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

Penetration Testing auf Dash Platform

Abschlusspräsentation im Rahmen des Masterprojektes 2021/22

Masterprojekt Team, 27.01.2022

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DoS auf Layer 1

Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 7 Fuzzing

Kapitel 8 DashPay / Race Conditions auf Dash Plattform / Spieltheoretische Angriffe

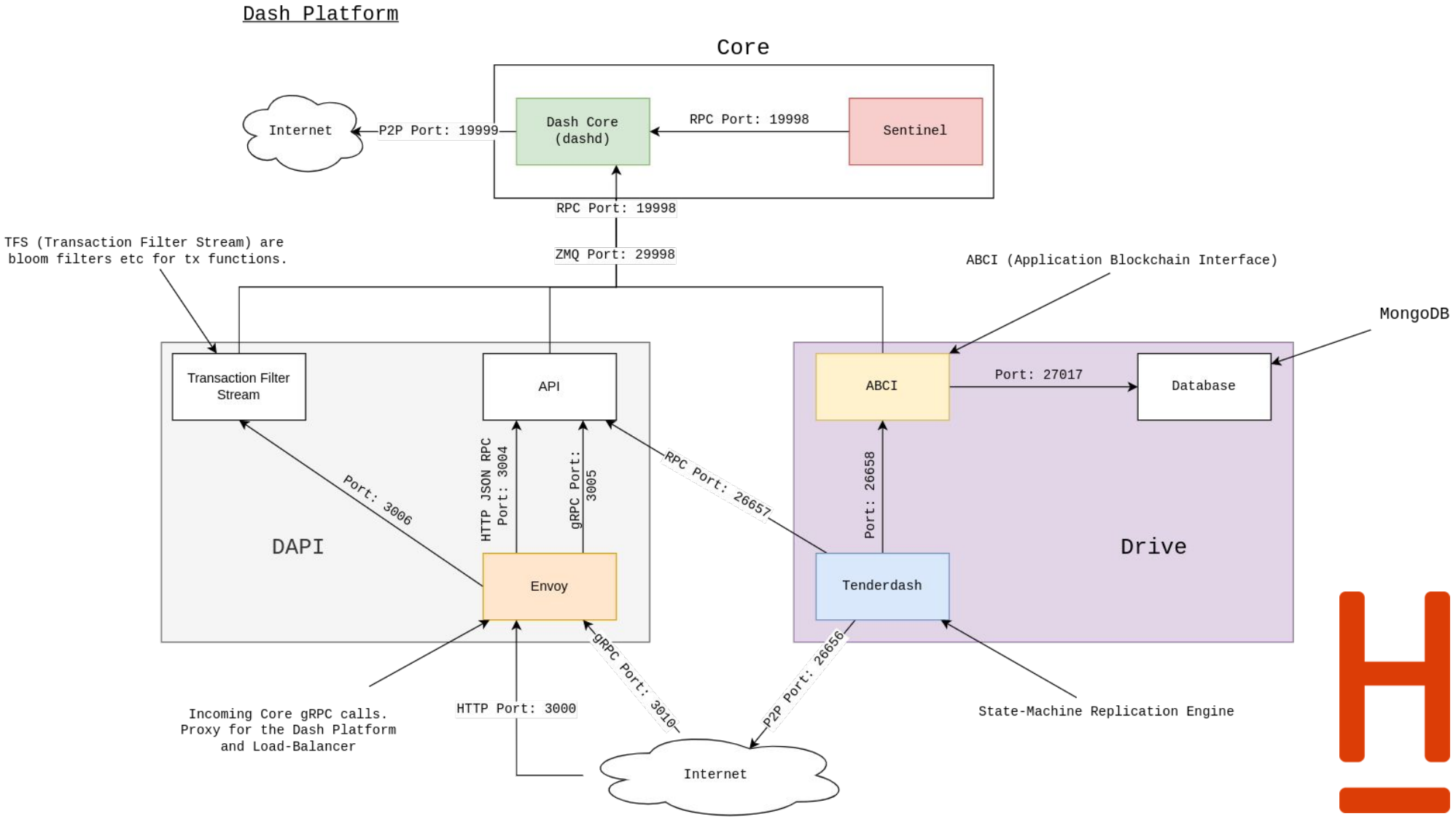
Kapitel 9 Fazit & Lessons Learned

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Dash Platform

Architekturdiagramm



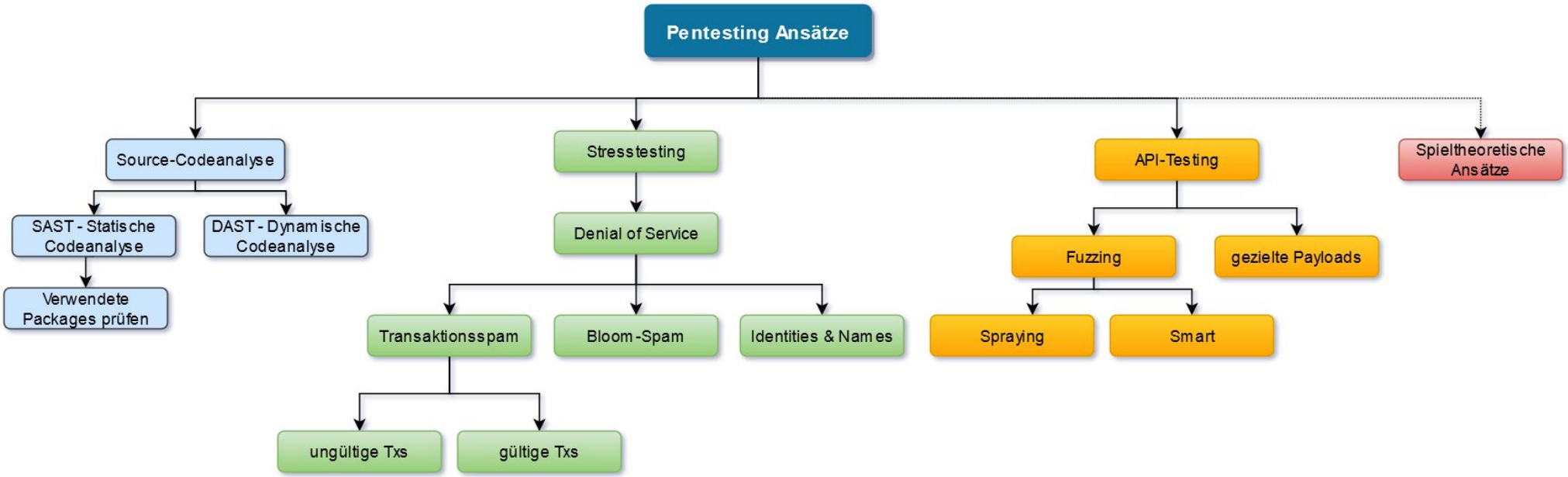
Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Methodik

Ansätze



Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Setup

Testnet und Devnet

- Setup Testnet
  - einzelne VM
  - Layer 1 & Layer 2
  - Masternode im Testnet
  - läuft auf dem Hochschul-VM-Server
    - Portfreigabe über Proxy
- Setup Devnet
  - 12 Pool - Rechner
  - lokales Netzwerk
    - 5 Subnetze
  - Netzlabor der Hochschule
  - nicht 24/7 aktiv, da sich die Rechner mit dem Bachelor-Team geteilt wurden.

Setup

Testnet Spezifikation

- Virtuelle Maschine
  - Proxmox Umgebung (QEMU Virtualisierung)
  - virtualisierte Hardware:
    - CPU: Intel(R) Xeon(R) Gold 5115 CPU @ 1x2.40GHz
    - RAM: 8GB
    - Disk: 107GB
  - 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 (X)*
      - *MongoDB v5.0.5*

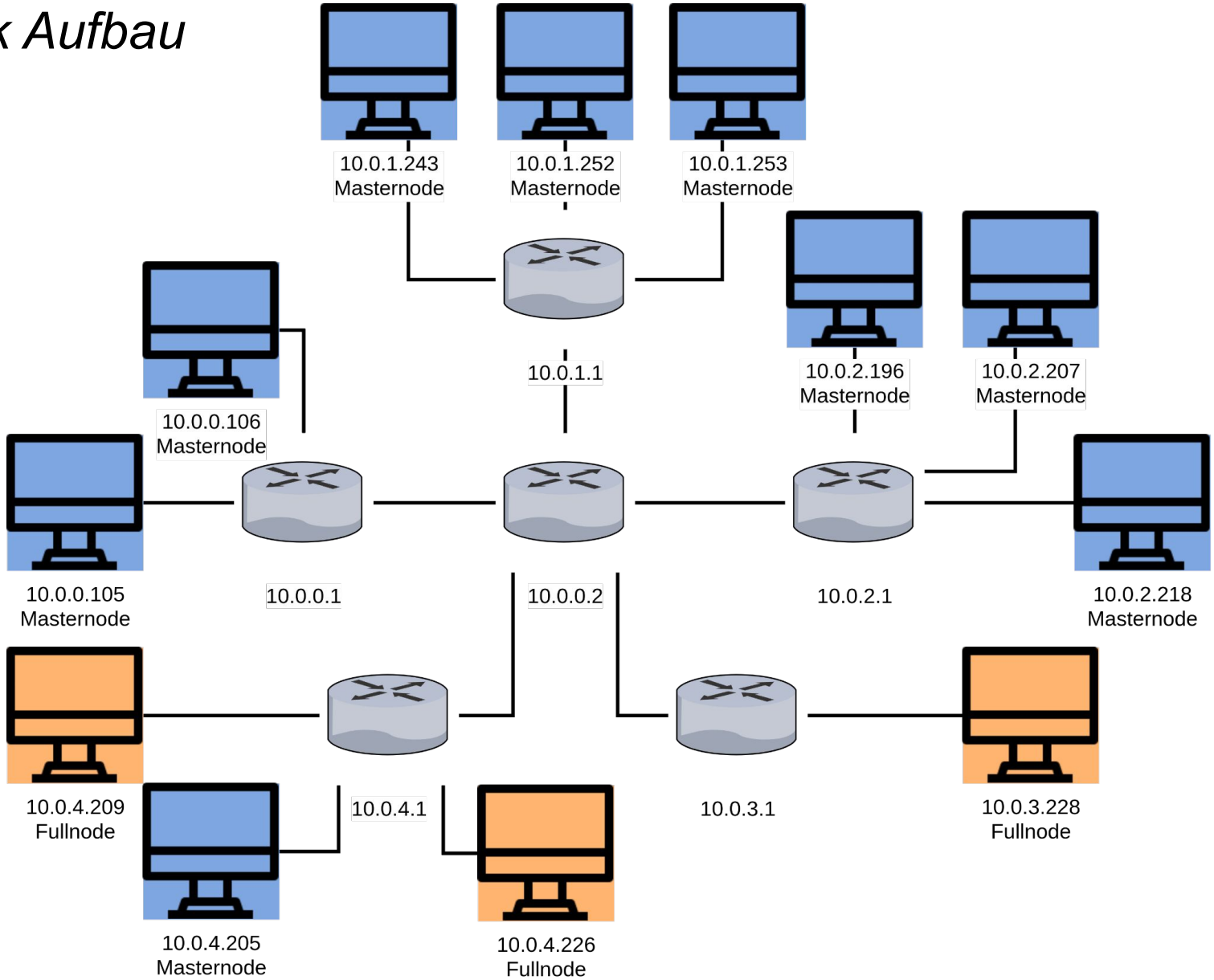
Setup

Devnet Spezifikation

- Netzlab Pool Rechner (12x)
  - 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 (Port: 27017) v5.0.5*

Devnet

Netzwerk Aufbau



Devnet

Dash Konfiguration

- 12x Nodes:
  - 9x Masternodes
  - 3x Fullnodes (inkl. Miner)
- Netzwerk = devnet = masterproject
- nur im lokalen Netz (allowprivatenet = 1)
- alle Sporks aktiv
- LLMQ-Type für Chainlock und Instantsend:
  - Devnet-Standard: llmq_devnet:
    - quorumSize: 10, quorumMinSize: 7, quorumThreshold: 6 (60%)

Devnet

Dash Konfiguration

```
devnet=masterproject
#----
[devnet]
port=19999
rpcport=19798
llmqchainlocks=llmq_devnet
llmqinstantsend=llmq_devnet
#----
listen=1
server=1
txindex=1
addressindex=1
spentindex=1
#----
rpcuser=dashrpc
rpcpassword=password
rpcallowip=127.0.0.1
#----
zmqpubrawtx=tcp://0.0.0.0:29998
zmqpubrawtxlock=tcp://0.0.0.0:29998
zmqpubhashblock=tcp://0.0.0.0:29998
zmqpubrawchainlock=tcp://0.0.0.0:29998
#----
allowprivatenet=1
masternodeblsprivkey=11619691910ede2dc343395305ab5258124ad1860917a52392326e16fe7b0214
sporkkey=cQYGsb1VN5ds1nxPqBMGxpAxuvhcuH6k41YZeuxqgdQUzm3c4JxJ
sporkaddr=ySRZxDixzb8SXtnX7uDXrTBHtRm8BNhKSU
```

