

Protokollspezifikation und Referenzimplementierung zur Abbildung generischer cyber-physischer Verträge

Vom Fachbereich IV der Universität Trier
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

von

Lars Creutz, M.Sc.

Trier, 2024

Betreuer: Prof. Dr. Peter Sturm

1. Berichterstatter: Prof. Dr. Peter Sturm

2. Berichterstatter: Prof. Dr.-Ing. Guido Dartmann

Datum der Disputation: 24.05.2024

Zusammenfassung

Sowohl national als auch international wird die zunehmende Digitalisierung von Prozessen gefordert. Die Heterogenität und Komplexität der dabei entstehenden Systeme erschwert die Partizipation für reguläre Nutzergruppen, welche zum Beispiel kein Expertenwissen in der Programmierung oder einen informationstechnischen Hintergrund aufweisen. Als Beispiel seien hier Smart Contracts genannt, deren Programmierung komplex ist und bei denen etwaige Fehler unmittelbar mit monetärem Verlust durch die direkte Verknüpfung der darunterliegenden Kryptowährung verbunden sind. Die vorliegende Arbeit stellt ein alternatives Protokoll für cyber-physische Verträge vor, das sich besonders gut für die menschliche Interaktion eignet und auch von regulären Nutzergruppen verstanden werden kann. Hierbei liegt der Fokus auf der Transparenz der Übereinkünfte und es wird weder eine Blockchain noch eine darauf beruhende digitale Währung verwendet. Entsprechend kann das Vertragsmodell der Arbeit als nachvollziehbare Verknüpfung zwischen zwei Parteien verstanden werden, welches die unterschiedlichen Systeme sicher miteinander verbindet und so die Selbstorganisation fördert. Diese Verbindung kann entweder computergestützt automatisch ablaufen, oder auch manuell durchgeführt werden. Im Gegensatz zu Smart Contracts können somit Prozesse Stück für Stück digitalisiert werden. Die Übereinkünfte selbst können zur Kommunikation, aber auch für rechtlich bindende Verträge genutzt werden. Die Arbeit ordnet das neue Konzept in verwandte Strömungen wie Ricardian oder Smart Contracts ein und definiert Ziele für das Protokoll, welche in Form der Referenzimplementierung umgesetzt werden. Sowohl das Protokoll als auch die Implementierung werden im Detail beschrieben und durch eine Erweiterung der Anwendung ergänzt, welche es Nutzenden in Regionen ohne direkte Internetverbindung ermöglicht, an ebenjenen Verträgen teilnehmen zu können. Weiterhin betrachtet die Evaluation die rechtlichen Rahmenbedingungen, die Übertragung des Protokolls auf Smart Contracts und die Performanz der Implementierung.

Wissenschaftlicher Werdegang des Autors

- 10/2013 - 03/2014
Bachelor Angewandte Informatik
Technische Universität Kaiserslautern
- 03/2014 - 02/2017
Bachelor Angewandte Informatik
Hochschule Trier, Umwelt-Campus Birkenfeld
Bachelorarbeit: Konzeption und Umsetzung einer Anwendung zur interaktiven Erstellung und Simulation von Numerical Control-Programmen in Siemens NX
Abschluss: Bachelor of Science (2,0)
- 03/2017 - 08/2018
Master Angewandte Informatik - Schwerpunkt Robotik
Hochschule Trier, Umwelt-Campus Birkenfeld
Masterarbeit: Konzeption und Umsetzung einer digitalen Werkzeugmaschinenbedientafel auf Grundlage des Siemens Virtual Numerical Control Kernel
Abschluss: Master of Science (1,3 - mit Auszeichnung)
- seit 12/2020
Promotionsvorhaben
Universität Trier
Thema: Protokollspezifikation und Referenzimplementierung zur Abbildung generischer cyber-physischer Verträge

Wissenschaftliche Veröffentlichungen des Autors

2020

- M. Dziubany, L. Creutz, S. Kopp, J. Schneider, A. Schmeink, G. Dartmann, „Development of a Cyber-Physical System for an Autonomous Indoor Transportation Service,“ 2020 9th Mediterranean Conference on Embedded Computing (MECO), 2020
- L. Creutz, G. Dartmann, „Cypher Social Contracts A Novel Protocol Specification for Cyber Physical Smart Contracts,“ 2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), 2020

2021

- M. Dziubany, S. Kopp, L. Creutz, J. Schneider, A. Schmeink, G. Dartmann, Artificial Intelligence for Fleets of Autonomous Vehicles: Desired Requirements and Solution Approaches, Chapter in Smart Transportation: AI Enabled Mobility and Autonomous Driving, Verlag CRC Press, 2021
- L. Creutz, S. Kopp, J. Schneider, M. Dziubany, Y. Becker, G. Dartmann, Simulation Platforms for Autonomous Driving and Smart Mobility: Simulation Platforms, Concepts, Software, APIs, Chapter in Smart Transportation: AI Enabled Mobility and Autonomous Driving, Verlag CRC Press, 2021
- L. Creutz, J. Schneider, G. Dartmann, „Fides: Distributed Cyber-Physical Contracts,“ 2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPSISA), 2021

2022

- L. Creutz, K. Wagner, G. Dartmann, „Cyber-Physical Contracts in Offline Regions“, 2022 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), 2022
- L. Creutz, J. Schneider and G. Dartmann, „Distributed Hash Table with Extensible Remote Procedure Calls“, 2022 5th International Conference on Computational Intelligence and Networks (CINE), 2022

2023

- L. Creutz, G. Dartmann, „Decentralized Policy Enforcement in Zero Trust Architectures“, 2023 IEEE Future Networks World Forum (FNWF), 2023

Vorträge

2020

- 9th Mediterranean Conference on Embedded Computing (MECO): „Development of a Cyber-Physical System for an Autonomous Indoor Transportation Service“
- International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics): „Cypher Social Contracts A Novel Protocol Specification for Cyber Physical Smart Contracts“

2021

- Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA): „Fides: Distributed Cyber-Physical Contracts“
- MINT Konferenz Umwelt-Campus Birkenfeld: „Digitale Wiedererweckung des Dorfes - LandLeuchten“
- BMVI mFUND-Fachaustausch Blockchain: „Digitale Wiedererweckung des Dorfes (Projekt LandLeuchten)“

2022

- International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics): „Cyber-Physical Contracts in Offline Regions“

2023

- IEEE Future Networks World Forum (FNWF): „Decentralized Policy Enforcement in Zero Trust Architectures“ (Symposium on Security in Future Networks)

Forschungsprojekte

- Autonome, personenbezogene Organisation des Straßenverkehrs und digitale Logistik (APEROL), 10/2018 - 12/2020, BMDV
- Digitale Wiedererweckung des Dorfes (LandLeuchten), 01/2021 - 01/2023, BMDV
- Energieeffiziente Analyse- und Steuerungsprozesse im dynamischen Edge-Cloud-Kontinuum für die industrielle Fertigung (EASY), seit 02/2023, BMWK

Reviewertätigkeiten

- IEEE Internet of Things Journal (2021)
- IEEE International Conference on Computer Communications and Networks (2024)

Inhaltsverzeichnis

1	Motivation	1
2	Ziele und Aufbau der Arbeit	4
2.1	Ziele der Arbeit	4
2.2	Aufbau der Arbeit	7
3	Grundlagen	9
3.1	Advanced Encryption Standard	9
3.2	Diffie-Hellman-Schlüsselaustausch	10
3.3	Kryptographische Hashfunktionen	14
3.4	HMAC-basierte Schlüsselableitung (HKDF)	15
3.5	Distributed Hash Table	16
4	Stand der Forschung	18
4.1	Smart Contracts	18
4.1.1	Definition	18
4.1.2	Ethereum	21
4.1.3	Orakel	27
4.1.4	Domainspezifische Sprachen	28
4.2	Ricardian Contracts	30
4.2.1	Definition Ricardian Contracts	30
4.2.2	Anwendungen	33
4.3	Rechtliche Aspekte	38
4.4	Zusammenfassung	39
5	Protokollspezifikation	42
5.1	Definitionen	42
5.1.1	Accounts	42
5.1.2	Vertragsvorlagen	43
5.1.3	Verträge	48

5.1.4	Transaktionen	49
5.2	Dokumentation der Vorlagen und Verträge	52
5.2.1	Aufbau und Struktur der Indizes	52
5.2.2	Zuordnung von Objekten zu Indexknoten	54
5.2.3	Aktualisierungen der Indizes	55
5.2.4	Angriffsvektoren auf das Netzwerk	58
5.3	Exemplarischer Ablauf	62
5.3.1	Erstellen und Publizieren einer Vorlage	63
5.3.2	Veröffentlichung eines Vertrags	64
5.3.3	Annehmen des Vertrags	65
5.3.4	Bestätigung von Vertragsschritten	65
5.4	Unterschiede zum Stand der Forschung	66
5.4.1	Smart Contracts	66
5.4.2	Ricardian Contracts	72
5.5	Zusammenfassung	74
6	Referenzimplementierung Fides	76
6.1	Allgemeines	76
6.2	Konfiguration	77
6.2.1	Netzwerk	77
6.2.2	Logging	79
6.2.3	Lokale Kommunikation	79
6.2.4	Abstraktionen	79
6.3	Datenmodelle	79
6.3.1	Datenbankmodelle	79
6.3.2	Serialisierung	80
6.3.3	Datentypen	82
6.3.4	Fides Instanzen	89
6.4	API	91
6.4.1	Abstraktionen	92
6.4.2	Kommandozeilenanwendung	92

6.4.3	Kernfunktionen	92
6.4.4	Externe Module	109
6.4.5	Automatisierung	110
6.4.6	Zusätzliche Funktionen	110
6.5	Command Line Interface	113
6.5.1	Berechtigungen bei Accounts	113
6.5.2	Übergabe von Passwörtern	114
6.5.3	Verfügbare Kommandos	114
6.5.4	Ausgaben	122
6.6	Netzwerkarchitektur	124
6.6.1	Allgemeines	124
6.6.2	Netzwerkknotten	124
6.6.3	Nutzende	134
6.6.4	Private Netzwerke	141
6.6.5	Isolation der Netzwerke	144
6.7	Verschlüsselung	146
6.7.1	Angriffsvektoren auf die Verschlüsselung	147
6.8	Automatisierung	154
6.8.1	Allgemeines Vorgehen	154
6.8.2	Wiederverwenbarkeit durch Aufgabendefinition	156
6.8.3	Import und Export	156
6.8.4	Beispiel einer Automatisierung	158
6.9	Zusammenfassung	164
7	Verträge ohne Internetverbindung	165
7.1	Allgemeines	165
7.2	Übersicht	166
7.3	Konfiguration	168
7.4	Hardware	169
7.5	Protokoll	169
7.6	Datenmodelle	170

7.6.1	Datenbankmodelle	170
7.6.2	Datentypen	173
7.6.3	Instanzen der Abstraktion	174
7.7	Blinde Operationen	174
7.8	API	176
7.8.1	Kommandozeilenanwendung	177
7.8.2	Kernfunktionen	177
7.8.3	Zusätzliche Funktionen	183
7.9	Command Line Interface	184
7.10	Middleware	186
7.10.1	Grundfunktionen	187
7.10.2	Zusatzfunktionen	190
7.11	Exemplarischer Ablauf	192
7.12	Zusammenfassung	195
8	Evaluation	197
8.1	Rechtliche Bewertung	197
8.1.1	Terminologie	197
8.1.2	Zivilrechtliche Betrachtung	198
8.1.3	Datenschutzrechtliche Betrachtung	205
8.1.4	Zusammenfassung	207
8.2	Messungen der Performanz	208
8.2.1	Werkzeuge zur Messung	208
8.2.2	Darstellung der Messergebnisse	208
8.2.3	Auslastung bei Netzwerkknoten	209
8.2.4	Auslastung bei regulärer Nutzung	226
8.3	Vergleich mit Ethereum	230
8.3.1	Datentypen und Funktionen	231
8.3.2	Ausführungskosten	234
8.3.3	Offene Fragen	235

9 Ausblick	237
9.1 Referenzimplementierung	237
9.2 Übertragung des Konzepts	239
10 Fazit	242
Anhang	246
A Grundlagen	246
A.1 Betriebsmodi von Blockchiffren	246
A.2 Distributed Hash Table	247
B Stand der Forschung	248
B.1 Smart Contract Anwendungen	248
B.2 Rechtliche Bewertung Smart/Ricardian Contracts	251
C Code	256
C.1 Fides	256
C.2 LoRaWAN Middleware	256
D Evaluation	263
D.1 Darstellung der Messergebnisse	263
D.2 Messungen der Performanz	264
D.3 Fides als Smart Contract	309

Abbildungsverzeichnis

1	Verschlüsselung im Modus CBC. Eigene Abbildung nach [73].	10
2	Entschlüsselung im Modus CBC. Eigene Abbildung nach [73].	11
3	Man-in-the-Middle Angriff beim Diffie-Hellman-Schlüsselaustausch	13
4	Zusammenhang zwischen Ricardian und Smart Contracts. Eigene Abbildung nach [98].	32
5	Merkle Baum aufgebaut anhand der Beschreibungen der Aufgaben \mathcal{D}	47
6	Zusammenhang zwischen Vertrag und Vertragsvorlage	48
7	Adressierung der zweiten Aufgabe des Vertrags mit zusätzlichen Beweiselementen zur Berechnung von r_M , um die Bestätigung durchzuführen.	52
8	Zusammenhang zwischen Transaktionen	53
9	Sichtweise bei einem Denial-of-Service Angriff eines Netzwerkknotens	62
10	Veröffentlichung einer Vorlage und deren Import von einer anderen Partei	63
11	Ablauf beim Erstellen und Beziehen eines Vertrags	64
12	Ablauf bei der Annahme des Vertrags	66
13	Bestätigung eines Vertragsschritts	67
14	Kapselung der RPC Methoden für Netzwerkknoten	126
15	Isolation von Netzwerken und zwischen Netzwerkknoten und Nutzenden	145
16	Verschlüsselung innerhalb der Bearbeitung von Verträgen in Fides	148
17	Bit Flip Angriff auf AES-CBC Verschlüsselung	149
18	Exemplarische Automatisierung eines Vertrags durch <i>Hooks</i>	160
19	Übersicht zur Verwendung von Fides mit LoRaWAN	168
20	Zusammenhang zwischen regulären und LoRaWAN Transaktionen	172

21	Automatisiert erstellte Integration der Middleware innerhalb des The Things Network	188
22	Platzhalter innerhalb der Middleware	191
23	Gesendete LoRaWAN Nachrichten und Übersetzung innerhalb der Middleware	191
24	LoRaWAN Up- und Downlink Nachrichten innerhalb des The Things Network	195
25	Mockup mobiler Bestellprozess	240
26	Mockup Kommunikation der Parteien	241
27	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 1	270
28	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 1	271
29	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 1	272
30	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Versuch 1	273
31	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Versuch 1	274
32	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 2	275
33	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 2	276
34	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 2	277
35	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Versuch 2	278
36	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Versuch 2	279

37	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 3	280
38	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 3	281
39	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 3	282
40	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Ver- such 3	283
41	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Ver- such 3	284
42	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 1	285
43	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 1	286
44	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 1	287
45	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Ver- such 1	288
46	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Ver- such 1	289
47	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 2	290
48	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 2	291
49	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 2	292
50	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Ver- such 2	293
51	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Ver- such 2	294

52	CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 3	295
53	CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 3	296
54	Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 3	297
55	Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Versuch 3	298
56	Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Versuch 3	299
57	CPU Auslastung (0-50 %) bei regulärer Nutzung bei Versuch 1	300
58	CPU Auslastung (0-25 %) bei regulärer Nutzung bei Versuch 1	301
59	Speicherauslastung (GB) bei regulärer Nutzung bei Versuch 1	302
60	Festplattenzugriff (MB) bei regulärer Nutzung bei Versuch 1 .	303
61	Netzwerkverkehr (MB) bei regulärer Nutzung bei Versuch 1 .	304
62	CPU Auslastung (0-60 %) bei regulärer Nutzung bei Versuch 2	305
63	CPU Auslastung (0-25 %) bei regulärer Nutzung bei Versuch 2	306
64	Speicherauslastung (GB) bei regulärer Nutzung bei Versuch 2	307
65	Netzwerkverkehr (MB) bei regulärer Nutzung bei Versuch 2 .	308

Tabellenverzeichnis

1	Elemente einer Ethereum Transaktion nach [32]	25
2	Vergleich Ethereum Smart Contract/ <i>Cypher Social Contract</i> anhand der Ziele aus [157]	72
3	Konstruktorformat für Validatoren	83
4	Datentyp <i>Template</i>	84
5	Datentyp <i>Contract</i>	85
6	Vorlagenindex \mathcal{I}_T	86
7	Vertragsindex \mathcal{I}_C	87
8	Datentyp <i>Transaction</i> (protobuf)	87
9	Transaktionstypen als Teil eines <i>oneof</i> Objekts innerhalb von Transaktionen (protobuf)	88
10	Datentyp <i>LogEntry</i> (protobuf)	89
11	Kernkomponenten des Hook Interfaces	155
12	LoRaWAN Protokoll	171
13	Erweiterungen des Datentyps <i>LoRaContract</i>	173
14	Transaktionstypen als Teil eines <i>oneof</i> Objekts innerhalb von LoRaWAN Transaktionen (protobuf)	175
15	Prozessorinformationen eines Netzwerkknottens	213
16	Simulation von zwei Clients in fünf Iterationen mit einer War- tezeit von 120s	225
17	Simulation von 80 Contracts mit zwei, vier und acht clients in fünf Iterationen mit einer Wartezeit von 120s	225
18	Prozessorinformationen des Notebooks	227
19	Gaskosten des Smart Contract	234
20	Ungefähre Kosten (USD) des Smart Contracts bei unterschied- lichen Preisen pro Ether.	235

Auflistungsverzeichnis

1	Übertragungsformat des Beweises	51
2	Dateistruktur eines Accounts	82
3	Verkürzter Import von protobuf Datentypen	108
4	Regulärer Import von protobuf Datentypen	108
5	Kommandozeilenanwendung: Aktualisieren und Anzeigen eines Vertrags	123
6	Kommandozeilenanwendung: Einzelne Transaktion entschlüsseln	123
7	Kommandozeilenanwendung: Temporären privaten Schlüssel auslesen	123
8	Konstruktor <i>FidesGrondNode</i> (Auszug)	125
9	Bit Flipping Angriff: Verschlüsselung der Eingabe	150
10	Bit Flipping Angriff: Änderung der verschlüsselten Daten . . .	150
11	Bit Flipping Angriff: Lokale Validierung einer Transaktion . .	151
12	Dateistruktur einer Hook	157
13	Beschreibung einer Vorlage innerhalb der Kommandozeilenanwendung (Auszug)	159
14	Automatisierung: Callback Methode bei neuen Vertragsangeboten	160
15	Automatisierung: Callback Methode bei Vertragsaktualisierungen	162
16	Akzeptieren eines Vertrags in der Kommandozeilenanwendung	176
17	Blinde Operation innerhalb der Kommandozeilenanwendung (LoRaWAN Abstraktion)	176
18	LoRaWAN Beispiel: Erstellen und Publizieren eines Vertrags .	192
19	LoRaWAN Beispiel: Blinde Operation zur Annahme des Vertrags	193
20	LoRaWAN Middleware: Uplink Nachrichten	257
21	LoRaWAN Middleware: Neue Vertragsanfragen	257
22	LoRaWAN Middleware: Bestätigung von Aufgaben	258

23	LoRaWAN Middleware: Vertragsstatus abrufen	259
24	LoRaWAN Middleware: Vorlagenstatus abrufen	260
25	LoRaWAN Middleware: Vertrag löschen	261
26	Standardkonfiguration zur Messung der Systemlast (CPU) . .	263
27	Angepasste Konfiguration zur Messung der Systemlast (CPU)	263
28	Konfiguration zur Messung der Systemlast (RAM)	264
29	Skript zur Simulation von Nutzenden	265

1 Motivation

Ideengebend für die Arbeit und das darin entwickelte *Cypher Social Contracts* Protokoll sind die von Eric Hughes bereits in 1993 beschriebenen Punkte für private Interaktion innerhalb des „Cypherpunk’s Manifesto“ [107]. Hughes beschreibt hier die Notwendigkeit von Privatsphäre innerhalb einer offenen Gesellschaft im elektronischen Zeitalter und erläutert Konzepte wie Datensparsamkeit, bzw. das Offenlegen von Informationen bei Interaktionen zwischen Parteien. Anonyme Transaktionssysteme werden als Notwendigkeit angesehen, um solche Kommunikation in Form von offenen Foren/Netzwerken umzusetzen. Die heute bestehenden Probleme im Bezug auf Privatsphäre und Datensicherheit wurden schon damals erkannt: In [107] gibt Hughes an, dass die Privatsphäre, in der Regel umgesetzt durch sichere Kryptographie, nur von den Nutzenden selbst forciert werden kann, da Regierungen oder große Firmen, deren Geschäftsmodelle auf Nutzerdaten fußen, per se nicht daran interessiert sind, die Privatsphäre für die Nutzenden umzusetzen. Dies ist für den Status Quo zutreffend: Regierungen sprechen sich für Staatstrojaner und Chatüberwachungen aus und möchten unter anderem Ende-zu-Ende verschlüsselte Nachrichtendienste aushebeln¹, während es möglich ist, detaillierte Profile der Nutzenden erstellen und nahezu über das gesamte Internet hinweg verfolgen können [53], [128].

Systeme, die die Privatsphäre der Nutzenden respektieren, können nach [107] fast ausschließlich durch die Nutzenden selbst realisiert werden, indem sie sich selbst organisieren und Software auf Systemen nutzen, die sie auch selbst überprüfen können. Betrachtet man hierfür die zuvor erwähnte Chatkontrolle, so ist es selbst erfahrenen Anwendern nahezu unmöglich zu validieren, welche Version der Software genutzt wird und ob die angeblich enthaltene Kryptographie tatsächlich funktioniert. Werden Smartphone- bzw. App-Store Anbieter per Gesetz forciert Personengruppen andere Versionen der Anwendung

¹<https://www.patrick-breyer.de/beitraege/chatkontrolle/> (Abgerufen im Juni 2023)

auszuliefern, so kann die Verschlüsselung auf einfache Weise unerkannt ausgehebelt werden, indem zum Beispiel bei der Aktualisierung der Anwendung eine veränderte Version der Software bezogen wird, die die Nachrichten derart verschlüsselt, dass sie von einer dritten Partei lesbar sind. Die in der Arbeit entwickelten generischen cyber-physischen Verträge sollen daher Nutzenden die Möglichkeit zur Selbstorganisation und Dezentralisierung geben und damit eine Plattform schaffen, die es ermöglicht, sich sicher austauschen zu können. Die Art und Weise des Austauschs hängt vom jeweiligen Anwendungskontext ab, so können entweder rechtlich bindende Verträge geschlossen werden oder lediglich kommuniziert werden. Zum Schutz der Privatsphäre soll nur das zwischen den beteiligten Parteien offengelegt werden, was für die jeweilige Übereinkunft notwendig ist. Die benötigten Informationen sollen zudem schon vor der Erstellung des Vertrags einsehbar sein, wodurch die Möglichkeit einer transparenten Kenntnissnahme eröffnet wird. All dies soll ohne die Kontrolle einer zentralen Instanz unmittelbar durch die Nutzenden geschehen, wodurch es einer zentralen Instanz nicht möglich ist, die Kommunikation einzuschränken oder Änderungen am System selbst vorzugeben. Zusätzlich soll die Arbeit alternative Ansätze für einige Probleme bei aktuell verfügbaren Smart Contract Systemen liefern: Die generischen Verträge sollen nicht an eine Kryptowährung gebunden sein, sondern nur auf die menschliche Interaktion abzielen. Entsprechend kann ein Bezahlvorgang abstrakt definiert und extern validiert werden. Die Entkopplung von Zahlung und Vertragsaufgabe erlaubt hiermit zwar Geldströme zu verfolgen (sofern keine Barzahlung verabredet wurde), jedoch wird der Kontext der Geldübertragung der beobachteten Parteien durch die genutzte Kryptographie nicht ersichtlich. Die transparente Natur einer Blockchain (z. B. bei Ethereum) hingegen erlaubt das einfache Verfolgen aller Geldströme und die unverschlüsselte Interaktion mit Smart Contracts. Weiterhin soll es für Entwicklerinnen und Entwickler einfacher sein, Verträge in Form von kollaborativen Prozessen in den jeweils eigenen Systemen zu automatisieren. Smart

Contracts sind in der Regel nicht änderbar und müssen daher fehlerfrei umgesetzt werden, was zu erheblichen monetären Verlusten führen kann, wenn Fehler im Code durch Dritte ausgenutzt werden können. Zudem ist es nicht möglich, z. B. interne Bibliotheken der jeweiligen Organisation einzubinden. Die Arbeit soll Ansätze liefern, wie zwei Parteien über einen Vertrag kollaborieren können und dessen Bearbeitung jeweils lokal in den eigenen Systemen umgesetzt werden kann. Motiviert ist dieses Vorgehen durch die Möglichkeit der kontinuierlichen Verbesserung der eigenen Implementierung und durch das Verwenden von Teilautomatisierungen zur kollaborativen (manuell und automatisch) Bearbeitung des Vertrags. Weiterhin erhalten die Nutzenden die volle Kontrolle über ihre Daten und können jene auf zusätzlichen Systemen archivieren, ohne von einem Netzwerk oder einem fremden Softwaresystem abhängig zu sein.

Letztlich ist die Arbeit motiviert durch das Einbeziehen von Nutzenden in Regionen, die bisher kaum digitalisiert sind. So soll es Nutzerinnen und Nutzern in Regionen ohne Internetverbindung ebenfalls möglich sein, an Verträgen teilzunehmen.

2 Ziele und Aufbau der Arbeit

Im Folgenden werden die allgemeinen Ziele und Forschungsfragen sowie der Aufbau der Arbeit beschrieben.

2.1 Ziele der Arbeit

Ziele der Arbeit sind die Schaffung eines Protokolls und dessen Referenzimplementierung zur Abbildung generischer cyber-physischer Verträge. Verträge können hier als Übereinkünfte verstanden werden, die Prozesse zwischen zwei Akteuren beschreiben. Somit soll es zum Beispiel möglich sein, die Prozesse zweier Computersysteme als cyber-physischen Vertrag zu beschreiben, obwohl die dortigen Abläufe im rechtlichen Sinne keine Verträge darstellen. Jene Abläufe sind während ihrer Bearbeitung durch die Systeme dennoch auf einer abstrakten Ebene durch Menschen begreifbar, da sie durch das Protokoll einfach verstanden werden können. Wird das Protokoll für menschliche Übereinkünfte verwendet, die computergestützt dokumentiert und bearbeitet werden, so handelt es sich um Abläufe der echten Welt, deren rechtliche Bewertung je nach Anwendungsfall betrachtet werden muss.

Die Arbeit soll einen Beitrag in den Bereichen Ricardian Contracts und Smart Contracts leisten und hierbei insbesondere auf das einfache Verstehen der Übereinkünfte in Form eines minimalen Vertragsmodells abzielen.

Nachfolgend werden einige Kernziele der Arbeit im Ganzen beschrieben, die in Teilen die in [107] genannten Probleme erneut aufgreifen.

- **Transparenz:** Nutzende sollen in den Übereinkünften eine direkte Übersicht über die von ihnen durchzuführenden Aufgaben erhalten, bevor die eigentliche Interaktion beginnt. Identitäten von Nutzenden sollen, sofern gewünscht, innerhalb der jeweiligen Übereinkunft selbst bekannt werden und nicht durch darunterliegenden Mechanismus des Protokolls. Allgemein erlaubt es die Transparenz des Protokolls auch regulären Nutzenden die Abläufe verstehen zu können, fernab von deren

Wissen im Bereich der Informatik. Weiterhin erlaubt die Transparenz das Durchsetzen der Datensparsamkeit, da offengelegt wird, welche Daten in welchem Vertragsschritt benötigt werden.

- **Privatsphäre:** Damit während der Interaktionen durch die Nutzenden deren jeweilige Privatsphäre gewährt werden kann, soll das System kryptographische Methoden verwenden, um jenes sicherzustellen. Hierbei darf die Privatsphäre nicht mit der Geheimhaltung verwechselt werden (nach [107]: „Privacy is not secrecy“). Es soll lediglich sichergestellt werden, dass es den Nutzenden möglich ist zu entscheiden, welche Informationen geteilt werden, und dass beim Teilen dieser Informationen sichergestellt wird, dass nur man selbst und die andere Partei unmittelbar auf jene Informationen zugreifen kann [107]. Die jeweilige Privatheit/Speicherung der Daten ist somit Teil der Übereinkunft (und ggf. der jeweiligen Gesetzeslage bei kommerziellen Nutzenden). Weiterhin erlaubt das Verwenden von Kryptographie die Benutzung von ungesicherten Kommunikationskanälen. Sofern die Nutzenden sich und ihre Identitäten innerhalb des Systems kennen, können Nachrichten problemlos auch sicher über unsichere Kanäle übertragen werden [66].
- **Dezentralisierung und Selbstorganisation:** Die Dezentralisierung bezieht sich nicht allein auf den Aufbau des Netzwerks und die Kommunikation, sondern auch auf das Protokoll und dessen Implementierung. Hierzu müssen jene dauerhaft open-source verfügbar und über eine entsprechende Lizenz verwend- und erweiterbar bleiben. Weiterhin soll keine zentrale Organisation das Protokoll und die darunterliegenden Ideen einschränken können. Zusätzlich soll sich die Implementierung nicht auf einen einzelnen Anwendungsfall beschränken. Nutzergruppen, die die Software verwenden möchten, sollen dies getrennt von anderen tun können, wodurch sich entweder mehrere unabhängige Netzwerke/Fo-

ren Formen können oder das Protokoll gar durch einen zentralen Server betrieben werden kann. Weiterhin kann durch Dezentralisierung die Übertragung der Ideen des Protokolls in andere Domänen verstanden werden, die entkoppelt von den Resultaten und der Implementierung dieser Arbeit sind, um so Prozesse transparent, verständlich und mit klaren Zuständigkeiten zu beschreiben und hierdurch die Sicherheit und Privatsphäre der Nutzenden zu verbessern. Es sollte zudem jedem möglich sein, das Protokoll bzw. die Implementierung frei für die eigenen Zwecke in dem jeweiligen Kontext einzusetzen, um sich so digital selbst zu organisieren.

- **Referenzimplementierung:** Neben des Protokolls soll die dazugehörige Referenzimplementierung vielfältig einsetzbar und auf den gängigsten Betriebssystemen einfach zu installieren sein. Weiterhin soll die Implementierung unter einer freien Lizenz genutzt und erweitert werden können, um so Anpassungen für etwaige eigene Bedürfnisse vornehmen zu können. Um der enormen Komplexität von aktuellen Smart Contract Systemen für Entwicklerinnen und Entwickler entgegenzuwirken, soll es zudem einfach sein, die Implementierung in eigene Projekte einzubauen und somit zum Beispiel Prozesse innerhalb von Verträgen automatisieren zu können. Die Referenzimplementierung soll weiterhin die technische Grundlage für weitere Implementierungen des Protokolls liefern, um jenes zum Beispiel in grafischen Anwendungen für größere Personengruppen zugänglich zu machen.
- **Digitale Teilhabe in datenbegrenzten Netzwerken:** Die Teilnahme an digitalen Diensten schließt zunehmend strukturschwächere Regionen aus, in welchen keine oder in Teilen sehr schlechte Internetverbindung verfügbar ist. Ein Ziel der Arbeit ist die Verwendung des Protokolls und der Referenzimplementierung in offline Regionen durch LoRaWAN Konnektivität zur Teilnahme an Verträgen.

- **Ressourceneffizienz:** Die Implementierung soll auf leistungsschwacher Hardware benutzbar sein, um die Ziele der Selbstorganisation und der digitalen Teilhabe zu unterstützen.

2.2 Aufbau der Arbeit

Zunächst werden in Kapitel 3 einige Grundlagen, zum Beispiel zu kryptographischen Verfahren, die innerhalb der Arbeit angewandt werden, erläutert.

Im darauf folgenden Kapitel 4 werden verwandte Arbeiten und Konzepte vorgestellt, um die Arbeit entsprechend einordnen zu können.

Kapitel 5 beschreibt das *Cypher Social Contracts* Protokoll zur Umsetzung von generischen cyber-physischen Verträgen, welches innerhalb der Referenzimplementierung, ergänzt durch einige Erweiterungen und Verbesserung, implementiert wurde. Weiterhin werden die Ideen des Protokolls in den Stand der Forschung eingeordnet.

Kapitel 6 thematisiert die Referenzimplementierung im Detail und beschreibt die dort umgesetzten Komponenten zur Nutzung des Protokolls innerhalb der Anwendung Fides.

In Kapitel 7 wird eine Abstraktion der Implementierung vorgestellt, welche es ermöglicht, an cyber-physischen Verträgen in Regionen ohne direkte Internetverbindung mittels LoRaWAN Verbindung teilzunehmen.

Nachfolgend wird die Arbeit, bezogen auf diverse Teilbereiche, in Kapitel 8 evaluiert. Hierbei wird nicht nur auf die gemessene Performance der umgesetzten Referenzimplementierung eingegangen, sondern es werden zusätzlich auch rechtliche Aspekte zur Nutzung des Protokolls betrachtet. Ferner wird das Konzept der *Cypher Social Contracts* exemplarisch in Teilen als Ethereum Smart Contract implementiert, um die Übertragbarkeit des Konzepts und dessen Nutzbarkeit als Smart Contract zu bewerten.

Abschließend beschreibt der Ausblick in Kapitel 9 weitere Einsatzmöglichkeiten für die in der Arbeit entwickelten *Cypher Social Contracts* und macht zudem Vorschläge für etwaige Änderungen zur Verbesserung der Referenzimplementierung, bevor die Arbeit in Kapitel 10 zusammengefasst wird.

3 Grundlagen

Im Folgenden werden einige Grundlagen erläutert, die in späteren Teilen der Arbeit aufgegriffen werden.

