitions* auf Dash Platform verwendet werden können. Bei diesem Lösungsansatz gibt es allerdings momentan zwei Probleme:

- Durch die fixierte Umwandlungsrate von *Credits* kann durch einen rasanten Anstieg des Marktwertes von Dash, welches als *underlying* für die *Credits* dient, ebenfalls zu einem rasanten Anstieg in den Kosten von *state transitions* kommen, was die Mehrheit von Benutzern von Dash Platform aussperren könnte.
- Momentan ist das Umwandeln von Dash in *Credits* eine Einbahnstraße.

Das erste Problem würde sich durch einen freien Markt, der eine dynamische Umwandlungsrate, die von Nachfrage und Angebot bestimmt ist, lösen. Demnach wären die *Credits*

allerdings nur etwas Wert, falls Dash Platform aktiv genutzt wird. Um Probleme wie Hohe *gas fees* wie z. B. auf dem Ethereum-Netzwerk zu beobachten, sind zu vermeiden, müsste Dash Platform allerdings gut skalieren und kein Hochbieten von *state transitions* durch die Benutzer ermöglichen.

Zu dem zweiten Problem findet sich in der Entwickler-Dokumentation der folgende Ausschnitt:

> *As of Dash Platform Protocol v0.13, credits deducted to pay state transition fees cannot be converted by masternodes back into Dash. This aspect of the credit system will come in a future release.* [11]

Daher lassen sich momentan keine Aussagen über mögliche spieltheoretische Angriffe in Verbindung mit *Credits* und Layer 2 treffen. Aktuell gilt das man ab dem Moment, wo man *Credits* erwirbt, im aktuellen System schon verloren hat, da das Geld weg ist und man stadtessen auf einem Berg von unumwandelbaren und somit wertlosen *Credits* sitzt. Das aktuelle System lässt sich daher wie in Abbildung 24 zu sehen ist metaphorisch am besten als eine Einbahnstraße mit einer dranhängenden Sackgasse beschreiben.

Abbildung 24: Passende Metapher zur Visualisierung der derzeitigen Situation bezüglich der Relation zwischen Dash und *Credits*

In dem aktuellen System ist ebenfalls fraglich, welche Initiativen es für Masternodes gibt, die Layer 2 Dienste bereitzustellen, da diese reale Hardware- und Operationskosten verursachen, welche nicht von unumwandelbaren *Credits* gedeckt werden können. Solange keine bidirektionale Umwandlungsmöglichkeit zwischen Dash $\Leftrightarrow$ *Credits* besteht, sind spieltheoretische Angriffe in Verbindung mit *Credits* und Layer 2 ausgeschlossen. Eine bidirektionale Umwandlungsmöglichkeit wird also für die Sinnhaftigkeit von Dash Platform dringend benötigt und sollte von den Entwicklern in zukünftigen Releases akribisch durchdacht, geplant und implementiert werden.

10 Fazit und Ausblick

Insgesamt scheint es, dass Dash Platform eine den Ansprüchen an ein *Multi-Purpose*-DLT genügende Sicherheit bietet. Bei genauerem Hinsehen ist dies auch nicht verwunderlich, basiert der Großteil der von Dash Platform bereitgestellten Funktionalitäten doch auf bekannter und bewährter Open Source Software wie zum Beispiel MongoDB, gRPC oder dem Envoy Reverse Proxy. Im weiteren Projektverlauf wurde jedoch klar, dass durchaus einige Angriffsvektoren existieren. So kann zum Beispiel durch den geeigneten Einsatz von Bloom Filtern ein Knoten zum Absturz gebracht werden, was potenziell das gesamte Netzwerk gefährden kann.

Mögliche Folgeprojekte könnten die Angriffe, die im Rahmen des Projekts zwar konzipiert, aber aufgrund von Zeitmangel nicht umgesetzt wurden, implementieren. Als Beispiel sei hier der in Unterabschnitt 5.3 erläuterte Angriff auf das DPNS genannt. Außerdem könnten zukünftige Versionen von Dash Platform auf neue Angriffsvektoren untersucht werden.