Devnet

Dash Platform Konfiguration

- 9x Masternodes:
- Netzwerk = devnet-masterproject
- anhand dem manuellen Installationsguide
- Drive
  - *Port: 26658*
- Dapi
  - *RPC: 3004, GRPC: 3005*
  - *TX-Filter: 3006*
- Tenderdash
  - *P2P: 26656, RPC: 26657*
  - *1 Seed-Node (10.0.1.253)*
  - *create_empty_blocks_interval = "3m"*
  - *quorum_type = “101” -> llmq_devnet*
- *MongoDB*
  - *Single Replication Set: driveDocuments*

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

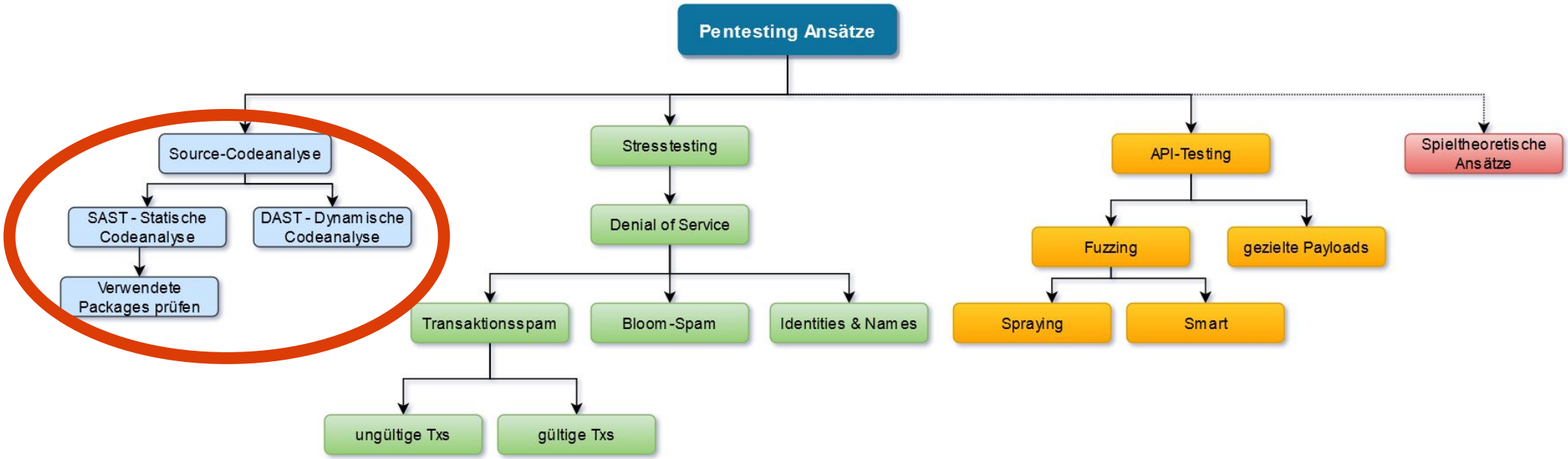
Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Statische Analyse

Übersicht



Statische Analyse

Übersicht

Was ist statische Analyse?

- Fehlersuche im Code
- White Box
- Regelbasiert
- Code wird nicht ausgeführt
- Werkzeuggestützt

Statische Analyse
SAST

Was ist SAST?

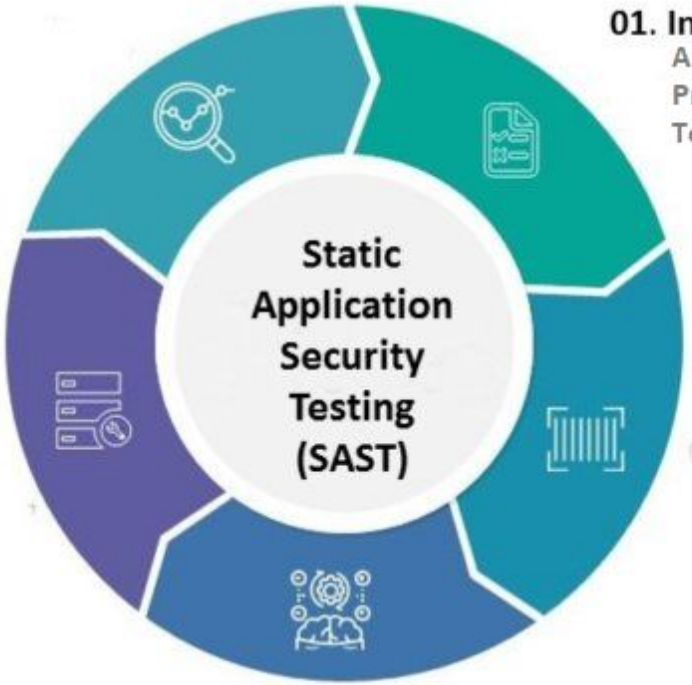
- Static application security testing
- Potenzielle Sicherheitslücken
- Genutzte Werkzeuge:
  - Semgrep
  - npm-audit

Warum SAST?

- automatisiert
- regelbasiert (übliche Fehler)
- Umfangreiche Codeabdeckung
- dokumentiert/protokolliert

Statische Analyse

SAST



01. Information Gathering
Anzugreifende Repositories, Programmiersprachen & Tools identifizieren

02. Preparation and compilation
Anzugreifende Repositories, relevante Abhängigkeiten & Tools herunterladen

03. Source Code Vulnerability Scanning
Automatisiertes Scannen des Quellcodes der Repositories & Auflistung der Findings

04. Analysis & Verification
Findings / potenzielle Schwachstellen analysieren & selektieren (False-Positives)

05. Reporting
Erstellung eines detaillierten Reports

Statische Analyse

Phase 1 - Relevante Repositories identifizieren

Dapi

- … stellt Ort der direkten Eingabe durch User dar
- … ist abhängig von anderen Packages

Zu untersuchende Repositories:

- 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

Statische Analyse

Phase 1 - Tools identifizieren

Anforderungen an die Tools:

- Open Source
- Fertige Rulesets
- Unterstützt genutzte Programmiersprachen (JS, TS)
- Unter Linux installierbar

Identifizierte Tools:

- Semgrep
- npm-audit

Statische Analyse

npm-audit

- In npm enthalten
- Ergänzt Semgrep
- Prüft dependencies
  - bekannte Schwachstellen
  - rekursiv

```
elliptic  <6.5.4
Severity: moderate
Use of a Broken or Risky Cryptographic Algorithm - https://github.com/advisories/GHSA-r9p9-mrjm-926w
fix available via `npm audit fix --force`
Will install @dashevo/dashcore-lib@0.18.15, which is a breaking change
node_modules/elliptic
  @dashevo/dashcore-lib  <=0.18.11 || >=0.19.0
  Depends on vulnerable versions of elliptic
  node_modules/@dashevo/dashcore-lib
    @dashevo/dpp  <=0.10.1 || 0.15.0-dev.3 - 0.15.0-dev.14 || >=0.17.0-dev.2
    Depends on vulnerable versions of @dashevo/dashcore-lib
    node_modules/@dashevo/dpp

9 moderate severity vulnerabilities
```

Statische Analyse

Ergebnisse

Häufige Findings:

- Object Injection via bracket notation
- Inefficient RegEx
- Insecure randomness
- Broken/insecure cryptographic algorithms
- …

Analyse hat ergeben, dass es sich um False Positives handelt

- Bisherige Entwicklung mit ESLint scheint zuverlässig zu sein

Ergebnisse wurden vollständig dokumentiert

- HTML-Report
- PDF-Document

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

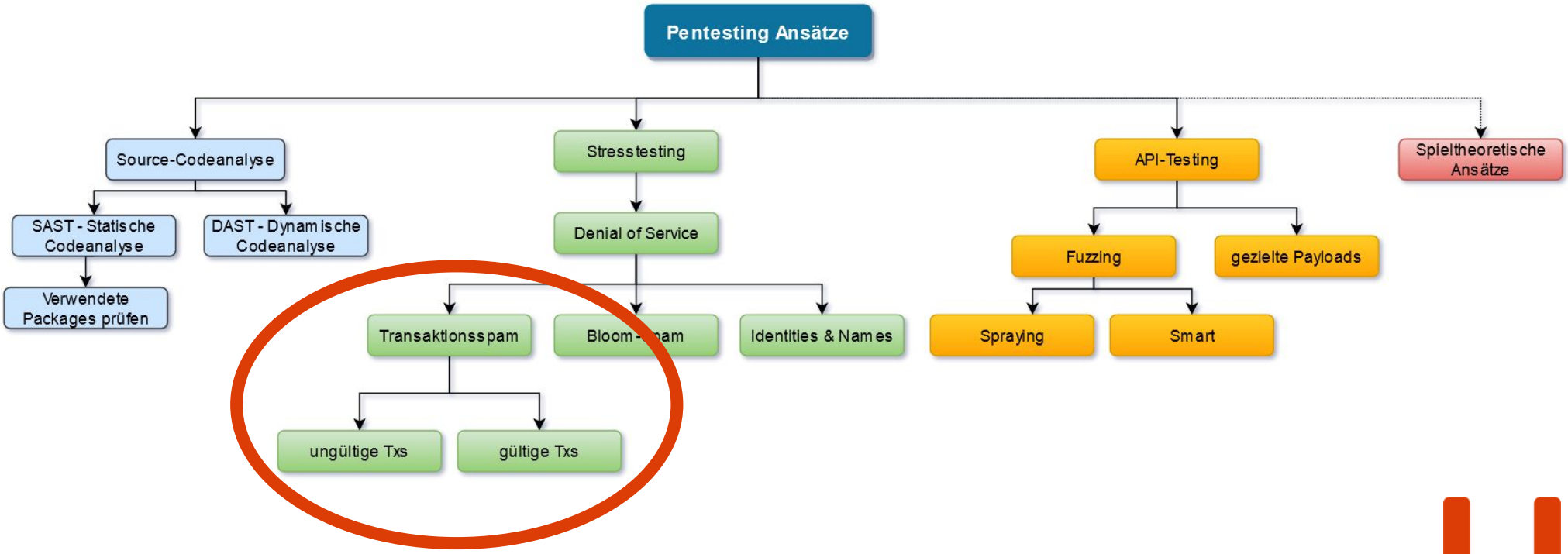
Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DOS auf Layer 1

Flooding auf Layer 1

Übersicht



Flooding auf Layer 1

Ansatz

- Dash Platform baut auf der Layer 1 Blockchain auf
  - Referenzierungen auf *Asset Locking Transactions*
  - Mining und Masternode-Rewards einzige Möglichkeit neue Credits zu generieren

➡ Ein DOS auf Layer 1 würde auch Layer 2 deaktivieren

(Und Layer 1-Funktionalität ebenso)

- Wie herbeiführen?
  - Last auf dem Node erzeugen
  - Zuspammen des Mempools mit Transaktionen
  - Netzwerk überlasten?
  - Vermutlich schwierig, da direkter Fork von Bitcoin

Flooding auf Layer 1

Erzeugen von Last

- Erste Idee: Erzeugen von Last durch gefälschte Transaktionen
  - Node muss Arbeit aufbringen um Validität zu überprüfen
  - Iterieren der Blockchain oder des UTXO-Sets
  - Prüfung von Signaturen
- Verwendung von Python
  - Multithreading über threading-Library
  - PyCoin
  - Aufruf dash-cli

- Versuch #1: Transaktionen referenzieren nicht existenten UTXO
  - Keine Auswirkung, LevelDB optimierter K/V-Store, Zugriff O(1)
- Versuch #2: UTXO-Set auslesen und Signatur fälschen
  - Erzeugt minimal mehr CPU-Last
  - Bitcoins ECDSA-Implementierung *libsecp256k1* ist extrem optimiert und effizient (~52 µs pro Verify ➡ ~19000 V/s auf Ryzen 5 2600)

Flooding auf Layer 1

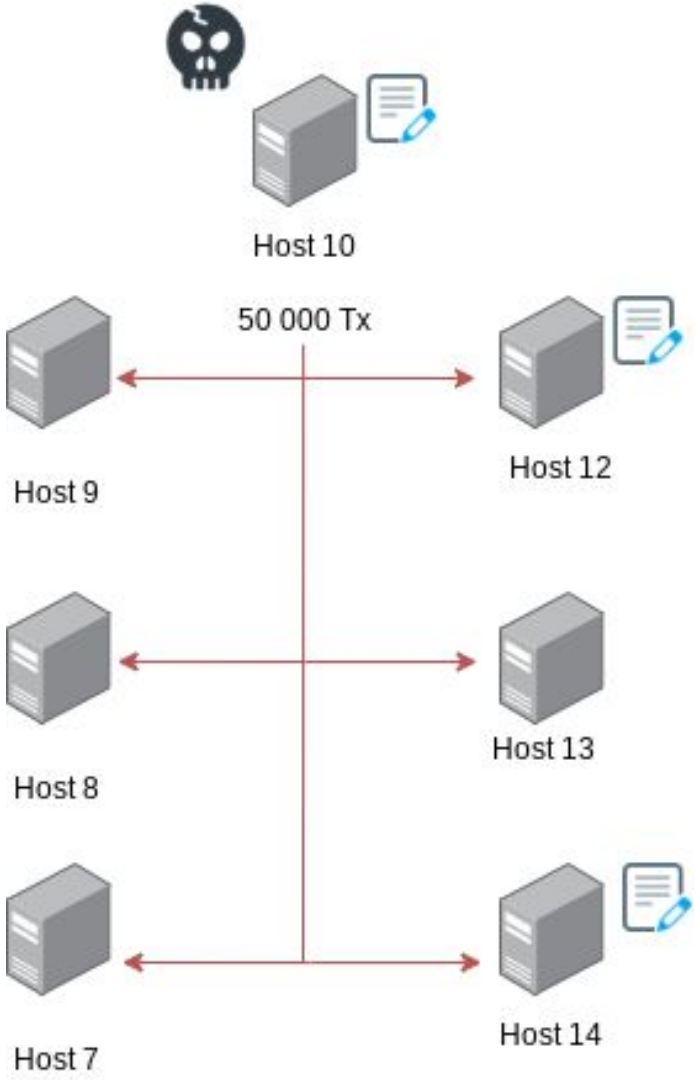
Überlasten des Mempools

- Zweite Idee: Zuspammen des Mempools
  - Verhindert, dass ehrliche Transaktionen aufgenommen werden
  - Beanspruchung von RAM und Festplatte

- Skript ähnlich wie zuvor: Python mit Multithreading
- Erster Durchlauf im lokalen Devnet
  - Angriff von Host 10 aus
  - In regelmäßigen Abständen Auslesen des Mempool-Status auf Host 10, 12 und 14
  - Aufzeichnen des Network-Traffics mit Wireshark
  - Insgesamt etwa 50 000 Transaktionen über einen Zeitraum von 2h.