3.1 Advanced Encryption Standard

Der Advanced Encryption Standard (AES), welcher in der Federal Information Processing Standards Publication (FIPS) 197 definiert ist [143], entspricht einer symmetrischen Blockchiffre, die auf dem Rijndael [63] Algorithmus aufbaut ist. Der einzige Unterschied zwischen AES und Rijndael liegt in den spezifizierten Werten für die Block- und Schlüssellänge [63] [143]. AES legt hier die Blocklänge auf 128 bits fest und unterstützt Schlüssellängen von 128, 192 oder 256 bits [63] [143]. Bei AES handelt es sich um den offiziellen Nachfolger des Data Encryption Standard (DES) [142], welcher mit seiner Schlüssellänge von 56 bits nicht mehr als sicher gilt.

Bei der Verschlüsselung wird ein Klartext-Block in Kombination mit einem geheimen Schlüssel verwendet, um so den verschlüsselten Block zu erzeugen. Bei der Entschlüsselung wird jener verschlüsselte Block mit dem gleichen geheimen Schlüssel verwendet, um so wieder den ursprünglichen Klartext zu erhalten [63]. Allgemein handelt es sich um eine iterierte Blockchiffre, was bedeutet, dass die Operationen in mehreren Schritten (sogenannten Runden) durchgeführt werden. Ausgewählte Betriebsmodi für AES werden in Anhang A.1 erläutert. Nachfolgend soll auf den in der Referenzimplementierung genutzten Modus CBC [75] genauer eingegangen werden.

Cipher-block Chaining (CBC) Bei CBC [75] werden bei der Verschlüsselung Daten der unverschlüsselten Blöcke mit den vorherigen verschlüsselten Blöcken verkettet/kombiniert [73]. Hierbei wird beim ersten Block ein Initialisierungsvektor (IV) verwendet, welcher zwar öffentlich (unverschlüsselt) vorliegen kann, jedoch nicht erratbar sein darf [73] [138]. Entsprechend eig-

nen sich für den IV kryptografisch sichere, zufällige Daten, die nicht wiederverwendet werden. Die Verschlüsselung mit der Betriebsart CBC wird in in Abbildung 1 verdeutlicht. Ändert sich der IV oder der erste Klartextblock (zum Beispiel durch einen Nachrichtenzähler), so ändert sich auch der verschlüsselte Text [138]. Durch die Verkettung bei der Verschlüsselung ist das Umstellen der Blöcke (siehe ECB) nicht mehr möglich. Ein Fehler/eine Änderung in einem verschlüsselten Block ruft eine Änderung in der gleichen Stelle des nächsten Klartextblocks hervor. Allgemein benötigt der CBC Modus Padding, sofern die Eingabedaten nicht einem Vielfachen der Blocklänge entsprechen [138]. Mögliche Angriffsvektoren im Bezug auf die Referenzimplementierung werden in 6.7.1 beschrieben.

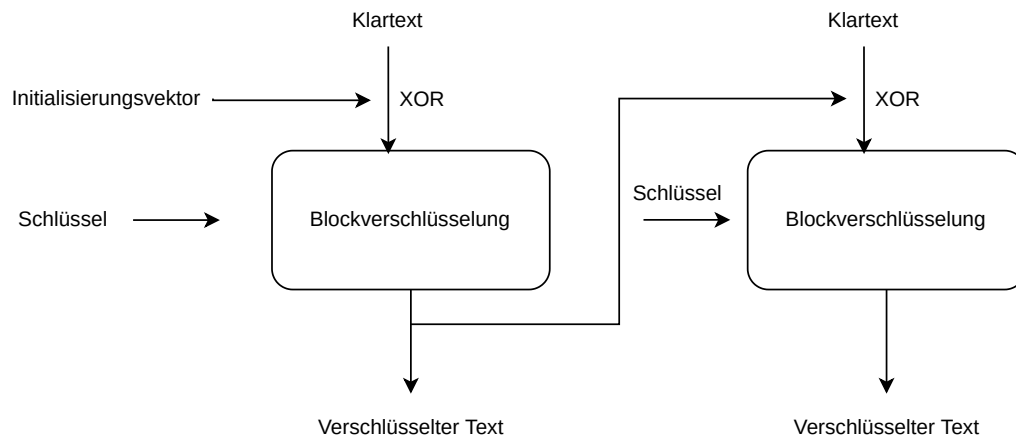


Abbildung 1: Verschlüsselung im Modus CBC. Eigene Abbildung nach [73].

Wie bereits erwähnt, sind die Operationen bei der Entschlüsselung umgedreht, was in Abbildung 2 verdeutlicht wird.

3.2 Diffie-Hellman-Schlüsselaustausch

Beim Diffie-Hellman-Schlüsselaustausch handelt es sich um den ersten, bereits in 1976 publizierten [67] Ansatz für ein asymmetrisches Kryptographie-

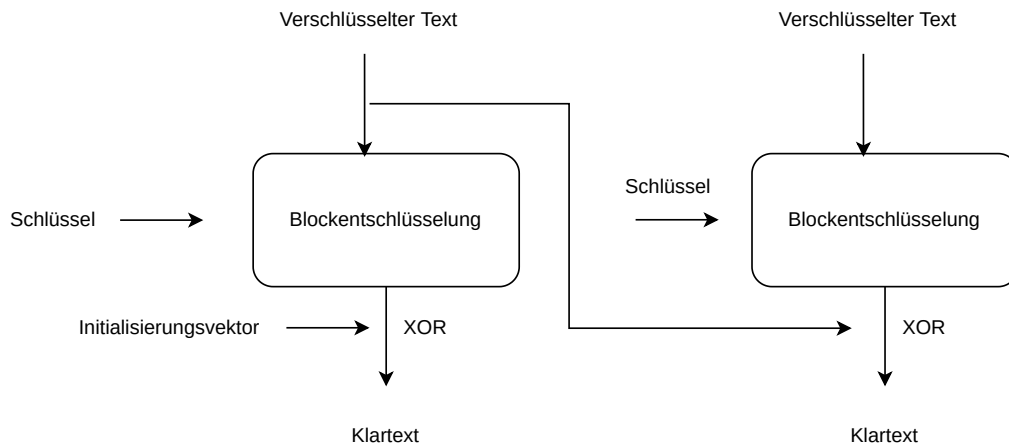


Abbildung 2: Entschlüsselung im Modus CBC. Eigene Abbildung nach [73].

verfahren, welches später unter anderem die Grundlage für RSA [150] war.² Das Verfahren greift zu dem Zeitpunkt der Publikation unveröffentlichte Ideen von Ralph Merkle, sogenannte Merkle Puzzle [140] (ein Verfahren zum Schlüsselaustausch über unsichere Kanäle), auf und wird daher zum Teil auch als Diffie-Hellman-Merkle-Schlüsselaustausch bezeichnet. Das Verfahren erlaubt die Berechnung eines geheimen Schlüssels über einen öffentlichen, nicht gesicherten Kommunikationskanal. Dieser Schlüssel wird dann in der Regel für symmetrische kryptografische Verfahren (z. B. AES) verwendet, um die darauffolgenden Nachrichten zu verschlüsseln. Im Folgenden soll das Verfahren in vereinfachter Weise erläutert werden. Wir nehmen die Kommunikation zwischen Alice und Bob an und betrachten zusätzlich Eve, eine weitere Partei, die die Nachrichten über den unsicheren Kommunikationskanal mitlesen kann. Nach [138] ergibt sich für den Ablauf:

1. Initialisierung (einmalig) einer Primzahl p und eines Erzeugers α einer zyklischen Gruppe \mathbb{Z}_p^* mit $2 \leq \alpha \leq p - 2$, die ausgewählt und veröffentlicht werden. Eine Gruppe G gilt als zyklisch, sofern es ein

²Ein ähnliches Verfahren wurde bereits zuvor innerhalb des britischen Geheimdienstes beschrieben [116] und war entsprechend nicht öffentlich zugänglich.

Element $\alpha \in G$ mit $G = \langle \alpha \rangle$ gibt, welches die gesamte Gruppe erzeugt: $\langle \alpha \rangle := \{\alpha^k | k \in \mathbb{Z}\}$ [119]. Die kommunizierenden Parteien Alice und Bob einigen sich im Vorfeld auf diese Parameter.

2. Beide Parteien bestimmen nun jeweils einen geheimen Schlüssel: Alice bestimmt x mit $1 \leq x \leq p - 2$. Äquivalent bestimmt Bob y mit $1 \leq y \leq p - 2$.
3. Bei der Kommunikation überträgt nun Alice $\alpha^x \bmod p$ und Bob $\alpha^y \bmod p$, also die öffentlichen Schlüssel zu den zuvor generierten geheimen Schlüsseln x und y .
4. Nachfolgend berechnet Alice für den geheimen Schlüssel $K = (a^y)^x \bmod p$ und Bob $K = (a^x)^y \bmod p$, wodurch beide den gleichen Wert für K erhalten. Die Sequenz an Nachrichten, gefolgt von der Berechnung für K , wird so bei der Bestimmung eines jeden Geheimnisses durchgeführt, zum Beispiel pro Nachricht, welche dann symmetrisch verschlüsselt werden soll.

Die Berechnung der öffentlichen Schlüssel verwendet die diskrete Exponentialfunktion, deren Umkehrung als diskreter Logarithmus bezeichnet wird [47]. Hierbei ist die eigentliche Funktion auch für große Exponenten effizient berechenbar, jedoch existiert kein Verfahren, um den diskreten Logarithmus effizient zu berechnen [47]. Für unser Beispiel bedeutet dies, dass Alice und Bob effizient ihre öffentlichen Schlüssel zu x und y berechnen können, es jedoch Eve, die diese Nachrichten beobachten kann, nicht effizient möglich ist, die privaten Schlüssel x und y zu berechnen. Ein möglicher Angriffsvektor auf den Diffie-Hellman-Schlüsselaustausch sind Man-in-the-Middle Angriffe. Unsere angreifende Partei Eve könnte sich zwischen Alice und Bob positionieren und deren Nachricht, welche die öffentlichen Schlüssel ankündigt, ändern. Hierzu würde Eve x_E und y_E und deren öffentliche Schlüssel berechnen. Bei der Übertragung von Alice zu Bob würde Eve dann $\alpha^x \bmod p$ abfangen, und

$\alpha^{x_E} \bmod p$ an Bob übertragen. Äquivalent würde Eve die Übertragung des öffentlichen Schlüssels von Bob verändern. Entsprechend würden Alice und Bob nun nicht die öffentlichen Schlüssel der jeweils anderen Partei verwenden, sondern die durch Eve geänderten Schlüssel. Eve, welche sich nun zwischen Parteien befindet, würde dann alle Nachrichten entschlüsseln, mitlesen und den ursprünglichen übertragenen öffentlichen Schlüssel erneut verschlüsseln und weiterleiten. Abbildung 3 verdeutlicht das Vorgehen. Daraus ergibt sich, dass öffentliche Schlüssel über andere Kanäle authentifiziert werden müssen, um dann eine sichere Kommunikation über unsichere Kanäle durchführen zu können.

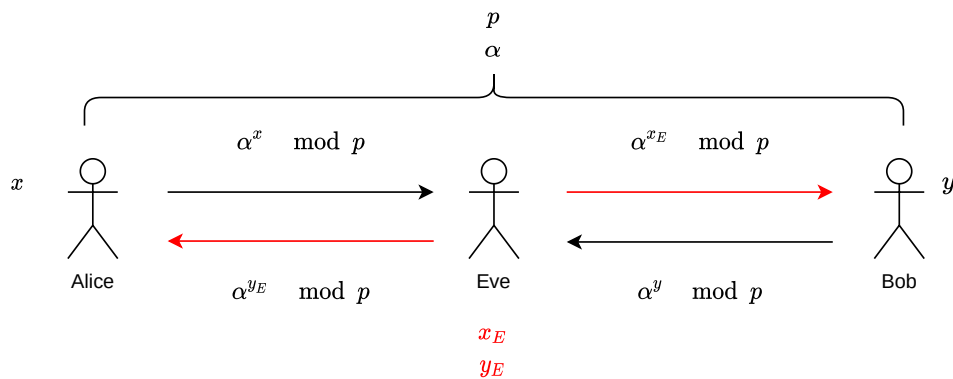


Abbildung 3: Man-in-the-Middle Angriff beim Diffie-Hellman-Schlüsselaustausch

Elliptic Curve Diffie-Hellman Elliptic Curve Diffie-Hellman (ECDH) ist eine Variante des Diffie-Hellman-Schlüsselaustausch auf Basis von elliptischen Kurven. Eine elliptische Kurve definiert als:

$$y^2 = x^3 + ax + b \quad (1)$$

Der Ablauf ergibt sich nach [135] wie folgt:

1. Zunächst einigen sich Alice und Bob auf eine elliptische Kurve. Für den weiteren Ablauf ist besonders der Generator der Kurve G relevant.
2. Weiterhin verfügt Alice über ihren privaten Schlüssel x , Bob über seinen Schlüssel y . Bei beiden Schlüsseln handelt es sich um Punkte auf der elliptischen Kurve. Die dazugehörigen öffentlichen Schlüssel ergeben sich aus der Punktmultiplikation ($*$) des privaten Schlüssels mit G .
3. Alice überträgt nun über einen (unsicheren) Kommunikationskanal ihren öffentlichen Schlüssel $x * G$, Bob überträgt entsprechend $y * G$.
4. Nachfolgend berechnet Alice das gemeinsame Geheimnis als $y * G * x$ und Bob als $x * G * y$, wodurch beide das gleiche Geheimnis erhalten.

Das gemeinsame Geheimnis kann dann wie bekannt in symmetrischen kryptografischen Verfahren verwendet werden.

3.3 Kryptographische Hashfunktionen

Hashfunktionen im Allgemeinen bilden Eingaben beliebiger Länge auf eine Ausgabe mit fester Länge ab und werden zum Beispiel verwendet, um die Integrität sicherzustellen [138]. Aufgeteilt werden diese Funktionen nach [138] in schlüssellose, deren Eingabe aus einer beliebigen Nachricht besteht, und schlüsselgebundene, die neben der Eingabe noch einen weiteren Schlüssel verwenden, Hashfunktionen. Im Folgenden sollen nur schlüssellose Hashfunktionen betrachtet werden, schlüsselgebundene werden in 3.4 kurz thematisiert. Sei $h(x) = y$ eine Hashfunktion (zum Beispiel SHA-256) mit der Eingabe x , die die Ausgabe y berechnet. Neben der einfachen Berechnung und der Kompression, welche sicherstellt, dass eine Eingabe beliebiger Länge eine Ausgabe einer konstanten Länge zuordnet, gelten nach [138] folgende Eigenschaften für kryptografische Hashfunktionen:

1. **Preimage Resistenz:** Für im Wesentlichen alle vorgegebenen Ausgaben

(Hash-Werte) ist es rechnerisch nicht möglich, die Eingabe x' zu finden, welche zu dieser Ausgabe y geführt hat ($h(x') = y$).

2. **2nd-Preimage Resistenz:** Es ist rechnerisch unwahrscheinlich, eine weitere Eingabe x' zu finden, die zur der gleichen Ausgabe führt, die sich aber von der eigentlichen Eingabe x , welche die Ausgabe y produzierte, unterscheidet ($(h(x) = h(x'), x \neq x')$).
3. **Kollisionsresistenz:** Es ist unwahrscheinlich, zwei unterschiedliche Eingaben x, x' zu finden, welche zur gleichen Ausgabe y führen ($h(x) = h(x') = y$). Im Unterschied zur *2nd-Preimage Resistenz* können hier die Eingaben x, x' frei gewählt werden.

3.4 HMAC-basierte Schlüsselableitung (HKDF)

HKDF [124] [125] beschreibt die Ableitung eines Schlüssels basierend auf Hashed Message Authentication Codes (HMAC) [123]. Unter HMAC versteht sich die Konstruktion eines Message Authentication Codes (MAC) auf Basis einer sicheren Hashfunktion (zum Beispiel SHA-256) in Kombination mit einem geheimen Schlüssel [123].

Allgemein wird MAC verwendet, um die Integrität von Nachrichten sicherstellen zu können, welche nicht rein durch die Verschlüsselung der Daten gegeben ist [63]. Somit erhält man nicht nur eine verschlüsselte Nachricht, sondern eine weitere Prüfsumme, die mit einer Nachricht übertragen wird und durch welche die andere Partei feststellen kann, dass die Nachricht während des Transports nicht verändert worden ist [63].

Das Ziel von HKDF ist es, aus dem initialen Schlüsselmaterial einen starken kryptografischen Schlüssel abzuleiten [124]. Hierzu existieren innerhalb von HKDF zwei Module [124]: HKDF-Extract, um einen pseudozufälligen Schlüssel abzuleiten und HKDF-Expand, um aus jenem Schlüssel weitere pseudozufällige Schlüssel zu erzeugen, die als eigentliche Ausgabe der Schlüs-

selableitung verstanden werden. Die Methoden sind wie folgt definiert [124]:

$$HKDF - Extract(salt, IKM) \rightarrow PRK \quad (2)$$

Hierbei beschreibt *salt* einen zufälligen, nicht-geheimen Wert. Sollte kein *salt* verwendet werden, wird ein String mit Nullen der Länge der verwendeten Hash Funktion verwendet. *IKM* definiert das initiale Schlüsselmaterial. Dies kann zum Beispiel einem in 3.2 beschriebenen Diffie-Hellman-Schlüsselaustausch entstammen. Der Rückgabewert der Funktion ist der bereits angesprochene pseudozufällige Schlüssel *PRK*. Für die konkrete Funktionsweise von HKDF ist es wichtig, dass der HKDF-Extract Schritt pro *IKM* nur einmal ausgeführt wird [125].

$$HKDF - Expand(PRK, info, L) \rightarrow OKM \quad (3)$$

Innerhalb von HKDF-Expand wird nun der zuvor berechnete Schlüssel *PRK* in Kombination mit einem Informationsparameter *info* und der Länge des zu berechnenden Schlüssels *L* verwendet, um den neuen Schlüssel *OKM* (output keying material) zu berechnen.

3.5 Distributed Hash Table

Unter einer verteilten Hashtabelle (distributed hash table - kurz DHT) versteht man eine reguläre Hashtabelle, welche über ein verteiltes System verwaltet wird. Eine Hashtabelle entspricht einer ungeordneten Liste von Schlüssel-Wert-Paaren (K, V) bei denen ein beliebiger Schlüssel *K* in der Regel nur einmal in jener Liste vorkommt und so klar dem Wert *V* zugeordnet werden kann [137]. In der Regel finden DHT Anwendung bei Peer-to-Peer (P2P) Systemen [171]. Hierbei ergeben sich für die Teilnehmenden Parteien innerhalb eines solchen Netzwerks einige Vorteile: Daten werden über die jeweilige Hash-Funktion innerhalb des Netzwerks zugeordnet und reduzieren somit

den Speicherbedarf der einzelnen Instanz [171]. Die Zuordnung einer Instanz erfolgt über deren ID innerhalb eines Netzwerks und in der Regel über ihre Entfernung zu K . Soll so nun der Wert V abgerufen werden, kann jene Anfrage innerhalb des Netzwerks entsprechend über die Nähe der ID zu K initiiert werden, was zu effizienten Anfragezeiten führt. Nachfolgend wird die DHT Implementierung Chord [155] kurz vorgestellt, die zum Teil später in der Referenzimplementierung samt Erweiterungen verwendet wird.³ Weitere DHT Implementierungen werden in Anhang A.2 beschrieben.

Chord Innerhalb von Chord [155] wird die ID von Knoten durch ihre IP-Adresse bestimmt und hiermit Werte V über die Nähe des Schlüssels K der gehashten IP-Adresse zugeordnet. Hierbei formt Chord ein ringförmiges Overlay Netzwerk, in dem jeder Knoten einen Zeiger auf seinen direkten Nachfolger verwaltet. Jener Knoten verfügt also über die IP-Adresse, deren Hash größer der ID des betrachteten Knotens ist. Bei der Zuordnung des Schlüssel-Wert-Paars wird der erste Knoten gewählt, bei dem die eigene ID größer oder gleich K ist. Um das Suchen innerhalb eines Netzwerks zu vereinfachen, verwalten die Knoten innerhalb von Chord zur Verbesserung des Routings sogenannte Finger-Tables [155]. Hierbei enthält der i -te Eintrag dieser Tabelle den Weg zu dem nächsten Knoten im Ring, der dem aktuellen Knoten mit einem Abstand von mindestens $(n + 2^{i-1}) \bmod 2^m$ folgt, wobei n dem betrachteten Knoten und m der Anzahl an Bits der Hashfunktion entspricht. Knoten innerhalb der verteilten Hashtabelle stabilisieren ihre Zustände in regelmäßigen Abständen selbst, indem sie die Korrektheit des eigenen Nachfolgers prüfen und die Routingtabelle kontinuierlich aktualisieren [155].

³Die Arbeiten verwenden aufgrund ihres Alters zum Teil veraltete Hashfunktionen wie SHA-1. Entsprechend greifen die Beschreibungen der verschiedenen verteilten Hashtabellen dies nur allgemein auf. Moderne Implementierungen verwenden in der Regel entsprechend sichere Hashfunktionen wie SHA-256.

4 Stand der Forschung

Um die Arbeit in den aktuellen Stand der Forschung einzuordnen, gibt dieses Kapitel eine Übersicht über verwandte Konzepte. Hierzu werden sowohl Smart als auch Ricardian Contracts vorgestellt.

4.1 Smart Contracts

Nachfolgend wird das Feld der Smart Contracts beschrieben. Zunächst werden hierbei deren Ursprung und die Begriffsdefinition betrachtet. Danach werden einige Projekte vorgestellt, welche Smart Contracts in unterschiedlicher Weise implementieren.

4.1.1 Definition

Der Begriff „Smart Contract“ wurde bereits in den 90er Jahren von Nick Szabo durch mehrere Arbeiten geprägt [156] [157] [158]. In [156] beschreibt er Smart Contracts als computergestütztes Transaktionsprotokoll, welches die Bedingungen eines Vertrages ausführt. Allgemein sollen verschiedene Bedingungen, wie zum Beispiel die Zahlungen, Pfandrechte oder deren jeweilige Durchsetzung, direkt an den Vertrag gebunden werden, um dadurch Ausnahmen und Fehler, sowohl versehentliche als auch solche bösartiger Natur, zu vermeiden. Bei den ökonomischen Zielen nennt Szabo das Senken der Kosten bei der Durchsetzung von Verträgen, die Minimierung von Verlusten bei betrügerischen Handlungen und allgemein die Reduktion von Transaktionskosten. Weiterhin sollen Smart Contracts das Vertrauen in Vermittler minimieren, um so direktere Interaktionen zwischen den Parteien zu ermöglichen. Als Beispiele für einfache Smart Contracts werden unter anderem Kassenterminals und der elektronische Datenaustausch genannt. Zusätzlich werden Verweise auf digitales Geld innerhalb von Smart Contracts in [156] deutlich. Hier beschränkt sich Szabo nicht nur auf die eigentliche Bezahlung, sondern fordert auch Protokolle, welche sicherstellen, dass zum Beispiel ein Produkt

auch geliefert wird. Die Notwendigkeit von Transparenz erklärt er anhand eines Beispiel eines Kassenterminals, welches möglicherweise ungefragt Daten von Kunden speichert, ohne dass dies den jeweiligen Kunden deutlich gemacht wird. Szabo beschreibt ein solches Verhalten als „hidden action“ der jeweiligen Software.

In einer anderen Arbeit [157] gibt er eine allgemeine Definition von Smart Contracts an. Hier definiert er Smart Contracts als eine Reihe von Versprechen, die in digitaler Form spezifiziert sind, einschließlich Protokollen, mit denen die Parteien diese Versprechen erfüllen. Er beschreibt die Inklusion der Protokolle in Hard- und Software zur Durchsetzung der Verträge am Beispiel eines Verkaufsautomaten. Die strikte Bindung an Hard- und Software widerspricht jedoch seiner allgemeineren Definition, da sich jene nicht auf eine automatische Ausführung bezieht. Des Weiteren nennt er das Einbetten von Smart Contracts in Besitz, sofern jener digital kontrolliert werden kann. Hierzu greift er das Beispiel eines Autos aus [156] auf, dessen Besitz über einen Smart Contract geregelt werden kann. Weiterhin beschreibt er vier Ziele der Vertragsgestaltung in [157]:

1. **Beobachtbarkeit:** Die Möglichkeit zur Prüfung des Vertrags durch die Parteien. Verträge sollen während der Bearbeitung observierbar sein, damit etwaige Vertragsbrüche unmittelbar erkannt werden können.
2. **Überprüfbarkeit:** Zum Nachweis, ob ein Vertrag korrekt abgearbeitet wurde, zum Beispiel durch einen Schiedsrichter. Zusätzlich erlaubt die Überprüfbarkeit auch die Unterscheidung zwischen Vertragsverletzungen und gutgläubigen Fehlern. Dieser Punkt ist erneut konträr zu der automatischen Ausführung auf Hard- und Softwareebene, da explizit auf menschliche Fehler eingegangen wird und die Wichtigkeit von rechtlichen Klärungen innerhalb eines Vertrags noch gegeben sein muss.
3. **Vertraulichkeit:** Nur die Parteien sollen Zugang zu vertraulichen Daten innerhalb der Bearbeitung des Vertrags erlangen. Ausgenommen

hiervon sind ausgewählte Schiedsrichter, die ggf. zwischen den Parteien schlichten sollen und durch jene Zugang erhalten können. Somit behalten die Vertragsparteien die volle Kontrolle über ihre Übereinkunft und Daten sollen nur so weit geteilt werden, wie es für die Erfüllung des Vertrags notwendig ist.

4. **Durchsetzbarkeit:** Szabo beschreibt ein Ziel als die Minimierung der Notwendigkeit der Durchsetzung durch entsprechende Protokolle, eingebettet in Hard- und Software, jedoch wird hier die Durchsetzbarkeit erneut allgemein erwähnt, was insbesondere für die Interaktion durch Menschen innerhalb von Smart Contracts von Relevanz ist.

Als Basis zur Erreichung dieser Ziele wird die Public-Key-Kryptographie [66] genannt und anhand von digitalen Signaturen verdeutlicht, wie Vertrauen zwischen zwei oder mehreren Parteien aufgebaut werden kann. Neben der Public-Key-Kryptographie [66] werden unter anderem noch Verfahren wie „blind signature“ [54] und „secret sharing“ [154] angegeben. Szabo beleuchtet auch mögliche Angriffe bei der Verwendung von Public-Key-Kryptographie [66], insbesondere vom Typ „Man in the Middle“ in Kombination mit PGP’s Konzept des „Web of Trust“ [173], indem durch Signaturen von anderen Schlüsseln für deren Integrität gebürgt werden kann. Hierbei ist es über indirekte Beziehungen schwierig eine konkrete Aussage zur Glaubwürdigkeit des Schlüssels zu treffen. Allgemein können dadurch Probleme entstehen, die Szabo auch in [157] aufgreift. Sofern die Schlüssel eindeutig einer Person zugeordnet werden können (zum Beispiel via der E-Mail auf wohl bekannten Schlüsselservern), müssen jene besonders geschützt werden. Ein Verlust des geheimen Schlüssels würde bedeuten, dass sich die angreifende Partei jederzeit als die angegriffene Person ausgeben kann, deren Schlüssel ggf. noch von weiteren Parteien als vertrauenswürdig eingestuft wurde. Abhilfe bietet PGP indem man seinen eigenen Schlüssel bei Verlust oder Kompromittierung durch ein Zertifikat zurückziehen kann. Dadurch beziehen andere Parteien diese Information über die genannten Schlüsselserver und stoppen die

Verwendung des jeweiligen Schlüssels. Szabo nennt auch die Problematik bei der Metadatenanalyse der Kommunikation. Dies ist besonders problematisch, wenn man im „Web of Trust“ dauerhaft mit dem jeweiligen Schlüssel kommunizieren möchte. Da Public-Key-Kryptographie [66] besonders bei unverschlüsselten Kommunikationskanälen, wie zum Beispiel öffentlichen P2P-Netzen, eine Rolle spielt, können Metadaten einfach analysiert werden. Klassische Schlüsselservers eignen sich hiermit nicht unbedingt zum Abbilden von Vertrauensbeziehungen. Ferner sollte es möglich sein, seine eigenen Schlüssel kontinuierlich wechseln zu können und jene ggf. nur für eine gewisse Art von Interaktionen offenzulegen.

Allgemein adressieren aktuelle Smart Contract Systeme, die in den folgenden Kapiteln beschrieben werden, Szabos Punkte in großen Teilen. Jedoch existieren einige Probleme bezogen auf seinen Annahmen:

- Transaktionskosten sind in modernen Systemen zum Teil deutlich höher als angenommen.
- Protokolle können zwar in Hard- und Software Aktionen forcieren und umsetzen, jedoch ist dies nicht der Fall sofern reale Personen Teil dieser Verträge sind. Ein Protokoll kann zwar rein rechtlich eine Bindung zur Aufgabe herstellen, jedoch die tatsächliche Aktion der Person nicht forcieren.
- Durch die Analyse von Metadaten sollte es Nutzenden möglich sein, ihre Identitäten bei der Bearbeitung von Verträgen wechseln zu können, wodurch klassische Schlüsselservers am Beispiel von PGP [173] nur bedingt Anwendung finden können.

4.1.2 Ethereum

Ethereum [50] beschreibt ein dezentrales System zur Umsetzung von Applikationen und Organisationen auf Grundlage der Kryptowährung Ether. Zu den Anwendungen gehören so zum Beispiel die Verwaltung von Vermögenswerten.

[36] Zum Erreichen von Konsens über die verteilten Teilnehmer des Ethereum-Netzwerks wird eine Blockchain verwendet. Im Gegensatz zu Bitcoin [141], was primär als Wertspeicher verwendet wird, handelt es sich bei Ethereum nicht nur um ein reines Zahlungsnetzwerk der genutzten Währung, sondern es erlaubt zudem die Programmierbarkeit von dezentralen Anwendungen, deren Zustand zusätzlich über ebenjene Blockchain dokumentiert wird. [36] Im Unterschied zu Bitcoin, was seit seiner Einführung auf den Konsensmechanismus „proof of work“ [141] setzt, stellte Ethereum am 15. September 2022 auf die Methode „proof of stake“ um, welche den Energieverbrauch des Netzwerks um 99% reduzierte [36].

Die Anwendungen, welche innerhalb des Ethereum Netzwerks operieren, werden als Smart Contracts bezeichnet. Im Allgemeinen gilt die genutzte Technologie innerhalb von Ethereum und deren Smart Contracts zum Zeitpunkt der Dissertation als „state of the art“ in jenem Bereich. Nachfolgend sollen jene Smart Contracts thematisiert werden, welche die zuvor beschriebenen Ideen von Szabo in Teilen aufgreifen. Zwecks der Konsensmechanismen sei auf die Originalarbeiten zu „proof of work“ [141] und „proof of stake“ [23] verwiesen.

Definition Smart Contracts (Ethereum) In [28] werden Smart Contracts innerhalb von Ethereum als nicht-änderbare, wohldefinierte Anwendungen auf der Blockchain beschrieben, die traditionelle Verträge in digitale Verträge überführen sollen. Weiterhin wird das Beispiel von Szabo eines digitalen Marktplatzes aufgegriffen, welches durch die Smart Contracts als umgesetzt beschrieben worden ist. Die Autoren beschreiben zudem, dass es sich bei den Verträgen (im juristischen Sinne) lediglich um Übereinkünfte handelt, die in Programmcode überführbar sind. Hierbei wird die Nutzbarkeit in rechtlichen Kontexten jedoch nicht klar herausgearbeitet und keine Aussage getroffen, dass der reine Programmcode bereits als rechtlich bindet verstanden werden kann. Einschätzungen, ob Smart Contracts tatsächlich

rechtlich bindend sein können, werden in späteren Teilen der Arbeit erneut aufgegriffen.

Entsprechend soll vermittelt werden, dass Vertrauen, welches die Grundlage menschlicher Interaktion ist, nicht notwendig ist, sofern die nicht-änderbaren Anwendungen innerhalb des verteilten Systems korrekt programmiert werden. Die Autoren beschreiben dies anhand eines Fahrradrennens zwischen zwei Akteuren, bei welchem die Auszahlung des Preisgelds in einem „nicht-smarten“ (offline) Vertrag verweigert werden könnte [36]. Die direkte Übertragbarkeit auf einen Smart Contract wird innerhalb des Artikels nicht vorgenommen, jedoch durch das Beispiel suggeriert. Ferner wird die Verlagerung des Problems in Bezug auf das angesprochene obsole Vertrauen ignoriert. Zwar könnte ein Smart Contract die Auszahlung sicherstellen, jedoch muss das Ergebnis des Rennens, also ein Ereignis aus der realen Welt, welches zum Zeitpunkt der Programmierung des Smart Contracts nicht bekannt ist, durch jenen Smart Contract verarbeitet werden. Hierbei werden in der Regel sogenannte Orakel verwendet, wodurch die Dezentralität des Netzwerks verringert wird, da nicht jeder Netzwerkknoten Orakelfunktionen erfüllt und man somit Daten von externen Parteien vertrauen muss. Entsprechend wäre es in dem Beispiel möglich, dass ein Orakel falsche Daten liefert und somit durch den Smart Contract das Preisgeld unwiderruflich an die falsche Partei ausgezahlt wird. Jenes Problem wird erneut in Kapitel 4.1.3, welches sich allgemein mit Orakeln beschäftigt, beschrieben.

Nachfolgend sollen einige Kernkomponenten von Ethereum kurz vorgestellt werden, um die Prozesse bei der Erstellung und Bearbeitung von Ethereum Smart Contracts nachvollziehen zu können.

Accounts Innerhalb von Ethereum muss zwischen zwei verschiedenen Accountarten unterschieden werden:

1. Nutzeraccounts („externally-owned account“), die von jedem mit dem Zugriff auf den privaten Schlüssel kontrolliert werden können.