10.1 Statement zum Stand von Dash Platform

Bei der Durchführung des Projektes stießen wir immer wieder auf Schwierigkeiten, wenn es darum ging den Dash Platform-Softwarestack zu installieren und konfigurieren. Die von der Dash Group bereitgestellten Dokumentationen waren stellenweise veraltet, fehlerhaft oder redundant vorhanden. Außerdem scheint die Installation durch die vielen verwendeten Komponenten unnötig kompliziert und schwerfällig. Hier könnte evtl. eine Integration aller Komponenten des Platform-Stacks in eine monolithische Software abhilfe schaffen. Das Anbieten verschiedener Installationsbinaries für verschiedene Systeme würde die Installation weiter vereinfachen. Außerdem muss vor dem Release von Dash Platform auf dem Mainnet eine Möglichkeit implementiert werden, die es erlaubt Credits wieder in Dash umzuwandeln. Ansonsten fehlt der Anreiz für Betreiber von Masternodes die Platform Services anzubieten. Aufgrund dieser Tatsachen glauben wir, dass Dash Platform sich in Zukunft schwer tun wird, sich in Konkurrenz zu anderen Multi-Purpose-DLTs, wie *Ethereum* [2] oder *Cardano* [30] durchzusetzen.

Literatur

[1] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly, Sebastopol, 2017.

[2] Vitalik Buterin et al. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] CoinMarketCap. Dash to usd. https://coinmarketcap.com/currencies/dash/. Zugriff am 2022-02-05.

[4] Bitcoin Core. libsecp256k1. https://github.com/bitcoin-core/secp256k1. Zugriff am 2022-02-04.

[5] Dash. Dash SDK. https://github.com/dashevo/platform/tree/master/packages/js-dash-sdk#readme. Zugriff am 2022-02-04.

[6] Dash. Platform gRPC Endpoints - Detailed platform gRPC endpoint reference. https://dashplatform.readme.io/docs/reference-dapi-endpoints-platform-endpoints. Zugriff am 2022-02-04.

[7] Dash. Dash manual installation guide. https://docs.dash.org/en/stable/masternodes/setup-testnet.html#manual-installation, 2021. Zugriff am 2022-01-29.

[8] Dash. Dashpay - data contract enabling wallets to provide a user centric payment experience. https://dashplatform.readme.io/docs/explanation-dashpay, 2021. Zugriff am 2022-01-29.

[9] Dash. Decentralized api (dapi) - a decentralized api for dash. https://dashplatform.readme.io/docs/explanation-dapi, 2021. Zugriff am 2022-01-29.

[10] Dash. Drive - decentralized storage. https://dashplatform.readme.io/docs/explanation-drive, 2021. Zugriff am 2022-01-29.

[11] Dash. Identity - blockchain-based identifier for users. https://dashplatform.readme.io/docs/explanation-identity, 2021. Zugriff am 2022-01-29.

[12] Dash. Name service (dpns) - enabling name-based user experiences. https://dashplatform.readme.io/docs/explanation-dpns, 2021. Zugriff am 2022-01-29.

[13] Dash. Platform consensus. https://dashplatform.readme.io/docs/explanation-platform-consensus, 2021. Zugriff am 2022-01-29.

[14] Dash. Platform state - global state of all platform data managed by a replicated state machine. https://dashplatform.readme.io/docs/explanation-drive-platform-state, 2021. Zugriff am 2022-01-29.

[15] Dash. State transition - the means of storing and updating dash platform data. https://dashplatform.readme.io/docs/explanation-platform-protocol-state-transition, 2021. Zugriff am 2022-01-29.

[16] Dash. Topup an identity's balance - increase the balance of a dash platform identity. https://dashplatform.readme.io/docs/tutorial-topup-an-identity-balance, 2021. Zugriff am 2022-01-29.

[17] Dash Core Group. DAPI Endpoints. https://dashplatform.readme.io/docs/reference-dapi-endpoints.

[18] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin utxo set. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 78–91, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

[19] Drew Dennison. sgrep 0.4.0 release. https://github.com/returntocorp/semgrep/releases/tag/0.4.0, 2020. Zugriff am 2022-02-02.

[20] Advanced Micro Devices. Ryzen 5 2600x. https://www.amd.com/de/products/cpu/amd-ryzen-5-2600x, 2018.

[21] Alis Technologies F. Yergeau. Utf-8, a transformation format of iso 10646. https://datatracker.ietf.org/doc/html/rfc2279, 1998. Zugriff am 2022-02-04.