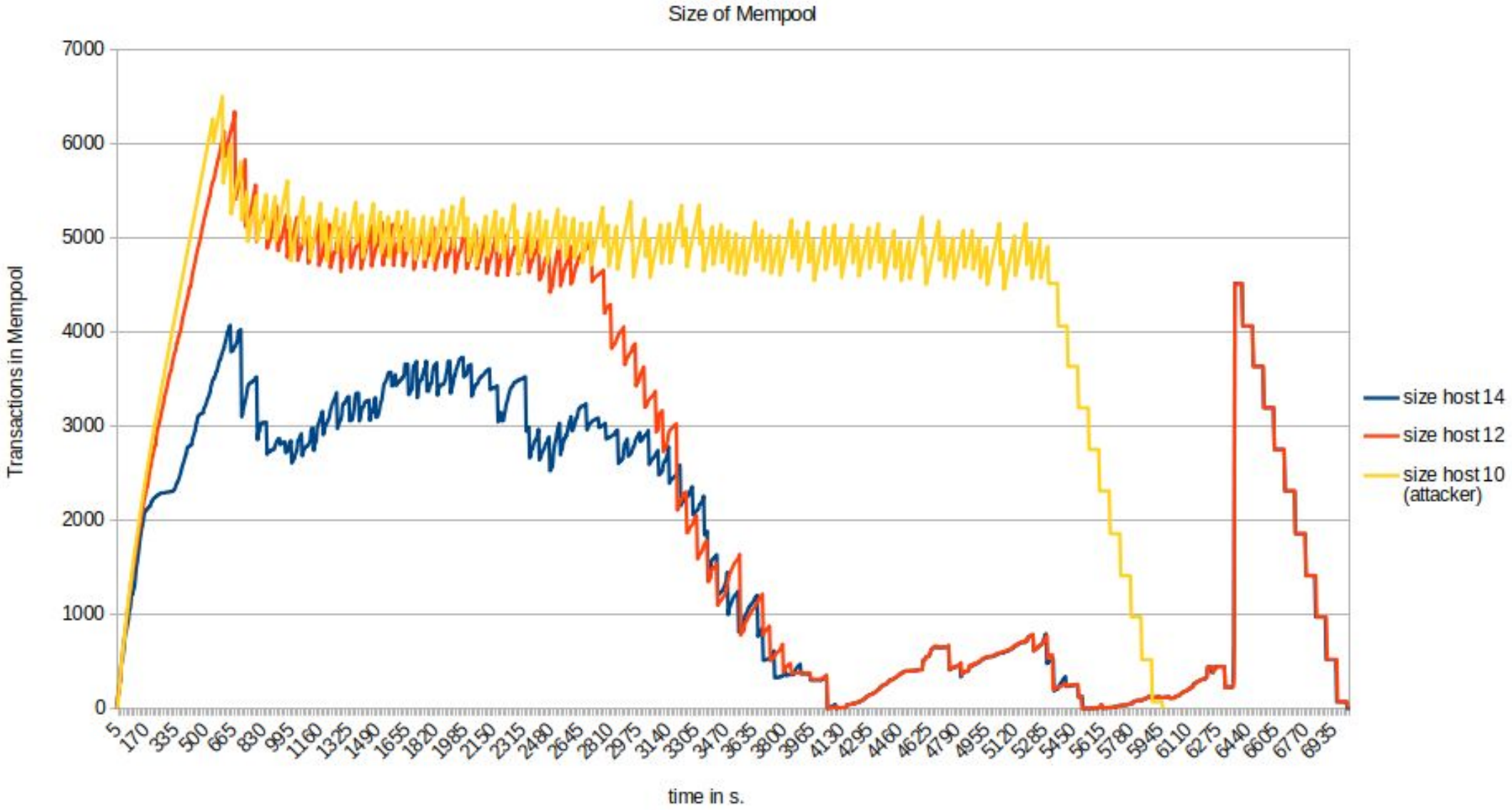
Flooding auf Layer 1

Überlasten des Mempools



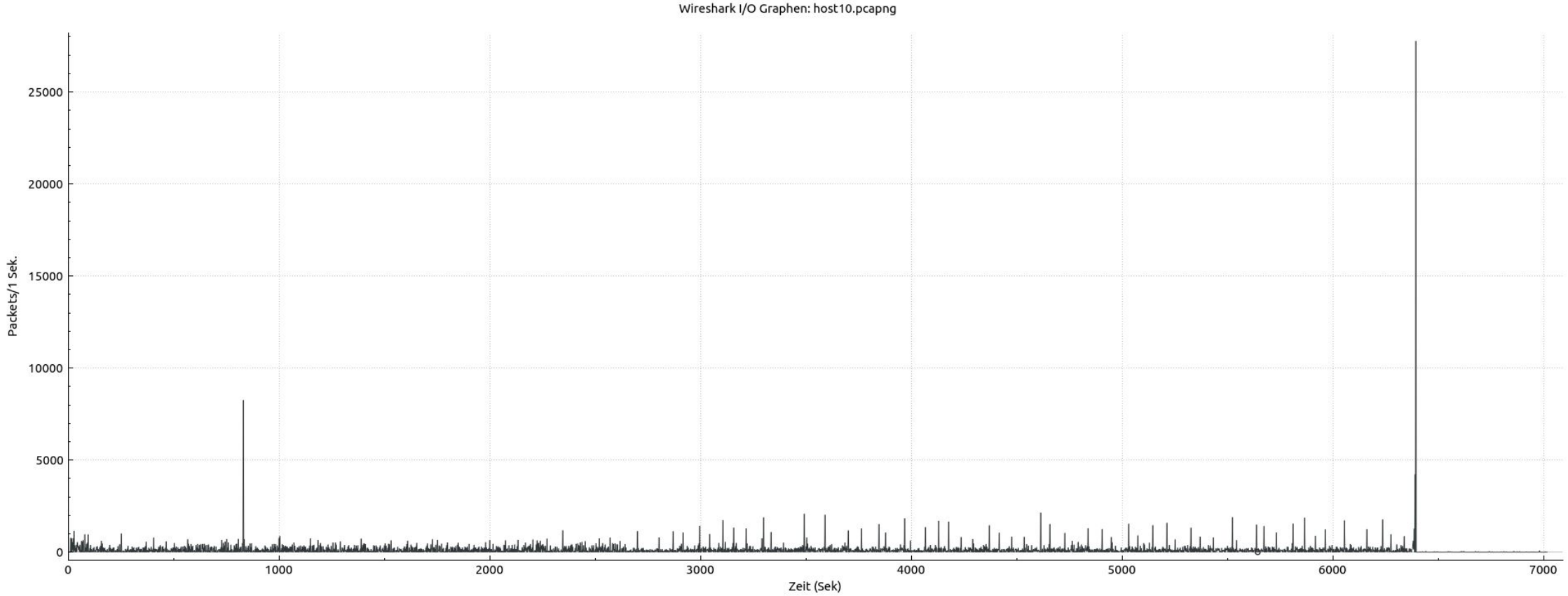
Flooding auf Layer 1

Ergebnisse



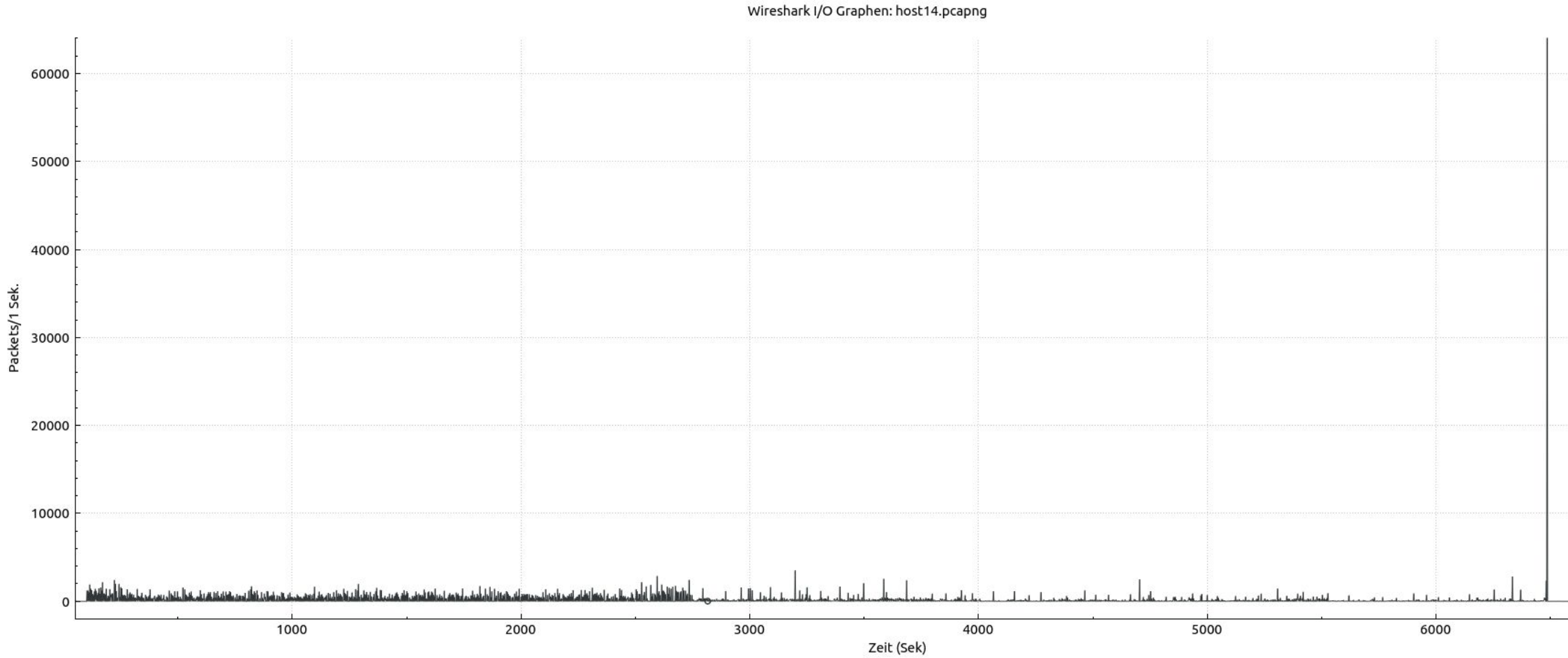
Flooding auf Layer 1

Ergebnisse



Flooding auf Layer 1

Ergebnisse



Flooding auf Layer 1

Ergebnis

- Ehrliche Transaktionen werden durchaus blockiert
- Nodes reagierten verzögert auf Eingaben
- Gebühren steigen
- Einen technischen DoS lässt sich so allerdings nicht herbeiführen