2. Contractaccounts, die von dem jeweiligen Programmcode des Smart Contracts kontrolliert werden.

Beide Arten von Accounts können sowohl die Kryptowährung Ether senden und empfangen als auch mit anderen Smart Contracts interagieren. [7] Die Accountarten unterscheiden sich darin, dass Nutzeraccounts bei ihrer Erstellung keine Kosten verursachen (lokale Berechnung des privaten und öffentlichen Schlüssels auf der elliptischen Kurve), wohin gegen Contractaccounts durch ihren Speicherbedarf innerhalb des Netzwerks entsprechende Kosten verursachen, da alle Netzwerkknoten die Informationen, die im Contract gespeichert werden sollen, auf ihren Systemen speichern müssen. Weiterhin können Contractaccounts nur Transaktionen als Antwort auf eine andere Transaktion verschicken. Nutzeraccounts können hingegen beliebig Ether untereinander transferieren. Weiterhin verfügen Contractaccounts nicht über eigene private Schlüssel, sondern sind nur durch die Programmierung des Smart Contracts und dessen Logik definiert. [7]

Transaktionen Eine Transaktion beschreibt die Zustandsänderung des Ethereum Netzwerks, ausgelöst von einem Nutzeraccount. Hierbei werden die Transaktion an alle Netzwerkteilnehmer, die einen Beitrag zum Konsens leisten, übermittelt und somit in dem verteilten Netzwerk ausgeführt. Der durch die Transaktion(en) geänderte neue Zustand wird dann an den Rest des Netzwerks übermittelt. Bei der Ausführung der Transaktionen entstehen Kosten, sogenannte Gaskosten, die von der sendenden Partei getragen werden müssen. [32]

Tabelle 1 beschreibt die Hauptelemente einer Transaktion innerhalb von Ethereum.

Allgemein lassen sich nach [32] drei Arten von Transaktionen unterscheiden:

1. Reguläre Transaktionen zwischen zwei Nutzeraccounts.

Element	Beschreibung
recipient	Ziel der Transaktion. Bei einem Nutzeraccount werden Werte übertragen (z. B. Ether), bei einem Smart Contract dessen Funktion(en) ausgeführt. Das Element wird geläufig auch als <i>to</i> bezeichnet.
signature	Signatur durch den privaten Schlüssel der sendenden Partei (<i>from</i> Element).
nonce	Inkrementierender Zähler, der die Anzahl an Transaktionen pro Account beschreibt.
value	Anzahl an Ether, die transferiert werden sollen.
data	Optionales Feld, welches in der Regel verwendet wird, um die Funktionen eines Smart Contracts von einem Nutzeraccount aufzurufen. Die Adressierung der Funktionen mit den jeweiligen Argumenten des Aufrufs richtet sich nach dem „application binary interface“ (ABI) [5].

Tabelle 1: Elemente einer Ethereum Transaktion nach [32]

2. Transaktionen, die einen Smart Contract veröffentlichen. Hierbei wird das Element *to* für den Programmcode des Smart Contracts verwendet.
3. Transaktionen, welche Funktionen eines Smart Contracts ausführen. Hierbei entspricht das Element *to* der Adresse des Smart Contracts.

Programmierung von Smart Contracts Wie bereits beschrieben, handelt es sich bei Smart Contracts innerhalb von Ethereum um Programme, die auf der Ethereum Blockchain ausgeführt werden. Jene Programme bestehen zum einen aus den Funktionen des Smart Contracts und den zugehörigen Daten, dem sogenannten Zustand. Jeder Smart Contract hat eine eigene Adresse auf der Blockchain und gilt so als Spezialfall eines Ethereum Accounts, wodurch Smart Contracts über ein aktuelles Guthaben verfügen und Transaktionen sowohl senden als auch empfangen können. [29]

Die Regeln zur Interaktion sind zum Zeitpunkt der Veröffentlichung des

Smart Contracts durch jene Programmierung festgelegt und somit nicht direkt von Nutzenden kontrolliert [29].

Smart Contracts innerhalb von Ethereum sind in der Regel nicht änder- bzw. löscherbar. Weiterhin sind alle Interaktionen final und können nicht rückgängig gemacht werden.[29]

Es steht jeder Partei frei, Smart Contracts innerhalb des Ethereum Netzwerks zu veröffentlichen. Je nach Komplexität des Smart Contracts entstehen hierbei bei der jeweiligen Transaktion entsprechende Gaskosten. Smart Contracts werden in einer höheren Programmiersprache, zum Beispiel Solidity [30], welche syntaktische Ähnlichkeiten zu Javascript aufweist, programmiert und in Bytecode kompiliert, welcher innerhalb der Transaktion übermittelt wird und von dem Ethereum Netzwerk interpretiert werden kann.

Neben regulären Smart Contracts können weiterhin sogenannte „multi-sig“ Contracts erstellt werden, die die Zugehörigkeiten zwischen mehreren Parteien aufteilen. Dies ist insbesondere relevant für dezentrale Organisationen, um Zugriffe auf Werte zu schützen und dem Verlust von privaten Schlüsseln vorzubeugen. Bei jenen Contracts müssen somit Transaktionen über eine Mindestanzahl an Signaturen aus einer zuvor definierten Menge von autorisierten Schlüsseln verfügen, damit die Transaktion als valide akzeptiert wird.[29]

Ethereum Virtual Machine Um die angesprochenen Smart Contracts dezentral ausführen zu können, wird die sogenannte Ethereum Virtual Machine (EVM) genutzt, auf welcher das Ethereum Protokoll basiert. Bei der EVM handelt es sich also um einen verteilten endlichen Automaten zur Dokumentation des Zustands des Ethereum Netzwerks. Innerhalb des Netzwerkes werden so durch die genutzte Blockchain nicht nur die monetären Transaktionen (Werttransfere) zwischen den Akteuren, sondern auch die Zustände und Änderungen dieser virtuellen Maschine dokumentiert [9]. Die Zustands-

übergangsfunktion Y ist nach [9] definiert als:

$$Y(S, T) = S' \quad (4)$$

Hierbei entspricht S dem aktuell gültigen Zustand, der mit einer Menge an validen Transaktionen T zum nächsten Zustand S' deterministisch überführt wird.

Bei der Ausführung des kompilierten Maschinencodes der Smart Contracts werden so verschiedene Maschinenbefehle (opcodes) ausgeführt [16]. Je nach Operation entstehen somit pro Befehl unterschiedliche Kosten, die innerhalb von Ethereum als Gas bezeichnet werden [10]. Neben den gängigen Operationen wie Addition, Subtraktion oder exklusivem Oder beinhalten die Maschinenbefehle der EVM weiterhin Blockchain-spezifische Operationen, zum Beispiel zur Bestimmung der Partei, die die jeweilige Berechnung angestoßen hat [9].

Neben Ethereum existieren noch weitere Smart Contract Anwendungen/Implementierungen auf unterschiedlichen Blockchains, die der Vollständigkeit halber in Anhang B.1 beschrieben werden.

4.1.3 Orakel

Nachfolgend sollen sogenannte Orakel vorgestellt werden, welche häufig im Kontext von Smart Contracts Verwendung finden. Orakel werden benötigt, da Smart Contracts in der Regel nur in der Lage sind Daten auf der jeweiligen Blockchain abzurufen. Dies schränkt die Nutzbarkeit eines solchen Programms erheblich ein. Daten der echten Welt werden daher mittels Orakeln an Smart Contracts übergeben. [20]

Nach [20] kann die Güte eines Orakels anhand folgender Kriterien bewertet werden:

- **Korrektheit:** Zustandsänderungen, ausgelöst von Smart Contracts, sollten nicht durch inkorrekte Orakeldata ausgelöst werden. Entspre-

chend muss das Orakel die Authentizität (Daten stammen von richtiger Quelle) und Integrität (Daten wurden nicht verändert) der Informationen garantieren.

- **Verfügbarkeit:** Aktionen von Smart Contracts sollen sich aufgrund von Orakel Daten nicht verzögern. Entsprechend muss gewährleistet werden, dass das Orakel ohne Unterbrechungen verfügbar ist.
- **Anreizkompatibilität:** Orakel sollen Parteien, die Daten zur Verfügung stellen (z. B. Wetterdaten), Anreize schaffen die Daten korrekt zur Verfügung zu stellen. Hierbei wichtig sind die Zuordenbarkeit und Verantwortlichkeit. Die externen Information sollten demnach klar der Partei zuzuordnen und die Qualität der Daten in jener Verantwortung sein. Eine entsprechende Entlohnung/Bestrafung soll dies umsetzen.

Die Orakel selbst lassen sich zudem unterscheiden in ihrer Architektur (dezentral/zentral), ihren Datenquellen und den jeweiligen Zusatzfunktionen, um den oben genannten Kriterien gerecht zu werden (z. B. Reputationssysteme).

Dennoch ist die Nutzung von Orakeln nicht unproblematisch, da selbst bei dezentralen Orakeln den Resultaten vertraut werden muss [74]. Durch die Nutzung von Orakeln geht somit ein Großteil der Dezentralität verloren und die eigentliche Anwendung wird durch externe Datenquellen angreifbarer. Weiterhin problematisch ist das Betreiben von langfristigen Smart Contract Anwendungen, die auf Orakel angewiesen sind, und bei denen man nicht sicher sein kann, ob der Orakeldienst über den gewünschten Zeitraum verfügbar bleiben wird.

4.1.4 Domainspezifische Sprachen

Um die Entwicklung von Smart Contracts zu vereinfachen und hierfür in Teilen natürliche Sprache zu nutzen, betrachten einige Arbeiten hierzu domainspezifische Sprachen (domain specific language - DSL). In [93] stellen

die Autoren einen semi-automatisierten Ansatz für eine DSL vor, die in Solidity übersetzt werden kann. Hierzu werden zunächst Regeln, Regularien, Gesetze o.ä. in Aussagen formuliert, die die Semantik erfassen. Nachfolgend werden jene Aussagen in Solidity Code abgebildet. Hierbei wird jedoch nur eine Grundform des Smart Contracts erstellt, welcher lediglich die einfach zu übersetzenden Aussagen mit Hinweisen für Entwicklerinnen und Entwickler enthält. Der Smart Contract ist somit nicht direkt nutzbar und es entstehen weitere Kosten für die Implementierung der generierten Funktionen durch die DSL.

Die Autoren in [169] beschreiben einen Ansatz für eine DSL, im Kontext des Papers „Contract Modeling Language“ (CML) genannt, und identifizieren hierfür einige wichtige Bausteine von Verträgen, die mit Hilfe der DSL in Solidity Code übersetzt werden können. Durch die CML sollen Absichten der jeweiligen Partei nicht nur beschreibbar, sondern auch flüssig lesbar sein. Hierdurch soll es ermöglicht werden, den Vertrag in Form der DSL syntaktisch einfacher verstehen zu können. Durch die Trennung der Vertragssemantik und der Implementierung durch die CML können plattformspezifische Aktionen (z. B. Einbinden von Bibliotheken) entkoppelt von der vertraglichen Beschreibung durchgeführt werden, was das System auch übertragbar auf andere Technologien macht.

Ein generischerer Ansatz, welcher natürliche Sprache zur Abstraktion von Smart Contracts mit einer DSL verwendet, findet sich in [149]. Die Autoren präsentieren einen Prototyp, welcher eine zuvor definierte Menge an verfügbaren Operationen und Datentypen unterstützt und exemplarischer Natur ist. Die Arbeit ist so eine Annäherung an eine einheitliche Vertragssprache, die die Semantik des Codes auf Abstraktionsebenen versucht verständlicher zu machen.

Zusammenfassend können domainspezifische Sprachen die Erstellung von Smart Contracts zukünftig vereinfachen. Für die breite Nutzbarkeit solcher

Anwendungen müssen jedoch weitere Fragestellungen betrachtet werden. Zum einen ist unklar, inwiefern sich zum Beispiel die rechtliche Betrachtung durch Nutzung von DSL zur Umsetzung von Smart Contracts verändert. Zum anderen ist es ggf. schwierig, domainspezifische Sprachen über verschiedene Domänen hinweg einzusetzen, um so die unterschiedlichsten Kollaborationen abbilden zu können. Die Komplexität bei der Abbildung der DSL Komponenten auf Smart Contract Code macht die Verwendung nur zum Teil einfacher für reguläre Anwenderinnen und Anwender. Betrachtet man die drei vorgestellten Arbeiten, so unterscheiden sich die jeweiligen DSL deutlich in ihrer Komplexität/Lesbarkeit. Wird eine DSL zu generisch, leidet darunter möglicherweise die Ausdruckskraft [169].

4.2 Ricardian Contracts

Ricardian Contracts können als Gegenspieler zu Smart Contracts verstanden werden und werden zum Teil mit jenen kombiniert. Hierbei adressieren Ricardian Contracts primär die semantische Seite von Übereinkünften, nicht deren automatisierte, starre Bearbeitung in Form von Programmcode.

4.2.1 Definition Ricardian Contracts

Ricardian Contracts wurden in [100] als Teil des Ricardo Systems vorgestellt und dann in andere Domänen übertragen. Ursprünglich wurden Ricardian Contracts entwickelt, um mit Anleihen zu handeln [99]. Der Autor gibt die Definition eines Ricardian Contracts wie folgt an [99]:

1. Ein Vertrag, der von Herausgebenden den Inhabenden angeboten wird,
2. sich auf einen Wert bezieht, der von den Inhabenden besessen und von den Herausgebenden verwaltet wird,
3. von Menschen leicht gelesen werden kann,
4. von Programmen gelesen werden kann,

5. digital signiert ist,
6. Informationen zu Schlüsseln und Servern enthält und
7. mit einer eindeutigen und sicheren Kennung verbunden ist.

Die Unterschiede zu gängigen Smart Contract Systemen werden unmittelbar deutlich, besonders im Punkt 3. Das Einbeziehen von Menschen als direkter Teil eines Vertrags erlaubt es, den Vertrag verstehen zu können, auch ohne Programmiererfahrung. Das Einbetten der Schlüssel in Punkt 6 schafft eine eigene Public-Key-Infrastruktur [99], die durch eine Signatur geschützt ist. Dennoch handelt es sich nicht um eine verteilte Anwendung, da zum Beispiel die Server innerhalb des Vertrags zentral betrieben werden. Weiterhin muss dem Schlüssel der Partei, die den Vertrag erstellt, vertraut werden. Punkt 7 bezieht sich auf das Adressieren von Verträgen, hierzu wurde durch den Autor SHA1 Hash-Werte angenommen [99], welche zu der Entstehungszeit der Idee noch als sicher galten. Transaktionen verweisen entsprechend auf diesen nicht veränderbaren Hash-Wert, der den Vertrag eindeutig identifiziert. Änderungen am Vertrag selbst würden die enthaltene Signatur ungültig machen bzw. den Hash-Wert (bei gültiger, neuer Signatur) ändern. Der Zusammenhang zwischen Smart und Ricardian Contracts wird in Abbildung 4 gezeigt.

Die Unterschiede der beiden Ansätze werden unmittelbar deutlich. Ricardian Contracts verfügen durch ihre Definition über einen hohen semantischen Anteil, was es allgemein möglich macht, die Übereinkünfte besser verstehen zu können als Smart Contracts. Bei Smart Contracts handelt es sich aus heutiger Sicht um dezentrale Programme, die entsprechend nur mit Vorwissen verstanden werden können. Die automatische Ausführung der Smart Contracts ist somit deutlich performanter als Ricardian Contracts, deren geschäftliche Logik zum Beispiel zusätzlich implementiert werden müsste. Im Gegensatz hierzu erlauben Ricardian Contracts aber prinzipiell eine Streit- und Änderbarkeit der Verträge, bilden also eher die mensch-

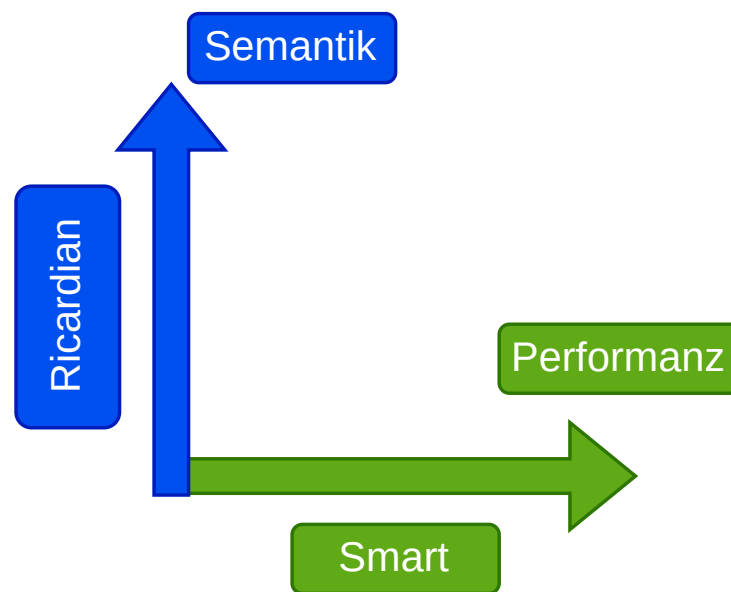


Abbildung 4: Zusammenhang zwischen Ricardian und Smart Contracts. Eigene Abbildung nach [98].

liche Übereinkunft ab. Auch wenn in [99] noch einige offene Punkte genannt werden, ist anzunehmen, dass Ricardian Contracts in einem legalen Kontext einsetzbar sind und hier weniger Hürden vorzufinden sind als es bei Smart Contracts der Fall ist. Dies ist entsprechend durch die Lesbarkeit der Verträge in Kombination mit digitalen Signaturen, welche Absichten bekunden, zu argumentieren.

4.2.2 Anwendungen

Ricardo System Bei dem Ricardo System vom Systemics [113], das die eigentliche Grundlage und den Namen für Ricardian Contracts vorgab, handelt es sich um ein System zur Abwicklung von Zahlungen durch verschiedenste Instrumente, die durch Ricardian Contracts beschrieben werden. Die Kommunikation findet durch die offene Software WebFunds [115] statt und verwendet das eigens entwickelte SOX Protokoll [106], welches unter anderem auch die verschlüsselte Interaktion mit dem jeweiligen Servern der Verträge sicherstellt [112]. An dieser Stelle sei erwähnt, dass die Projektseiten schon seit Längerem nicht aktualisiert wurden. Downloads, zum Beispiel für den WebFunds Client, sind nicht länger verfügbar und es wirkt allgemein so, als wäre das Projekt bereits länger eingestellt. Ob das Ricardo System noch aktiv verwendet wird, lässt sich mit den vorhandenen Quellen nicht eindeutig beantworten. Alle Ricardian Contracts, welche nicht von Systemics selbst stammen, sind auf der offiziellen Homepage nicht mehr abrufbar [109].

Der allgemeine Aufbau eines solchen Ricardian Contracts innerhalb des Ricardo Systems wird in [114] beschrieben. Im Folgenden soll nur auf das allgemeine Format und einige Kernaspekte eingegangen werden.

Das Format des Vertrags orientiert sich an dem INI Format [133] und verwendet 7 bit ASCII Zeichen zur Beschreibung [114]. Innerhalb der erlaubten Syntax existieren einfache Felder, Text über mehrere Zeilen, Arrays und Sektionen. Innerhalb der Ricardian Contracts wird bei den Sektionen zwischen primären und erweiterten Sektionen unterschieden [114]. Primäre Sektoren

enthalten nach [114] unter anderem:

- **entity**: Beschreibt den Ausstellenden des Vertrags, also wer den Vertrag unterzeichnet. Zusätzliche Felder der Definition enthalten Informationen zu dem Namen der Ausstellenden, einen Kurznamen (z. B. für Handelssoftware), den juristischen Namen, ISO-Ländercode, Logo und URLs zum Abrufen wichtiger Dokumente. [114]
- **issue**: Technische Verwaltung des Finanzinstruments. Enthält Informationen über Server, die den Dienst anbieten, die Art des Instruments (z. B. Anleihe oder Aktie) und Kontaktadresse für technischen Support (E-Mail). [114]
- **definitions**: Vorangestellter Abschnitt mit allgemeinen Definitionen des Vertrags [114].
- **conditions**: Sektion nach **entity** und **issue**, die den eigentlichen Inhalt des Vertrags, also dessen Klauseln, vor allem für menschliche Lesende beschreibt. [114]
- **keys**: Öffentliche Schlüssel, die möglicherweise relevant für den Vertrag sind. Die Gültigkeit der Signatur muss entsprechend separat geprüft werden. Hierbei muss der Vertrag mit dem jeweiligen Schlüssel für den Vertrag signiert werden. Dieser Schlüssel wird zusätzlich mit dem Schlüssel des Unternehmens signiert (Sektion **entity**). Somit entsteht eine Public-Key-Infrastruktur innerhalb des Vertrags. [114].
- **signatures**: Sektion, die die PGP Signatur in Klartext enthält. [173] [114]

Erweiterte Sektionen sollen nur einmal vorkommen und auf semantische Unterschiede zu der Sektion **issue** hinweisen. Daher muss die Art der Sektion **issue** der erweiterten Sektion entsprechen. Zu den erweiterten Sektionen

zählen nach [114]: **bond**, **share**, **currency** und **task**. Innerhalb der Sektionen werden die Instrumente entsprechend genauer beschrieben.

Weiterhin kann der Vertrag durch weitere Sektionen ergänzt werden, sofern jene einzigartig im Vertrag sind und sich allgemein an die Bedingungen des Formats halten [114]. Die obige Beschreibung bezieht sich auf die wichtigsten Elemente der Ricardian Contracts. Für alle weiteren Elemente eines Vertrags sei auf [114] verwiesen.

Abschließend sei erwähnt, dass die Spezifikation in Teilen nicht genau/aktuell ist. So wird die zusätzliche Sektion **task** zum Beispiel nicht unter den primären Sektionen erwähnt, auf die sich die Sektion **issue** bezieht. Im Allgemeinen lässt sich das Konzept der Ricardian Contracts jedoch auf andere Domänen in ähnlicher Weise übertragen.

OpenBazaar Bei OpenBazaar handelt es sich um eine peer-to-peer Anwendung zur Abbildung von dezentralen Marktplätzen. Das Projekt wurde eingestellt, die offizielle Website ist nur noch in Webarchiven verfügbar [19]. Die jeweiligen Implementierungen finden sich weiterhin in den offiziellen Repositories des Projekts [17]. Innerhalb des WebFunds Projekts werden zum Teil Abbildungen von OpenBazaar verwendet, um Ricardian Contracts zu beschreiben [111] [110]. OpenBazaar verwendet Ricardian Contracts, um die angebotenen Waren der Nutzenden und zusätzlich innerhalb des Protokolls dezentrale Reputationen abbilden zu können [6]. Bezahlungen finden über Kryptowährungen wie Bitcoin [141] statt [19]. Im Folgenden wird die Struktur und Benutzung von Ricardian Contracts in OpenBazaar erläutert. Hierzu betrachten wir zunächst eine Übersicht über die Inhalte der Verträge nach [6]:

- **Kryptographische Schlüssel:** Zur Zuordnung von Identitäten als Teilen des Vertrags (pseudonym).
- **Semantische Daten:** Der eigentliche Vertrag und dessen Geschäftsbedingungen.

- **Digitale Signaturen:** Nachweise, dass den Bedingungen des Vertrages zugestimmt wurde.
- **Kryptographischer Hash:** Zur Identifizierung und Sicherstellung der Unveränderlichkeit.

OpenBazaar erweitert die ursprünglichen Ricardian Contracts um eine Art Register für Transaktionen zwischen den Vertragsparteien. Hierzu dient der Ricardian Contract auch als Quittung („trade receipt“) der Übereinkunft. [6] Die Verträge werden in vier Stufen aufgeteilt [6]:

1. **Angebot des Anbieters:** Hier wird das Objekt (physisch, digital oder auch Dienstleistungen) durch die verkaufende Partei angeboten und beschrieben. Weiterhin wird ein Moderator ausgewählt, der unter anderem für die Vertrauensbeziehung und das Auflösen etwaiger Probleme verantwortlich ist.
2. **Bestellung:** Die kaufende Partei initiiert den Vorgang, der dann von der anbietenden Partei durch eine Signatur bestätigt wird. Zusätzlich verwendet die kaufende Partei eine Treuhandadresse der jeweiligen Kryptowährung und zahlt für das Produkt. Die Treuhandadresse ist besonders wichtig, um Probleme auflösen zu können, da die Auszahlung von dieser Adresse nur veranlasst wird, wenn zwei von drei Signaturen vorliegen. Besteht so Uneinigkeit zwischen den Parteien A, B und dem Moderator M, können entweder A und M oder B und M die Zahlung auslösen. Treten keine Probleme bei der Bearbeitung auf, sorgen die Parteien A und B für die jeweiligen Signaturen, wodurch das eingezahlte Geld der anderen Partei der Übereinkunft ausgezahlt wird.
3. **Auftragsbestätigung:** Die verkaufende Partei bestätigt den Auftrag und dessen Bearbeitung. Entsprechend enthält diese Bestätigung weitere Informationen (z. B. Tracking-Nummern bei physischen Waren).

Zusätzlich teilsigniert die verkaufende Partei eine Transaktion zur Auszahlung der Gelder durch die Treuhandadresse, die noch von der kaufenden Partei bestätigt werden muss.

4. **Empfang der Waren:** Die kaufende Partei bestätigt den Erhalt der Waren und signiert ihren Teil der Transaktion aus Punkt 3. Hierdurch werden die Gelder in der Treuhandadresse freigegeben und an die verkaufende Partei ausgezahlt. Zusätzlich kann eine Bewertung abgegeben werden, die von dem Moderator und der anderen Partei als eine Art Zusammenfassung gespeichert wird.

Zur Abbildung der Verträge verwendet OpenBazaar im Gegensatz zu den Ricardian Contracts von Ricardo [113] keine Textdateien im INI Format [133], sondern nutzt das Serialisierungsformat *protobuf* (Protocol Buffers) [65] von Google [18].

Weitere Anwendungen Ricardian Contracts wurden in den unterschiedlichsten Formen in weiteren Anwendungen verwendet. Da die meisten der Anwendungen jedoch bereits eingestellt wurden oder im Allgemeinen wenig Traktion erhielten, sollen jene der Vollständigkeit halber nur kurz angesprochen werden. Nach [110] finden sich Ricardian Contracts unter anderem in einem Fork [69] von OpenAssets, einem Protokoll das den Austausch von Vermögenswerten auf Grundlage der Bitcoin [141] Blockchain erlaubt [145]. Jener Fork, dessen Änderungen bereits im November 2014⁴ vorgeschlagen wurden, wurde jedoch bis zum aktuellen Zeitpunkt (Februar 2023) kein Teil der offiziellen Spezifikation.

FreiMarkets [94] ist eine Protokollerweiterung von Bitcoin [141], um Nutzenden die Möglichkeit zu geben eigene Vermögenswerte oder Währungen ausgeben zu können. Hierzu verweist das Projekt auf die Möglichkeit, Ricardian Contracts als externe Ressource außerhalb der Blockchain zu nutzen,

⁴<https://github.com/NicolasDorier/open-assets-protocol/commit/793320eec090a04f799f606f64eb4dc96af79e99> (Abgerufen im Februar 2023)

um dort Details der Vermögenswerte anzugeben.

Askemos [168] beschreibt das Konzept eines verteilten Betriebssystems, welches Informationen fälschungssicher verteilt darstellen soll. Hierzu verwendet die virtuelle Maschine des Projekts Baumstrukturen, die von den Teilnehmenden betrachtet werden [168]. Jene Nutzenden stimmen dann über die Korrektheit der vorliegenden Daten auf Basis ihres lokalen Wissens ab. BALL ist die Referenzimplementierung des Askemos-Projekts mit einigen Beispieldiensten, dessen Projektseite zum aktuellen Zeitpunkt (Februar 2023) nur über Webarchive erreichbar ist [167].

4.3 **Rechtliche Aspekte**

Die Neuheit der Technologie in Kombination mit den diversen Einsatzmöglichkeiten führt dazu, dass es weiterhin keine Klarheit darüber gibt, ob Smart Contracts oder Ricardian Contracts tatsächlich Verträge im Rechtssinn darstellen. Einige rechtliche Einschätzungen werden in Anhang B.2 aufgegriffen. Das für die Arbeit erstellte unabhängige Rechtsgutachten [95], welches die rechtlichen Rahmenbedingungen des Protokolls und der Referenzimplementierung untersucht, wird im Detail in Kapitel 8 aufgegriffen.

Probleme bei der Verwendung von Smart Contracts Nachfolgend soll auf einige Probleme bei der Nutzung von Smart Contracts für rechtlich bindende Übereinkünfte hingewiesen werden. Betrachtet man die rechtliche Betrachtung von Smart Legal Contracts in England [56], so lassen sich in der Arbeit Widersprüche zu der eigentlichen Idee von Smart Contracts erkennen. Die Autoren empfehlen Smart Contracts derart auszulegen, dass die Ausführung sich durch die Parteien beenden oder anhalten lässt. Dies entspricht einer klassischen Hintertür und macht die verteilte Ausführung eines Smart Contracts (in der Regel auf einer Blockchain) nichtig. Weiterhin wird in [56] mehrfach empfohlen, einige Aspekte/Rahmenbedingungen des Smart Legal Contract in natürlicher Sprache festzuhalten. Dies ist zwar im Kon-

text von hybriden Verträgen durchaus sinnvoll, verursacht jedoch für beide Parteien erhebliche Kosten. In jenem Fall müssen nicht nur das Programm korrekt implementiert werden und die Bedingungen an den Vertrag erfüllt werden, sondern zusätzliche Bedingungen auch noch im Kontext des klassischen Rechts korrekt formuliert werden. Entsprechend entstehen zusätzliche Kosten, was die Adaption eines solchen Vorgehens fraglich macht.

Ein weiteres Problem ist der Umgang mit sensiblen Daten je nach verwendeter Blockchain. Die Bundesregierung gibt hierzu in [49] an, dass die „Blockchain-Technologie datenschutzkonform ausgestaltet und angewendet“ werden muss. Interaktionen mit Smart Contracts sind in der Regel öffentlich einsehbar. Wird ein Smart Contract für die Durchsetzung eines Vertrags verwendet und prüft daher bestimmte Bedingungen, so müssen jene öffentlich gemacht werden und weitere Eingaben diesbezüglich auch. Nutzt zum Beispiel ein Smart Contract Informationen über einen Standort, so wird dieser publik kommuniziert. Sofern dieser Standort durch weitere Daten oder die Relation zu dem öffentlichen Schlüssel der übermittelnden Partei einer Person zugeordnet werden kann, sind diese Informationen in einer öffentlichen Blockchain durch die dort geltenden Konsensmechanismen nicht mehr löscher. Ansätze wie Zero-Knowledge-Proofs können dem entgegenwirken, verursachen jedoch erneute Kosten und werden in der Regel nicht auf der Kern-Blockchain ausgeführt, was zu weiteren Problemen führen kann [37].

Zusätzlich können unterschiedliche Rechtslagen in den jeweiligen Ländern der involvierten Parteien zu Hindernissen bei der Nutzung globaler Smart Contracts führen.

4.4 Zusammenfassung

Zusammenfassend sind zwei Konzepte verwandt mit der Arbeit: Ricardian Contracts und Smart Contracts. Während Ricardian Contracts über ein hohes Maß an Semantik verfügen, adressieren Smart Contracts primär die Automatisierbarkeit von Übereinkünften. Bei ihren Anwendungen unterscheiden

sich die beiden Konzepte. Ricardian Contracts eignen sich besser, um Verträge der realen Welt korrekt abzubilden und diese ggf. computergestützt bearbeiten zu können. Jedoch wurden Ricardian Contracts nie einheitlich spezifiziert und somit in diversen Anwendungen unterschiedlich definiert und implementiert. Das eigentliche Konzept erlangte in den Jahren nach der ersten Publikation jedoch allgemein wenig Traktion.

Bei Smart Contracts hingegen handelt es sich um programmierte Anwendungen, welche in der Regel in dezentralen Netzwerken von unterschiedlichen Netzwerkknoten ausgeführt werden. Ethereum gilt als „state of the art“ im Bereich der Smart Contracts und beeinflusste die Entwicklung vieler neuer Systeme mit ähnlichem Funktionsumfang. Bei der Entwicklung von Smart Contracts für reale Geschäftsbeziehungen existieren weiterhin Hürden, die deren Nutzung einschränken:

- **Beziehen von Daten der Außenwelt über Orakel:** Hierdurch wird die Dezentralität der Anwendungen ausgehebelt, welche bei Angriffen auf die Orakel zu direkten monetären Verlusten führen können, sofern die Berechnungen eines Smart Contracts auf falscher Datenlage erfolgen. Zusätzlich ergeben sich Probleme bei langfristigen Smart Contracts (z. B. Versicherungen), wenn dort sichergestellt werden muss, dass die Orakel über den gesamten Zeitraum fehlerfrei funktionieren.
- **Unterschiedliche/Unklare rechtliche Rahmenbedingungen:** Bei Kollaboration, die ggf. in unterschiedlichen Teilen der Welt stattfinden, müssen die jeweiligen Rechtslagen beachtet werden. Allgemein ist die Rechtslage für Smart Contracts nicht final geklärt und muss pro Einzelfall betrachtet werden. Hierdurch entstehen Mehrkosten für beide Parteien, da nicht nur die Rahmenbedingungen in natürlicher Sprache geklärt, sondern dies auch korrekt implementiert werden muss.
- **Datenschutzrechtliche Bedenken:** In öffentlichen Blockchains können Transaktionsdaten Rückschlüsse auf die involvierten Parteien zu-

lassen. Je nach Gesetzeslage kann dies problematisch sein, da die Daten nicht mehr gelöscht werden können.

- **Komplexität der Entwicklung:** Unveränderliche Smart Contracts zu programmieren und Fehler im Code nicht korrigieren zu können, kann zu erheblichen monetären Verlusten führen. Über die Jahre wurden so durch Angriffe, Betrugsversuche oder Fehler im Code immense Summen von Kryptowährungen verloren.⁵

Weiterhin ist es für Laien schwierig bis unmöglich Smart Contracts sicher zu programmieren. Ansätze für domainspezifische Sprachen können hier etwas Abhilfe schaffen, sind jedoch aus technischer Sicht auch als komplex einzustufen. Zusätzlich werden innerhalb von Smart Contracts kaum Interaktionen betrachtet, die nicht auf monetären Transferen beruhen. Geschäftsbeziehungen, deren gewünschte Zahlungsart nicht der darunterliegenden Kryptowährung des Smart Contract Systems entspricht, eignen sich somit nur bedingt für die Umsetzung mit einem Smart Contract. Transaktionskosten, die bei der Bearbeitung entstehen (und je nach System zum Teil hohe Kosten verursachen), müssen dennoch in der darunterliegenden Kryptowährung bezahlt werden.