[22] Fraunhofer FOKUS. Fuzzino. https://github.com/fraunhoferfokus/Fuzzino, July 2021.

[23] ForAllSecure. Experimental: gRPC Fuzzing - Mayhem for API. https://mayhem4api.forallsecure.com/docs/grpc.html. Zugriff am 2022-02-02.

[24] GitLab. GitLab Docs - SAST Analyzers. https://docs.gitlab.com/ee/user/application_security/sast/. Zugriff am 2022-02-02.

[25] Google. american fuzzy lop. https://github.com/google/AFL, February 2022.

[26] Dash Core Group. Dash platform names search. https://dashevo.github.io/platform/SDK/platform/names/search/. Zugriff am 2022-02-04.

[27] Dash Core Group. Platform Proofs. https://dashplatform.readme.io/docs/reference-platform-proofs.

[28] Dash Core Group. Topup an identity's balance. https://dashplatform.readme.io/docs/tutorial-topup-an-identity-balance. Zugriff am 2022-02-04.

[29] Wireshark Group. Wireshark. https://wireshark.com/. Zugriff am 2022-02-05.

[30] Charles Hoskinson. Cardano whitepaper. https://whitepaper.io/document/581/cardano-whitepaper, 2017.

[31] Google inc. Leveldb. https://github.com/google/leveldb. Zugriff am 2022-02-04.

[32] Anton Suprunchuk Ivan Shumkov. Dip12: Dpns. https://github.com/dashpay/dips/blob/master/dip-0012.md, 2020. Zugriff am 2022-02-04.

[33] Eyal Katz. Top 10 static application security testing (sast) tools in 2021. https://spectralops.io/blog/top-10-static-application-security-testing-sast-tools-in-2021, 2021. Zugriff am 2022-02-02.

[34] Richard Kiss. pycoin – python cryptocoin utilities. https://pypi.org/project/pycoin/, 2020.

[35] Martin Glinz, Harald Gall. Statische Analyse. https://files.ifi.uzh.ch/rerg/arvo/ftp/se/Folien/Kapitel_11_StatAnalyse-1.pdf, 2009. Zugriff am 2022-02-02.

[36] Adam Muntner. fuzzdb-project/fuzzdb. https://github.com/fuzzdb-project/fuzzdb, February 2022.

[37] James Newton-King. Security considerations in grpc. https://docs.microsoft.com/en-us/aspnet/core/grpc/security?view=aspnetcore-6.0#message-size-limits, 2022. Zugriff am 2022-02-04.

[38] npmjs. npm Docs - npm-audit. https://docs.npmjs.com/cli/v8/commands/npm-audit. Zugriff am 2022-02-02.

[39] Dennis Schirrmacher. Eine million android-nutzer laden falschen whatsapp-messenger aus google play. https://www.heise.de/security/meldung/Eine-Million-Android-Nutzer-laden-falschen-WhatsApp-Messenger-aus-Google-Play-3880190.html, 2017. Zugriff am 2022-02-04.

[40] Ivan Shumkov et al. Dip11: Identities. https://github.com/dashpay/dips/blob/master/dip-0011.md, 2020.

[41] Strophy. chore: update to grpc-js. https://github.com/dashevo/js-grpc-common/pull/30. Zugriff am 2022-02-04.

[42] trailofbits. protofuzz. https://github.com/trailofbits/protofuzz.

[43] Unknown. proto-fuzzer. https://www.npmjs.com/package/proto-fuzzer.

[44] Jinqiu et al. Yang. Towards better utilizing static application security testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 51–60. IEEE, 2019.

[45] Ömer Gin, Jan Müller, Markus Neuber. SAST - Detailed Findings. https://lab.it.hs-hannover.de/m0s-m4z-u1/masterprojekt-blockchain-2021/-/blob/main/SAST/SAST_-_Detailed_Findings.pdf, 2021. Zugriff am 2022-02-04.

[46] Ömer Gin, Jan Müller, Markus Neuber. Static Application Security Testing (SAST). https://lab.it.hs-hannover.de/m0s-m4z-u1/masterprojekt-blockchain-2021/-/wikis/Dokumentation/Static-Application-Security-Testing-(SAST), 2021. Zugriff am 2022-02-04.