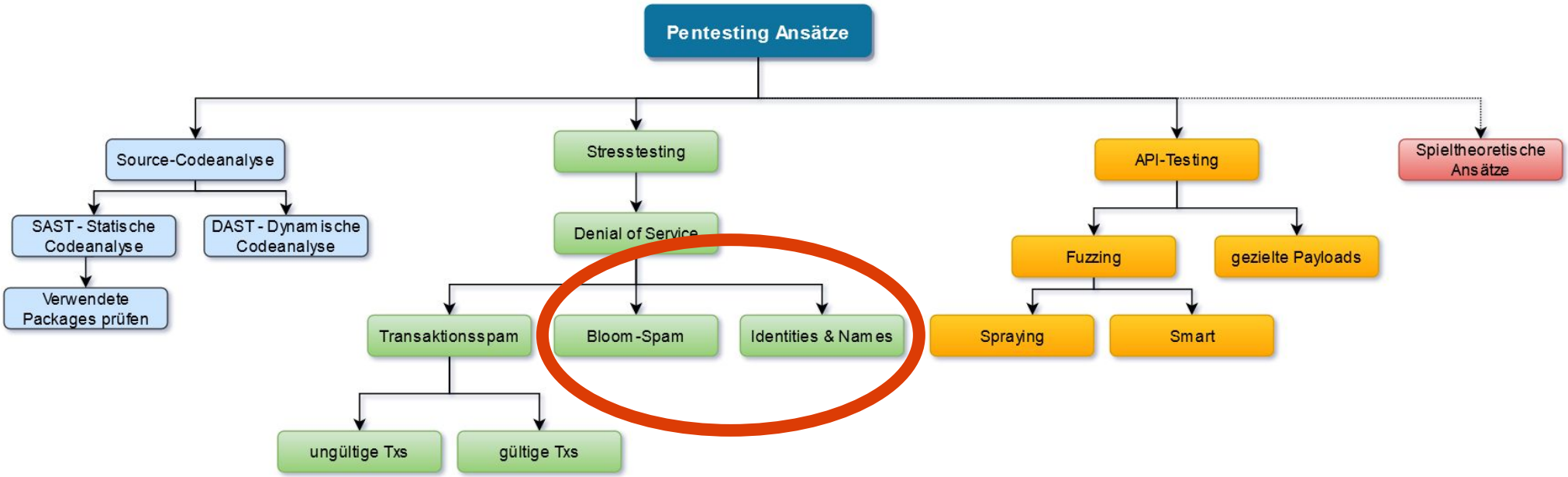
- Wie realistisch ist so ein Angriff auf ein reales Netzwerk?
  - Enorme Kosten durch Fee
  - Transaktionen verzögern sich höchstens um 2h.
    - In diesen 2h. über 100$ verloren
  - Mempoolminfee steigt
  - Node kann blockiert werden
  - Gewinn?
- Mögliche Folgeprojekte
  - Verteilt durchführen
  - Verwendung von aufwändigeren Transaktionen

Inhaltsverzeichnis

Kapitel 1	Einführung in die Dash Platform Architektur
Kapitel 2	Verwendete Methodik
Kapitel 3	Setup Testnet und Devnet
Kapitel 4	Statische Analyse
Kapitel 5	Flooding/DoS auf Layer 1
Kapitel 6	Flooding/DoS auf Layer 2

Flooding/DoS auf Layer 2

Übersicht



Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DoS auf Layer 1

Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 6.1 Bloom-Filter

Bloom-Filter

Einleitung

Core gRPC Endpoint:

- Bietet mehrere Methoden um diverse Informationen abzufragen
- Methode subscribeToTransactionsWithProofs
  - Matching erfolgt via Bloom-Filter (vom Benutzer definiert)
  - Rückgabe beliebiger Anzahl von Transaktionen ab beliebigem Block
  - Wird in dem Dash SDK und von Diensten wie der DashPay-App verwendet

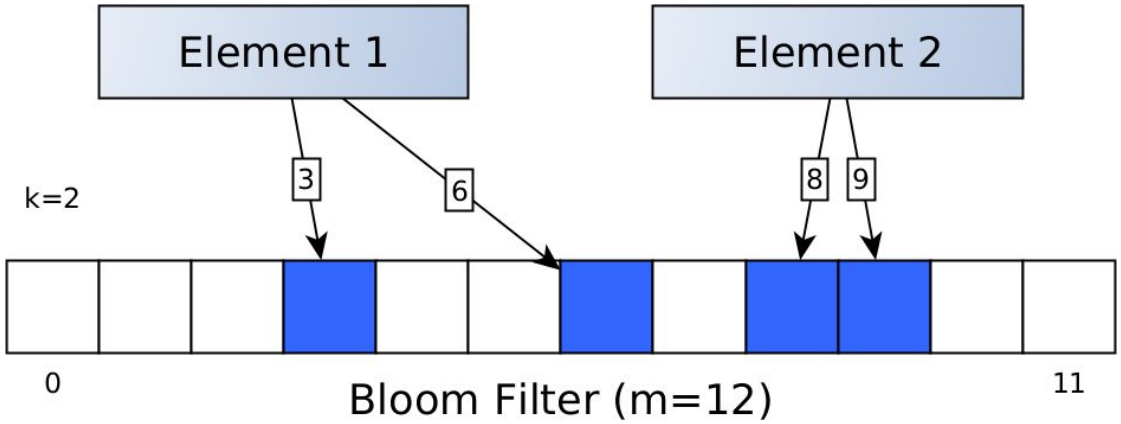
Idee #1:

- DoS durch Ausnutzung der Filter-Funktion
- Konstruktion möglichst ineffizienter Bloom-Filter
- Herbeiführung durch Erzeugung von CPU-Last (Masternode)

Bloom-Filter

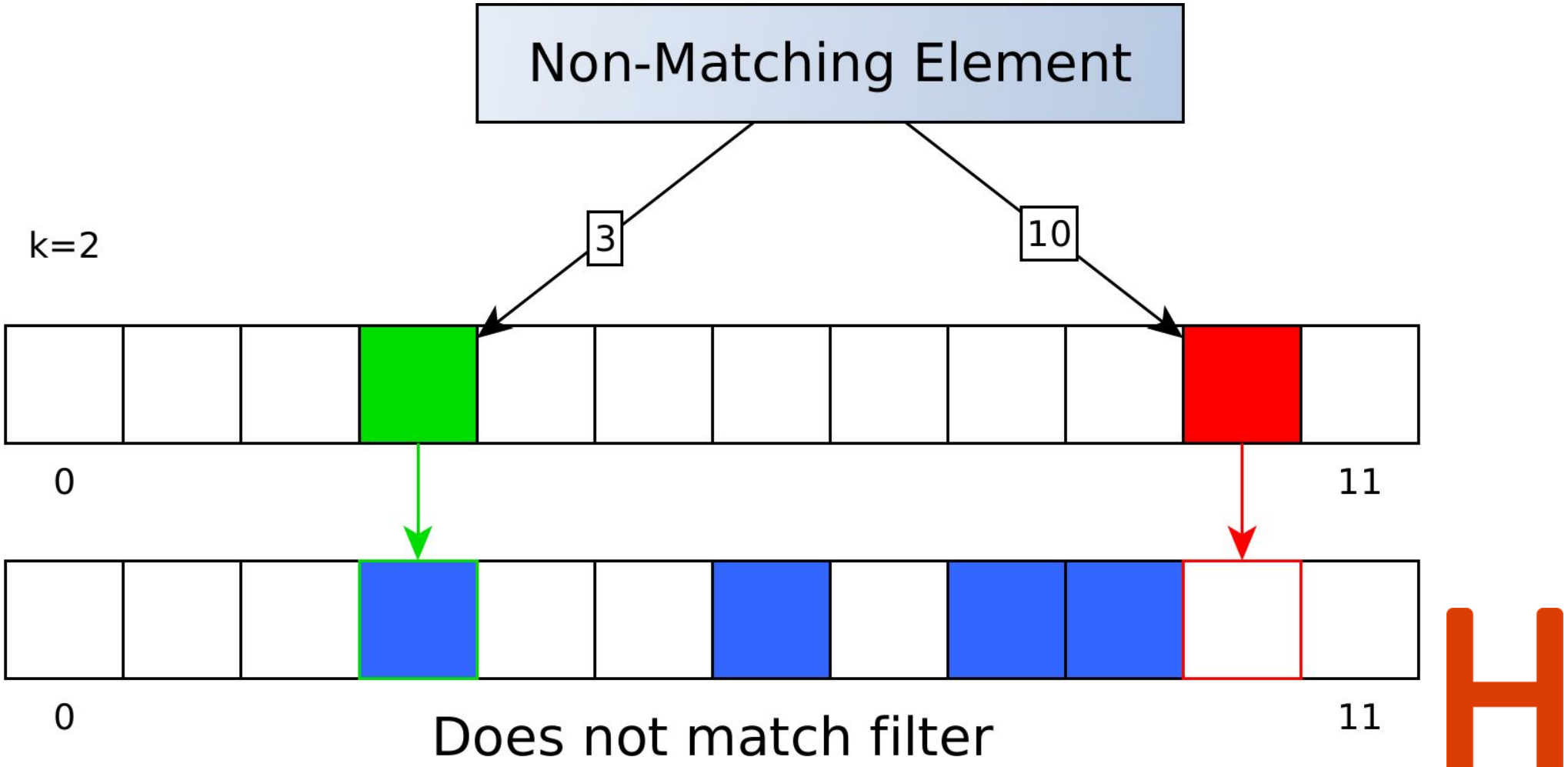
Erläuterung

- Filter
  - Erfasst mindestens die Elemente, für die dieser Konstruiert wurde
  - Kann auch weitere Ergebnisse liefern (false positives)
  - Array mit m Bits
- Für jedes zu filternde Element werden k unterschiedliche Abbildungen erzeugt
  - Wertebereich 0 bis m-1 (= Index für Bit im Array)
  - Filterkonstruktion durch Oder-Verknüpfung
- Treffer, wenn für jeden Hash das zugehörige Bit im Array auf 1 steht
- Abbruch, sobald für einen Hash das Bit auf 0 steht



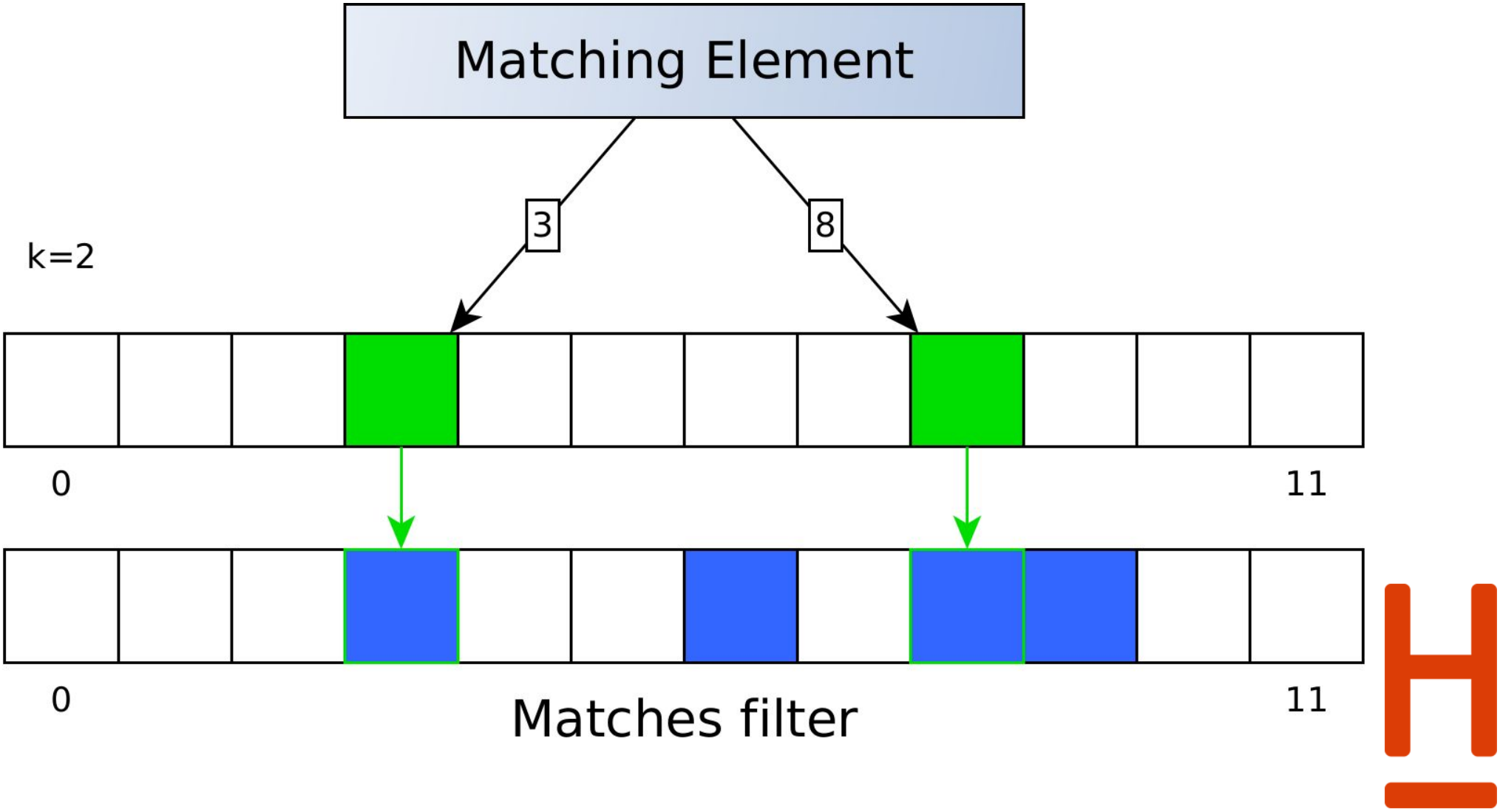
Bloom-Filter

Erläuterung



Bloom-Filter

Erläuterung



Bloom-Filter

Idee #1

Ineffizienz durch:

- Große Menge an zu berechnenden Hashfunktionen pro Element (großes k)
- Filter mit vielen Einsen
  - Früher Abbruch unwahrscheinlich
  - Hohe Wahrscheinlichkeit, dass alle k Hashes berechnet werden müssen
- Worst Case: n Elemente * k Hashfunktionen

Problem:

- k auf 50 begrenzt
- Dadurch mit diesem Parameter allein nur relativ wenig Last erzeugbar
- Daher Idee #1 so nicht umsetzbar

Bloom-Filter

Praktikabler Angriffsvektor

Idee #2:

- Überlastung durch Abfrage möglichst vieler Daten
  - Konstruktion eines Catch-All-Filters (alle Bits auf 1)
  - Abfrage der gesamten Blockchain (ab Höhe 0, ohne Limit)

Umsetzung:

- Mehrere parallele Abfragen
- In JavaScript durch Nutzung des DAPI-Clients

Ergebnisse:

- Dienst wird überlastet und wiederholt zum Absturz gebracht
- Grund: Heap Overflow in Node.js

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DoS auf Layer 1

Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 6.2 Identities

Identities

Übersicht

Dezentraler Identifier für Dash Platform

- Gebunden an Public Key(s)
- Vor Erzeugung: Asset Lock Transaction auf Layer 1
  - Transformiert Dash zu Platform Credits auf Layer 2
  - Credits zum Bezahlen von Aktivitäten auf Dash Platform

Genutzt für:

- Registrierung von Namen für Blockchain Accounts
- Registrierung von Data Contracts
- Signieren von State Transitions
  - Z.B. für Update von Dokumenten, die in einem Data Contract definiert wurden
  - Ggfs. Race Conditions möglich

Identities

Angriffsvektor

Idee:

- Herbeiführen eines DoS in der DAPI
  - Anlegen und Abfragen vieler Identities im lokalen Devnet
- Abfrage: Übergabe eines Arrays mit Public Key Hashes möglich
  - Kein Limit für Public Key Hashes angegeben
  - *Was passiert man, wenn man sehr viele Hashes angibt?* (gRPC Payload/Suche)

Problem:

- Anlegen von Identities nur mit Dash SDK möglich
- Dash SDK konnte im lokalen Devnet nicht genutzt werden
  - Benötigt Envoy; Envoy funktionierte nicht korrekt
  - gRPC Healthcheck im Envoy durch Dash Entwickler ausgeschaltet
    - Kommunikation mit DAPI und Transaction Filter Stream nicht möglich
- Versuch: Healthcheck in der DAPI auszuschalten o. selbst zu implementieren
  - Ohne Erfolg
  - Angebot der Entwickler einen Pull Request zu erstellen

Identities

Ausweichlösung

Anlegen von Identities auf dem globalen Testnet

- In einer einzigen Wallet
- Versuch mehrere Identities gleichzeitig anzulegen wurde serverseitig blockiert
- Anzahl angelegter Identities nicht ausreichend für geplante Abfrage
  - Bei mehreren Abfragen erneut serverseitig blockiert
  - Alternative: Über mehrere Proxies realisieren
    - Aus Zeitgründen nicht mehr durchgeführt

Anmerkungen

- In der DashPay App konnten zwei interessante Verhaltensweisen beobachtet werden:
  - Benachrichtigung über versuchten Double Spend
  - Überweisung von 3-5 Testdash
  - Dash Entwickler wurden kontaktiert und über das Verhalten informiert

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

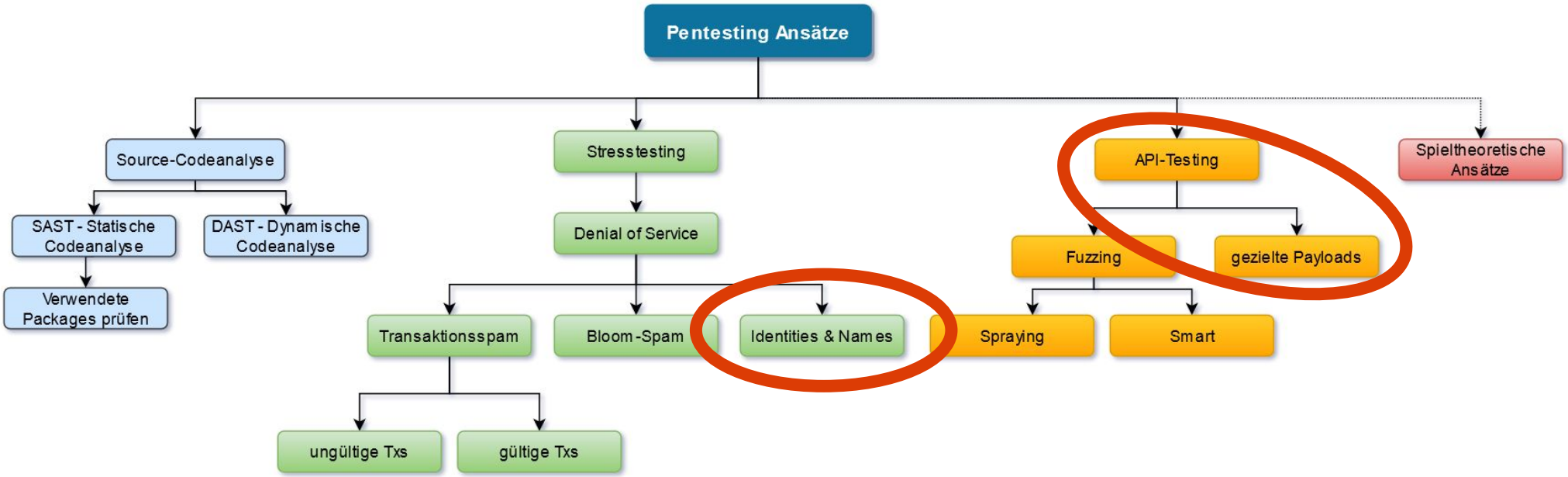
Kapitel 4 Statische Code-Analyse

Kapitel 5 Flooding/DoS auf Layer 1

Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 6.3 DPNS

DPNS
Übersicht



DPNS

Grundlagen

- **D**ash **P**latform **N**ame **S**ervice
  - Erweiterung des Konzepts von Nutzer-Identitäten
  - Nutzer können einen eigenen Namen festlegen
  - Wird von Diensten wie der DashPay-App verwendet
  - Einfacher zu merken / verwenden als Public Key Hashes
  - Realisiert durch spezifische DPNS-Contracts

- Ursprüngliche Idee: Nutzer mit speziellen Namen in Anwendungen
  - Verwendung von UTF-Sonderzeichen
    - Imitierung (Impersonation)
  - Strings ohne Null-terminator
  - Testen auf generelles Fehlverhalten

DPNS

Validierung von Namen

Frage: Was für Namen sind erlaubt und werden vom Netzwerk akzeptiert?

- **Versuch #1**: Den Namen dashtothemoon🚀 für eine Identity anlegen
  - Erweiterung durch einen UTF-Charakter in Form eines Emojis
    - Nicht möglich, wird bereits in der dashsdk (Node.js) validiert und gar nicht erst abgesendet

- **Versuch #2**: Den gleichen Namen erneut anlegen, jedoch diesmal die clientseitige Validierung deaktivieren
  - Einfach möglich (Danke JavaScript)
    - Konsens-Fehlermeldung vom Masternode, wird auch dort validiert!

Resultat: Spezielle Namen nicht direkt möglich aufgrund von strenger Validierung

DPNS

Suche von Namen

Zwei Möglichkeiten zur Namenssuche:

- Exakte Suche (=== Operator in js, String-equals Methoden in anderen Sprachen)
  - Generell eher uninteressant
    - Wenig Rechenleistung
    - Zusätzlich optimierbar durch zum Beispiel Hash-basierte Suchverfahren
- Wildcard Suchen für einen beliebigen Suffix (zum Beispiel dashto*)
  - Angriffsvektoren denkbar
    - gRPC-Payloads haben eine maximale Größe
      - *Was passiert wenn Wildcard-Query sehr viele Namen liefert?*
    - Wildcard Suche hat eine Laufzeit von O(n)
      - *Was passiert wenn sehr viele Namen durchsucht werden müssen?*
      - *Suche mit O(m*n) kann bereits problematisch werden*

DPNS

Suche von Namen

Szenario #1 - Payload Größe (DoS)

- **Vorgehen**
  - >1000 Namen angelegt, die a* entsprechen
  - Entsprechende Namen mit Wildcard-Suche a* abgefragt
- **Resultate**
  - Suche wird auf 100 Resultate eingeschränkt
    - Invalides Payload selbst mit Namen maximaler Länge nicht möglich

Szenario #2 - Suche über eine Vielzahl an Namen (DoS)

- **Idee**
  - Man lege so viele Namen wie möglich an
  - Namen sollten einem bestimmten Suchmuster entsprechen
  - Rechenaufwand soll maximiert werden

DPNS

Szenario #2 - Suche über eine Vielzahl an Namen

Vorgehen: Man nutze die maximale Länge der Namen aus (63)

- Tool zum Erstellen von Namen implementiert

  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABZ00001

- AAA
  - Beliebiger Präfix
- B
  - Maschinen Infix, ermöglicht Aufteilung auf mehrere Rechner
- Z
  - Thread Index, erlaubt Multi-Threading
- 00001
  - Counter, gezählt wird von 0-9, a-z
  - 34^5 Möglichkeiten

DPNS

Szenario #2 - Suche über eine Vielzahl an Namen

Probleme

- Erstellen von Namen generell eher langsam
  - Man muss gegebenenfalls auf mehrere State Transitions warten
- IP rate limiting hindert Geschwindigkeit
  - Mehrere Threads erreichen das Limit schnell
    - Aufteilen auf unterschiedliche Masternodes nicht direkt möglich unter Verwendung des dashsdk
    - Alternativer Lösungsansatz wären Proxies

DPNS

Szenario #2 - Suche über eine Vielzahl an Namen

Erkenntnisse während der Namenserstellung

- Erstellt man viele Namen gleichzeitig, so stößt man häufig auf einen internen Server Fehler
  - Tenderdash ist nicht mehr per TCP-Socket für die DAPI erreichbar

Resultate

- Durchführung der Suche aufgrund des Zeitmangels leider nicht mehr geschehen
  - Query würde dem Präfix und einem nicht existierenden Maschinen Infix entsprechen
- Auswertung der Fehlermeldung aus dem selben Grund nicht erfolgt
- Bachelor Projekt könnte hier gegebenenfalls anschließen

Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

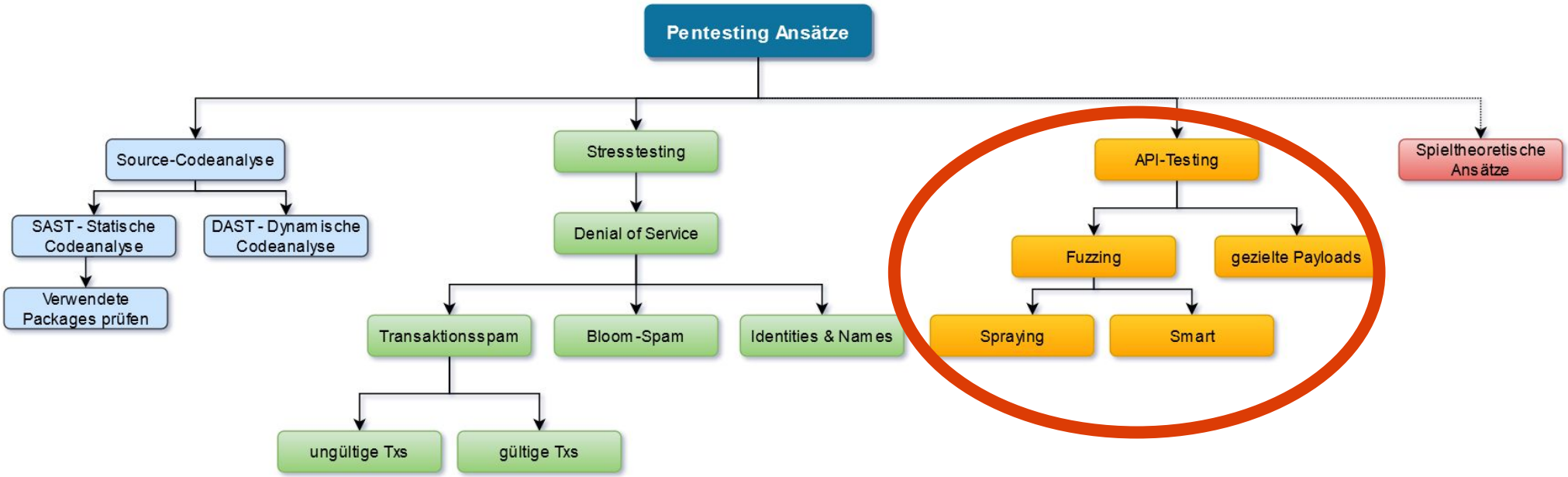
Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DoS auf Layer 1

Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 7 Fuzzing

Fuzzing
Übersicht



Fuzzing
Einführung

Schnittstellen mit mehr oder weniger zufälligen Daten “beschießen”