Um Verträge digital umzusetzen und dies gleichwohl einer diverseren Personengruppe, die nicht vorrangig aus Programmierinnen und Programmierern besteht, zugänglich zu machen, bedarf es einer Kombination aus Ricardian Contracts und Smart Contracts, um generische Übereinkünfte abzubilden, deren Fokus nicht auf der monetären, sondern menschlichen Interaktion liegt. Das nachfolgend beschriebene Protokoll soll dies realisieren.

⁵<https://rekt.news/leaderboard/> (Abgerufen im Februar 2023)

5 Protokollspezifikation

Im Folgenden soll zunächst auf das Protokoll und dessen Kernaspekte eingegangen werden. Eine ursprüngliche Version des Protokolls wurde in [57] unter dem Namen *Cypher Social Contracts* als Hommage an das Cypherpunk's Manifesto [107], welches bereits in der Motivation der Arbeit thematisiert wurde, veröffentlicht.

Die Arbeiten an der Referenzimplementierung [61] ergänzten die finale Spezifikation des Protokolls um weitere Änderungen, die in der folgenden Beschreibung enthalten sind.

Dieses Kapitel setzt den Fokus auf die Verträge im Allgemeinen und beschreibt die jeweiligen Definitionen und Abläufe. Punkte, die sich im Speziellen auf die Implementierung beziehen, zum Beispiel die Art und Weise der Kommunikation und Verschlüsselung, werden in Kapitel 6 genauer erklärt. Dies soll die Trennung zwischen der eigentlichen Idee des Protokolls und dessen Implementierung realisieren, da sich die Implementierung im Laufe der Zeit, zum Beispiel beim Einsatz von stärkerer Verschlüsselung, ändern kann, das Konzept jedoch allgemein gültig sein soll.

5.1 Definitionen

Nachfolgend werden die Definitionen des Protokolls eingeführt und erläutert, um anschließend auf die Dokumentation der Übereinkünfte einzugehen und das Protokoll anhand von exemplarischen Abläufen zu zeigen.

5.1.1 Accounts

Das Protokoll definiert einen Account \mathcal{A} des Users i als:

$$\mathcal{A}_i = (\mathcal{K}_{i,S}, \mathcal{K}_{i,P}) \tag{5}$$

Hierbei handelt es sich um die Kombination aus privatem Schlüssel $\mathcal{K}_{i,S}$

und öffentlichem Schlüssel $\mathcal{K}_{i,P}$ im Bereich der Public-Key-Kryptographie [66]. Entsprechend wird der private Schlüssel verwendet, um Daten zu signieren, deren Signatur dann von anderen Parteien mit dem öffentlichen Schlüssel geprüft werden kann. Um innerhalb der Übereinkünfte sicher Daten übertragen zu können, existiert zudem ein weiterer Accounttyp, welcher nur kurzfristig pro Übereinkunft verwendet wird. Dieser temporäre (ephemeral) Account \mathcal{A}_ε wird verwendet, um Diffie-Hellman-Schlüsselaustausche durchzuführen und aus dem abgeleiteten Geheimnis Daten der Übereinkunft symmetrisch zu verschlüsseln.

$$\mathcal{A}_{\varepsilon_i} = (\mathcal{K}_{\varepsilon_i,S}, \mathcal{K}_{\varepsilon_i,P}) \quad (6)$$

Hierbei sei angemerkt, dass es allen Nutzenden dennoch möglich ist, den regulären Account \mathcal{A} regelmäßig zu rotieren, um so die eigene Privatsphäre zu erhöhen und die Analyse von Metadaten zu erschweren.

5.1.2 Vertragsvorlagen

Vertragsvorlagen, sogenannte Templates, werden verwendet, um Verträge abzuleiten. Gültige Verträge entstammen somit einer Vorlage, welche die Kernpunkte eines Vertrags definiert. Vertragsvorlagen schaffen 1-N Beziehungen zwischen der Vorlage und den daraus abgeleiteten Verträgen. Hierdurch können Vertragsvorlagen beliebig wiederverwendet werden, was in automatisierten Verträgen zum Beispiel dazu führt, dass Dienstleistende ihr Angebot lediglich einmal beschreiben müssen, welches dann von einer beliebigen Anzahl an Kundinnen und Kunden verwendet werden kann. Definiert sind die Templates wie folgt:

$$\mathcal{O}_j = (\mathcal{H}_{\mathcal{O}_j}, \mathcal{K}_{j,P}, B, T, R, r_M, V) \quad (7)$$

Hierbei beschreibt \mathcal{O}_j eine Vorlage des Users j . Die Vorlage wird durch $\mathcal{H}_{\mathcal{O}_j}$ identifiziert, was dem SHA-256 Hash der Vorlage entspricht. Hierbei

ist wichtig zu erwähnen, dass SHA-256 zum Zeitpunkt der Arbeit als entsprechend sicher galt [96]. Sollte sich diese Beurteilung langfristig ändern, sollte eine entsprechend neuere und sichere Hashfunktion verwendet werden. Zusätzlich sind in dessen Definition der öffentliche Schlüssel $\mathcal{K}_{j,P}$ von User j und eine allgemeine Beschreibung B enthalten. Die restlichen Elemente beziehen sich auf die in der Vorlage enthaltenen Aufgaben, welche bei der Vertragsabarbeitung bestätigt werden müssen. Hierbei entspricht T einer Liste von Beschreibungen der Aufgaben. Jene Beschreibungen bestehen aus generischem Text und können hierdurch von allen Parteien verstanden werden, sofern natürliche Sprache verwendet wird. Die Möglichkeit, primär natürliche Sprache für die Definition der Vorlagen zu verwenden, ordnet das Protokoll und die darin geschlossenen Verträge eher in die Richtung der Ricardian Contracts ein. Eine Aufgabe t_i ist definiert als Hash ihrer Beschreibung \mathcal{D}_i :

$$t_i = \mathcal{H}(\mathcal{D}_i) \quad (8)$$

Für T gilt:

$$T = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_N\} \quad (9)$$

Die Zuordnung jener Aufgaben zu der verantwortlichen Partei wird in R definiert:

$$R = \{r_1, r_2, \dots, r_N\} \quad (10)$$

Hierbei handelt es sich um eine Liste der gleichen Länge wie T , in jener boolesche Werte r die Zuordnung zur jeweiligen Partei vornehmen, also $r \in \{true, false\}$. Hierbei sei *true* definiert als die Zuständigkeit von j , also der Partei die die Vorlage erstellt, und *false* als die Zuständigkeit einer anderen Partei i , welche einen Vertrag erstellt und die Vorlage verwendet. Für N gilt: $0 < N \leq 255$. Entsprechend können 255 Aufgaben Teil eines Vertrages sein. Zusätzlich werden die Aufgaben mit minimalem Padding p aufgefüllt, damit gilt:

$$\log_2(N + p) \in \mathbb{N} \quad (11)$$

Hierbei werden leere Aufgaben verwendet, also $D_i =$, die nicht übertragen werden müssen und von dem Protokoll ergänzt werden.

Neben der Aufgabendefinition und deren Zuständigkeit sind in V Validatoren enthalten, welche pro Aufgabe definiert werden können und die Automatisierbarkeit des Protokolls gewährleisten sollen. Wie schon bei der Zuständigkeit muss eine korrekte Definition einer Vorlage die gleiche Anzahl an Validatoren wie Aufgaben enthalten, welche von den folgenden Typen sein können:

- *Plaintext*: Keine Validierung der Eingabedaten.
- *Range*: Überprüfung, ob die eingegebenen Daten innerhalb des angegebenen Bereichs pro Datentyp liegen.
- *Regex*: Kontrolle, ob die Daten einem regulären Ausdruck entsprechen.
- *Signature*: Test, ob die Daten von einem zuvor bekannten, in der Vorlage definierten Account signiert wurden.
- *Multiple signature*: Überprüfung der Eingabe auf mehrere Signaturen von verschiedenen Accounts.
- *SHA-256*: Überprüfung, ob die eingegebenen Daten einem gegebenen SHA-256 Hash entsprechen.

Im Folgenden wird die Nützlichkeit der Validatoren im Kontext der Automatisierung anhand kurzer Beispiele erläutert.

Range Validatoren prüfen Wertebereiche pro Datentyp und können zum Beispiel verwendet werden, um die maximale Bestellmenge eines Produkts festlegen zu können. Da offene Protokolle fehlerhaftes Verhalten nicht verhindern können, stellt somit jener Validator sicher (sofern beide Parteien eine korrekte, unveränderte Version der Software benutzen), dass Falscheingaben vermieden oder (bei fehlerhaftem Verhalten) zuverlässig erkannt werden.

Der Validator zur Prüfung von regulären Ausdrücken *Regex* macht klare Vorgabe an das gesamte Eingabeformat, sofern dieses sich aus verschiedenen

Komponenten zusammensetzt.

Die Validatoren *Signature* und *Multiple signature* stellen sicher, dass es sich bei den Eingabedaten um richtige Signaturen der zuvor angegebenen Partei handelt. Hierdurch ist es ggf. möglich, kritische Bereiche der Infrastruktur zu trennen, so dass die Parteien, welche die Signatur(en) ausstellen, nicht unbedingt direkter Teil einer Übereinkunft sein müssen. Der *SHA-256* Validator kann abschließend verwendet werden, um deutlich zu machen, dass es sich bei den übertragenen Informationen um unveränderte Daten handelt. Dies ist besonders nützlich, wenn zum Beispiel sensitive Daten wie eine Bankverbindung übermittelt werden soll. Sofern sich jene Daten ändern, müsste eine neue Vorlage erstellt und von den Nutzenden importiert werden.

Zusätzlich enthalten Vertragsvorlagen noch einen Hash r_M , was dem top Hash eines Merkle Baums [139] entspricht. Dieser Wert wird verwendet, um eine Zuordnung zwischen Aufgaben und deren Bestätigungen bei der Abarbeitung der Verträge zu machen, indem anderen Netzwerkteilnehmern durch einen einfachen Beweis klar gemacht wird, dass jene Aufgabe Teil des Vertrags ist, ohne auf die Vorlage zuzugreifen. Abbildung 5 verdeutlicht den Zusammenhang zwischen Aufgaben und dem daraus resultierenden Baum. Durch das hinzugefügte Padding bei der Anzahl der Aufgaben entsteht ein perfekter Binärbaum [51]. Neben den oben genannten Elementen beinhaltet eine Vorlage innerhalb der Implementierung noch einen Zufallswert, eine sogenannte *nonce*, welcher sicherstellen soll, dass der Hash des Vertrags eindeutig bleibt und somit keine Konflikte entstehen. Dies ergibt sich unmittelbar aus den verfügbaren Zuständen einer Vertragsvorlage:

- *active*: Die Vorlage wurde im Netzwerk veröffentlicht und ist aktiv. Teilnehmende, die die Vorlage kennen, können somit Verträge ableiten und jene publizieren.
- *inactive*: Die Partei, welche die Vorlage veröffentlicht hat, zog jene zurück. Es können folglich keine weiteren Verträge abgeleitet werden. Inaktive Vorlagen können nicht erneut aktiviert werden.

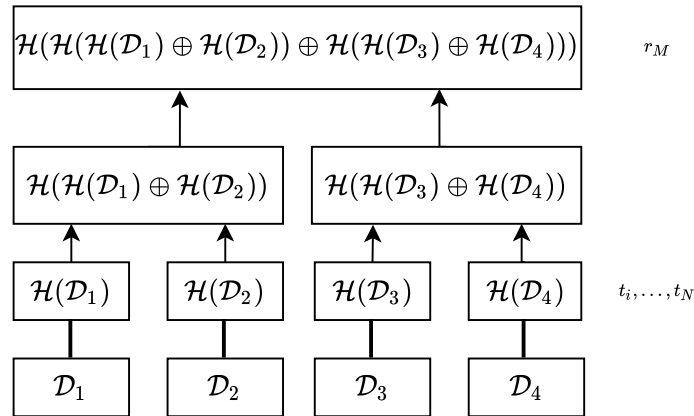


Abbildung 5: Merkle Baum aufgebaut anhand der Beschreibungen der Aufgaben \mathcal{D}

Geht man nun von einer inaktiven Vorlage aus, deren Inhalte man in der gleichen Form erneut veröffentlichen möchte (zum Beispiel bei einem Sonderangebot, was nur in bestimmten Zeiträumen verwendet werden darf), würde dies in dem gleichen Hash $\mathcal{H}_{\mathcal{O}_j}$ resultieren. Eine zufälliger Wert (*nonce*), welcher in der Berechnung des Hashs mitgenutzt wird, verhindert diese Doppeldeutigkeit.

Für eine valide Vorlage definiert das Protokoll folgende Bedingungen:

1. *nonce* ist vorhanden und nicht größer als 4 Byte
2. Für die Anzahl an Aufgaben N gilt $0 < N \leq 255$
3. Anzahl der Elemente in R, T, V sind gleich
4. B ist vorhanden
5. $\forall x \in R$: x entspricht booleschem Wert
6. r_M wurde korrekt aus T berechnet

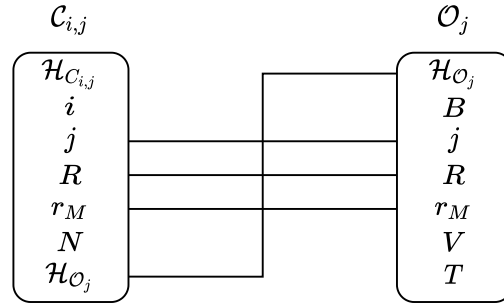


Abbildung 6: Zusammenhang zwischen Vertrag und Vertragsvorlage

5.1.3 Verträge

Wie bereits zuvor erwähnt, leiten sich Verträge jeweils aus einer Vorlage ab und referenzieren somit einige Elemente jener. Ein Vertrag \mathcal{C} zwischen den Parteien i und j ist definiert als:

$$\mathcal{C}_{i,j} = (\mathcal{H}_{C_{i,j}}, \mathcal{K}_{i,P}, \mathcal{K}_{j,P}, N, R, r_M, \mathcal{H}_{O_j}) \quad (12)$$

Hierbei wird i auch als *sender* und j als *receiver* bezeichnet. Eindeutig identifiziert wird jener Vertrag durch seinen Hash $\mathcal{H}_{C_{i,j}}$. Weiterhin enthalten sind die öffentlichen Schlüssel der Parteien, die Anzahl der Aufgaben N in Kombination mit der Zuordnung der Verantwortlichkeit R , eine Referenz auf den Merkle Baum r_M und der Hash der verwendeten Vorlage \mathcal{H}_{O_j} . Der Zusammenhang zwischen Vorlage und Vertrag wird in Abb. 6 erneut verdeutlicht. Jeder Vertrag befindet sich in einem Zustand, welcher implizit durch die Anzahl und Art der Transaktionen, also den Interaktionen mit dem Vertrag, bestimmt ist:

- *offer*: Der Vertrag wurde aus einer gültigen Vorlage erstellt und im Netzwerk veröffentlicht. Die Transaktion der Vertragserstellung durch die Partei i kann als erste Willenserklärung verstanden werden.
- *rejected/live*: Die andere Partei, welche die Vorlage verwaltet, hat den

zuvor veröffentlichten Vertrag abgelehnt/angenommen. Sofern die Partei j den Vertrag annimmt, gibt sie durch die Transaktion die zweite Willenserklärung der Übereinkunft ab.

- *finished*: Alle Aufgaben des Vertrags wurden von den beiden Parteien bearbeitet und bestätigt. Der Vertrag ist somit abgeschlossen.

Für valide Verträge gelten die folgenden Bedingungen, die in Kombination mit einer Vorlage geprüft werden:

1. Die verwendete Vorlage \mathcal{O}_j ist nicht inaktiv und wird korrekt im Vertrag referenziert.
2. r_M, R stimmen mit der Vorlage überein.
3. N entspricht der Länge von R .
4. Die erstellende Partei des Vertrags verwaltet nicht die Vorlage: $\mathcal{K}_{i,P} \neq \mathcal{K}_{j,P}$.

5.1.4 Transaktionen

Transaktionen verändern die Zustände der Objekte (Vorlagen und Verträge) und sind definiert als:

$$\mathcal{T}_i = (\mathcal{H}_{T_i}, \mathcal{S}_i, \mathcal{K}_{i,P}, type, \mathcal{H}_{T_i}) \quad (13)$$

Hier beschreibt \mathcal{H}_{T_i} den Hash der Transaktion, \mathcal{S}_i die digitale Signatur, welche sich mit $\mathcal{K}_{i,P}$ prüfen lässt und \mathcal{H}_{T_i} den Verweis auf die vorherige Transaktion. Die Art der Transaktion ist durch *type* bestimmt. In Abhängigkeit jenes Typs enthält die Transaktion weitere Felder:

- *Template publish*: Bei der Veröffentlichung einer neuen Vertragsvorlage \mathcal{O}_j ist jene entsprechend in der Transaktion enthalten. Dies ist die einzige Art von Transaktion bei der \mathcal{H}_{T_i} leer ist, da die Publikation als

erster Schritt bei jeder Interaktion mit der Vorlage verstanden werden kann.

- *Template revoke*: Um zu verhindern, dass von einem bereits publiziertem Template weitere Verträge abgeleitet werden können, ändert jene Art der Transaktion den Zustand der Vorlage. Die Vorlage wird durch \mathcal{H}_{O_j} adressiert.
- *Contract offer*: Sofern eine aktive Vorlage verwendet wird, um einen neuen Vertrag zu erstellen, wird jener Vertrag $\mathcal{C}_{i,j}$ als Teil der Transaktion versendet. Entsprechend hat nun der Besitzer der Vorlage die Möglichkeit jenen Vertrag anzunehmen oder abzulehnen. Die Transaktion enthält weiterhin einen temporären öffentlichen Schlüssel $\mathcal{K}_{\mathcal{E}_{i,P}}$, welcher später innerhalb dieses Vertrags genutzt wird, um die ausgetauschten Informationen derart zu verschlüsseln, dass jene nur den beiden Parteien zugänglich sind.
- *Contract accept/decline*: Akzeptieren/Ablehnen eines Vertrags, welcher durch $\mathcal{H}_{\mathcal{C}_{i,j}}$ identifiziert wird. Sofern der Vertrag akzeptiert wird, enthält die Transaktion zusätzlich den anderen temporären öffentlichen Schlüssel $\mathcal{K}_{\mathcal{E}_{j,P}}$, was den Schlüsselaustausch abschließt.
- *Confirm task*: Bestätigung eines Vertragsschrittes, sofern sich jener im Zustand *LIVE* befindet. Diese Transaktion enthält den zuvor angesprochenen Beweis, um zu zeigen, dass die zu bestätigende Aufgabe Teil des Merkle Baums r_M ist und in der korrekten Reihenfolge bearbeitet wird. Hierzu müssen lediglich $\log_2(N + p)$ weitere Hash Werte innerhalb der Transaktion übertragen werden. Das Berechnen dieses Beweises dient auf der Seite der Nutzenden als Art Spam-Vermeidung und erlaubt es anderen Netzwerkteilnehmern die jeweilige Transaktion schneller zu validieren, da sie nicht das Template erneut laden müssen.⁶ Wir neh-

⁶Innerhalb der Referenzimplementierung ist jene Prüfung bei $N = 255$ etwa um den Faktor 10 schneller als das Laden der Vorlage.

men an: $N = 4$ und die aktuell zu bestätigende Aufgabe sei $t_{i=2}$. Für das Padding gilt durch N entsprechend $p = 0$. Durch die Struktur des Baumes muss jener Beweis die Hash Werte $\mathcal{H}(\mathcal{D}_1)$, $\mathcal{H}(\mathcal{D}_3 \oplus \mathcal{D}_4)$ und $\mathcal{H}(\mathcal{D}_2)$ enthalten. Da $\mathcal{H}(\mathcal{D}_2)$ $t_{i=2}$, also der zweiten Aufgabe, entspricht, muss jener Hash Wert nicht Teil des Beweises sein, sondern kann in dem Beweis an der richtigen Stelle ersetzt werden. Da Transaktionen immer die jeweilige Aufgabe ansprechen, gilt für den Index des Hash Werts von $t_{i=2}$, da durch das in (11) hinzugefügte Padding:

$$index = i + (N + p) - 2 \quad (14)$$

Somit ergibt sich für den Beweis \mathbf{p} :

$$\mathbf{p} = [nil, nil, \mathcal{H}(\mathcal{D}_3 \oplus \mathcal{D}_4), \mathcal{H}(\mathcal{D}_1), \mathcal{H}(\mathcal{D}_2), nil, nil] \quad (15)$$

Um keine leeren Werte innerhalb dieses Feldes übertragen zu müssen, gilt für die Übertragung innerhalb der Implementierung folgendes Format:

`index;Hash Wert`

Daraus ergibt sich für das obige Beispiel:

<p>2; $\mathcal{H}(\mathcal{D}_3 \oplus \mathcal{D}_4)$ 3; $\mathcal{H}(\mathcal{D}_1)$</p>

Auflistung 1: Übertragungsformat des Beweises

Abbildung 7 verdeutlicht das Vorgehen. Andere Netzwerkteilnehmer können anhand dieser Informationen schnell validieren, ob die angesprochene Aufgabe Teil der Vorlage ist und ob jene zum korrekten Zeitpunkt bearbeitet wird, da i und N Teil des lokalen Zustands sind. Alle Eingabedaten der Nutzenden, enthalten in dem Feld *input_data*, werden verschlüsselt übertragen. Hierzu werden die Schlüssel aus den

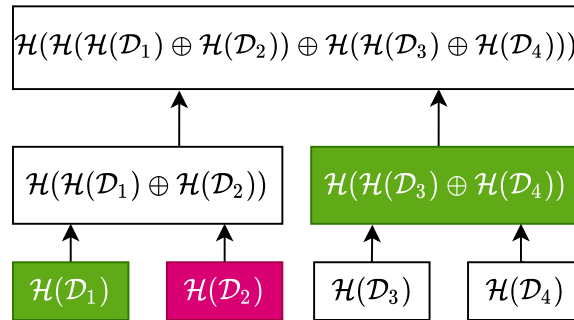


Abbildung 7: Adressierung der zweiten Aufgabe des Vertrags mit zusätzlichen Beweiselementen zur Berechnung von r_M , um die Bestätigung durchzuführen.

Transaktionen „Contract offer“ und „Contract accept“ verwendet, um daraus durch den Diffie-Hellman-Schlüsselaustausch [66] ein gemeinsames Geheimnis abzuleiten, was dann für die jeweilige Transaktion zur Generierung eines Schlüssels durch HKDF [124] [125] benutzt wird, um mit jenem die Eingabedaten symmetrisch zu verschlüsseln.

Abbildung 8 beschreibt den Zusammenhang der zuvor genannten Transaktionen. Endzustände werden hierbei fett dargestellt.

5.2 Dokumentation der Vorlagen und Verträge

Im Folgenden wird die Dokumentation der Verträge und Vorlagen innerhalb des Netzwerks beschrieben.

5.2.1 Aufbau und Struktur der Indizes

Die Zustände der Vorlagen und Verträge werden in der Regel innerhalb eines dezentralen Systems dokumentiert, können jedoch aber auch gleichwohl in zentralen Systemen verwendet werden. Hierzu sei \mathcal{I}_T der Index für Vorlagen (Templates) und \mathcal{I}_C der Index für Verträge (Contracts). Das Trennen der beiden Indizes bezieht sich auf die Anwendung des Protokolls in dezentralen Systemen. Durch die Trennung werden einzelne Knoten schwerer anzugreifen,

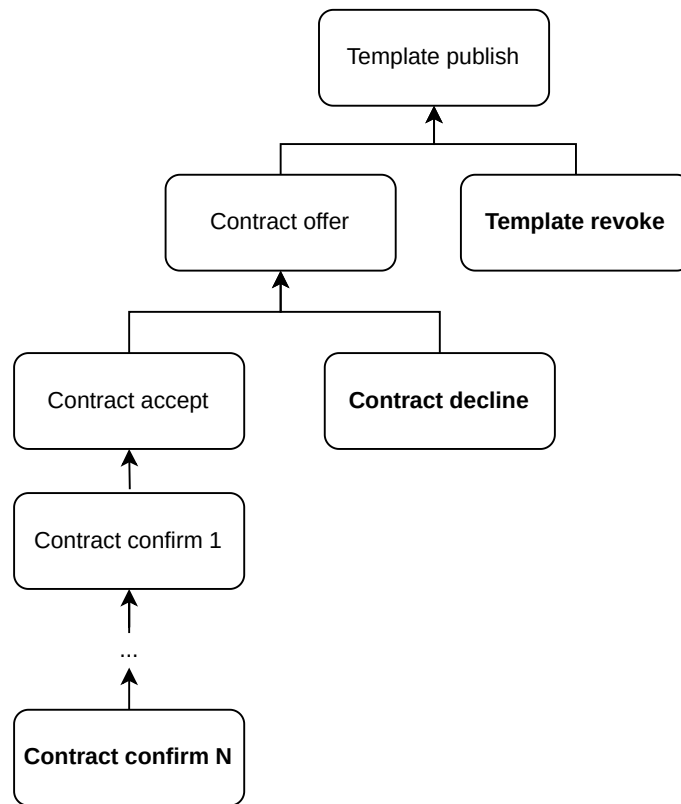


Abbildung 8: Zusammenhang zwischen Transaktionen

sofern eine angreifende Partei die Nutzung einer Vorlage einschränken will. Der Vorlagenindex referenziert lediglich neu erstellte Verträge und benötigt daher im Vergleich zum Vertragsindex, welcher auch die Transaktionen mit Eingabedaten verwaltet, weniger Speicher. Durch die verwendete Hashfunktion wird es daher in einem dezentralen Netzwerk erschwert, einen Knoten gezielt mit der Dokumentation von Verträgen zu belasten, da jene entsprechend im Netzwerk verteilt werden. Sofern innerhalb des Netzwerks nur ein Knoten existiert, übernimmt jener entsprechend die Verwaltung von \mathcal{I}_T und \mathcal{I}_C und ist gleichbedeutend mit einer zentralen Verwendung des Systems. Die Datentypen und Implementierung zu den Indizes werden separat im Kapitel zur Referenzimplementierung beschrieben.

5.2.2 Zuordnung von Objekten zu Indexknoten

Grundlegend bietet sich die Zuordnung von Objekt zu Netzwerkknoten in Form einer Distributed Hash Table an (siehe 3.5), sofern ein dezentrales System verwendet wird. Hierzu müssen die jeweiligen Netzwerkknoten die Zustände der Objekte dokumentieren, Transaktionen prüfen und Nutzenden die Möglichkeit geben, ihre lokalen Zustände zu aktualisieren. Die Netzwerkknoten sind über eine ID identifizierbar. Jene ID ist der Hash-Wert von mehreren Parametern, die den Knoten eindeutig bestimmen:

$$id = \mathbf{SHA-256}(network \oplus ip \oplus port) \quad (16)$$

Hierbei beschreibt *network* den Namen des Netzwerks, *ip* die öffentliche IP-Adresse des Knotens und *port* der hierbei verwendete TCP Port. Entsprechend muss der Netzwerkknoten sicherstellen, auf dieser IP:Port Kombination erreichbar zu sein. Um die allgemeinen Abläufe in Kombination mit den durchgeführten Aktionen im Netzwerk zu veranschaulichen, nehmen wir innerhalb der folgenden Darstellungen die Struktur des Netzwerks als Chord [155] DHT an, welche im Grundlagenkapitel bereits beschrieben wurde und

auch in der Referenzimplementierung verwendet wird. Grundlegend werden die Indizes innerhalb des Netzwerks getrennt. Somit werden Verträge an zum Teil zwei Stellen des Netzwerks referenziert: Während der Vertragserstellung sowohl auf \mathcal{I}_T als auch auf \mathcal{I}_C und bei der Bearbeitung des Vertrags nur noch auf \mathcal{I}_C . Dies ist besonders hilfreich, da es die neu erstellten Verträge entsprechend im Netzwerk verteilt. Verwaltet somit ein Knoten zum Beispiel eine bestimmte Vorlage, kann jener nicht mit etwaigen Verträgen „geflutet“ werden, da dort lediglich Verweise auf den neuen Vertrag verwaltet werden. Die eigentliche Abarbeitung der Verträge, inklusive der größeren Transaktionen beim Bestätigen von Aufgaben, wird dann auf anderen Netzwerkknoten gespeichert.

5.2.3 Aktualisierungen der Indizes

Im Folgenden wird die allgemeine Struktur bei der Aktualisierung der Indizes beschrieben. Die konkreten Anfragen und Antworten in allen Ausprägungen werden im Kapitel der Referenzimplementierung genauer behandelt. Für die Beschreibung der Aktualisierungen nehmen wir an, dass die Anfrage an den korrekten Knoten gerichtet wurde und somit das Routing in der verteilten Hashtabelle bereits stattfand. Allgemein richten sich einzelne Anfragen entweder an den Vorlagenindex \mathcal{I}_T oder an den Vertragsindex \mathcal{I}_C . Bei der Bearbeitung der jeweiligen Anfrage zusätzlich muss der Transaktionstyp unterschieden werden. Weiterhin kann eine korrekte Bearbeitung der Anfrage nur erfolgen, wenn sich auf dem Indexknoten das angesprochene Objekt im gleichen Zustand befindet wie bei der anfragenden Partei. Sollte dies nicht der Fall sein, muss jener Zustand korrekt wiederhergestellt oder über den Indexknoten bezogen werden, bevor die eigentliche Anfrage wiederholt werden kann. Vor der eigentlichen Bearbeitung der Anfrage müssen Indexknoten die folgenden Bedingungen, unabhängig der Anfrage, sicherstellen:

1. Transaktion überschreitet nicht die Größe von 1MB.
2. Erstellende Partei der Transaktion hat jene korrekt signiert.

Vorlagenindex Nachfolgend werden die möglichen Aktualisierungen des Vorlagenindex betrachtet. Hierzu erfolgt eine Betrachtung der Transaktionstypen, die korrekte Zustandsänderungen am Index hervorrufen können.

- *Template publish*: Wenn der Indexknoten eine neue Vorlage verwalten soll, muss hier zunächst geprüft werden, ob die in der Transaktion enthaltene Vorlage korrekt ist. Hierzu prüft der Indexknoten die geltenden Regeln, die bereits in Kapitel 5.1.2 beschrieben wurden. Weiterhin muss der Knoten sicherstellen, dass die sendende Partei der Transaktion mit der empfangenden Partei der Vorlage übereinstimmt und diese Transaktion tatsächlich von der jeweiligen Partei stammt. Hierzu prüft der Indexknoten die digitale Signatur anhand des gegebenen öffentlichen Schlüssels.
- *Template revoke*: Sofern eine Vorlage von einer Partei zurückgezogen wird, muss jenes innerhalb des Index vermerkt werden, damit andere Parteien davon abgehalten werden, neue Verträge zu erstellen und somit die Vorlage zu verwenden. Hierbei muss für die bereits bekannte Vorlage erneut geprüft werden, ob die Transaktion von der berechtigten Partei stammt, welche bereits die Vorlage publiziert hat.
- *Contract offer*: Damit die Partei, die die Vorlage verwaltet, auf neue Verträge aufmerksam werden kann, müssen neue Vertragsangebote entsprechend auch auf dem Vorlagenindex referenziert werden. Hierzu sendet die Partei, die den Vertrag erstellt hat, eine entsprechende Anfrage zur Aktualisierung des Index, welche auf die gültige Vorlage verweist und den Vertrag enthält. Der Indexknoten muss nun zunächst prüfen, ob die genutzte Vorlage noch aktiv ist und danach die Bedingungen

an den Vertrag, die in 5.1.3 beschrieben wurden, prüfen. Zusätzlich stellt der Indexknoten noch sicher, dass der Hash des Vertrags korrekt berechnet und die Transaktion von der Partei korrekt signiert wurde.

- *Contract accept/decline*: Bei der Annahme/dem Ablehnen eines Vertrags kann die Referenz auf dem Vorlagenindex entfernt werden. Hierzu wird durch den Indexknoten sichergestellt, dass diese Änderung nur von der Partei durchgeführt werden kann, die auch die Vorlage erstellt hat. Bei einer korrekten Anfrage löscht der Indexknoten dann die Referenz auf den Vertrag und die Transaktion des Typs *Contract offer*.

Vertragsindex Die notwendigen Operationen zur Verwaltung der Verträge bei ihrer Bearbeitung sind ähnlich zu den zuvor beschriebenen Vorgängen bei der Aktualisierung eines Vorlagenindex und werden nun für die jeweiligen Transaktionstypen eingeführt.

- *Contract offer*: Ein Knoten, welcher den Index für den jeweiligen Vertrag verwaltet, muss nicht zwangsläufig auch den Index für die Vorlage verwalten. Daher muss bei der Erstellung des Vertragsindex innerhalb der Anfrage die Transaktion enthalten sein, welche die Vorlage publiziert hat. Der Indexknoten prüft dann, ob es sich um ein korrektes Template handelt und prüft anschließend, ob der Vertrag die Vorlage korrekt verwendet. Nachfolgend wird noch validiert, dass der Hash des Vertrags korrekt berechnet wurde und die Übereinkunft in den Index des Knotens aufgenommen.
- *Contract accept/decline*: Bei der ersten Interaktion durch die andere Partei muss ein Indexknoten prüfen, ob der angesprochene Vertrag sich im Zustand *OFFER* befindet und somit der Typ der Transaktion korrekt ist. In der Regel kann diese Überprüfung über die Metadaten des Index geprüft werden, wodurch innerhalb der Implementierung der

Vertrag für diese Prüfung noch nicht extra geladen werden muss. Handelt es sich um eine valide Transaktion mit korrektem Typ, muss der ursprüngliche Vertrag aus der jeweiligen Transaktion geladen werden und die neue Transaktion auf jenen angewendet werden. Sofern jene Operation erfolgreich durchgeführt wurde, passt der Indexknoten den Zustand des Vertrags an, aktualisiert die lokalen Metadaten des Objekts und speichert die empfangene Transaktion, so dass jene von den beteiligten Parteien für die Zustandsprüfung der Objekte erfragt werden kann.

- *Confirm task*: Bei der Bestätigung von Aufgaben während der Bearbeitung des Vertrags wird äquivalent vorgegangen wie schon bei der Annahme/dem Ablehnen des Vertrags. Nachdem der Indexknoten den Vertrag in seinem bekannten Zustand geladen hat, prüft jener, ob die Transaktion auf den aktuellen Zustand des Vertrags anwendbar ist und ändert bei einer erfolgreichen Änderung des Objekts den Indexzustand und speichert die Transaktion. Lediglich bei der Anzahl der Aufgaben muss unterschieden werden, ob der nächste Zustand des Vertragsobjekts weiterhin im Zustand *LIVE* oder im Zustand *FINISHED* ist.

5.2.4 Angriffsvektoren auf das Netzwerk

Im Folgenden werden einige Angriffsvektoren auf das Netzwerk und mögliche Abhilfen für zukünftige Versionen des Protokolls und dessen Implementierung vorgestellt. Die vorgestellten Angriffe beziehen sich auf klassische Methoden im Bereich der verteilten Hashtabellen. Weitere Möglichkeiten zur Prävention bzw. Verhinderung solcher Angriffsmöglichkeiten werden im Ausblick zu der Arbeit vorgestellt und dort mit der Referenzimplementierung in Verbindung gebracht.

Sybil Angriff Unter einem Sybil Angriff [70] versteht man die Störung eines Netzwerks durch das Erstellen von vielen falschen Identitäten. Da es in ei-

nem komplett verteilten System (ohne zentrale Stelle oder Einschränkungen) nicht möglich ist sicherstellen zu können, dass eine Identität innerhalb des Netzwerks zu genau einer physikalischen Partei gehört [165].

Innerhalb des Protokolls wäre ein Sybil Angriff durch den *port* Parameter der ID von Netzwerkknoten (16) umsetzbar, da jener prinzipiell frei wählbar ist. Da es sich bei dieser Arbeit um die erste Implementierung des Protokolls handelt, sollten verschiedene Instanzen auf verschiedenen Ports betreibbar sein. Sollten zunehmend Angriffe auf diverse Netzwerk auftreten, wäre eine einfache Abhilfe den Parameter *port* als Teil des Protokolls auf seinen Standardwert 3301 festzulegen. Um dann Sybil Angriffe auszuführen, bedarf es unterschiedlichen IP Adressen der angreifenden Partei, was zu einem Mehraufwand bei einem Angriff auf ein Netzwerk führen würde. Dennoch ist ein solcher Angriff weiterhin problemlos durchführbar, sofern genug Ressourcen zur Verfügung stehen [70]. Ein Sybil Angriff kann daher als Isolation eines gesamten Netzwerks verstanden werden.

Eclipse Angriff Bei einem Eclipse Angriff [165] wird versucht, einzelne Teile des Netzwerks (ggf. einzelne Knoten) zu isolieren. Entsprechend platziert sich die angreifende Partei derart im Netzwerk, dass der anzugreifende Knoten in seinem Routing vollumfänglich gestört wird und dessen Anfragen an durch die angreifende Partei verwaltete Knoten gerichtet werden. Daher hängt der genaue Ablauf und Aufwand eines Eclipse Angriffs von dem verwendeten Routingprotokoll ab. Eclipse Angriffe wurden so zum Beispiel schon für Bitcoin [105] und Ethereum [134] gezeigt. Für die im Protokoll genutzte verteilte Hashtabelle (Chord) sind Eclipse Angriffe durch die Berechnung der ID von Netzwerkknoten unmittelbar schwieriger, jedoch nicht unmöglich, durchzuführen [165]. Die Anfälligkeit für Sybil Angriffe durch das *port* Element der ID würde einen solchen Angriff zusätzlich erleichtern.

Speicherung/Routing Angriffe Bei einem Angriff auf die Speicherung bzw. auf das Routing, handelt es sich um falsches Verhalten von Indexknoten

innerhalb des Netzwerks [165]. Durch die Offenheit der Software und (in öffentlichen Netzwerken) die uneingeschränkte Teilnahme an einem Netzwerk sind solche Angriffe nicht zu verhindern. Eine angreifende Partei könnte daher bewusst falsche Informationen an Clients zurückgeben.

Republizieren eines inkorrekten Zustands Ein möglicher Angriff innerhalb des Protokolls und seiner Implementierung ist das falsche Wiederherstellen eines Indexzustands, wenn jener republiziert werden muss. Nachfolgend soll jenes Szenario kurz beleuchtet werden. Hierzu wird ein Vertrag mit vier Vertragsschritten angenommen. Die Transaktion, welche die Vorlage veröffentlicht hat, sei \mathcal{T}_0 . Die Transaktionen, die den Vertrag erstellen und dessen Aufgaben bestätigen, seien $\mathcal{T}_1 \dots \mathcal{T}_6$. Weiterhin wird von den Parteien A und B ausgegangen. Partei A wird später den Angriff durchführen. Die aktuelle Transaktion sei \mathcal{T}_3 , also die erste Bestätigung einer Aufgabe, die von A durchgeführt, aber von B noch nicht importiert wurde. Somit ergeben sich die lokalen Zustände zu: $A : \mathcal{T}_1 \dots \mathcal{T}_3$ und $B : \mathcal{T}_1 \dots \mathcal{T}_2$. Sollte nun der Indexknoten, welcher die Vorlage verwaltet, z. B. durch einen Stromausfall offline gehen oder ein näherer Knoten das Netzwerk betreten, welcher für den Vertrag zuständig wäre, kann A \mathcal{T}_3 lokal löschen und \mathcal{T}'_3 publizieren. Aktualisiert B nun den lokalen Zustand, wird die neuere Transaktion \mathcal{T}'_3 importiert. Ebenso kann der Indexknoten kein Fehlverhalten von A erkennen. Entsprechend kennt B nur die Inhalte der geänderten Transaktion und nimmt jene als letzten Zustand auf (da es sich aus der Sicht von B um eine korrekte Transaktion handelt). Ein solcher Angriff wäre unmittelbar nicht möglich, wenn die geänderte Transaktion bei dem ursprünglichen Indexknoten landet, da eine Validierung scheitern würde. Weiterhin würde die Änderung (selbst bei dem Republizieren eines falschen Index) bei B auffallen, wenn B bereits \mathcal{T}_3 lokal importiert hat, da sich die Transaktion \mathcal{T}'_3 nicht auf den lokalen Zustand von B anwenden lässt.

Überlastung von Netzwerkknoten Überlastungen einzelner Netzwerkknoten, zum Beispiel durch Denial-of-Service (DoS) Angriffe, können dazu führen, dass einzelne Teile des Netzwerks für Nutzende nicht erreichbar sind und somit Übereinkünfte nicht korrekt bearbeitet werden können. Intensiviert wird ein solches Problem, wenn die Nutzenden nur einzelne Netzwerkknoten kennen und dadurch in Gänze im Zugang zu dem Netzwerk beschnitten werden. Wird der überlastete Knoten durch seine schlechte Erreichbarkeit somit von der verteilten Hashtabelle getrennt und durch die Nutzenden weiterverwendet, teilt sich das Netzwerk auf. Verwenden hierbei die anderen Parteien der Übereinkünfte funktionierende Netzwerkknoten, verzerrt dies die Bearbeitung jener. Abbildung 9 visualisiert das Beschriebene. Alice greift in diesem Fall nur über einen Knoten, welcher angegriffen wird, auf das Netzwerk zu. Durch die entstehenden Verzögerungen bei der Vielzahl von Anfragen können die anderen Netzwerkknoten den betroffenen Knoten nicht mehr erreichen und entfernen jenen aus der verteilten Hashtabelle. Da der angegriffene Knoten die anderen Netzwerkknoten auch nicht erreichen kann, verwaltet jener die Indizes selbst und dokumentiert somit die Anfragen von Alice. Würde Bob in diesem Fall seine Anfragen korrekt im eigentlichen Netzwerk verteilen, entstehen zwei unterschiedliche Sichtweisen, was die Bearbeitung der Übereinkünfte entsprechend einschränkt, da die Transaktionen gegenseitig nicht bezogen werden können.

Einordnung der Angriffsvektoren In offenen Peer-to-Peer Anwendungen auf Basis quelloffener Software/Protokollen lassen sich gewisse Angriffe nicht verhindern. Die möglichen Angriffe müssen daher immer im Kosten/Nutzen Verhältnis betrachtet werden. Die zuvor genannten Angriffe, gerichtet auf ein Netzwerk wie Bitcoin oder Ethereum, in welchem monetäre Transaktionen verarbeitet werden, sind deutlich wahrscheinlicher als ein Angriff auf ein öffentliches Netzwerk, welches das Protokoll dieser Arbeit verwendet. Nicht nur verursacht ein solcher Angriff Kosten bei der angreifenden

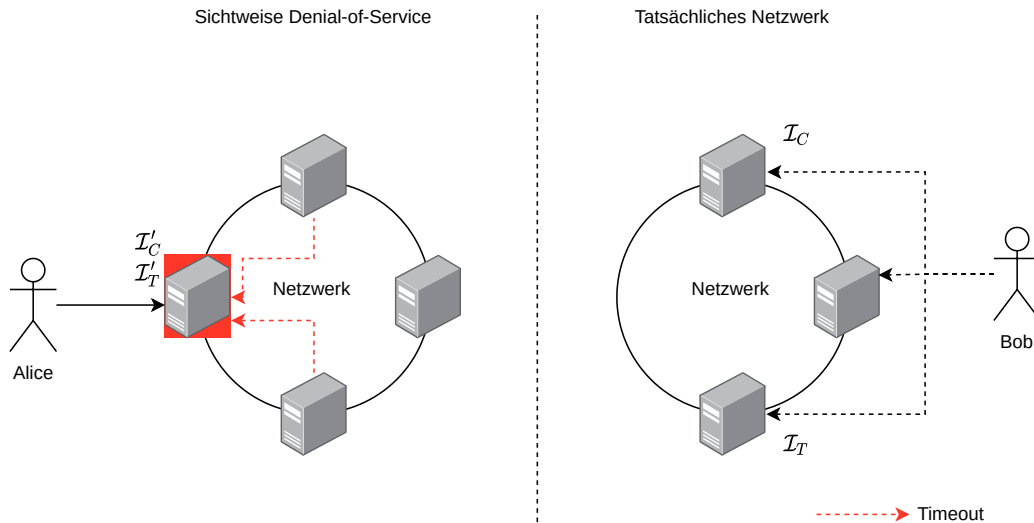


Abbildung 9: Sichtweise bei einem Denial-of-Service Angriff eines Netzwerkknotens

Partei (ggf. mehrere unterschiedliche IP-Adressen zur Platzierung innerhalb des Netzwerks), sondern ist ein etwaiger monetärer Gewinn nur bei Netzwerken möglich, wo tatsächlich digitale Währungen ausgetauscht werden. Durch die Zuordnung der Objekte durch die Indexknoten (siehe 5.2.2) und die Aufteilung in Vorlagen- und Vertragsindex ist es zudem erschwert, konkrete Knoten eines Netzwerks zu isolieren.

5.3 Exemplarischer Ablauf

Um das zuvor Beschriebene anhand eines Beispiels zu veranschaulichen, werden erneut zwei Parteien betrachtet: Alice, die eine Vertragsvorlage erstellt und verwaltet, und Bob, welcher aus jener Vorlage einen Vertrag ableitet. In Anlehnung an die eingeführten Definitionen sei $j = Alice$ und $i = Bob$.

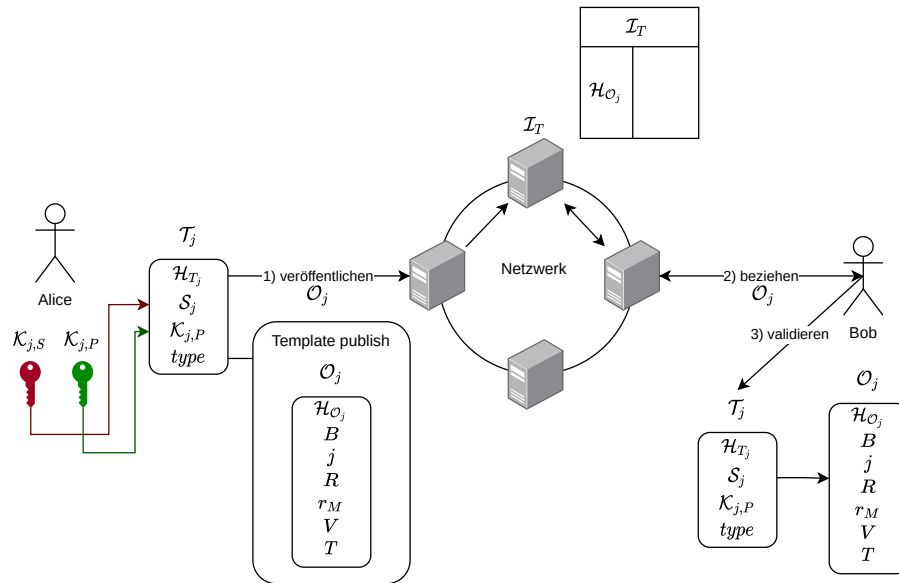


Abbildung 10: Veröffentlichung einer Vorlage und deren Import von einer anderen Partei

5.3.1 Erstellen und Publizieren einer Vorlage

Zunächst definiert Alice die Inhalte der Vorlage, nämlich die durchzuführenden Aufgaben, deren Zuordnung zu der jeweiligen Partei, die dazu gehörigen Validatoren und eine allgemeine Beschreibung. Aus jener Vorlage O_j wird nun eine Transaktion vom *type* „Template publish“ erstellt, kryptographisch mit ihrem privaten Schlüssel signiert und dann im Netzwerk veröffentlicht. Das Netzwerk führt danach die Zuordnung von Netzwerkknoten zum Hash der Vorlage durch und erstellt an jener Stelle den Index für die Vorlage \mathcal{I}_T . Möchte nun Bob jene Vorlage verwenden, wird ein beliebiger Knoten des Netzwerks angefragt, um die entsprechende Transaktion zu beziehen. Jene wird danach lokal validiert, indem nicht nur die Signatur, sondern auch die allgemeine Struktur der Vorlage geprüft wird. Nach erfolgreicher Prüfung wird die Vorlage dann lokal gespeichert und kann verwendet werden. Diese Prüfung wird auch durch den Netzwerkknoten durchgeführt, welcher den

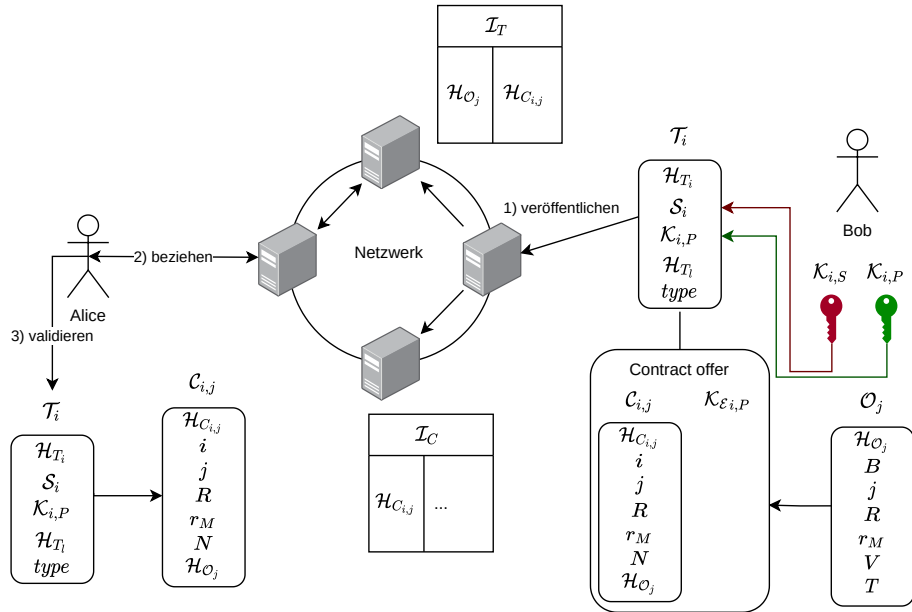


Abbildung 11: Ablauf beim Erstellen und Beziehen eines Vertrags

Index verwaltet. Somit können falsche Transaktionen nicht den Zustand des Index ändern.

Abb. 10 illustriert den allgemeinen Ablauf von der Veröffentlichung bis zum Import einer Vorlage.

5.3.2 Veröffentlichung eines Vertrags

Um nun einen Vertrag zu veröffentlichen, muss Bob ihn aus der Vorlage ableiten und dann eine Transaktion erzeugen, deren *type* „Contract offer“ ist. Das Vorgehen ist äquivalent zu der bereits beschriebenen Veröffentlichung der Vorlage. Die Transaktion enthält neben der Signatur und dem öffentlichen Schlüssel noch den neu erstellten Vertrag und den temporären öffentlichen Schlüssel, um, sofern der Vertrag akzeptiert wird, verschlüsselt kommunizieren zu können. Zusätzlich verweist diese Transaktion auf H_{T_i} , also die letzte Transaktion, die die Vorlage veröffentlicht hat. Bei der Veröffentlichung wird der Vertrag dann nicht nur auf \mathcal{I}_T referenziert, sondern auch der Index \mathcal{I}_C auf

dem zuständigen Knoten erstellt. Da Alice lokal in regelmäßigen Abständen überprüft, ob jemand ihre Vorlage verwendet hat, bezieht sie anschließend die Transaktion über das Netzwerk und importiert den Vertrag bei sich lokal. Auch hier findet erneut innerhalb des Netzwerks und lokal eine Validierung statt. Abb. 11 verdeutlicht die beschriebenen Abläufe. Das Erstellen des Vertrags wird als erste Willenserklärung verstanden. Weiterhin initiiert Bob in unserem Beispiel den Schlüsselaustausch, indem die Transaktion seinen temporären öffentlichen Schlüssel für diese Übereinkunft enthält.

5.3.3 Annehmen des Vertrags

Nachfolgend will Alice den Vertrag von Bob annehmen und durch eine Transaktion sowohl \mathcal{I}_T als auch \mathcal{I}_C ändern. Mit der erfolgreichen Annahme des Vertrags wird jener auf \mathcal{I}_T gelöscht und folgend nur auf \mathcal{I}_C verwaltet. Innerhalb der Transaktion gibt auch Alice ihren temporären öffentlichen Schlüssel an, welche dann bei der inhaltlichen Abarbeitung des Vertrags verwendet wird, um die jeweiligen Eingaben verschlüsseln zu können. Abb. 12 verdeutlicht die Annahme des Vertrags. Um die Übersichtlichkeit der Abbildung zu gewährleisten, werden die Accountschlüssel der Parteien nicht gezeigt. Die gesendeten Transaktion sind dennoch weiterhin mit dem jeweiligen Account der Partei signiert. Die Annahme des Vertrags entspricht der zweiten Willenserklärung und schließt den Schlüsselaustausch der Übereinkunft ab.

5.3.4 Bestätigung von Vertragsschritten

Nachdem der Vertrag nun im Zustand *LIVE* ist, beginnt die inhaltliche Abarbeitung. Für unser Beispiel wird angenommen, dass der erste Vertragsschritt von Bob bestätigt werden soll. Für die Bestätigung dieser Aufgabe muss eine Transaktion an das Netzwerk gesendet werden, die dann von Alice bezogen wird, um ihren lokalen Zustand zu ändern. Bei der Übertragung von Daten werden die zuvor ausgetauschten Schlüssel für die Verschlüsselung von Eingabedaten benutzt, die nur durch die andere Partei gelesen wer-

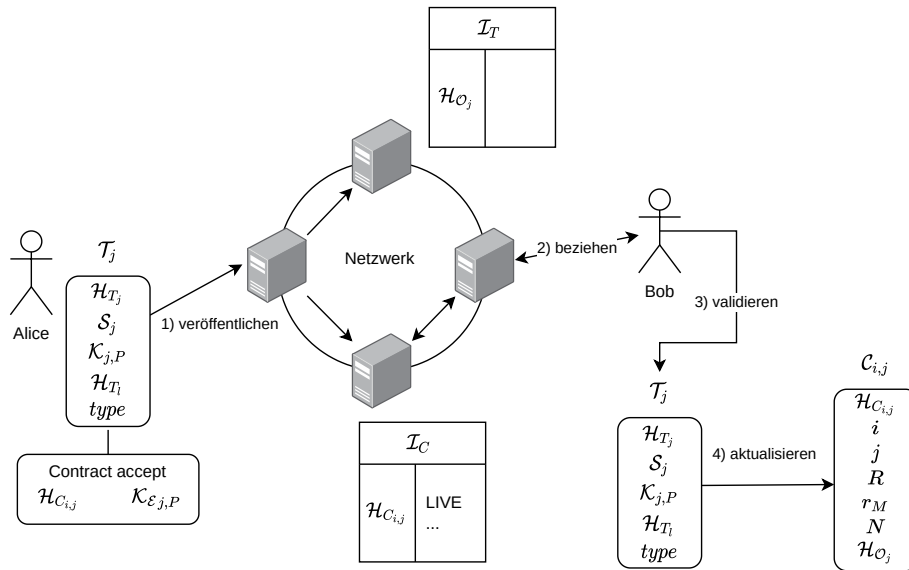


Abbildung 12: Ablauf bei der Annahme des Vertrags

den sollen. Abbildung 13 verdeutlicht das Vorgehen. Auch hier wird der Übersichtlichkeit halber auf die Darstellung der Accountschlüssel verzichtet. Äquivalent werden nachfolgend noch die anderen Aufgaben des Vertrags durch die zuständige Partei bestätigt, bis sich der Zustand nach der letzten Bestätigung zu *FINISHED* ändert.

5.4 Unterschiede zum Stand der Forschung

Nachdem das Protokoll und dessen Abläufe nun vorgestellt wurden, wird jenes mit den verwandten Arbeiten verglichen und eingeordnet.

5.4.1 Smart Contracts

Das Konzept der *Cypher Social Contracts* unterscheidet sich deutlich von dem der Smart Contracts. Bei der Betrachtung der Unterschiede werden hierzu Ethereum Smart Contracts angenommen, welche als „state of the art“ in jenem Feld gelten. Die primären Anwendungsfelder dieser Smart Contracts

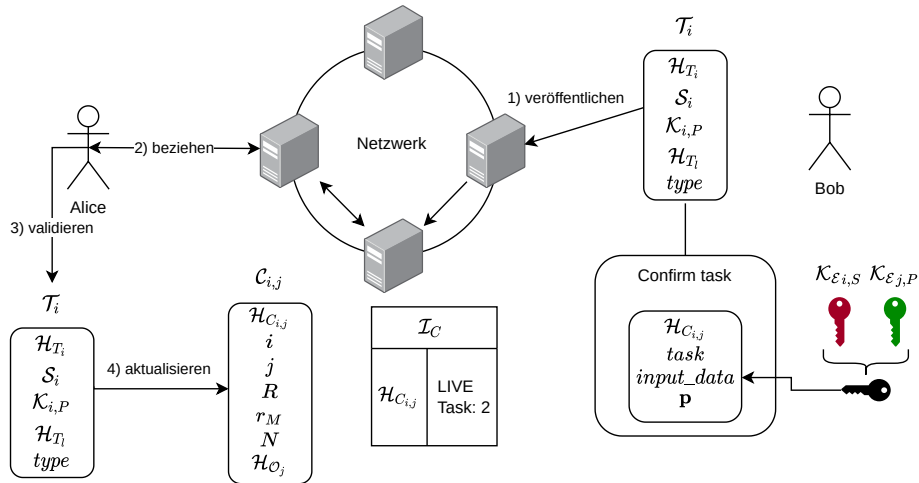


Abbildung 13: Bestätigung eines Vertragsschritts

sind nach [50] finanzieller Natur. Als nicht-monetäre Anwendung werden die dezentrale Verwaltung und Abstimmungen durch Smart Contracts genannt [50]. Dies ist konträr zu der eigentlichen Idee der *Cypher Social Contracts*, die auf einfach zu verstehende, reguläre Übereinkünfte ihren Fokus richten. Finanzielle Interaktionen sind in jenen als Teilschritte zu verstehen, die auf andere Art und Weise abgewickelt werden. Veröffentlichte, nicht-änderbare Smart Contracts werden zudem oft im Kontext „code is law“ genannt. Dies ergibt sich unmittelbar aus der Nicht-Änderbarkeit des Programms in Kombination mit der deterministischen Berechnung des jeweiligen Vertragszustands über das Ethereum Netzwerk. Prozesse der echten Welt sind jedoch komplexer, wodurch die Aussage „code is law“ irreführend wirken kann (vgl. Kapitel 4.3 und 8.1). Ferner ist es erheblich komplexer Ereignisse der realen Welt als Eingaben für bereits veröffentlichte Smart Contracts zu verwenden (Orakel) und das korrekte Verhalten des jeweiligen Programms dauerhaft sicherzustellen. Hierzu sei als Beispiel eine Versicherung für Starkwetterereignis-

se genannt, die Einzahlungen der Versicherungsbeiträge via Smart Contract erlaubt und Auszahlungen anhand Informationen von Orakeln durchführt. Zwar ist eine solche Anwendung aktuell umsetzbar, jedoch kann jedes technische Problem, welches ggf. erst in Jahrzehnten auftritt, zu monetären Verlusten bei den versicherten Personen führen. Hier zu nennen seien Angriffe auf das Orakel oder das Einstellen eines solchen Dientes, was den kompletten Smart Contract nutzlos machen und im schlimmsten Falle die eingezahlten Summen sperren würde. Zwar lässt sich argumentieren, dass man für einen solchen Fall eine Art Notfall innerhalb des Smart Contracts behandeln kann, jedoch widerspricht es der eigentlichen Idee von „code is law“ und einem dezentralen, durch Nutzende kontrollierten Finanzwesen. Ein Smart Contract kann also gewisse Aufgaben der realen Welt im Optimalfall abbilden; kommt es jedoch zu Streitigkeiten durch menschliche Akteure der Übereinkunft, so stoßen Smart Contracts schnell an ihre Grenzen aufgrund ihrer Nicht-Änderbarkeit.

Betrachtet man das genannte Beispiel in der realen Welt, so kann es auch hier zu Problemen kommen. Verweigert die Versicherung eine Zahlung trotz erfüllter Bedingungen der Police, haben jedoch hier Versicherte die Möglichkeit sich mit der anderen Vertragspartei vor Gericht zu streiten, um die Situation aufzulösen. Nutzt man für eine solche Übereinkunft also das in der Arbeit vorgestellte Protokoll zur Dokumentation oder zur Abwicklung des Schadens, sind deren Inhalte und Zuständigkeiten für Menschen ohne Programmiererfahrung zu verstehen, was die einfachere Benutzbarkeit des Protokolls unterstreicht. Weiterhin ist es beiden Parteien möglich, den jeweils eigenen, privaten Zustand der Übereinkunft in einem solchen Verfahren offenzulegen und somit nachzuweisen, welche Aussage von welcher Partei getroffen wurde. Durch die digitalen Signaturen können die jeweiligen Absichten so glaubhaft dargelegt werden.

Die Trennung von Vertrag und Bezahlung bietet weiterhin diverse Vorteile: Zum einen können Prozesse, die keine Zahlung vorsehen, als Übereinkunft

abgebildet und bearbeitet werden, ohne dass die reine Bearbeitung innerhalb eines Netzwerks mit einer Kryptowährung bezahlt werden muss. Zum anderen erlaubt es die Inklusion von Stakeholdern, die Zahlungen mit Kryptowährungen ablehnen, um so diversen Risiken bei jener Nutzung vorzubeugen [127]. Der generische Ansatz des Protokolls erlaubt es einen Zahlungsvorgang arbiträr zu beschreiben. Somit kann z. B. innerhalb eines cyber-physischen Vertrags nach Erhalt einer Ware auch via Überweisung gezahlt werden. Gleichzeitig kann so die Privatsphäre der Nutzenden weiterhin gestärkt werden, da die Kontexte der Zahlungen ggf. nicht mit dem Vertrag in Verbindung gebracht werden können. Werden so Anzahl und Art der Ware innerhalb der Bearbeitung des Vertrags festgelegt und dort die Zahlungsdaten übermittelt, so könnte ein Zahlungsdienst zwar die Zahlung zwischen den Parteien wahrnehmen, jedoch nicht genau wissen welche Waren gekauft wurden. In Zeiten von personenbezogener Werbung und Tracking würde dieses Vorgehen also die Privatsphäre der kaufenden Partei stärken.

Ein weiterer Unterschied ist die durch das Protokoll mögliche Teilautomatisierung, die im Kontext der Referenzimplementierung im Kapitel der Automatisierung technisch behandelt wird. Wird so ein cyber-physischer Vertrag automatisiert, geschieht das auf dem eigenen System über das man die volle Kontrolle besitzt. Der Programmcode muss aufgrund der Entkopplung einer digitalen Währung somit nicht von einem ganzen Netzwerk ausgeführt werden, wodurch es auch möglich ist, private Bibliotheken, zum Beispiel in Unternehmenskontexten, für die Bearbeitung dieser Verträge zu nutzen, ohne sensible Daten innerhalb eines öffentlichen Netzwerks offenlegen zu müssen. Die Übereinkünfte können dort als verstehbare Schicht zwischen zwei Akteuren verstanden werden, deren Systeme jeweils aber nicht direkt miteinander verbunden sind. Dies ist insbesondere interessant bei der stetigen Zunahme an Supply-Chain-Angriffen, wo angreifende Parteien Zugänge zu Systemen erhalten, indem sie das schwächste Glied der Lieferkette angreifen und von dort aus Angriffe auf die anderen Systeme initiieren. Durch die Teilautoma-

tisierung ist weiterhin die Änder- und Wartbarkeit der eigenen Software gegeben: Hierbei kann die eigene Implementierung der Automatisierung zum Beispiel Stück für Stück erweitert oder auch Fehler korrigiert werden. In absoluten Notfällen könnten die Übereinkünfte sogar manuell bearbeitet werden. Abschließend greift Tabelle 2 die von Nick Szabo in [157] definierten Ziele an einen Smart Contract auf und stellt Ethereum Smart Contracts (SC) den *Cypher Social Contracts* (CSC) gegenüber.

Ziel	SC	CSC	Bemerkung
Beobachtbarkeit	✓	✓	Etwaige Vertragsbrüche können innerhalb der <i>Cypher Social Contracts</i> lediglich von den zwei involvierten Parteien erkannt werden. Bei Ethereum Smart Contracts kann die Interaktion von jedem mitgelesen werden, da jede Interaktion mit einem Smart Contract auf der öffentlichen Blockchain stattfindet.
Überprüfbarkeit	(✓)	✓	Die Unterscheidung zwischen Vertragsverletzungen und gutgläubigen Fehlern wird in Ethereum durch den Smart Contract selbst bestimmt. Ein Schiedsrichter hätte hier keinen Einfluss auf die allgemeine Übereinkunft. Innerhalb der <i>Cypher Social Contracts</i> könnte ein Schiedsrichter/Gericht nach der Abarbeitung noch über die Korrektheit entscheiden und ggf. Änderungen, welche bei Smart Contracts ohne eine Hintertür nicht durchsetzbar wären, anordnen.

Vertraulichkeit	(X)	✓	Innerhalb von Smart Contracts in Ethereum kann jeder die Interaktionen mit einem Smart Contract nachvollziehen. Bei <i>Cypher Social Contracts</i> haben nur die beiden Parteien Zugriff auf die verschlüsselten Daten. Bei jeder Übereinkunft werden weiterhin neue Schlüsselpaare generiert, was bedeutet, dass, sofern eine Übereinkunft einem Schiedsrichter/Gericht offengelegt wird, jene Instanz keinen Zugriff auf alle vorherigen oder nachfolgenden Übereinkünfte besitzt.
-----------------	-----	---	--

Durchsetzbarkeit	(✓)	(✓)	Die Ausführung eines Smart Contracts ist nicht rückgängig zu machen, was bedeutet, dass genau der entwickelte Code auch dezentral ausgeführt wird. Die Durchsetzbarkeit durch ein Gericht bei etwaigen Fehlern ist hiermit nicht gegeben, da jenes keine Kontrolle über die verteilte Blockchain besitzt. Innerhalb der <i>Cypher Social Contracts</i> wird menschliches Vertrauen benötigt, da jenes die Grundlage für die Interaktion zweier Parteien ist. Sofern eine Partei eine definierte Aktion in der realen Welt nicht durchführt, kann die Übereinkunft und deren Aktionen, z. B. in einem Gericht, offengelegt und auf dieser Basis Urteile gefällt werden, wodurch die Durchsetzbarkeit gegeben ist und direkter monetärer Verlust (durch die Aktionen eines Smart Contract) ausgeschlossen werden kann.
------------------	-----	-----	--

Tabelle 2: Vergleich Ethereum Smart Contract/*Cypher Social Contract* anhand der Ziele aus [157]

5.4.2 Ricardian Contracts

Ricardian Contracts weisen im Vergleich zu Smart Contracts wegen ihrem hohen semantischen Anteil eine höhere Ähnlichkeit zu *Cypher Social Contracts* auf. Obwohl Ricardian Contracts ihren Fokus auf Lesbarkeit des Vertrags für Menschen (in Kombination mit maschineller Lesbarkeit) legen, wurde jene

Idee dennoch als Teil von Finanzinstrumenten entwickelt (siehe Kapitel 4) und daher schon zu Beginn an Zahlungen gekoppelt. Vereinfacht lässt sich sagen, dass Ricardian Contracts versuchen rechtsgültige Verträge in Papierform zu übersetzen und somit eine Art Digitalisierung durchzuführen. Da das Konzept an sich recht generisch ist, wurden Ricardian Contracts in den unterschiedlichsten Domänen verwendet, ohne dass sich ein gewisser Standard bei ihrer Erstellung abzeichnete. Zum Beispiel verwenden die in Kapitel 4 vorgestellten Systeme unterschiedliche Komponenten, um einen Vertrag zu beschreiben und setzen zudem auf unterschiedliche Serialisierungsverfahren - sie sind somit nicht kompatibel. Das *Cypher Social Contracts* Protokoll grenzt sich daher von Ricardian Contracts derart ab, dass die Nutzung primär auf die Kommunikation zweier Parteien abzielt. Hierbei kann es sich um die Kommunikation zwischen Maschinen, eine menschliche Interaktion zur Kommunikation oder um die konkrete Bearbeitung eines Vertrags im Rechtssinn handeln. Für jede beliebige Art der inhaltlichen Ausgestaltung der Kommunikation greift das gleiche mentale Modell der Übereinkunft, was die allgemeinen Abläufe klar verständlich macht. Zusätzlich werden Ricardian Contracts von einer Partei ausgestellt und nachfolgend von anderen digital signiert (wachsendes Dokument), während sich bei *Cypher Social Contracts* der Zustand einer Übereinkunft durch die Interaktion mit Transaktion implizit ergibt. Die Unveränderlichkeit der Transaktionsobjekte garantiert somit ein klares Verständnis über Abläufe, Zustandsänderungen und Historien.