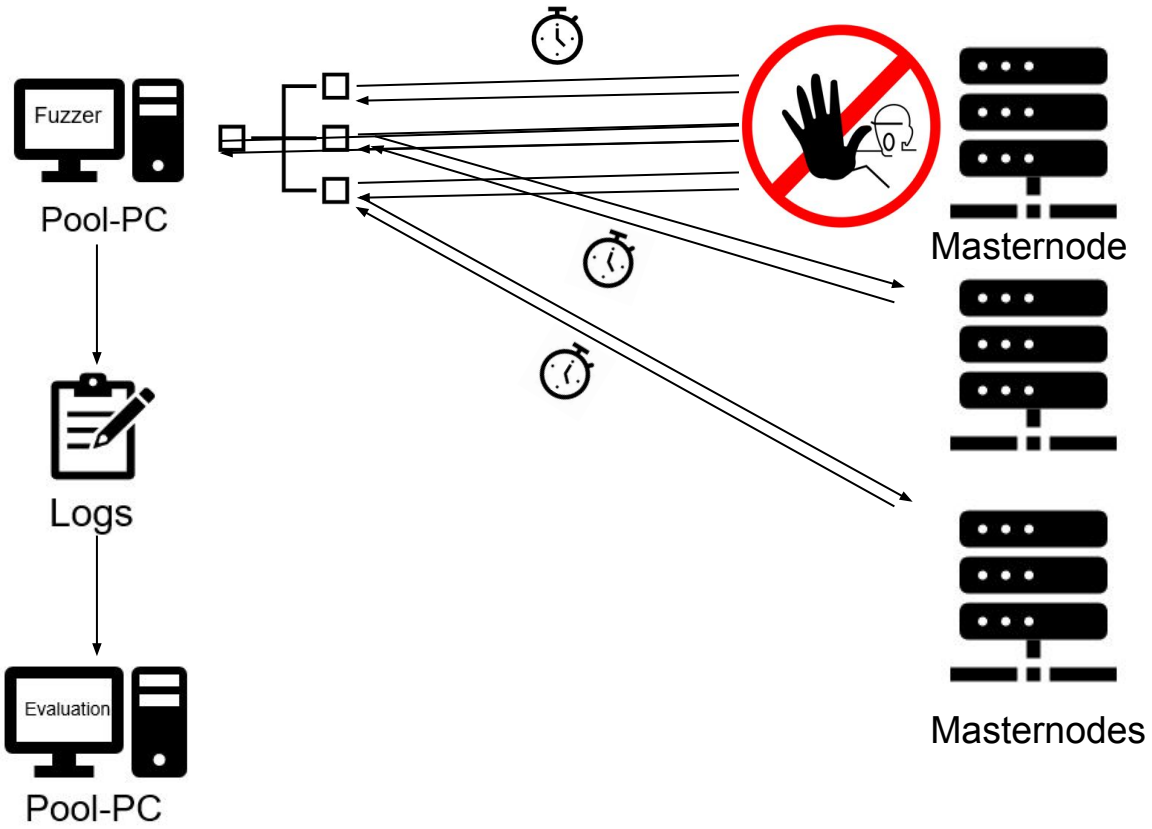
- **Ziel**: ungewolltes Verhalten provozieren
- *Probleme im Anschluss weiter analysieren*

gRPC und JSON-RPC-Schnittstelle der DAPI als Entrypoints von außen auf die Masternodes

- *gRPC bietet alle Funktionen, JSON-RPC nur get(Best)BlockHash und getMnListDiff*
  - gRPC nutzt Protokoll Buffer zur Kodierung der Daten (Bytestream) → nicht jeder Fuzzer ist anwendbar
    - Interface-Sprache beschreibt die Struktur der Daten
  - Betrachtete Ansätze für das Fuzzing der gRPC-Schnittstelle
    - AFL, Fuzzino, proto-fuzzer, Mayhem for API, Eigenentwicklung, **ProtoFuzz**

Fuzzing mit ProtoFuzz

Beschreibung, Probleme & Lösungen



Fuzzing

Auswertungsansatz

- Enorme Datenmenge
- Skript zur Auswertung an Hand der Status Codes der Responses
  - Zu analysierende Status Codes:
    - *INTERNAL, UNAVAILABLE, RESOURCE_EXHAUSTED, DEADLINE_EXCEEDED*
  - Resultierende Report per Hand prüfen
    - Durch Verständnis oder manuelle Requests
- Beispielreport siehe Folgefolie

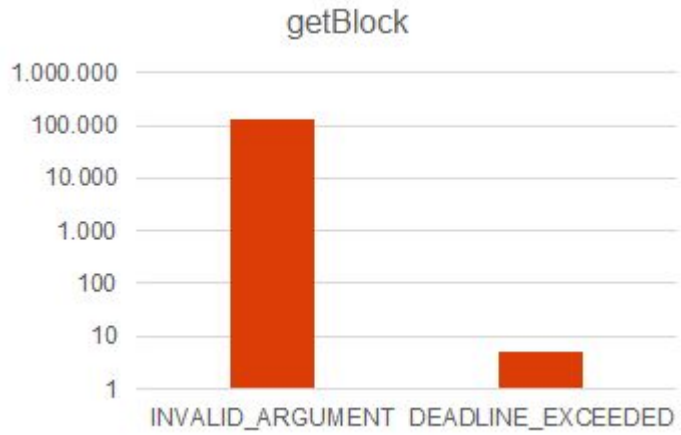
Fuzzing

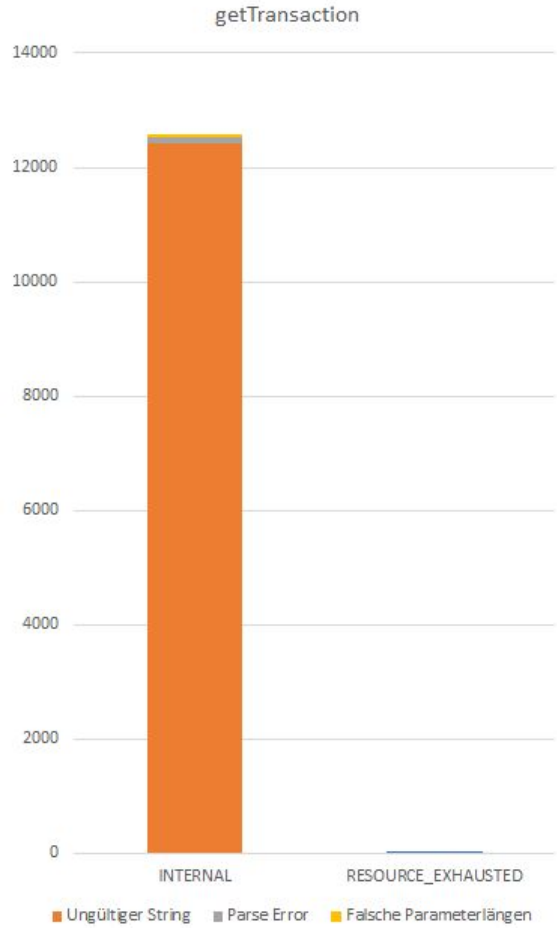
Auswertungsreport - Beispiel

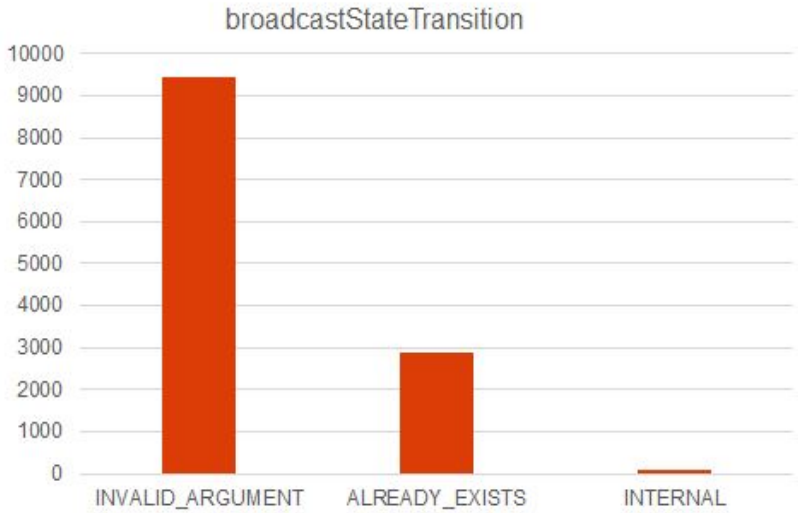

```
{
 "ID": "GetTransactionRequest",
 "TESTNET": true,
 "REQUESTS": 12417,
 "SUCCESS": 0,
 "EXCEPTION": 12417,
 "SUCCESS_QUOTA": 0.0,
 "EXCEPTION_QUOTA": 1.0,
 "ERROR_CODES_RAW_DISTRIBUTION": {
  "INTERNAL": 12414,
  "RESOURCE_EXHAUSTED": 3
 },
 "ERROR_CODES_QUOTA_DISTRIBUTION": {
  "INTERNAL": 0.9997583957477651,
  "RESOURCE_EXHAUSTED": 0.00024160425223483932
 },
 "ANOMALY_COUNT": 0,
 "ANOMALY_HASHES": [],
 "OCCURED_ERROR_MESSAGES": {
  "0": {
   "error_returned": "<_InactiveRpcError of RPC that terminated with:\n\tstatus = StatusCode.INTERNAL\n\tdetails = \"parameter 1 must be hexadecimal string not the provided
   "count": 12239,
   "occuredByInput": [
  },
  "1": {
   "error_returned": "<_InactiveRpcError of RPC that terminated with:\n\tstatus = StatusCode.INTERNAL\n\tdetails = \"Parse error at /usr/src/app/lib/externalApis/dashcore/r
   "count": 118,
   "occuredByInput": [
  },
  "2": {
   "error_returned": "<_InactiveRpcError of RPC that terminated with:\n\tstatus = StatusCode.RESOURCE_EXHAUSTED\n\tdetails = \"received trailing metadata size exceeds limit
   "count": 3,
   "occuredByInput": [
    "id: \"[\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",\\\"1\\\",
    "id: \"[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    "id: \"-999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ]
  },
  "3": {
   "error_returned": "<_InactiveRpcError of RPC that terminated with:\n\tstatus = StatusCode.INTERNAL\n\tdetails = \"parameter 1 must be of length 64 (not 5628) at /usr/src
   "count": 1,
   "occuredByInput": [
    "id: \"999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ]
  },
  "4": {
```