Weiterhin anzumerken ist die Beziehung zwischen Vorlagen und Verträgen innerhalb des Protokolls. Während es bei den vorgestellten Systemen mit Ricardian Contracts notwendig ist, für jede Übereinkunft einen neuen Vertrag zu erstellen und zu signieren, kann innerhalb des Protokolls dieser Arbeit die hierzu genutzte Vorlage wiederverwendet werden, um aus jener einen Vertrag abzuleiten. Dies trennt nicht nur die zwei Willenserklärungen der Übereinkunft klar ab, sondern erlaubt es z. B. Dienstleistenden auch ihre Vorlage nur einmal zu veröffentlichen, was sich allgemein positiv auf die Do-

kumentation und Verwaltung etwaiger Vertragsangebote auswirkt.

5.5 Zusammenfassung

Das vorgestellte Protokoll eignet sich, um generische Übereinkünfte nahezu jeder Art abzubilden und erlaubt es Nutzenden, sich so über das Protokoll sicher auszutauschen. Über die lokal erstellten Accounts verfügen Nutzende über die volle Kontrolle ihrer digitalen Identitäten innerhalb des Systems und hängen nicht von anderen Parteien ab. Vorlagen, deren Hash zum Beispiel über andere Webseiten bezogen wurde, können lokal importiert und begutachtet werden, bevor eine möglicherweise ungewollte Handlung in einem zentralen System ausgelöst werden kann (dark pattern).

Der allgemeine Ablauf einer Übereinkunft kann die Willenserklärungen der involvierten Parteien sicher abbilden: Sofern die Inhalte der Vorlage für beide Parteien eindeutig sind, stellen die Transaktionen „Contract offer“ und „Contract accept“ jeweils die Willenserklärung dar, welche durch die digitale Signatur der jeweiligen Transaktion gesichert ist. Sollten die Kernaspekte der Übereinkunft erst während der Bearbeitung der Aufgaben festgelegt werden, so können die Willenserklärungen separat als Bestätigung von Vertragsschritten abgegeben werden. Im letzten Fall beziehen sich die zuvor genannten Transaktionen („Contract offer/accept“) dann auf die Willenserklärung zur Kommunikation.

Dokumentiert werden die Übereinkünfte auf Indizes für Verträge und Vorlagen, die durch Transaktionen aktualisiert werden. In dezentralen Systemen findet die Zuordnung zwischen Netzwerkknotten und Index über den Hash des Objekts statt, in zentralen Systemen kann ein einzelner Knotten verwendet, oder eine separate Implementierung zur Dokumentation der Übereinkünfte realisiert werden, die sich zum Beispiel an der Referenzimplementierung des Protokolls orientiert.

Insgesamt richtet sich das Protokoll an einfache, generische Übereinkünfte, die nicht nur zwischen Menschen, sondern auch in rein cyber-physischen Sys-

tem stattfinden können. Die Verständlichkeit des Protokolls (im Vergleich zu Smart Contracts) resultiert in einem einfachen mentalen Modell einer Übereinkunft, das auch von Laien ohne Programmiererfahrung nachvollzogen werden kann.

Zudem kann das Protokoll auf die eigenen Bedürfnisse angepasst werden, sofern die Kernabläufe gleich bleiben. So limitiert das hier vorgestellte Protokoll die Aufgabenanzahl einer Vorlage, um später innerhalb der Implementierung effizienter mit LoRaWAN kompatibel zu sein. Da es sich bei der LoRaWAN Nutzung um einen Edge-Case handelt, könnte die Übertragung des Protokolls auch eine höhere Anzahl an Aufgaben pro Vorlage vorsehen und hier einige Anpassungen (z. B. bei der Validierung der Transaktionen) vornehmen. Diese Änderungen wirken sich jedoch nicht auf die Sicherheit bei der Nutzung oder die rechtliche Betrachtung aus, sofern weiterhin die Abläufe nach Abbildung 8 gegeben sind.

6 Referenzimplementierung Fides

Nachfolgend wird die Referenzimplementierung des zuvor beschriebenen *Cypher Social Contracts* Protokolls vorgestellt. Die in Python 3 geschriebene Implementierung erlaubt die Verwendung des Protokolls in eigenen Programmen bzw. die reine Nutzung als Software über eine Kommandozeilenanwendung, die Basis für die Entwicklung einfach zu bedienender grafischer Anwendungen sein kann. Die Software lässt sich mit dem Ausführen eines einzelnen Befehls installieren und ist kompatibel mit den gängigsten Betriebssystemen (Linux, MacOS, Windows).

6.1 Allgemeines

Das Projekt wurde nach der römischen Personifikation für Vertrauen, Ehre und der Einhaltung von Wort und Eid Fides benannt [147]. Der Name soll die Idee verdeutlichen und unterstreichen, dass Vertrauen die Grundlage bei menschlicher Interaktion ist. Jenes Vertrauen darin, dass eine Aktion durchgeführt wird, kann nicht durch Smart Contracts forciert werden, was bereits in Kapitel 4 beschrieben wurde.

Der Name wurde bereits in anderen Projekten verwendet, so auch in einer Optimierungs-Bibliothek, die fast zeitgleich mit der Implementierung dieser Arbeit entstanden ist und den Namen auf dem offiziellen Python Package Index (PyPi) verwendet [161]. Die erste Veröffentlichung dort ist datiert auf November 2020 [77].

Die Resultate dieser Arbeit wurden mit der Einreichung des Papers [61] im Juni 2021 unter der MIT Lizenz [144] quelloffen verfügbar gemacht [129]. Jene Arbeit ist die unmittelbare Weiterentwicklung der Ideen und Implementierungen aus [57], die die allgemeine Idee und das Konzept der Anwendung vorgestellt hat und schon zum Teil den Namen Fides verwendete. Wegen der

schieren Größe des Projekts⁷ wurde sich gegen eine Umbenennung entschieden. Zusätzlich profitiert hier die Sicherheit der Nutzenden, da das Projekt im Allgemeinen weniger anfällig für Supply-Chain Angriffe wird, sofern man es nur aus seinem offiziellen Repository [129] beziehen kann [48].

6.2 Konfiguration

Fides erlaubt die Konfiguration der kompletten Anwendung innerhalb einer Konfigurationsdatei. Teile dieser Konfiguration können durch Umgebungsvariablen überschrieben werden. Im Folgenden werden die Möglichkeiten der Konfiguration beleuchtet, um den folgenden Kapiteln zu der Referenzimplementierung einfacher folgen zu können. Innerhalb der Implementierung können die Parameter der Konfiguration über das Modul *fides.core.config* bezogen werden. Für Abstraktionen, die Fides für andere Anwendungsbereiche aufbereiten, gilt die Konvention *fides.abst.{Name der Abstraktion}.core.config*. Im Allgemeinen geben die jeweiligen Konfigurationsmodule Standardwerte vor, welche verwendet werden, sofern die Nutzenden keine eigenen Werte festlegen.

6.2.1 Netzwerk

Konfigurationen, die sich auf die Betriebsart von Fides und auf das Netzwerk beziehen, werden in der Konfigurationsdatei innerhalb der Sektion *Networkd* festgelegt. Folgende Konfigurationsparameter sind vorhanden:

- **Mode:** Legt die Art und Weise der Fides-Instanz fest. Mögliche Ausprägungen sind *REGULAR* (einfacher Client, der sich mit einem Netzwerk verbindet), *FULL* (Netzwerkknoten, der eingehende Verbindungen akzeptiert und Objekte dokumentiert) und *LORA* (Nutzende der LoRaWAN Abstraktion).

⁷Das erste Commit (c1dbdabcc88a2439fbff0cc9db567eed4e3ad983) der öffentlichen Version von Fides umfasst 91 Dateien mit insgesamt 10253 Zeilen an Code/Konfiguration.

- **Network:** Der Name des Netzwerks.
- **Interface:** Das zu nutzende Interface bei Netzwerkknoten (die öffentliche IP Adresse). Kann durch die Umgebungsvariable *FIDES_NETWORK_INTERFACE* überschrieben werden.
- **Port:** TCP-Port, auf dem Verbindungen akzeptiert werden (bei Netzwerkknoten). Kann durch die Umgebungsvariable *FIDES_NETWORK_PORT* überschrieben werden.
- **Refresh:** Intervall in Sekunden, in welchem auf neue Zustände von Verträgen/Vorlagen geprüft wird.
- **IndexDrop:** Wartezeit in Sekunden bis ein Objekt auf dem Netzwerkknoten gelöscht wurde, wenn es nicht aktualisiert wird.
- **MaxTries:** Anzahl an Versuchen, die unternommen werden, um einen Netzwerkknoten zu kontaktieren.
- **UseImportedNodes:** Flag, um auch ggf. importierte Netzwerkknoten (nicht nur manuell hinzugefügte) zu verwenden.
- **Cert:** Zertifikat der ID des Netzwerkknotens (bei **Mode FULL**)
- **ClientCert:** Zertifikat des aktuell genutzten Accounts
- **CertNode:** Public key des Accounts, welcher die Zertifikate in privaten Netzwerken ausstellen darf.
- **CheckClientCert:** Flag, welches in privaten Netzwerken festlegt, ob nur Netzwerkknoten oder auch Nutzende Zertifikate brauchen, um am Netzwerk teilnehmen zu können.

6.2.2 Logging

- **Level:** Das zu nutzende Loglevel. Kann durch die Umgebungsvariable *FIDES_LOG_LEVEL* überschrieben werden.
- **File:** Flag, welches angibt, dass die Ausgaben des aktuellen Loggers auch in eine Datei geschrieben werden sollen.
- **FileSize:** Dateigröße in MB pro Logdatei

6.2.3 Lokale Kommunikation

Die lokale Kommunikation kann innerhalb der Sektion *Fidesd* konfiguriert werden. Hierbei ist es möglich, im Element **Port** den TCP-Port der lokalen Kommunikation zu wählen.

6.2.4 Abstraktionen

Abstraktionen setzen ihre Konfigurationsparameter in dem jeweiligen Modul. Für die LoRaWAN Abstraktion sei hier auf Kapitel 7.3 verwiesen.

6.3 Datenmodelle

Bei der Beschreibung des Datenmodells muss zwischen der lokalen Repräsentation der Objekte und deren Repräsentation bei ihrer Übertragung unterschieden werden. Grundlegend werden innerhalb der Referenzimplementierung sowohl objektorientierte als auch prozedurale Ansätze verfolgt, welche später in 6.4 beschrieben werden.

6.3.1 Datenbankmodelle

Innerhalb der Referenzimplementierung wird sqlite3 [90] als Standarddatenbank zur Speicherung der Objekte verwendet. Sqlite3 eignet sich besonders gut für die gewählte Programmiersprache Python, da die Bibliothek bereits

Teil der Standardbibliothek der Programmiersprache ist und somit nicht separat installiert und konfiguriert werden muss. Bei der Installation von Fides müssen so lediglich die Datenbankschemata angelegt werden.

Grundsätzlich spiegeln die Datenbanken die jeweiligen Python Klassen zur Speicherung der Objekte und orientieren sich an den genutzten Datentypen. Hierbei werden die Objekte selbst in serialisierter Form und ergänzende Informationen (z. B. der Zustand) separat gespeichert. Komplexe Fremdschlüsselbeziehungen zwischen den einzelnen Datenbanken existieren nicht. Entsprechend wird an dieser Stelle auf die Visualisierung der Datenbankschema verzichtet und auf die in 6.3.3 beschriebenen Datentypen verwiesen.

6.3.2 Serialisierung

Serialisierung beschreibt die Transformation von Daten, zum Beispiel Objekte einer Klasse, in ein einheitliches Übertragungsformat. Die Serialisierung der Objekte innerhalb der Implementierung soll hier kurz beleuchtet werden, da die genutzte Bibliothek `protobuf` [65] in Teilen unklar operiert. [122] beschreibt eine Zusammenfassung möglicher Probleme bei der deterministischen Serialisierung mittels `protobuf`:

- Deterministische Serialisierung ist nicht kanonisch zwischen verschiedenen Programmiersprachen. Weiterhin gilt die Serialisierung zudem als nicht zwangsläufig stabil zwischen verschiedenen Versionen mit Schemaänderungen bei unbekanntem Feldern.
- Die Reihenfolge des Datentyps `Map` innerhalb eines `protobuf` Schemas ist undefiniert. Elemente einer `Map` können somit nicht deterministisch angeordnet sein.
- Die Felder eines `protobuf` Objekts können in beliebiger Reihenfolge angegeben werden (nummeriert), was zu Problemen bei unterschiedlichen Parsern führen kann.

- *Packed repeated fields* können durch ihre Verkettung (bei mehreren pro Element einer protobuf Nachricht) zu nicht deterministischen Resultaten führen.

Daher könnte es, in Abhängigkeit von zukünftigen Versionen und/oder Designentscheidungen innerhalb von protobuf, bei späteren Versionen von Fides zu Problemen bei der Kompatibilität kommen, insbesondere, wenn die Anwendung für andere, auch von protobuf unterstützte, Programmiersprachen entwickelt wird. Primär bezieht sich das Problem auf die Art und Weise, wie Transaktionen signiert und validiert werden (siehe 6.4.3). Die Referenzimplementierung verwendet das serialisierte protobuf Objekt als Eingabe für die Signatur, da somit alle Elemente des Objekts für die jeweilige Signatur betrachtet werden, um etwaige Fehler und die Manipulation von Objekten ausschließen zu können. Zusätzlich können alle Transaktionen verschiedener Typen auf die gleiche Art und Weise signiert werden. Würde es hier bei der prüfenden Partei zu einem anderen Resultat bei der Serialisierung der kopierten Nachricht kommen, würde die Signaturprüfung entsprechend fehlschlagen. Allgemein verwenden die Datenmodelle von Fides keine der oben genannten problematischen Datentypen. Da es nicht seriös abschätzbar ist, wann bzw. ob ein solcher Fall überhaupt eintritt, soll nachfolgend dennoch eine etwaige Lösung beschrieben werden. Eine spätere Version von Fides müsste lediglich eigene Regeln für die Reihenfolge und die Repräsentation der jeweiligen Teilobjekte (zum Beispiel der Hash eines Vertrags) machen, welche dann uniform als Eingabe zur Erstellung der Signatur genutzt wird. Das Übertragungsformat (Serialisierung durch protobuf) wäre dadurch egal, da die Prüfung auf dem resultierenden Objekt durchgeführt wird, dessen Teilobjekte gleich sind. Innerhalb der Tests und dem Betrieb von Fides konnten keine Probleme bei der Serialisierung festgestellt werden. Hierbei wurden neben diversen Linux Distributionen (Ubuntu, Fedora, Raspbian) auch die Kombination mit Windows 10 und MacOS (Big Sur, Monterey) und mit diversen, zum Teil unterschiedliche, Versionen von Python (3.6-3.10) und der

genutzten protobuf Bibliothek (ab Version 3.17.3) untersucht.

6.3.3 Datentypen

Im Folgenden werden die Kerndatentypen, welche in der Regel in einfachen SQLite Datenbanken [90] gespeichert werden, vorgestellt. Zusätzlich wird, wenn vorhanden, das jeweilige Gegenstück bei der protobuf Serialisierung erläutert. Hierbei geht die Beschreibung nur auf die Objekte der Klassen ein, die Methoden werden im Kapitel der API/Kernfunktionen (siehe 6.4.3) genauer beleuchtet.

Accounts Grundlegend handelt es sich bei Accounts um einfache Textdateien im json-Format, die folgendermaßen aufgebaut sind:

```
{
  "id": "CP9hgZImo0lXn9OUWHrM1QqDRMN1jOMjHrJjr7JlbyM
    =",
  "name": "main",
  "description": "Hauptaccount",
  "encrypted": true,
  "key_data": "LS0t[...]tLQo="
}
```

Auflistung 2: Dateistruktur eines Accounts

Hierbei entspricht das Feld *id* dem öffentlichen Schlüssel $\mathcal{K}_{i,P}$, *name* dem Namen des Accounts \mathcal{A}_i bzw. dem Namen der Datei, die auf dem System angelegt wurde, und *description* einer optionalen Beschreibung, um den Account ggf. besser zuordnen zu können. Das private Element des Accounts $\mathcal{K}_{i,S}$ ist in dem Feld *key_data* gegeben. Sofern jener Schlüssel mit einem Passwort geschützt wurde, wird dies innerhalb des Feldes *encrypted* als boolescher Wert angegeben. Das Passwort zu den Accounts \mathcal{A} kann von den Nutzenden frei gewählt werden. Die temporären Accounts $\mathcal{A}_{\mathcal{E}}$ verwenden als Passwort die

Validator	Konstruktorformat
PLAINTEXT	
REGEX	Regulärer Ausdruck
RANGE	[Datentyp];[untere Grenze];[obere Grenze]
SIG	Öffentlicher Schlüssel
MULTLSIG	[Öffentlicher Schlüssel 1];[...];[öffentlicher Schlüssel N]
SHA-256	SHA-256 Hash

Tabelle 3: Konstruktorformat für Validatoren

base64 encodierte Signatur des Vertragshashes durch den primären Account \mathcal{A} , wodurch sich die temporären Accounts, die immer verschlüsselt erzeugt werden, entsperren lassen.

Das Account Module wurde vollständig prozedural implementiert. Daher existiert keine separate Klasse für Accounts und jene werden auch nicht serialisiert über das Netzwerk übertragen. Das Modul verwendet die Objekte der Bibliothek *cryptography* [160] somit ohne weitere Abstraktion.

Vorlagen Implementierungen zu den Vorlagen verwenden die Klasse *Template*. Tabelle 4 gibt eine Übersicht über die Elemente der Klasse und deren Serialisierung. Der direkt verwendete protobuf Datentyp *Validator* ergibt sich aus seinem Typ (Enumeration *VALIDATOR_TYPE*) und dem Feld *constructor*, welches die Parameter des Validators festlegt. Tabelle 3 beschreibt das Format des Felds *constructor* für die vorhandenen Validatoren.

Verträge Verträge sind innerhalb der Klasse *Contract* implementiert. Die Elemente der Klasse sind in Tabelle 5 beschrieben.

Netzwerkknoten Netzwerkknoten setzen sich aus ihrer ID, der öffentlichen IP-Adresse, dem genutzten Port und dem jeweiligen Netzwerknamen zusammen. Die entsprechenden Elemente werden in einer einfachen sqlite3 Datenbank gespeichert. Zusätzlich wird noch ein Flag *imported* dokumentiert, wel-

Element	Python	protobuf	Bemerkung
j	receiver (str)	receiver (string)	Erstellende Partei der Vorlage
T	tasks (list[str])	tasks (repeated string)	Aufgaben der Vorlage
R	responsibility (list[bool])	tasks (repeated bool)	Zuständigkeiten
V	validators (list[Validator])	tasks (repeated Validator)	Validatoren der Aufgaben. Direkte Verwendung des protobuf Objekts.
r_M	merkle_top (str)	merkle_top (string)	Root hash des Merkle Baums
B	description (str)	description (string)	Beschreibung der Vorlage
\mathcal{H}_{O_i}	hash (str)	hash (string)	Hash der Vorlage
	nonce (int [4 Byte big endian])	nonce (fixed32)	Nonce der Vorlage
	tx_ref (str)		Hash der Transaktion, mit welcher die Vorlage veröffentlicht wurde
	state (str)		Zustand der Vorlage
	_imported (bool)		Flag, das anzeigt, ob die Vorlage importiert wurde
	_published (bool)		Flag, das anzeigt, ob die Vorlage aus dieser Instanz veröffentlicht wurde
	storage (Storage)		Objekt der Klasse <i>Storage</i> zur Verwaltung der lokalen Datenbank
	logger (Logger)		Objekt der Klasse <i>Logger</i>

Tabelle 4: Datentyp *Template*

Element	Python	protobuf	Bemerkung
$\mathcal{H}_{c_{i,j}}$	hash (str)	hash (string)	Hash des Vertrags
i	sender (str)	sender (string)	Erstellende Partei des Vertrags
j	receiver (str)	receiver (string)	Erstellende Partei der Vorlage
N	num_tasks (int)	num_tasks (uint32)	Anzahl an Aufgaben der Vorlage
r_M	merkle_top (str)	merkle_top (string)	Root hash des Merkle Baums
\mathcal{H}_{o_j}	template (str)	template (string)	Hash der Vorlage
R	responsibility (list[bool])	tasks (repeated bool)	Zuständigkeiten
	nonce (int [4 Byte big endian])	nonce (fixed32)	Nonce des Vertrags
	state (str)		Zustand des Vertrags
	tx_ref (str)		Hash der Transaktion, mit welcher der Vertrag veröffentlicht wurde
	last_tx (str)		Hash der Transaktion, die den aktuellen Zustand des Vertrags verändert hat
	current_task (int)		Aktuelle, zu bearbeitende Aufgabe
	_published (bool)		Flag, das anzeigt, ob der Vertrag aus dieser Instanz veröffentlicht wurde
	storage (Storage)		Objekt der Klasse <i>Storage</i> zur Verwaltung der lokalen Datenbank
	logger (Logger)		Objekt der Klasse <i>Logger</i>

Tabelle 5: Datentyp *Contract*

Tabelle	Inhalte	Beschreibung
Metadata	Template, state, owner, transaction	Notwendige Informationen über eine Vorlage. Enthält den aktuellen Zustand der Vorlage, die Transaktion, welche jene Vorlage publiziert hat, und den öffentlichen Schlüssel der Partei, die die Vorlage veröffentlicht hat.
Templates	Template hash, contract hash, contract transaction	Verbindung zwischen Vertrag und der verwendeten Vorlage. Zusätzlich enthält jeder Eintrag der Tabelle den Hash der Transaktion, welche den Vertrag publiziert hat, der die jeweilige Vorlage verwendet.
Transactions	Hash, data	Transaktionen, die die Zustandsänderungen auslösten und von Nutzenden bezogen werden. Hier werden sowohl Vorlagen- als auch Vertragstransaktionen gespeichert.

Tabelle 6: Vorlagenindex \mathcal{I}_T

ches beschreibt, ob der Knoten manuell hinzugefügt wurde oder beim Durchsuchen des Netzwerks importiert wurde.

Index Innerhalb der Indexkomponente wird zwischen Vertragsindex \mathcal{I}_C und Vorlagenindex \mathcal{I}_T unterschieden. Die Indizes werden in getrennten Datenbanken verwaltet:

Vorlagenindex

Die Datenbank von \mathcal{I}_T setzt sich aus den Tabellen *Metadata*, *Templates* und *Transactions* zusammen, deren Elemente in Tabelle 6 beschrieben sind.

Vertragsindex

Der Vertragsindex \mathcal{I}_C besteht aus den Tabellen *Contracts* und *Transactions*, deren Elemente in Tabelle 7 beschrieben sind.

Tabelle	Inhalte	Beschreibung
Contracts	Contract hash, state, last_tx, transaction, cur- rent_task	Benötigte Informationen, um den Zustand eines Vertrags zu verwalten. Indexknoten speichern die Verträge nicht in Fides-Instanzen und bauen die Objekte somit aus dem aktuellen Zustand dieser Tabelle auf, bevor neue Transaktionen, welche den Zustand des Objekts verändern, auf diesen temporären Zustand angewendet werden.
Transactions	Hash, data	Transaktionsdaten, die mit den Verträgen zusammenhängen.

Tabelle 7: Vertragsindex \mathcal{I}_C

Transaktionen Transaktionen nutzen ähnlich zu Validatoren auch den direkten protobuf Datentyp, welcher sich aus folgenden Elementen aus Tabelle 8 zusammensetzt. Durch das Element *type* ist die Art der Transaktion bestimmt. Hierbei wird innerhalb von protobuf ein *oneof* Objekt verwendet, welches aus einer zuvor definierten Menge an möglichen Nachrichten stammen kann. Tabelle 9 beschreibt die möglichen Nachrichten und deren zusätzlichen Felder.

Element	protobuf	Bemerkung
\mathcal{H}_{T_i}	hash (string)	Hash der Transaktion
\mathcal{S}_i	signature (string)	Signatur der Transaktion
$\mathcal{K}_{i,P}$	sender (string)	Öffentlicher Schlüssel der Partei, die die Transaktion erstellt und signiert hat
	timestamp (string)	Zeit, zu der die Transaktion signiert wurde
\mathcal{H}_{T_i}	prev_trans (string)	Verweis auf die vorherige Transaktion
<i>type</i>	type (oneof)	Typ der Transaktion

Tabelle 8: Datentyp *Transaction* (protobuf)

Typ	Weitere Felder	Bemerkung
Confirm task	$\mathcal{H}_{c_{i,j}}$ (string), t (string), \mathbf{p} (repeated string), $input_data$ (string)	Addressierung des Vertrags und der jeweiligen Aufgabe durch den Hash. Zusätzlich enthalten sind der passende Beweis und ggf. verschlüsselte Eingabedaten.
Contract offer	$\mathcal{C}_{i,j}$ (Contract), $\mathcal{K}_{\mathcal{E}_{i,P}}$ (string)	Vertragsobjekt und temporärer öffentlicher Schlüssel der Partei i .
Contract accept	$\mathcal{H}_{c_{i,j}}$ (string), $\mathcal{K}_{\mathcal{E}_{j,P}}$ (string)	Addressierung des anzunehmenden Vertrags durch dessen Hash. Weiterhin enthalten ist der temporäre öffentliche Schlüssel von Partei j .
Contract decline	$\mathcal{H}_{c_{i,j}}$ (string)	Addressierung des abzulehnenden Vertrags durch dessen Hash.
Template publish	\mathcal{O}_j (Template)	Vorlage
Template revoke	$\mathcal{H}_{\mathcal{O}_j}$ (string)	Zurückziehen einer Vorlage, adressiert durch deren Hash.

Tabelle 9: Transaktionstypen als Teil eines *oneof* Objekts innerhalb von Transaktionen (protobuf)

protobuf	Bemerkung
component (string)	Komponente, von wo die Lognachricht stammt
msg (string)	Aufbereitete Lognachricht
tx (string)	Transaktion, welche die Änderung hervorgerufen hat
timestamp (string)	Zeitpunkt der Lognachricht
network (string)	Fides Netzwerk

Tabelle 10: Datentyp *LogEntry* (protobuf)

Logs Lognachrichten sind in zwei Kategorien aufgeteilt:

1. Logs, die unmittelbar aus der Referenzimplementierung stammen und Informationen über die Anwendung liefern.
2. Logs, die für die Nutzenden aufbereitet über die Kommandozeilenanwendung bereitgestellt werden.

Die erste Kategorie verwendet einfache Textdateien, um die jeweiligen Nachrichten des Standard Python Logging Moduls zu speichern. Die zweite Kategorie verwendet zur Speicherung eine sqlite3 Tabelle, welche neben der serialisierten Lognachricht (*LogEntry*) noch die ID des Eintrags und ein Flag ob die Nachricht bereits gesichtet wurde, speichert. Lognachrichten der zweiten Kategorie verwenden ebenso protobuf zu Serialisierung. Tabelle 10 beschreibt den Aufbau der Lognachrichten.

6.3.4 Fides Instanzen

Innerhalb von Fides ist es möglich, beliebig viele lokale Instanzen zu verwenden. Hierbei ist eine Instanz definiert als der Pfad zu einem Ordner (in der Regel benannt *.fides*), welcher wie folgt aufgebaut ist:

```
.fides/
|-- abst
|   |-- lora
|-- account
```

```
| |-- ephemeral
|-- contracts
|-- endpoints
|-- hooks
|-- index
|-- log
|-- templates
|-- transactions
```

Nachfolgend werden die einzelnen Unterordner und deren Inhalte genauer beschrieben. Für die inhaltliche Einordnung der Komponenten sei auf die Kernfunktionen (siehe 6.4.3) verwiesen.

- **abst**: Daten der Abstraktionen. Zum aktuellen Zeitpunkt existiert nur die LoRaWAN Abstraktion. Die Inhalte folgen der Grundstruktur von Fides und sind genauer in 7.3 beschrieben.
- **account**: Ordner für die Accounts der Instanz. Weiterhin enthalten sind die symbolische Verknüpfung für den Hauptaccount, die Tabelle für etwaige Accountaliasse und innerhalb des Ordners *ephemeral* jeweils in Unterordnern die temporären Schlüssel für die Vertragsabarbeitung.
- **contracts**: Datenbanken für die Verträge und Vertragsaliasse.
- **endpoints**: Datenbank für die Netzwerkknoten.
- **hooks**: Exportierte Hook Dateien. Bei der Ausführung weiterer Dateien, die auf die Prozess ID (PID) hinweisen, oder Sperren, damit Hooks nicht doppelt ausgeführt werden.
- **index**: Datenbanken für die Vorlagen und Verträge für Netzwerkknoten.
- **log**: Logdateien, aufgesplittet in vier Dateien, deren Größe pro Datei in der Konfiguration festgelegt werden kann. Weiterhin eine Datenbank,

die Zustandsänderungen des Systems dokumentiert und über das Command Line Interface angezeigt wird.

- **templates:** Datenbanken für die Vorlagen und Vorlagenaliasse.
- **transactions:** Datenbank für die Transaktionen, die die Zustandsänderungen auslösten.

Weiterhin enthält der *.fides* Ordner die Datei *fides.config*, die die gesamten Konfigurationsparameter der Anwendung steuert.

Die Anwendung versucht die Hauptinstanz in *\$HOME/.fides* zu verwenden, die jedoch mittels eine Umgebungsvariable gesteuert werden kann (siehe 6.2).

6.4 API

Die Folgenden Beschreibungen beziehen sich auf die gemachten Implementierungen und darauf, wie die API im Allgemeinen strukturiert wurde. Jene API wird unter anderem von dem Kommandozeilenwerkzeug, welches später eingeführt wird, verwendet und kann direkt von Entwicklerinnen und Entwicklern benutzt werden, um Fides in die eigenen Anwendungen zu integrieren. Die Implementierungen sind wie folgt aufgebaut:

```
fides
|-- abst
|   |-- lora
|       |-- cli
|       |-- core
|           |   |-- network
|           |   |-- types
|           |-- utils
|-- cli
|-- core
```

```
|  |-- network
|  |  |-- chord
|  |-- types
|-- external
|  |-- grond
|      |-- [...]
|-- hooks
|-- utils
```

6.4.1 Abstraktionen

Innerhalb des Moduls *fides.abst* sollen Implementierungen gebündelt werden, die das System abstrahieren und es somit in anderen Domänen nutzbar machen. Allgemein sollen die Abstraktionen von der Ordnerstruktur äquivalent zu der Hauptanwendung aufgebaut sein, wie in der Übersicht zu sehen ist. Die aktuell einzige Abstraktion nutzt LoRaWAN [11], um in Regionen ohne Internet mittels jener Funktechnologie auch an Verträgen teilnehmen zu können. Die LoRaWAN Abstraktion, welche in [58] veröffentlicht wurde, wird genauer in Kapitel 7 beschrieben.

6.4.2 Kommandozeilenanwendung

Implementierungen, die die Kommandozeilenanwendung realisieren, sind im Modul *fides.cli* enthalten. Die verfügbaren Kommandos sowie allgemeine Designentscheidungen bei der Entwicklung der Anwendung sind in Kapitel 6.5 beschrieben.

6.4.3 Kernfunktionen

Die Kernfunktionen von Fides sind innerhalb des Moduls *fides.core* zu finden und werden nachfolgend beleuchtet.

Accounts Die in 5 eingeführten Accounts verwenden innerhalb der Referenzimplementierung die elliptische Kurve Curve25519 [45]. Hierbei verwenden reguläre Accounts \mathcal{A} das Edwards-Signaturschema [46] und temporäre Accounts \mathcal{A}_E X25519⁸ [45], das System zum Diffie-Hellman-Schlüsselaustausch. Alle Implementierungen, die sich auf die Kernfunktionen der Accounts beziehen, sind in *fides.core.account* zu finden. Nachfolgend sollen die Funktionen des Moduls und deren Ein- und Ausgaben kurz beschrieben werden:

- **change_password**

Beschreibung: Passwort eines Accounts ändern

Eingaben: Pfad zum Account, altes Passwort, neues Passwort

Ausgabe: -

- **_opener**

Beschreibung: Funktion zum Anlegen der Accountdateien mit den korrekten Berechtigungen (siehe 6.5.1)

Eingaben: -

Ausgabe: File Descriptor [79] zum Anlegen von Accountdateien mit den korrekten Berechtigungen

- **generate_key**

Beschreibung: Erstellen eines neuen Accounts

Eingaben: Name des Accounts, optionale Beschreibung, Passwort, Speicherpfad, Flag zur Unterscheidung von Ed25519/X25519 [45] [46]

Ausgabe: Pfad zum erstellten Account

- **get_public_key**

Beschreibung: Öffentlichen Schlüssel abrufen

⁸Die Benennung bezieht sich hierbei auf den Erfinder der elliptischen Kurve, der X25519 für den Diffie-Hellman-Schlüsselaustausch, Ed25519 für das Edwards-Signaturschema und Curve25519 zur allgemeinen Beschreibung der Kurve verwendet: https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5R0Rux30o_oDDRksU/ (Abgerufen im Februar 2023)

Eingabe: Geheimer Schlüssel

Ausgabe: Öffentlicher Schlüssel zu dem übergebenen geheimen Schlüssel

- **load_key**

Beschreibung: Privaten Schlüssel des Accounts laden

Eingaben: Pfad zum Account, Passwort

Ausgabe: Geladener privater Schlüssel, sofern das Passwort korrekt ist

- **sign**

Beschreibung: Daten mit privatem Schlüssel des Accounts signieren

Eingaben: Privater Schlüssel, zu signierende Daten

Ausgabe: Signatur der Eingabedaten

- **verify**

Beschreibung: Signierte Daten anhand des öffentlichen Schlüssels überprüfen

Eingaben: Öffentlicher Schlüssel, Eingabedaten, Signatur

Ausgabe: *True*, sofern Signatur korrekt ist, ansonsten *False*

- **construct_public_key**

Beschreibung: Öffentlicher Schlüssel (Objekt) aus String-Repräsentation erstellen

Eingaben: String Repräsentation des öffentlichen Schlüssels (base64), Flag zur Unterscheidung von X25519/Ed25519 [45] [46]

Ausgabe: Öffentlicher Schlüssel

- **create_shared_secret**

Beschreibung: Generierung eines gemeinsamen Schlüssels für die symmetrische Verschlüsselung von Eingaben

Eingaben: Geheimer Schlüssel, anderer öffentlicher Schlüssel, *salt* und Informationen

Ausgabe: Abgeleitetes Geheimnis durch HKDF [126] nach Diffie-Hellman-Schlüsselaustausch [66]

- **dump_private_key_raw**

Beschreibung: Private Schlüssel des Accounts extrahieren

Eingabe: Geheimer Schlüssel

Ausgabe: Geheimer Schlüssel (unverschlüsselt) im PEM [130] Format

- **load_private_key_raw**

Beschreibung: Extrahierten privaten Schlüssel eines Accounts laden

Eingabe: Geheimer Schlüssel im PEM [130] Format

Ausgabe: Geheimer Schlüssel

Konfiguration Die in 6.2 beschriebenen Konfigurationsparameter werden von dem Modul *fides.core.config* verwaltet, von wo sie von den anderen Modulen importiert werden. Hierzu wird die in Python enthaltene Komponente *ConfigParser* [83] verwendet, um die vorliegende Konfigurationsdatei, die vom Format her INI Dateien von Microsoft [133] ähnelt, zu lesen und die entsprechenden Werte zu übernehmen. Innerhalb des Moduls wird zusätzlich geprüft, ob die Umgebungsvariable *FIDES_PATH* gesetzt wurde. Sofern dies der Fall ist, wird an jenem Ort die Fides Instanz innerhalb des Ordners *.fides* verwendet. Der Standardpfad der Anwendung ist für Linux/MacOS⁹: „*\$HOME/.fides*“ und innerhalb von Windows „*%USERPROFILE%/.fides*“. Die korrekte Bestimmung des Pfads wird während der Installation durch die in Python enthaltene Funktion *expanduser* [81] sichergestellt.

Vorlagen Vorlagen werden in der Klasse *Template* innerhalb des Moduls *fides.core* implementiert und können von dort verwendet werden. Nachfolgend werden die Funktionen des Moduls und deren Ein- und Ausgaben kurz beschrieben:

- **__init__**

⁹Die Umgebungsvariable *\$HOME* existiert in beiden System. In der Regel wird jene innerhalb von Linux-Distributionen zu „*/home/{Nutzername}/*“ und bei MacOS zu „*/Users/{Nutzername}/*“ aufgelöst.

Beschreibung: Konstruktor der Klasse

Eingaben: -

Ausgabe: Template Objekt

- **finalize**

Beschreibung: Finalisiert ein Template, berechnet dessen Hash und prüft das finale Objekt.

Eingabe: Privater Schlüssel

Ausgabe: Boolescher Wert

- **db_load**

Beschreibung: Lädt ein Template aus der lokalen Datenbank

Eingabe: Hash der Vorlage

Ausgabe: Boolescher Wert

- **save**

Beschreibung: Speichert das Objekt in der lokalen Datenbank

Eingabe: -

Ausgabe: -

- **validate**

Beschreibung: Validiert, ob eine Transaktion die Vorlage erfolgreich zurückziehen kann.

Eingabe: Transaktion vom Typ *Template revoke*

Ausgabe: Boolescher Wert

- **serial_load**

Beschreibung: Laden eines Template von serialisierten Daten

Eingaben: Serialisiertes Template, Flag, ob es sich um ein Protobuf Objekt handelt oder um die String-Repräsentation des Protobuf Objekts

Ausgabe: Boolescher Wert

- **serialize**

Beschreibung: Konvertierung eines Template Objekts zu einem Proto-

buf Objekt

Eingaben: -

Ausgabe: Protobuf Objekt der Vorlage

- **check**

Beschreibung: Prüft allgemeine Bedingungen an die Vorlage.

Eingaben: -

Ausgabe: Boolescher Wert

- **print**

Beschreibung: Ausgabe des Template Objekts

Eingaben: -

Ausgabe: Informationen zur Vorlage (via STDOUT)

- **_get_rpc_stub**

Beschreibung: Interne Funktion, um eine Verbindung zum lokalen Kommunikations-Daemon herzustellen

Eingabe: -

Ausgabe: Verbindungsobjekt

- **_publish_to_fidesd**

Beschreibung: Veröffentlichung einer Transaktion an den lokalen Kommunikations-Daemon, welcher die Transaktion an das Netzwerk weiterleitet

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **publish**

Beschreibung: Funktion zur Veröffentlichung einer Vorlage

Eingabe: Privater Schlüssel zu dem in der Vorlage verwendeten Account

Ausgabe: Boolescher Wert

- **revoke**

Beschreibung: Zurückziehen einer bisher aktiven Vorlage

Eingabe: Privater Schlüssel zu dem Account, welcher die Vorlage veröffentlicht hat

Ausgabe: Boolescher Wert

- **tx_import**

Beschreibung: Import eines Templates aus einer Transaktion des Typs *Template publish*

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **delete**

Beschreibung: Löschen eines Templates aus der lokalen Datenbank

Eingabe: -

Ausgabe: -

Verträge Verträge sind in der Klasse *Contract* implementiert und wie schon die Vorlagen im Modul *fides.core* enthalten. Innerhalb der Klasse sind folgende Funktionen implementiert:

- **__init__**

Beschreibung: Konstruktor der Klasse

Eingaben: -

Ausgabe: Contract Objekt

- **print**

Beschreibung: Ausgabe des Contract Objekts

Eingabe: -

Ausgabe: Informationen zum Vertrag (via STDOUT)

- **create_eph_keypair**

Beschreibung: Temporären Account für den jeweiligen Vertrag erstellen

Eingabe: Privater Schlüssel, welcher je nach Partei entweder den Ver-

trag oder die Vorlage erstellt hat

Ausgabe: Tupel aus temporärem, privatem und öffentlichem Schlüssel

- **load_eph_keypair**

Beschreibung: Laden der temporären Schlüssel des Vertrags

Eingabe: Privater Schlüssel zum Zugriff auf den temporären privaten Schlüssel

Ausgabe: Tupel (privater, temporärer Schlüssel, None), sofern Vertrag im Zustand *OFFER*, sonst Tupel aus privatem, temporärem Schlüssel und öffentlichem, temporären Schlüssel der anderen Partei

- **db_load**

Beschreibung: Lädt einen Vertrag aus der lokalen Datenbank

Eingabe: Hash des Vertrags

Ausgabe: Boolescher Wert

- **save**

Beschreibung: Speichert das Objekt in der lokalen Datenbank

Eingabe: -

Ausgabe: -

- **serial_load**

Beschreibung: Laden eines Vertrags von serialisierten Daten

Eingaben: Serialisierter Vertrag, Flag, ob es sich um ein Protobuf Objekt handelt oder um die String-Repräsentation des Protobuf Objekts

Ausgabe: Boolescher Wert

- **create**

Beschreibung: Neuen Vertrag aus bekannter Vorlage erstellen

Eingabe: Hash der Vorlage

Ausgabe: Boolescher Wert

- **finalize**

Beschreibung: Finalisiert einen Vertrag, berechnet dessen Hash und

prüft das finale Objekt. Zusätzlich werden die temporären Schlüssel generiert.

Eingaben: Privater Schlüssel, Nonce (optional)

Ausgabe: Boolescher Wert

- **check**

Beschreibung: Prüft die allgemeinen Bedingungen an den Vertrag in Kombination mit der zu nutzenden Vorlage.

Eingabe: Template Objekt

Ausgabe: Boolescher Wert

- **tx_import**

Beschreibung: Import eines Vertrags aus einer Transaktion des Typs *Contract offer*

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **_get_rpc_stub**

Beschreibung: Interne Funktion, um eine Verbindung zum lokalen Kommunikations-Daemon herzustellen

Eingabe: -

Ausgabe: Verbindungsobjekt

- **_publish_to_fidesd**

Beschreibung: Veröffentlichung einer Transaktion an den lokalen Kommunikations-Daemon, welcher die Transaktion an das Netzwerk weiterleitet

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **publish**

Beschreibung: Funktion zur Veröffentlichung eines Vertrags

Eingabe: Privater Schlüssel zu dem in dem Vertrag verwendeten Account

Ausgabe: Boolescher Wert

- **serialize**

Beschreibung: Konvertierung eines Contract Objekts zu einem Protobuf Objekt

Eingabe: -

Ausgabe: Protobuf Objekt des Vertrags

- **validate**

Beschreibung: Validiert, ob eine Transaktion den Zustand des Vertrags ändern kann, zum Beispiel bei der Bestätigung einer Aufgabe des Vertrags.

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **update**

Beschreibung: Lokalen Vertrag anhand einer Transaktion aktualisieren. Hierbei wird zunächst die Funktion **validate** verwendet.

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **_responsible**

Beschreibung: Interne Funktion zur Überprüfung, ob der angegebene öffentliche Schlüssel für die aktuelle Aufgabe des Vertrags zuständig ist.

Eingabe: Öffentlicher Schlüssel

Ausgabe: Boolescher Wert

- **confirm**

Beschreibung: Bestätigung der aktuellen Aufgabe des Vertrags

Eingaben: Privater Schlüssel, zu verschlüsselnde Eingabedaten

Ausgabe: Boolescher Wert

- **accept**

Beschreibung: Einen Vertrag im Zustand *OFFER* annehmen

Eingabe: Privater Schlüssel, der die Vorlage verwaltet, die im Vertrag verwendet wird

Ausgabe: Boolescher Wert

- **decline**

Beschreibung: Einen Vertrag im Zustand *OFFER* ablehnen

Eingabe: Privater Schlüssel, der die Vorlage verwaltet, die im Vertrag verwendet wird

Ausgabe: Boolescher Wert

- **export**

Beschreibung: Vertragsinhalte exportieren

Eingabe: Privater Schlüssel

Ausgabe: String, der allgemeine Statusinformationen zum Vertrag, alle notwendigen Transaktionen der Übereinkunft samt unverschlüsselten Eingabedaten und den privaten, temporären Schlüssel des Vertrags enthält.

- **delete**

Beschreibung: Vertrag und dazugehörige Schlüssel lokal löschen

Eingabe: -

Ausgabe: -

- **get_tx_data**

Beschreibung: Entschlüsselung und Validierung von verschlüsselten Eingabedaten bei der Bestätigung von Aufgaben des Vertrags

Eingaben: Hash der Transaktion, Privater Schlüssel

Ausgabe: Tupel (unverschlüsselte Daten, boolescher Wert zur Validierung) bei erfolgreicher Operation, Tupel (None, None) sonst.

Verschlüsselung Für die Verschlüsselung werden innerhalb von der Referenzimplementierung die temporären Accounts eines Vertrags verwendet, die bei dem Erstellen und der Annahme des Vertrag ausgetauscht werden. Hierbei wird ein Diffie-Hellman-Schlüsselaustausch durchgeführt (siehe 3.2) und dieses gemeinsame Geheimnis innerhalb von HKDF (siehe 3.4) verwendet. Das Resultat von HKDF wird dann verwendet, um die eingegebenen Daten symmetrisch mit AES im Modus CBC (siehe 3.1) zu verschlüsseln. Der genaue Ablauf der Verschlüsselung und deren Praktikabilität wird in Kapitel 6.7 im Kontext der Referenzimplementierung erneut behandelt. Die Funktionen zur Ver- und Entschlüsselung werden von den Kernkomponenten innerhalb von Fides verwendet und sind im Modul *fides.core.encryption* implementiert, welches die folgenden Methoden enthält:

- **encrypt**

Beschreibung: Verschlüsselt die Eingabedaten mittels AES-CBC. Etwaiges Padding der Eingabedaten wird vor der Verschlüsselung durchgeführt.

Eingaben: Schlüssel, Eingabedaten

Ausgaben: Zufällig generierter Initialisierungsvektor, verschlüsselter Text

- **decrypt**

Beschreibung: Entschlüsselt zuvor verschlüsselte Daten durch Verwendung des gleichen Schlüssels und entfernt etwaiges Padding.

Eingaben: Schlüssel, Initialisierungsvektor, verschlüsselte Daten

Ausgabe: Unverschlüsselte Eingabedaten

Merkle Bäume Die Implementierungen zu den in der Vorlage und Verträgen verwendeten Merkle Bäumen sind innerhalb des Moduls *fides.core._merkle* implementiert, welches nicht direkt verwendet werden soll, sondern lediglich

durch die anderen Kernkomponenten benutzt wird. Das Modul besteht aus den nun folgenden Funktionen:

- **hash**
Beschreibung: Helfer-Funktion zur Berechnung eines SHA-256 Hashs
Eingabe: Daten
Ausgabe: SHA-256 Hash der Eingabedaten (hex Format)
- **tree_from_tasks**
Beschreibung: Berechnet den Merkle Baum für die Aufgaben einer Vorlage.
Eingaben: Liste an Aufgaben, Flag, ob der gesamte Baum ausgegeben werden soll oder nur r_M
Ausgabe: Liste des Merkle Baums oder erstes Element der Liste
- **get_minimum_padding**
Beschreibung: Berechnet die Anzahl „leerer Aufgaben“, um einen balancierten Baum zu erstellen.
Eingabe: Anzahl der Aufgaben
Ausgabe: Anzahl der zusätzlich benötigten „leeren Aufgaben“
- **is_in_tree**
Beschreibung: Prüft, ob die angegebene Aufgabe in Kombination mit dem Beweis Teil des Merkle Baums ist.
Eingaben: r_M , Aufgabe (Hash), Beweis \mathbf{p} , aktuelle Aufgabe (Integer), Anzahl der Aufgaben
Ausgabe: Boolescher Wert
- **generate_proof**
Beschreibung: Berechnet den notwendigen Beweis zur Bestätigung der aktuellen Aufgabe.
Eingaben: Liste an Aufgaben, aktuelle Aufgabe (Integer)
Ausgabe: Beweis \mathbf{p}

- **generate_proof_path**

Beschreibung: Berechnet den Pfad innerhalb des Merkle Baums, welcher den Beweis darstellt.

Eingaben: Liste an Aufgaben, aktuelle Aufgabe (Integer)

Ausgabe: Liste von booleschen Werten, die beschreiben, welche Elemente innerhalb des Beweises adressiert werden müssen

Speicherung Die Kernobjekte des Systems werden von der zentralen Komponente *fides.core.storage* geladen, gespeichert und gelöscht. Hierbei werden innerhalb des Moduls Verträge, Vorlagen, Transaktionen und Informationen zu den Netzwerkknoten innerhalb der jeweiligen Datenbank (siehe 6.3) verwaltet.

Aufgabenspezifische Implementierungen Zur Validierung des Inhalts von Transaktionen bei der Verwendung von Validatoren existiert innerhalb der Referenzimplementierung das Modul *fides.core.task_impl*, welches die Funktionsweisen der in 5.1.2 eingeführten Validatoren umsetzt. Die Funktionen des Moduls werden von den Kernkomponenten benutzt und müssen nicht zwangsläufig von Entwicklerinnen und Entwicklern verwendet werden. Als Beispiel sei hier die Funktion **get_tx_data** des Moduls *fides.core.contract* genannt. Das Modul enthält die folgenden Funktionen:

- **validate_task**

Beschreibung: Globale Funktion zur Validierung einer unverschlüsselten Eingabe, die die internen Funktionen aufruft. Bei Validatoren des Typs *Plaintext* ist der Rückgabewert immer *True*.

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

- **_validate_sha256**

Beschreibung: Interne Funktion zur Prüfung des Validators des Typs *SHA-256*

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

- **_validate_multi_sig**

Beschreibung: Interne Funktion zur Prüfung des Validators des Typs

Multiple signature

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

- **_validate_sig**

Beschreibung: Interne Funktion zur Prüfung des Validators des Typs

Signature

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

- **_validate_regex**

Beschreibung: Interne Funktion zur Prüfung des Validators des Typs

Regex

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

- **_validate_range**

Beschreibung: Interne Funktion zur Prüfung des Validators des Typs

Range

Eingaben: Eingabedaten, Validator

Ausgabe: Boolescher Wert

Transaktionsspezifische Implementierungen Die Implementierung zu den Transaktionen arbeiten direkt auf dem protobuf Datentyp. Daher existiert keine separate Klasse, wie es bei Vorlagen oder Verträgen der Fall ist. Um die API konsistent zu halten, implementieren die Funktionen innerhalb des Moduls *fides.core.transaction_impl* daher einige Funktionen, deren

Benennung an die Kernmodule angelehnt ist. Hierbei wird die Konvention $\{Name\ der\ Funktion\}_tx$ verwendet. Die nachfolgenden Funktionen sind Teil des Moduls:

- **save_tx**
Beschreibung: Speichern einer Transaktion
Eingabe: Transaktion
Ausgabe: -
- **db_load_tx**
Beschreibung: Laden einer Transaktion
Eingabe: Hash der Transaktion
Ausgabe: Transaktion oder None
- **sign_tx**
Beschreibung: Signieren einer Transaktion. Hierbei wird der Hash der Transaktion berechnet und der gesamte Inhalt der Transaktion signiert.
Eingaben: Transaktion, privater Schlüssel
Ausgabe: Exception bei bereits signierter Transaktion oder sofern der Typ der Transaktion falsch ist
- **verify_tx**
Beschreibung: Signaturprüfung der Transaktion
Eingabe: Transaktion
Ausgabe: Boolescher Wert

Netzwerk Implementierungen, welche sich im allgemeinen auf die Kommunikation beziehen, sind in dem Modul *fides.core.network* enthalten. Innerhalb des Moduls ist ein Submodul *chord* enthalten, was die Funktionen der DHT für die Anwendung innerhalb von Fides implementiert. Die Implementierung hierzu werden in Kapitel 6.6 separat beschrieben. Weiterhin ist die bereits angesprochene Klasse zur lokalen Kommunikation, *fidesd*, innerhalb des Netzwerkmoduls enthalten. *Fidesd* wird innerhalb der Anwendung verwendet,

wenn Nutzende die Verbindung zu einem Netzwerk herstellen. Hierbei prüft die Klasse die aktuell genutzte lokale Konfiguration und startet die entsprechenden Dienste in Abhängigkeit des gewählten Modus. Zusätzlich empfängt *fidesd* die finalisierten Transaktionen (zum Beispiel von dem Command Line Interface) und leitet jene an die bekannten Netzwerkknoten weiter. Auf technischer Ebene verwendet die Komponente ebenfalls RPC Aufrufe, erlaubt jedoch nur lokale Verbindungen auf dem jeweiligen Gerät. Ergänzend zu den bereits angesprochenen Methoden enthält die Klasse weitere RPC Methoden zum sicheren Stoppen des Netzwerks und zum Abrufen des aktuellen Status der Komponenten, die von *fidesd* verwaltet werden (zum Beispiel zum Veröffentlichen von Transaktionen im Netzwerk). Neben den vorgestellten Elementen des Moduls wird hier auch die Logik der Indexverwaltung implementiert, welche von Netzwerkknoten verwendet wird. Neben den Anforderungen an die Dokumentation der Objekte (siehe 5.2) des Protokolls werden technische Aspekte in 6.6.2 beschrieben.

Datentypen Die in 6.3 beschriebenen *protobuf* [65] Spezifikationen sind Teil des Moduls *fide.core.types*. Hier liegen zudem die dazu passenden Codedateien, die von dem *protobuf* Compiler generiert werden. Alle generierten Objekte werden innerhalb der *__init__.py* Datei des Moduls importiert und sind somit global über das Modul verfügbar, was insgesamt das Importieren einfacher und besser lesbar macht:

```
from fides.core.types import Transaction
```

Auflistung 3: Verkürzter Import von *protobuf* Datentypen

ist daher äquivalent zu:

```
from fides.core.types.transaction_pb2 \
import Transaction
```

Auflistung 4: Regulärer Import von *protobuf* Datentypen

6.4.4 Externe Module

Grund Die Implementierung zur verteilten Hash Tabelle, welche von Fides ergänzt wird, um die RPC Anfragen im Netzwerk zu verteilen, ist innerhalb des Moduls *fides.external.grond* enthalten und wurde separat im Kontext der Referenzimplementierung kollaborativ mit Jens Schneider entwickelt. Die Ergebnisse der Arbeit wurden in [62] veröffentlicht und sollen im folgenden kurz zusammengefasst werden. Da zum Zeitpunkt der Implementierung keine Kandidaten für stabile DHT Implementierung in der spezifizierten Programmiersprache Python existierten, wurde das Chord Protokoll [155] mit einigen Änderungen für die Nutzung in Fides implementiert. Die Implementierungen wurden entsprechend generisch aufgebaut, um das Projekt auch in andere Domänen übertragen zu können. Ziel des Projekts war die Umsetzung des Chord Protokolls in Kombination mit der Erweiterbarkeit durch beliebige gRPC Services. Eine der wichtigsten Änderungen der Implementierung ist die eingeführte **check** Methode zur Prüfung eingehender RPC Anfragen innerhalb der Ringstruktur. Hierbei prüft jeder *full node* für jede Anfrage, die sich auf die Ringstruktur/DHT bezieht, ob jene von einem korrekten Knoten kommt. Hierzu nutzt der Knoten die IP Adresse des aufrufenden Knotens in Kombination mit der angegebenen *id* des Knotens (16) und prüft, ob die angegebene *id* korrekt ist, sich der andere Knoten somit im gleichen Netzwerk befindet und ob jener Knoten auf der angegebenen IP:Port Kombination erreichbar ist, bevor die Anfrage bearbeitet wird.

Weitere Module Neben des Moduls zur Realisierung des Netzwerks als verteilte Hashtabelle nutzt Fides weitere Bibliotheken, die nicht Teil der Pythonumgebung sind. Folgende Module werden bei der Installation von Fides zusätzlich installiert:

- **click** [41] zur Umsetzung des Command Line Interface,
- **cryptography** [160] für die kryptographischen Funktionen (Verschlüs-

selung, Signaturen),

- **grpcio-tools** [159], um Remote Procedure Calls mittels gRPC [101] und protobuf [65] verwenden zu können,
- **python-daemon** [38] für die Realisierung von Hintergrundprozessen (Daemons) auf Linux und MacOS.

6.4.5 Automatisierung

Die API von Fides stellt über das Modul *fides.hooks* sogenannte Hooks zur Automatisierung von Vorlagen und Verträgen bereit, welche im Detail in Kapitel 6.8 erläutert werden.

6.4.6 Zusätzliche Funktionen

Zusätzliche Funktionen, welche von unterschiedlichen Komponenten benutzt werden, wurden im Modul *fides.utils* gebündelt. Nachfolgend werden die Teilmodule kurz vorgestellt.

Alias Innerhalb des Moduls *fides.utils.alias* werden über das Interface *IAlias* verschiedene Klassen zur Verwaltung von lokalen Aliassen, welche die Les- und Nutzbarkeit der Anwendung verbessern sollen, umgesetzt. Allgemein definiert die Klasse die folgenden Methoden, welche von der jeweiligen konkreten Implementierung bereitgestellt werden müssen:

- **add**: Alias hinzufügen
- **delete**: Alias löschen
- **get**: Alias abrufen
- **list**: Aliasse auflisten

Innerhalb des Moduls werden Klassen für das Festlegen von Aliassen für Vorlagen, Verträge und Accounts (öffentliche Schlüssel) implementiert, die von dem Command Line Interface bei der Darstellung und Verwendung der Anwendung benutzt werden.

Config Im Modul *fides.utils.config* werden die folgenden drei Methoden bereitgestellt, um die Konfigurationsdatei der Fides Instanz zu verändern:

- **_validate**
Beschreibung: Interne Funktion zur Validierung der zu setzenden Konfigurationsparameter
Eingaben: Sektion, Name und Wert des Elements
Ausgabe: Boolescher Wert
- **get_config**
Beschreibung: Konfigurationselement auslesen
Eingaben: Sektion, Name des Elements
Ausgabe: Wert des gesuchten Elements oder None
- **set_config**
Beschreibung: Konfigurationselement festlegen
Eingaben: Sektion, Name des Elements, Wert des Elements
Ausgabe: Boolescher Wert
- **delete_element**
Beschreibung: Element einer Sektion löschen
Eingaben: Sektion, Name des Elements
Ausgabe: Boolescher Wert

Log Das Modul *fides.utils.log* stellt allen weiteren Modulen eine Klasse zum Loggen der Ereignisse bereit. Zusätzlich implementiert das Modul eine Klasse zur Sicherung von Log-Nachrichten, die im Command Line Interface mit den Kommandos *fds status/log* abgerufen werden können. Hierzu zählen wichtige

Änderungen an dem lokalen Zustand, zum Beispiel bei dem Empfangen von Transaktionen während eines Vertrags. Die Klasse bereitet die Transaktionen und deren Inhalte auf und verwendet auch hier, sofern vorhanden, Aliasse zur besseren Lesbarkeit.

Time Innerhalb des Moduls *fides.utils.time* werden die folgenden Funktionen zur Behandlung von Zeitstempeln global definiert:

- **now_to_str**
Beschreibung: Aktuelle Systemzeit im festgelegten Format beziehen
Eingaben: -
Ausgabe: Aktueller Zeitstempel als String
- **now_to_date**
Beschreibung: Aktuelles Datum abrufen
Eingaben: -
Ausgabe: Aktuelles Datum
- **str_to_date**
Beschreibung: Zeitstempel von String in Datumsformat konvertieren
Eingabe: Zeitstempel als String
Ausgabe: Zeitstempel im Datumsformat
- **add_delta**
Beschreibung: Delta zu Zeitstempel hinzufügen
Eingaben: Zeitstempel als Datumsformat, Delta in Sekunden
Ausgabe: Zeitstempel+Delta im Datumsformat

Die Methoden werden unter anderem verwendet, um einen Zeitstempel in eine Transaktion einzufügen oder Elemente des Index zu entfernen, sofern die letzte Aktualisierung zu lange zurückliegt.

6.5 Command Line Interface

Fides wird mit einer Kommandozeilenanwendung, kurz CLI für Command Line Interface, ausgeliefert, mit welcher das System verwendet werden kann. Hierzu sind nahezu alle Funktionen innerhalb des CLI abgebildet. Ausnahmen werden in den folgenden Abschnitten, welche die verfügbaren Kommandos beschreiben, erklärt. Das CLI `fds`, abgeleitet von **Fides**, wurde primär für Linux entwickelt, jedoch funktioniert die Anwendung auch auf MacOS und mit kleineren Einschränkungen auf Windows.

Allgemein wurde sich für ein einfaches CLI entschieden, da es wenig Overhead im Repository bedeutet, man Fides sehr einfach steuern kann und die hierzu verwendete Library `click` [41], welche unter der BSD-3-Clause Lizenz steht, sich sehr gut mit der gewählten MIT Lizenz kombinieren lässt. Somit hat man bei der Einbindung von Fides innerhalb einer API noch zusätzlich ein CLI, ohne dass jene Funktionen selbst entwickelt werden müssen. Dies ist besonders hilfreich, wenn man gewisse Prozesse anhand von Skripten automatisieren möchte.

6.5.1 Berechtigungen bei Accounts

Um Accounts, unabhängig davon, ob jene verschlüsselt oder unverschlüsselt sind, vor anderen Nutzenden des gleichen Systems zu schützen, wird eine spezielle Funktion als Parameter der Python Funktion `open` verwendet, welche die Datei des Accounts mit folgenden Berechtigungen anlegt:

- **Besitzer:** Lese- und Schreibzugriff
- **Gruppe:** Kein Zugriff
- **Andere:** Kein Zugriff

Das Vorgehen soll auf falsch konfigurierten Systemen den Zugriff auf die Accountdateien von anderen Nutzenden verhindern.

6.5.2 Übergabe von Passwörtern

Bei einigen Funktionen innerhalb des CLI müssen Passwörter übermittelt werden, um so zum Beispiel den zugehörigen Account zu entsperren, um die jeweilige Aktion durchführen zu können. Hierzu bietet die verwendete Library `click` [41] einen Decorator `password_option` an. Zum Zeitpunkt der Implementierung des CLI nutzte die Funktion jedoch die Standardausgabe (STDOUT), um den Nutzenden dazu aufzufordern, sein Passwort einzugeben. Entsprechende Fehler passieren bei Umleitungen der Ausgabe, beziehungsweise sofern man das Passwort aus einer anderen Quelle mittels Pipes übergeben möchte oder jenes über die Standardeingabe (STDIN) lesen will. Daher wurde eine globale Helferfunktion eingeführt, was das sichere Übergeben für Passwörter plattformunabhängig handhabt und zusätzlich die Unix Philosophie in der Form von Pipes unterstützt. Zusätzlich orientiert sich unsere Implementierung an den Vorgaben von [39]. Allgemein erwartet die Funktion `password_file_or_getpass` innerhalb von `fides.cli.helpers` durch ein übergreifend genutztes Flag „`-password-file`“ entweder eine Datei, in welcher das Passwort zu finden ist, oder verwendet `getpass` [84], um plattformunabhängig das Passwort zu lesen.

Sofern die zu lesende Datei über die korrekten Berechtigungen verfügt, ist jenes Vorgehen sicherer, da keine sensiblen Informationen innerhalb der Prozessübersicht sichtbar gemacht werden. Zudem kann die Anwendung so deutlich einfacher in Skripten verwendet werden, wo man ggf. das Passwort nicht immer eingeben möchte.

Somit stellt die Implementierung des Command Line Interface von Fides eine sichere und plattformunabhängige Variante zur Übermittlung von Passwörtern bereit.

6.5.3 Verfügbare Kommandos

Im Folgenden werden die Kommandos von `fds` vorgestellt. Die Aufteilung jener ist ähnlich zu der API. Eine allgemeine Hilfe für Nutzende kann mit-

tels des „-help“ Flag angezeigt werden, welches jeden Befehl und dessen Argumente genau erklärt. Bevor auf die genauen Kommandos eingegangen wird, werden zunächst noch übergreifende Flags erklärt, die über verschiedene Kommandos hinweg verwendet werden:

- **-password-file**: Wie bereits zuvor beschrieben, werden über dieses Flag entweder Passwörter aus einer Datei gelesen oder, sofern nicht vorhanden, *getpass* [84] genutzt, um jenes durch Eingabe der Nutzenden einzulesen.
- **-account oder -a**: Angabe des Accountnamens, der für den jeweiligen Befehl verwendet wird. Als Standard wird hier „default“ angenommen.
- **-data oder -d**: Eingabedaten, z. B. bei der Bestätigung von Aufgaben.
- **-editor oder -e**: Zeigt die Informationen in dem bevorzugten Editor an.
- **-verbose oder -v**: Startet das Kommando im Vordergrund.