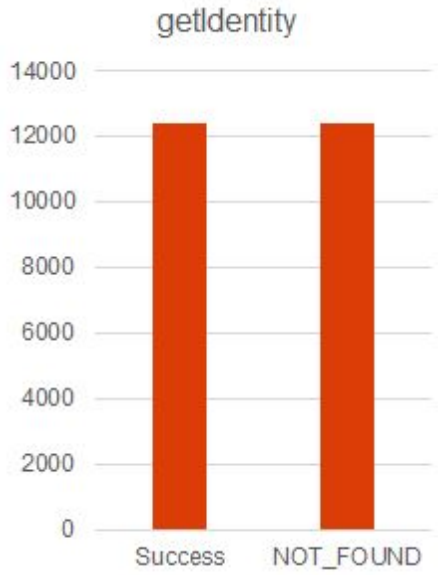
Fuzzing

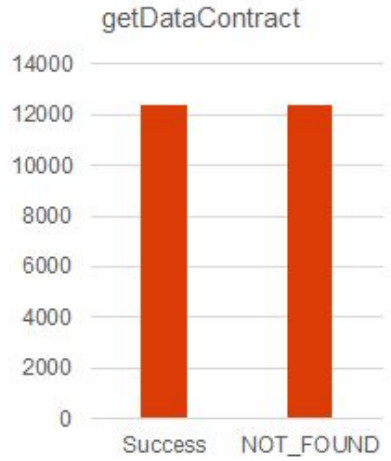
Auswertungsergebnisse





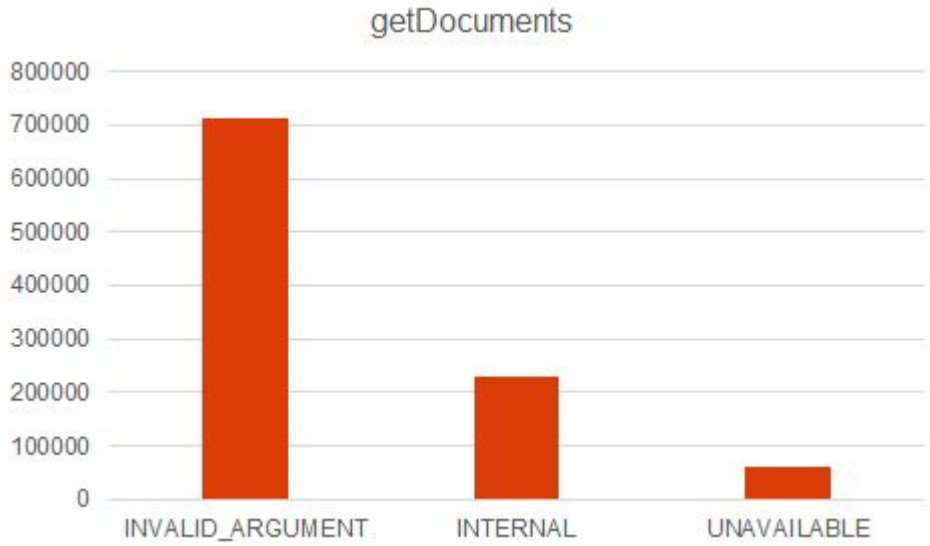


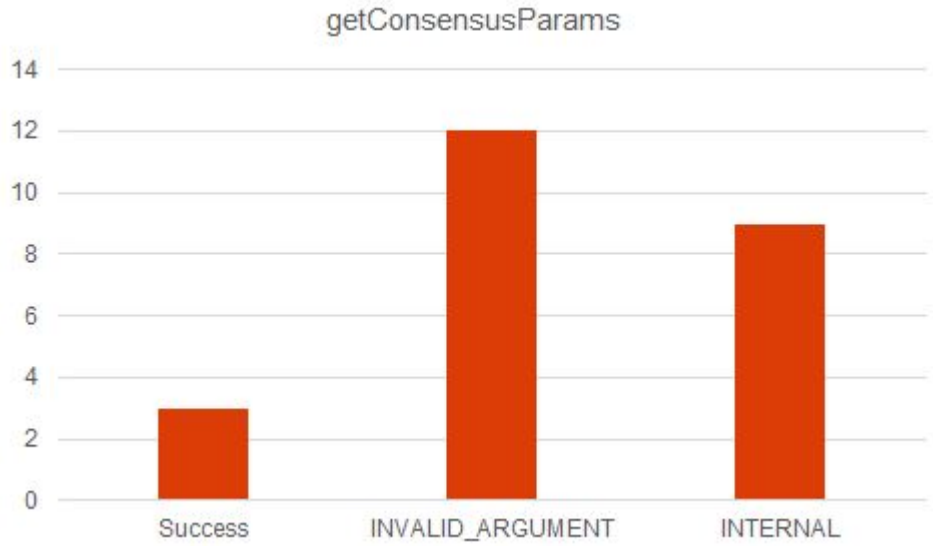


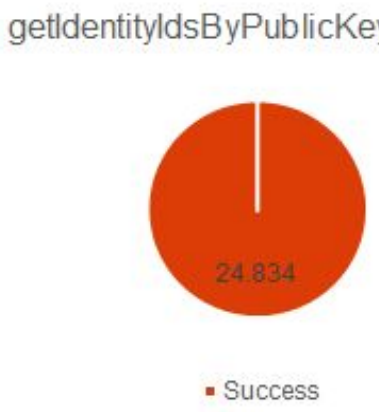


Fuzzing

Auswertungsergebnisse







Inhaltsverzeichnis

Kapitel 1 Einführung in die Dash Platform Architektur

Kapitel 2 Verwendete Methodik

Kapitel 3 Setup Testnet und Devnet

Kapitel 4 Statische Analyse

Kapitel 5 Flooding/DoS auf Layer 1

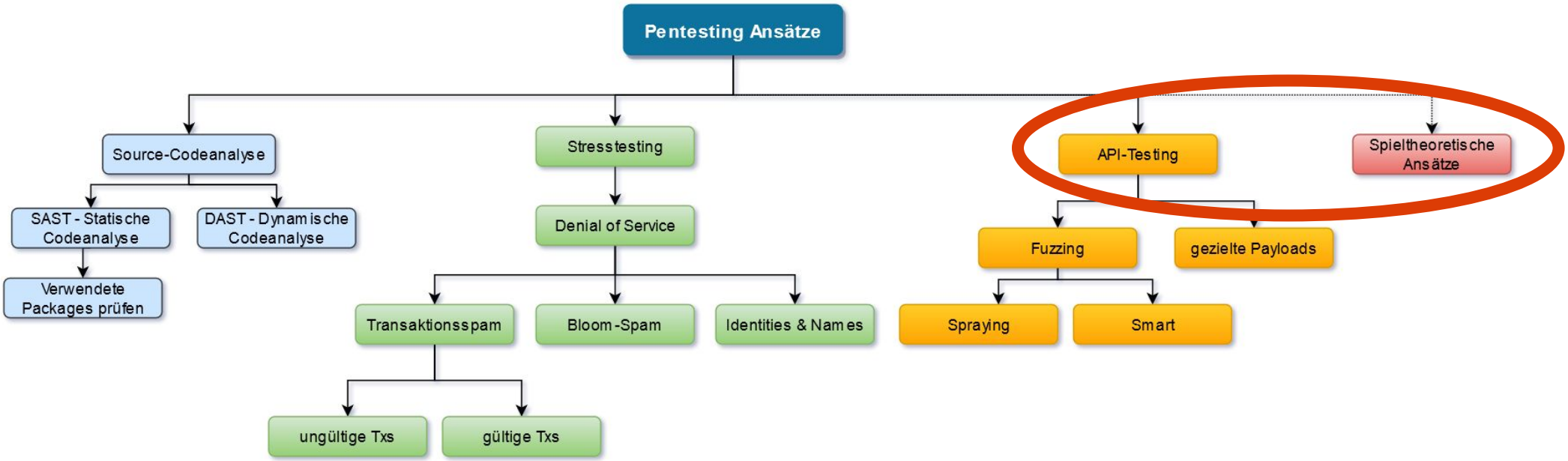
Kapitel 6 Flooding/DoS auf Layer 2

Kapitel 7 Fuzzing

Kapitel 8 DashPay / Race Conditions auf Dash Plattform / Spieltheoretische Angriffe

Dash Pay / Race Conditions auf Dash Platform / Spieltheoretische Angriffe

Übersicht



https://www.dash.org/wp-content/uploads/homepageimage-3-1.png

DashPay

Hintergrund und Ansatz

- Android/iOS App als Frontend für die Interaktion mit dem Dashpay Data Contract

Grundlegende Funktion:

- Nutzer können Profile erstellen, die einer Identity zugeordnet sind und sich über einen *DPNS*-Eintrag suchen und vernetzen
- Über diese Profile (welche in Dash Platform persistiert sind) können Nutzer dann untereinander Layer 1 Zahlungskanäle erstellen und sich bidirektional Dash senden
- Damit ist der *»komplizierte«* Teil mit den Adressen vor den Nutzern verborgen
- DashPay bietet somit eine UX die vergleichbar mit PayPal ist (allerdings dezentralisiert)

Ansatz:

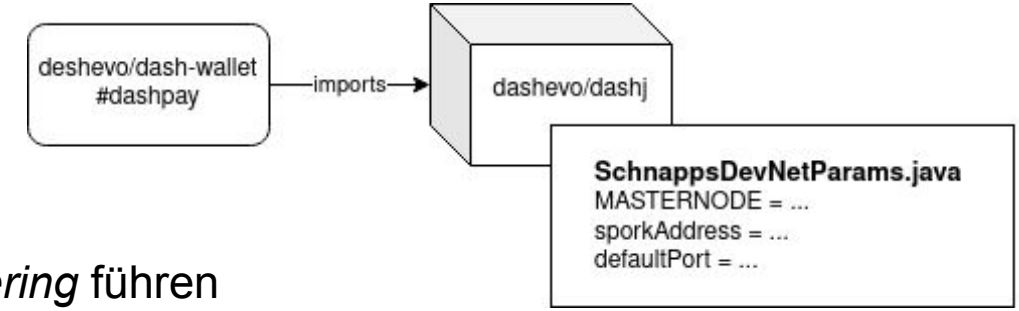
- DashPay App (Android) in Kombination mit unserem lokalen Devnet verwenden
- schauen ob Namen angelegt werden können, die nicht dem Validations-Standard entsprechen
- gibt es hier Probleme bei der App? Abstürze, fehlerhafte Zustände?

DashPay

Bauen der APK und Probleme bei der Installation des Artefaktes

Versuch #1:

- APK dekompilieren und in Java-Klassen eingebettete IP-Adressen der Masternodes ändern.
- Anschließend rekompilieren und auf physischen Gerät / Emulator wieder installieren.
- Kein Erfolg: Tooling unzureichend und Rekompilationsschritt nicht erfolgreich.

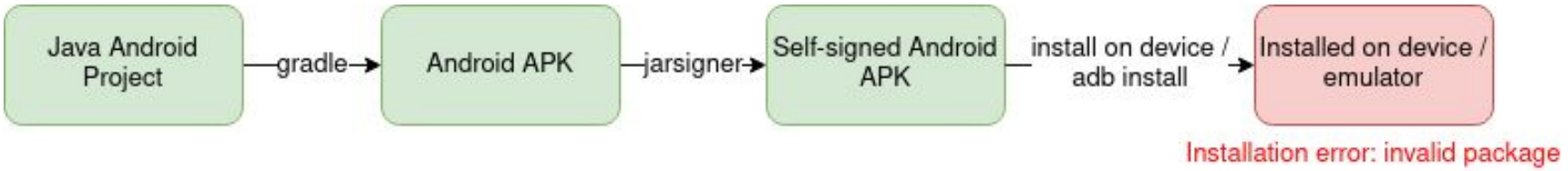


Versuch #2:

- Absprache mit Dashpay-Entwickler *hashengineering* führen
- Notwendigkeit für unser Vorgehen: Ändern der Abhängigkeit *dashj* (*bitcoinj* fork)
- Definition der notwendigen Parameter unseres lokalen Devnets in *SchnappsDevNetParams*

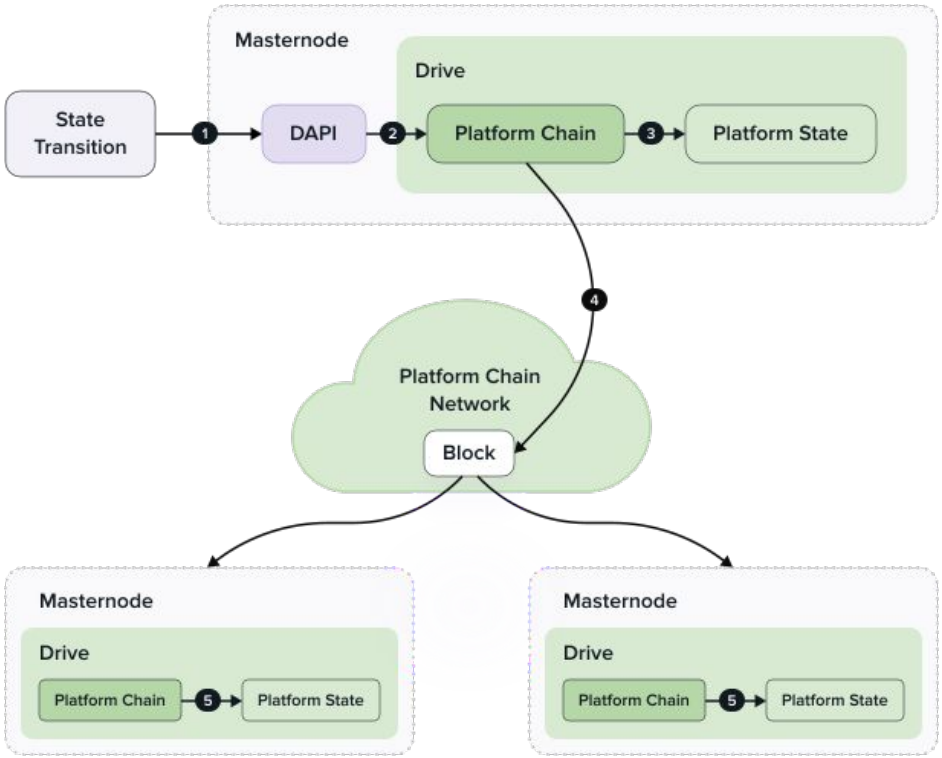
Kein Erfolg: Bauen der APK zwar möglich - nach anschließendem Signieren aber auf keinem Gerät / Emulator installierbar (*invalid package*).

- **Aufgeben**: Projekt kurz vor Ende, keine Zeit mehr und ohne Android-Erfahrung auch durch Absprache mit Entwicklern nicht auf unserer Seite lösbar.

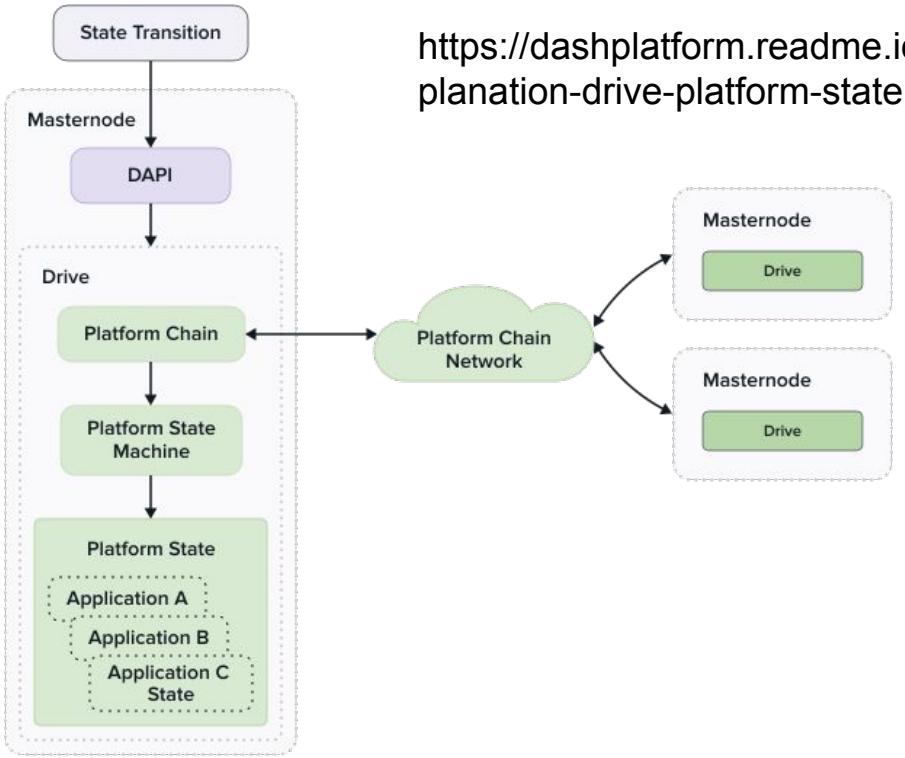


Race Conditions auf Dash Platform

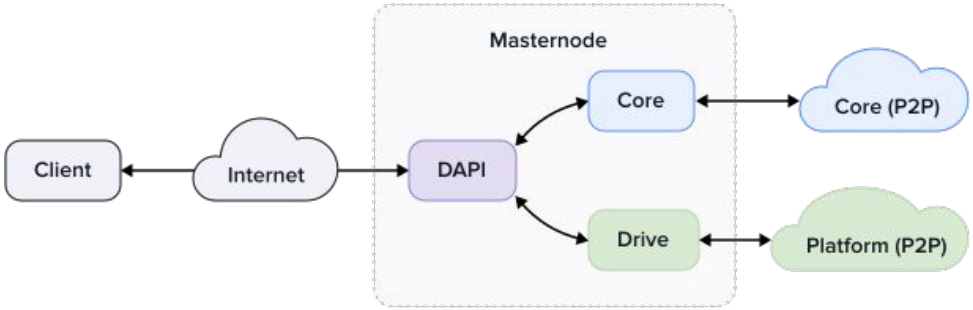
Hintergrund: Funktionalität von Dash Platform



https://dashplatform.readme.io/docs/explanation-drive



https://dashplatform.readme.io/docs/explanation-drive-platform-state



https://dashplatform.readme.io/docs/explanation-dapi

Race Conditions auf Dash Platform

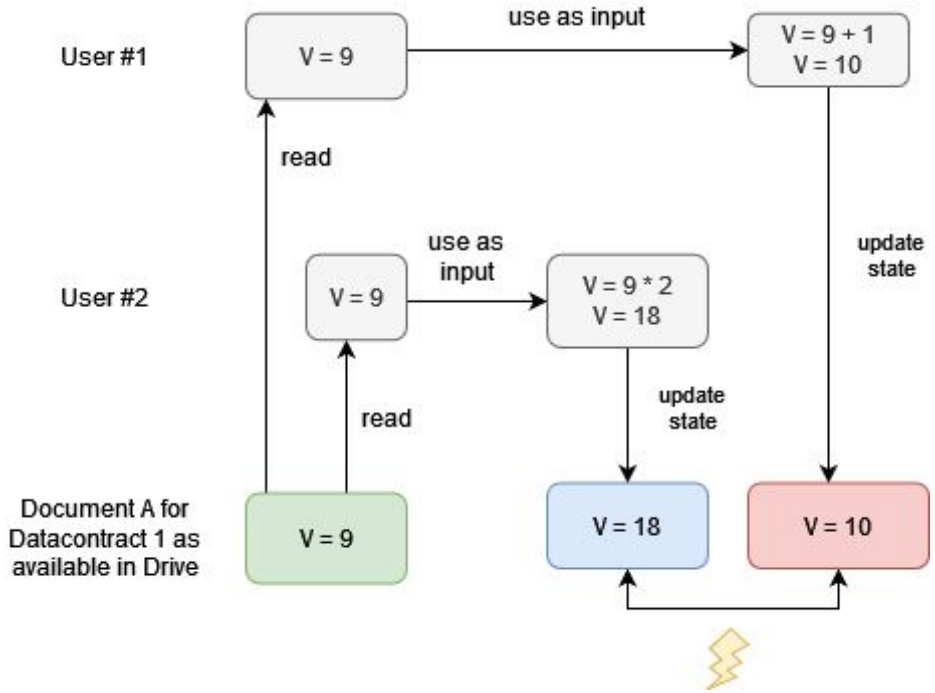
Ansatz und mögliche Race Conditions

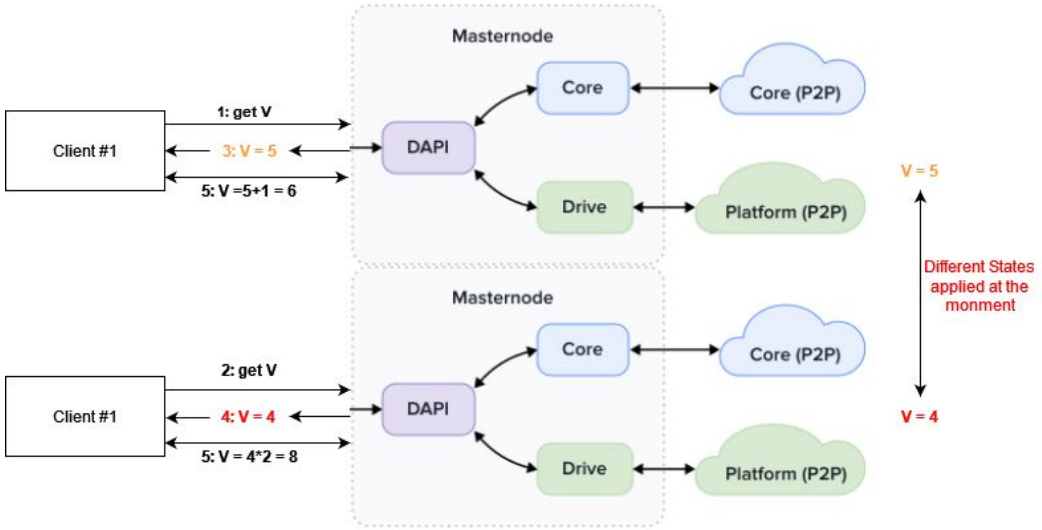
Szenario #1:

- Gleicher lesender Zugriff
- Unterschiedliche schreibende Ergebnisse

Szenario #2:

- Verbindung mit unterschiedlichen Masternodes in unterschiedlichen States
- Ungleicher lesender Zugriff, unterschiedliche schreibende Ergebnisse





Race Conditions auf Dash Platform

Educated Guess über potenzielle Folgen für Data Contracts

» 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. «

» 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. «

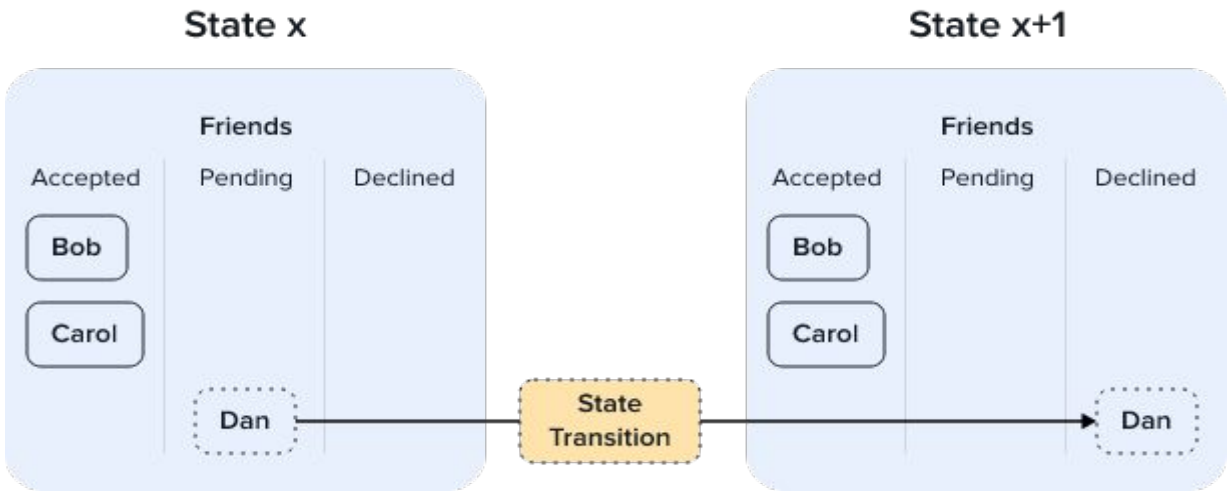
https://dashplatform.readme.io/docs/explanation-platform-consensus

- Durch spärliche Umschreibung in der Dokumentation unklar, ob Race Conditions überhaupt existieren können.
- Es scheint so als ob durch Tendermint immer **eine** Version als die gültige und aktuelle State Transition gewählt wird und durch das *ordering* ein wer zuerst bei der Mehrheit ankommt, mahlt zuerst gilt.
- Testen aus zeitlichen Gründen nicht mehr möglich, falls doch Race Conditions auftreten => Probleme bei Data Contracts à la *account basierten Tokens*.

Spieltheoretische Angriffe

Hintergrund

- Auf Layer 2 werden sogenannte *Credits* verwendet, um *state transitions* auf Dash Platform auszulösen
- *Credits* sind also die "*Währung"* auf L2
- Lesezugriffe sind also *kostenlos*, Schreibzugriffe *kosten Credits*
- *Credits* können durch eine sogenannte *asset lock transaction* auf Layer 1 beim Erstellen einer Identity erworben werden



https://dashplatform.readme.io/docs/explanation-platform-protocol-state-transition

Spieltheoretische Angriffe

Probleme

- *Credits* sind an eine fixierte Umwandlungsrate gebunden => Steigt das *Dash*, welches als *underlying* für die *Credits* dient im Preis, könnte es passieren das *state transitions* für eine Mehrheit unbezahlbar werden.
- Momentan ist das Umwandeln in *Credits* eine Einbahnstraße (laut Dev. Doku).

» You can check in at any time you like but you can never leave. «

- Man wird also implizit zum *bag holder* von wertlosen *Credits*, die nicht wieder in Dash umgewandelt werden können.

» Wo sind meine Dash? «

- Ab dem Moment wo man *Credits* erwirbt, hat man mit dem aktuellen System quasi schon verloren (Geld weg, *Credits* da).
- Fraglich ist auch wie Knoten, die mit *Credits* bezahlt werden so von ihren Diensten (welche Kosten verursachen) profitieren.
- Solange keine bidirektionale Umwandlungsmöglichkeit zwischen Dash ⇔ *Credits* besteht, sind spieltheoretische Angriffe in Verbindung mit *Credits* und L2 ausgeschlossen.

Eine bidirektionale Umwandlungsmöglichkeit wird also für die Sinnhaftigkeit von DP dringend benötigt!

Inhaltsverzeichnis

Kapitel 1	Einführung in die Dash Platform Architektur
Kapitel 2	Verwendete Methodik
Kapitel 3	Setup Testnet und Devnet
Kapitel 4	Statische Analyse
Kapitel 5	Flooding/DoS auf Layer 1
Kapitel 6	Flooding/DoS auf Layer 2
Kapitel 7	Fuzzing
Kapitel 8	DashPay / Race Conditions auf Dash Plattform / Spieltheoretische Angriffe
Kapitel 9	Fazit

Fazit & Lessons Learned

Ergebnisse:

- viele Ansätze sind durch Fork von Bitcoin schon gescheitert
- starke serverseitige Datenvalidierung:
  - verhindert Flooding
  - robust gegen Fuzzing
- TX-Filter anfällig durch Bloom Filter
- Anschluss des Bachelorprojekt könnte neue Erkenntnisse bringen (mehr Zeit)

Projektverlauf:

- steile Lernkurve (keine Vorerfahrung in den Gebieten durch das Studium)
  - Dash / Blockchain
  - Pentesting
- wenig Zeit für die Projektanforderungen
- Dash-Projekt noch nicht abgeschlossen
  - lückenhafte veraltete Dokumentation (Verweise auf mehrere Versionen)
  - veraltete Abhängigkeiten wurden nicht aufgelöst (Envoy, Config-Parameter, Tendermint, Dashmate)
  - erschwerter Aufbau der Testumgebung

Reflexion

- anfängliche Frustration durch Setup-Schwierigkeit
- gute Teamarbeit
  - sowohl Online als auch vor Ort
- Selbstorganisation
  - Aufteilung in Subteams
  - paralleles Arbeiten an mehreren Ansätzen
- Unterstützung durch den Projektbetreuer
  - Ideenfindung
  - Kommunikationsaufbau
- Projektübergreifende Unterstützung mit dem Bachelorprojekt
- Teamübergreifende Unterstützung (innerhalb des Projekts)
- Pentesting mitten in der Entwicklung schwierig
  - fehlende Tools & Vorerfahrung, hoher Setup Aufwand

Vielen Dank für Ihre Aufmerksamkeit!