Nachfolgend werden die verfügbaren Subkommandos, Subsubkommandos, deren Parameter und mögliche Optionen vorgestellt.

account

- **fds account alias**: Verwaltung von Aliaswerten für Accounts
 - **add**: Alias hinzufügen
Argumente: Öffentlicher Schlüssel, neuer Alias
 - **delete**: Alias entfernen
Argument: Öffentlicher Schlüssel
 - **list**: Aliasse auflisten

- **fds account create:** Neuen Account erstellen
Argumente: Name, optionale Beschreibung
Optionen: `-password-file` zur Übergabe des Passworts
- **fds account delete:** Account löschen
Argumente: Name
- **fds account dump-raw:** Privaten Schlüssel des Accounts im PEM Format ausgeben
Argument: Name
Optionen: `-password-file`
- **fds account list:** Vorhandene Accounts auflisten
- **fds account password:** Passwort eines Accounts ändern
- **fds account set-default:** Standardaccount festlegen
Argument: Name
- **fds account sign:** Eingabedaten mit gewähltem Account signieren
Argumente: -
Optionen: `-password-file`, `-data`, `-account`
- **fds account verify:** Signierte Daten validieren
Argumente: Öffentlicher Schlüssel, Daten (Ausgabeformat von **sign**)

config

- **fds config delete:** Element aus der Konfiguration löschen
Argumente: Gruppe, Element (z. B. Logging, Level)
- **fds config get:** Element aus der Konfiguration abrufen
Argumente: Gruppe, Element
- **fds config set:** Element in der Konfiguration setzen
Argumente: Gruppe, Element, Wert

contract

- **fds contract accept:** Vertrag annehmen
Argumente: Contract, neues Alias (optional)
Optionen: `-password-file`, `-account`
- **fds contract alias:** Aliasse für Verträge verwalten
 - **add:** Alias hinzufügen
Argumente: Vertrag, neues Alias
 - **delete:** Alias entfernen
Argumente: Vertrag
 - **list:** Aliasse auflisten
- **fds contract apply-tx:** Anwenden einer Transaktion (Datei, STDIN) auf den lokalen Vertrag (ohne Internetverbindung)
Argumente: Vertrag, Pfad zur Transaktion
- **fds contract confirm:** Vertragsschritt bestätigen
Argument: Vertrag
Optionen: `-password-file`, `-account`, `-data`
- **fds contract create:** Vertrag aus Vorlage erstellen
Argumente: Vorlage, Alias für Vertrag (optional)
Optionen: `-password-file`, `-account`, `-publish` bzw. `-p` (sofern der Vertrag direkt veröffentlicht werden soll)
- **fds contract decline:** Ein Vertragsangebot ablehnen
Argumente: Vertrag
Optionen: `-password-file`, `-account` (Empfänger der Vorlage)
- **fds contract delete:** Vertrag vollständig löschen
Argumente: Vertrag

Optionen: `-force` (sofern Vertrag nicht im Status *FINISHED* ist trotzdem löschen)

- **fds contract export**: Informationen über Vertrag exportieren (Transaktionen, temporärer privater Schlüssel)
Argumente: Vertrag, Pfad zur Datei (optional)
Optionen: `-password-file`, `-account`
- **fds contract fetch**: Vertrag aus Netzwerk beziehen
Argumente: Vertrag
- **fds contract import**: Vertrag über lokale Transaktion beziehen
Argumente: Pfad zur Transaktion
- **fds contract list**: Alle Verträge auflisten
- **fds contract offers**: Alle Verträge im Zustand *OFFER* auflisten, bei denen der angegebene Account die Vorlage verwaltet
Optionen: `-account`
- **fds contract publish**: Vertrag veröffentlichen
Argument: Vertrag
Optionen: `-password-file`, `-account`
- **fds contract show**: Informationen über Vertrag anzeigen
Optionen: `-editor`
- **fds contract status**: Detaillierte Statusinformationen zum Vertrag anzeigen (z. B. entschlüsselte Transaktionen)
Argumente: Vertrag
Optionen: `-password-file`, `-account`, `-editor`
- **fds contract update**: Manuell den Zustand des Vertrags aktualisieren
Argumente: Vertrag

endpoint

- **fds endpoint add**: Hinzufügen eines neuen Knotens innerhalb des Netzwerks
Argumente: IP, Port (optional - Standardwert 3301), Netzwerk (optional - entnommen aus Konfiguration)
- **fds endpoint delete**: Löschen eines Knotens
Argumente: ID des Knotens
- **fds endpoint list**: Verfügbare Knoten, unabhängig vom Netzwerk, auflisten

hooks

- **fds hooks start**: Hook starten
Argument: Name der Hook-Datei
Optionen: `-verbose`
- **fds hooks stop**: Hook stoppen
Argument: Name der Hook-Datei
- **fds hooks status**: Statusinformationen zu laufenden Hooks
Optionen: `-cleanup` zum Aufräumen von alten „pid“ Dateien bei nicht laufenden Hooks

init

Initialisiert die lokale Fides Instanz

Argument: Pfad (optional)

Optionen: `-default`, um die Standardkonfiguration zu verwenden

log

Zeigt einen Log der Transaktionen im Terminal an. Die neusten Transaktion werden oben dargestellt.

Optionen: `-cleanup`, um den Log vollständig zu löschen.

lora

Verwalten der LoRaWAN Abstraktion. Siehe Kapitel 7.

network

- **fds network crawl**: Das Netzwerk nach neuen Knoten durchlaufen
- **fds network create-id**: ID für Netzwerkknoten berechnen
Argumente: IP, Port, Netzwerk
- **fds network index-drop**: Den lokalen Index vollständig löschen
- **fds network index-status**: Den Status des lokalen Index abrufen
Optionen: `-editor`
- **fds network node-info**: Informationen über einen Netzwerkknoten abrufen
Argumente: IP, Port
- **fds network start**: Netzwerk starten
Optionen: `-verbose`
- **fds network status**: Allgemeinen Netzwerkstatus abrufen
- **fds network stop**: Netzwerk stoppen

status

Zeigt alle neuen Transaktionen seit dem letzten Aufruf des Befehls an. Ähnlich zu **fds log** werden hier jedoch die Transaktion beginnend mit der Ältesten zuerst angezeigt.

template

- **fds template alias**: Aliasse für Vorlagen verwalten. Gleiches Subkommandos wie **fds contract alias**
- **fds template create**: Erstellen einer neuen Vorlage in dem bevorzugten Texteditor. Hier können jedoch nur Validatoren vom Typ „Plain-text“ verwendet werden.
Argument: Alias der Vorlage (optional)
Optionen: `-account`, `-password-file`, `-publish` zum direkten Veröffentlichen
- **fds template delete**: Vorlage lokal löschen
Argument: Vorlage
- **fds template fetch**: Vorlage aus dem Netzwerk beziehen
Argument: Vorlage
- **fds template import**: Vorlage aus lokaler Transaktion beziehen
Argument: Pfad zur Transaktion
- **fds template list**: Vorhandene Vorlagen mit kurzer Beschreibung auflisten
- **fds template publish**: Vorlage veröffentlichen
Argumente: Vorlage
Optionen: `-account`, `-password-file`
- **fds template revoke**: Vorlage zurückziehen
Argument: Vorlage

- **fds template show:** Vorlage anzeigen
Argument: Vorlage
Optionen: `-editor`
- **fds template update:** Vorlage aktualisieren (berücksichtigt neue Angebote)
Argument: Vorlage

tx

- **fds tx decrypt:** Transaktionsdaten (bei Bestätigungen) entschlüsseln
Argumente: Transaktion, Passwort
Optionen: `-validate`, um Validator der Aufgabe auf entschlüsselte Daten anzuwenden
- **fds tx extract-secret:** Passwort zur Entschlüsselung der Transaktion extrahieren
Argument: Transaktion
Optionen: `-account`, `-password-file`
- **fds tx show:** Transaktion anzeigen
Argument: Transaktion

6.5.4 Ausgaben

Im Allgemeinen wurde darauf geachtet, die klassische Unix Philosophie bei den jeweiligen Kommandos einzuhalten. Hierdurch ist es möglich, die jeweiligen Ausgaben vernünftig umzuleiten oder sie in anderen Befehlen in dem CLI verwenden zu können. In der Regel nutzt die Ausgabe an die Nutzenden `STDERR` für zusätzliche Informationen und `STDOUT` für weiterleitbare Informationen. Dies wird kurz an einigen Beispielen gezeigt:

- Aktualisieren eines Vertrags \mathcal{C} (Hash oder Alias), dessen Ausgabe der Hash oder Alias des Vertrags selbst ist, welche entsprechend an den nächsten Befehl weitergegeben wird:

```
fds contract update  $\mathcal{C}$  | xargs fds contract show
```

Auflistung 5: Kommandozeilenanwendung: Aktualisieren und Anzeigen eines Vertrags

- Entschlüsseln von Transaktionsdaten mit dem zuvor extrahierten Schlüssel für die jeweilige Transaktion:

```
pass fds \  
| fds tx extract-secret  $\mathcal{T}$  --password-file - \  
| xargs fds tx decrypt  $\mathcal{T}$ 
```

Auflistung 6: Kommandozeilenanwendung: Einzelne Transaktion entschlüsseln

- Extraktion eines temporären privaten Schlüssels des Vertrags \mathcal{C} , dessen Passwort der Signatur von \mathcal{H}_C entspricht:

```
pass fds \  
| fds account sign --data " $\mathcal{H}_C$ " \  
  --password-file - \  
| awk -F : '{print $2}' \  
| fds account dump-raw \  
  ephemeral/ $\mathcal{H}_C$ /sender.priv --password-file -
```

Auflistung 7: Kommandozeilenanwendung: Temporären privaten Schlüssel auslesen

Entsprechend ist es möglich, Fides und dessen Command Line Interface in diverse Skripte zu integrieren, um somit Vorgänge, wie zum Beispiel das Archivieren von Übereinkünften, automatisieren zu können ohne hierfür ein

separates Programm erstellen zu müssen, welches die API direkt verwenden würde.

6.6 Netzwerkarchitektur

Nachfolgend wird die Netzwerkarchitektur von Fides beschrieben. Hierbei wird allgemein zwischen Netzwerkknoten, die im Zusammenschluss die Zustände der Verträge und Vorlagen dokumentieren, und Anwenderinnen und Anwendern, die lediglich mit Verträgen und Vorlagen interagieren, unterschieden.

6.6.1 Allgemeines

Im Allgemeinen wird zur Kommunikation in Fides gRPC [101] verwendet. Die Bibliothek ist unmittelbar kompatibel mit dem genutzten Serialisierungsformat protobuf [65]. Innerhalb der Bibliothek sind Transaktionen bis zu 4 MB zugelassen. Fides hingegen limitiert die Transaktionsgröße auf 1 MB.

6.6.2 Netzwerkknoten

Netzwerkknoten, auch *full nodes* genannt, formen die Fides Netzwerke und dokumentieren den Status der Übereinkünfte auf den jeweiligen Indizes. Aus technischer Sicht erlauben Netzwerkknoten die Ausführung der RPC Methoden und öffnen hierzu einen TCP Port, welcher von anderen Teilnehmenden oder weiteren Netzwerkknoten benutzt werden kann, um die Zustände der Verträge und Vorlagen im Netzwerk zu dokumentieren oder selbst als *full node* Teil des Netzwerks zu werden. Zur Umsetzung der distributed hash table (siehe 3.5) wurde die Bibliothek *Grond* [117] verwendet, welche gemeinsam mit Jens Schneider entwickelt wurde. Bei *Grond* handelt es sich um eine Implementierung des Chord Protokolls [155] in Python, was gRPC [101] zur Kommunikation verwendet. Das Modul ist direkt im offiziellen Fides Repository [129] enthalten.

Die Kapselung des Moduls erlaubt das Spezifizieren von Kernfunktionen des Chord Protokolls als RPC Methoden, welche durch anwendungsspezifische RPC Methoden (hier Fides) ergänzt werden. Abbildung 14 verdeutlicht das Beschriebene. Zunächst implementiert die Klasse *Node* des Moduls *Grond* die in 6.4.4 beschriebenen Kernfunktionen zur Umsetzung der DHT und wird ergänzt um die Anwendungsspezifischen RPC Methoden zur Dokumentation der Vorlagen und Verträge. Seit der Veröffentlichung von Version 0.1.0¹⁰, welche einige Verbesserungen der Stabilität der DHT ergänzt hat, unterstützt der Konstruktor der Klasse *Node* zudem einen Parameter zur Übertragung weiterer RPC Services. Diese Klasse *Node* dient als Basis für die Klasse *FidesGrondNode*, die die darunterliegende DHT Funktionen, also Chord, benutzt und um weitere RPC Methoden ergänzt, die dann von anderen Netzwerkknoten oder regulären Anwenderinnen und Anwendern von Fides benutzt werden. Der folgende Auszug zeigt einen Teil des Konstruktors der *FidesGrondNode* Klasse zur Verdeutlichung. Die genutzte Methode *add_indexServicer_to_server* wird aus dem dazugehörigen Service *index* generiert, in welcher die in 5.2 beschriebenen Indexfunktionen implementiert werden. Die dazugehörige Klasse *indexServicer* enthält die unimplementierten RPC Methoden der Spezifikation, welche dann von der Klasse *FidesGrondNode* implementiert werden.

```
class FidesGrondNode(Node, indexServicer):

    def __init__(self, id, ip, port, network, cert):
        services = [add_indexServicer_to_server]
        super().__init__(id, ip, port,
                        network, cert, services)
```

Auflistung 8: Konstruktor *FidesGrondNode* (Auszug)

¹⁰Die Version 0.1.0 des Projekts *Grond* wurde mit Fides in Version 0.5.0 veröffentlicht: <https://gitlab.rlp.net/1.creutz/fides/-/commit/45959fee5ed19e9be9bf2cd3259dadbd8c1562bc> (Abgerufen im Februar 2023)

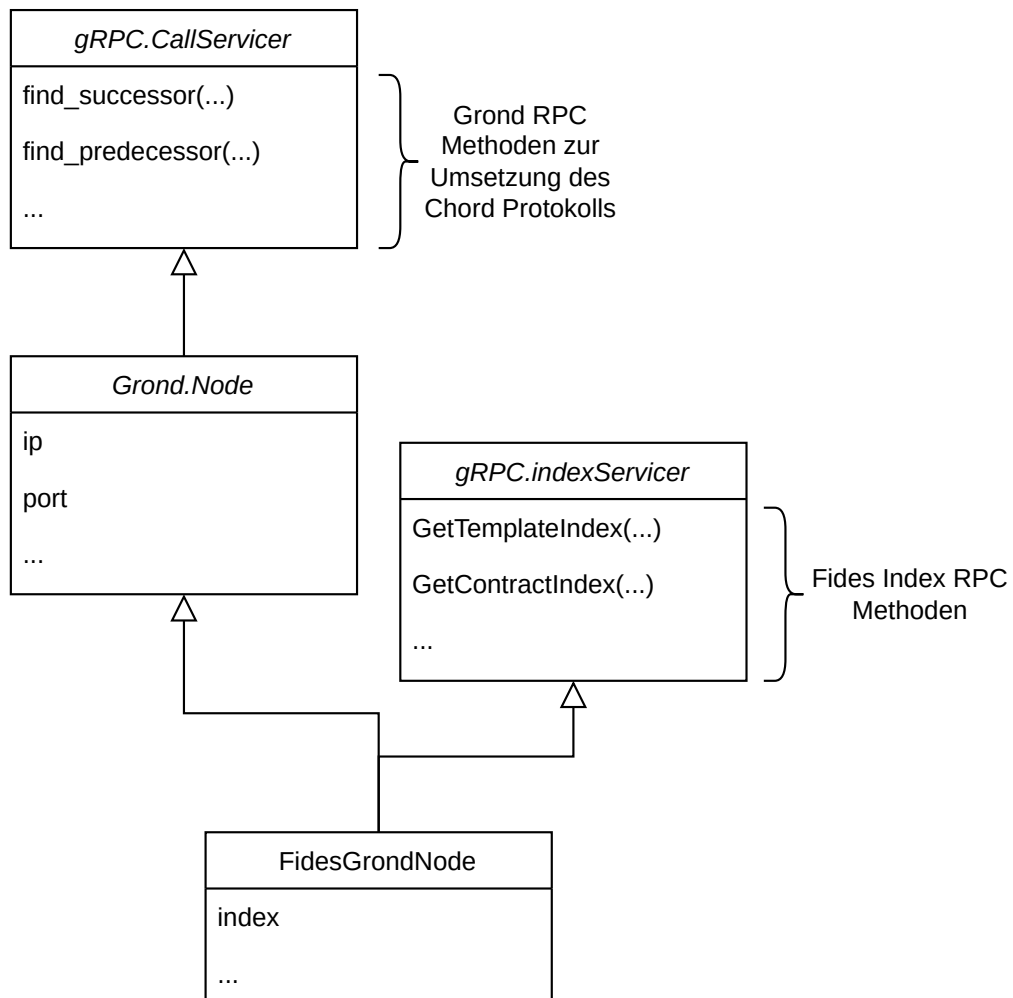


Abbildung 14: Kapselung der RPC Methoden für Netzwerkknoten

Bevor konkreter auf die Änderungen von Fides eingegangen wird, wird zunächst auf die RPC Methoden der Netzwerkknoten genauer eingegangen. Neben dem Service *index* werden folgende protobuf Datentypen definiert, welche in Anfragen (Request) und Antworten (Response) unterteilt werden können:

Anfragen

- **GetTemplateRequest**: Anfrage, eine Vorlage anhand ihres Hashs zu beziehen.
- **GetTemplateIndexRequest**: Beziehen des Index der angegebenen Vorlage.
- **UpdateTemplateIndexRequest**: Index der Vorlage aktualisieren. Die Anfrage adressiert eine bestimmte Vorlage und enthält die Transaktion, welche mit jener interagiert. In privaten Netzwerken enthält die Nachricht noch das Zertifikat der Partei, die die Transaktion erstellt hat (siehe 6.6.4).
- **TransferIndexRequest**: Anfrage zur Übertragung des eigenen Index (als *full node*) zu dem unmittelbaren Nachfolger im Netzwerk. Die Anfrage beschreibt im Feld *object* das zu übertragende Objekt (Vorlage/Vertrag) und gibt den Typ des Objekts über eine Enumeration (*TRANSFER_TYPE*) an.
- **PingTemplateIndexRequest**: Ping Anfrage, die verhindert, dass die jeweilige Transaktion nicht automatisch vom Vorlagenindex gelöscht wird. Die Anfrage enthält neben der Art des Objekts (Vorlage/Vertrag) den Hash der Vorlage und eine Referenz auf die Transaktion, die die Zustandsänderung auf dem Index hervorgerufen hat.
- **GetTemplateTransactionRequest**: Beziehen einer Transaktion, welche den Index der Vorlage verändert hat. Innerhalb der Anfrage wird

die Vorlage über das Feld *hash* angegeben. Das Feld *tx.hash* gibt den Hash der zu beziehenden Transaktion an.

- **GetContractRequest:** Anfrage zum Beziehen eines Vertrags anhand seines Hashs.
- **GetContractIndexRequest:** Beziehen der letzten Transaktion, die den Zustand des Vertrags verändert hat.
- **GetEntireContractIndexRequest:** Anfrage zum Erhalt des kompletten Vertragsindex und somit allen Transaktionen, die mit dem Vertrag interagierten. Der Indexknoten antwortet über einen *stream* und übermittelt die jeweiligen Einträge.
- **UpdateContractIndexRequest:** Ändern oder Erstellen eines Vertragsindex. Die Anfrage enthält den Hash des Vertrags, die Transaktion, die die Änderung durchführt und zusätzlich die Transaktion, welche die verwendete Vorlage veröffentlicht hat, sofern der Vertragsindex neu erstellt wird. Weiterhin enthalten ist ein Zertifikat, sofern es sich um ein privates Netzwerk handelt (siehe 6.6.4).
- **GetContractTransactionRequest:** Beziehen einer Transaktion von dem Vertragsindex. Der Vertrag wird über das Feld *hash* adressiert, die Transaktion über das Feld *tx.hash*.
- **PingContractIndexRequest:** Ping Anfrage an den Vertragsindex zur Anzeige, dass jener Index nicht von dem Netzwerkknoten gelöscht werden soll.

Antworten

- **PingResponse:** Antwort auf einen Ping des Objekts. Enthält einen booleschen Wert *success*.

- **GetTemplateResponse**: Beziehen einer Vorlage vom Index. Enthält neben dem booleschen Wert *success* noch das Feld *tx*, in welchem die Transaktion enthalten ist, welche die Vorlage veröffentlicht hat.
- **TemplateIndexEntry**: Gibt einen Eintrag auf dem Vorlagenindex zurück, der eine Transaktion eines Vertrags im Feld *tx_ref* enthält, welche die Vorlage referenziert. Wird genutzt, um sogenannte *stream* Antworten auf RPC Anfragen zu übermitteln, also zum Beispiel, um alle Transaktion zu beziehen, die auf die Vorlage verweisen.
- **UpdateTemplateIndexResponse**: Antwort auf den Request, den Zustand des Index für die Vorlagen zu verändern. Enthält neben dem *success* Feld einen etwaigen Fehlercode, der beschreibt, welche Überprüfung zur Ablehnung der Transaktion geführt hat. Zu den Fehlercodes zählen:
 - **NO_ERROR**: Kein Fehler.
 - **NO_TEMPLATE_METADATA** : Dem Indexknoten sind keine Daten über die Vorlage bekannt.
 - **MALFORMED_REQUEST** : Falscher Request.
Entweder stimmt der im Request angegebene Hash nicht mit dem der enthaltenen Transaktion überein (beim Veröffentlichen und Zurückziehen einer Vorlage) oder der Vertrag, welcher auf die Vorlage verweist, ist bereits Teil des Index (Ping ausreichend).
 - **TEMPLATE_CHECK_FAILED** : Überprüfung der Vorlage fehlgeschlagen. Hier führt der Netzwerkknoten die Methode **check**

auf der Vorlage aus.

- **WRONG_OWNER** : Die Partei, die die Transaktion signiert hat, besitzt nicht die Vorlage. Entsprechend kann zum Beispiel kein Vertrag angenommen oder die Vorlage zurückgezogen werden.
- **TEMPLATE_INACTIVE** : Die verwendete Vorlage ist bereits inaktiv. Es können somit z. B. keine weiteren Verträge erstellt werden.
- **WRONG_TX_TYPE** : Die Transaktion entspricht einem unbekanntem/unbehandeltem Typ.
- **NO_TEMPLATE_DATA**: Dem Indexknoten sind keine Daten über das Template bekannt.
- **CONTRACT_CHECK_FAILED**: Überprüfung des Vertrags ist fehlgeschlagen. Hier werden zusätzlich zu der *check* Methode der Klasse der Verträge noch weitere Überprüfungen durchgeführt (Anfrage von der korrekten Partei, passt die Referenz zur Vorlage).
- **NO_CONTRACT_DATA**: Dem Indexknoten sind keine Daten über den Vertrag bekannt.
- **SIG_WRONG**: Die Transaktion wurde nicht korrekt signiert.
- **TEMPLATE_ALREADY_PRESENT**: Die Vorlage, die auf dem Index übernommen wurde, ist dem Netzwerkknoten bereits

bekannt.

- **TEMPLATE_TX_SIZE_TOO_BIG**: Die Größe der Transaktion überschreitet 1 MB und wird daher abgelehnt.
 - **CLIENT_CERT_INVALID**: Der *client* besitzt kein korrektes Zertifikat, um den Index in dem Netzwerk zu ändern (siehe 6.6.4).
-
- **TransferIndexResponse**: Gibt im Feld *success* an, ob der angefragte Transfer an den nachfolgenden Netzwerkknoten erfolgreich war.
 - **GetTemplateTransactionResponse**: Resultat beim Beziehen einer Transaktion von dem Vorlagenindex. Enthält neben dem Indikation *success* noch das Feld *tx*, welches die angeforderte Transaktion inkludiert.
 - **GetContractResponse**: Bezieht einen Vertrag über den Index, welcher im Feld *tx* enthalten ist. Das Flag *success* zeigt an, ob sich das Objekt auf dem Index befindet.
 - **GetContractIndexResponse**: Ruft die letzte Transaktion ab, die den Zustand des Vertrags verändert hat. Jene Transaktion ist innerhalb der Antwort im Feld *tx* enthalten, das Feld *success* gibt den Status der Abfrage zurück.
 - **ContractIndexEntry**: Wird verwendet, um mehrere Transaktionen eines Vertrags zu beziehen. Notwendig, wenn in einem Vertrag mehrere Aufgaben nacheinander von der gleichen Partei bearbeitet werden und die aktuell letzte Transaktion nicht auf den lokalen Zustand angewendet werden könnte. Die jeweilige Transaktion ist im Feld *tx.ref* enthalten, die als *stream* RPC Antwort übermittelt wird.

- **UpdateContractIndexResponse**: Antwort auf die Anfrage zum Ändern des Vertragsindex. Enthält neben dem Flag *success* noch einen Fehlercode, welcher auf der Seite der *clients* behandelt wird:
 - **NO_ERROR**: Es trat kein Fehler auf.
 - **SIG_WRONG**: Die Signatur der Transaktion ist falsch.
 - **TEMPLATE_CHECK_FAILED**: Die Überprüfung der Vorlage ist fehlgeschlagen. Geprüft wird hier die Vorlage im Allgemeinen, da jene Teil der Anfrage ist und vom Indexknoten, welcher den Vertrag verwalten soll, nicht separat aus dem Netzwerk bezogen wird.
 - **CONTRACT_CHECK_FAIL**: Die Überprüfung des Vertrags ist fehlgeschlagen.
 - **NO_CONTRACT_DATA**: Der Vertrag konnte nicht gefunden werden, daher konnten Transaktion, die jenen annehmen/ablehnen oder Aufgaben bestätigen, nicht darauf angewendet werden.
 - **VALIDATION_FAIL**: Die Transaktion, welche den Zustand des Vertrags verändern möchte, konnte nicht erfolgreich validiert werden. Entsprechend wird eine Änderung des Zustands abgelehnt.
 - **WRONG_STATE**: Der Vertrag befindet sich im falschen Zustand und daher kann die Transaktion nicht angewendet werden.
 - **ALREADY_PRESENT**: Der Vertrag ist dem Indexknoten bereits bekannt.
 - **CONTRACT_TX_SIZE_TOO_BIG**: Die Größe der Transaktion überschreitet das Limit von 1 MB und wird daher abgelehnt.
 - **CLIENT_CERT_INVALID**: Der Client verfügt nicht über ein korrektes Zertifikat, um die Anfrage durchzuführen (siehe 6.6.4).
- **GetContractTransactionResponse**: Bezieht eine Transaktion, welche mit dem angegebenen Vertrag interagiert hat. Das Flag *success*

gibt den Status der Anfrage an. Im Feld *tx* ist die Transaktion enthalten, sofern die Anfrage korrekt war.

Weitere Methoden Neben den vorgestellten Anfragen und deren Antworten, enthalten Indexknoten weitere RPC Methoden, die von anderen Teilen der Anwendung verwendet werden. Die Funktion **GetNodeInfo** antwortet mit dem Typ *grond.NodeInfo*, in welchem die Angaben zu der ID des Netzwerkknotens, seiner IP:Port Kombination und in privaten Netzwerken (siehe 6.6.4) sein Zertifikat enthalten sind. Verwendet wird die Methode, um zum Beispiel auf Seite der Nutzenden feststellen zu können, ob der Knoten das Netzwerk gewechselt hat (veränderte ID im Vergleich zum lokalen Zustand).

Eine weitere Methode **GetPredecessor** antwortet ebenfalls mit dem Typ *grond.NodeInfo* und erlaubt es *clients* weitere Netzwerkknoten innerhalb des Netzwerks zu finden. Im Prinzip wird so durch die Ringstruktur „gelaufen“. Die Methode wird zum Beispiel innerhalb des Command Line Interface verwendet (*fds network crawl*).

Umsetzung des Index Die bei den Kernfunktionen genannte Klasse zur Umsetzung der Indizes wird innerhalb der Klasse *FidesGrondNode* verwendet. Hierbei ist es wichtig, dass die in der Klasse *Index* verwendeten SQLite Datenbankverbindungen mit dem Attribut *check_same_thread=False* geöffnet werden, die RPC Methoden innerhalb von gRPC [101] verschiedene Threads verwenden und SQLite es sonst nur erlauben würde die Datenbank von dem Thread, der sie öffnete, zu verwalten [91]. Entsprechend muss die Synchronisierung separat implementiert werden [91], was innerhalb von Fides über einen Lock [92] realisiert wurde. Die Realisierung der RPC Methoden wird dann innerhalb der Klasse *FidesGrondNode* für alle recht ähnlich behandelt, indem wie folgt vorgegangen wird:

1. Prüfe, ob der eigene Knoten zuständig für die Anfrage ist. Hierbei wird

die interne Methode `_responsible` aufgerufen, welche *Grond* verwendet, um den Knoten im Netzwerk zu finden, der für den angegebenen Hash des Objekts zuständig ist.

2. Sofern man nicht für die Anfrage zuständig ist, wird die gesamte Anfrage an den im ersten Schritt gefundenen Knoten weitergeleitet.
3. Sollte man als Netzwerkknoten selbst zuständig sein, wird die Anfrage bearbeitet. Sollte es sich lediglich um das Beziehen von Informationen handeln, ist keine zusätzliche Synchronisierung notwendig. Sofern die Anfrage jedoch den Zustand des Index verändern will, muss zunächst die Sperre bezogen werden. Danach wird die Anfrage an die jeweilige Methode der Klasse *Index* weitergeleitet und der jeweilige Rückgabewert als Antwort auf die RPC Anfrage übermittelt.

Lediglich bei privaten Netzwerken (siehe 6.6.4) ändert sich das Vorgehen bei Anfragen, die den Index verändern, da dort zunächst noch das jeweilige Zertifikat geprüft werden muss.

6.6.3 Nutzende

Reguläre Anwenderinnen und Anwender verbinden sich mit den zuvor vorgestellten Netzwerkknoten, um ihre Vorlagen und Verträge auf jenen zu verwalten. Hierzu müssen *clients* keine Verbindungen von Außen erlauben und können sich somit problemlos hinter einer abgeriegelten Firewall befinden. Entsprechend einfach ist die Verwendung der Anwendung, da sie keiner weiteren Konfiguration des Rechners oder Heimnetzwerks bedarf. Allgemein verwaltet die Klasse *ChordClient* des Moduls *fides.core.network.chord.client* die gesamte Kommunikation um Vorlagen und Verträge. Nachfolgend werden die Methoden der Klasse kurz vorgestellt:

- `__init__`
Beschreibung: Konstruktor der Klasse

Eingaben: -

Ausgabe: ChordClient Objekt

- **status**

Beschreibung: Statusmeldung, die von *fidesd* z. B. dem Command Line Interface bereitgestellt wird

Eingaben: -

Ausgabe: Status der Komponente als String

- **check_node_cert**

Beschreibung: Überprüfung des Zertifikats eines *full nodes*

Eingaben: ID des Knotens, Zertifikat

Ausgabe: Boolescher Wert

- **get_random_endpoint**

Beschreibung: Zufällige *full nodes* aus lokalem Datenbestand beziehen

Eingaben: -

Ausgabe: RPC Verbindungselement oder None

- **publish**

Beschreibung: Transaktion, die über *fidesd* weitergeleitet wurde, im Netzwerk veröffentlichen

Eingaben: Anfrage/Transaktion

Ausgabe: Tupel (Boolescher Wert, None/Fehlermeldung)

- **start**

Beschreibung: Starten der Netzwerkverbindung und Threads zur Überwachung der Objekte (Templates/Contracts)

Eingaben: -

Ausgabe: Boolescher Wert

- **_handle_failed_contract_update**

Beschreibung: Versucht ein fehlgeschlagenes Vertragsupdate zu korrigieren

Eingaben: Contract Objekt, Transaktion

Ausgabe: Boolescher Wert

- **_handle_template_republish**

Beschreibung: Republikation der Vorlage durchführen

Eingabe: Hash der Vorlage

Ausgabe: Boolescher Wert

- **_handle_contract_republish**

Beschreibung: Vertrag republizieren

Eingabe: Contract Objekt

Ausgabe: Boolescher Wert

- **check_contract_state**

Beschreibung: Aktuellen Zustand des Vertrags mit Index vergleichen

Eingabe: Hash des Vertrags

Ausgabe: Boolescher Wert

- **check_template_state**

Beschreibung: Zustand einer Vorlage abrufen

Eingabe: Hash der Vorlage

Ausgabe: Boolescher Wert

- **_check_contract_states**

Beschreibung: Interne Funktion, die regelmäßig alle Verträge der Fides Instanz auf Aktualität überprüft

Eingaben: -

Ausgabe: -

- **_check_template_states**

Beschreibung: Interne Funktion zur Prüfung der eigenen Vorlagen

Eingaben: -

Ausgabe: -

- **fetch_template**

Beschreibung: Template über das Netzwerk beziehen

Eingabe: Hash der Vorlage

Ausgabe: Tupel (boolscher Wert, Transaktion/None)

- **fetch_contract**

Beschreibung: Contract aus dem Netzwerk beziehen

Eingabe: Hash des Vertrags

Ausgabe: Tupel(boolscher Wert, Transaktion/None)

- **stop**

Beschreibung: ChordClient stoppen

Eingaben: -

Ausgabe: -

Neben der Klasse *ChordClient* enthält das Modul noch zwei weitere Methoden, welche global gültig sind:

- **get_tx_until**

Beschreibung: Menge an Transaktionen zwischen zwei Transaktionen aus lokalen Informationen bestimmen

Eingaben: Letzte Transaktion, Referenztransaktion

Ausgabe: Liste an Transaktionen

- **check_contract_advanced**

Beschreibung: Prüfen, ob lokaler Zustand eines Vertrags vor oder nach der gegebenen Transaktion ist

Eingaben: Contract Objekt, Transaktion

Ausgabe: Tupel (boolscher Wert, Liste an Transaktionen/None)

Die Komponente *ChordClient* wird von dem allgemeinen Kommunikations-Daemon *fidesd* gestartet und operiert dann im Hintergrund. Initialisiert wird die Klasse mit dem gewählten Netzwerknamen, in welchem die Fides Instanz

angesiedelt ist, dem gewählten Aktualisierungsintervall zur Prüfung auf neue Zustände und in privaten Netzwerken mit dem aktuellen Zertifikat. Alle Elemente werden aus der Konfigurationsdatei über das Modul *fides.core.config* bezogen.

Mit dem Start von *fidesd* wird die Methode **start** der *ChordClient* Klasse aufgerufen, welche zwei Threads startet, die im Hintergrund anhand des gegebenen Intervalls prüfen, ob sich die Zustände der Verträge oder Vorlagen der aktuellen Fides Instanz verändert haben. Innerhalb der von den Threads ausgeführten Methoden werden folgende Aktionen durchgeführt:

1. Verbindung zur jeweiligen Datenbank (Verträge/Vorlagen) herstellen.
2. Zertifikatselement der Konfiguration dynamisch neu laden. Dies erlaubt das Wechseln von Zertifikaten bei der Verwendung von unterschiedlichen Accounts in einem privaten Netzwerk. Weiterhin erlaubt dieses Vorgehen es Abstraktionen etwaige Zertifikate für verschiedene *clients* zu verwalten.
3. Abrufen von Verträgen, die nicht im Zustand *REJECTED* sind oder von Vorlagen, die man selbst publiziert hat.
4. Status von Vertrag/Vorlage prüfen und ggf. Objekt republishieren (sofern nicht im Netzwerk vorhanden).

Nachfolgend wird kurz erläutert, wie jeweils der Status von Verträgen und Vorlagen geprüft oder aktualisiert wird.

Prüfen des Zustands eines Vertrags Um den Zustand eines Vertrags korrekt zu prüfen, müssen einige Randbedingungen beachtet werden. Im Allgemeinen wird wie folgt verfahren:

1. Lade den Vertrag aus der lokalen Instanz.

2. Prüfe, ob das interne Feld *tx_ref* gesetzt ist, was bedeutet, dass der Vertrag korrekt mit einer Transaktion publiziert wurde. Wurde der Vertrag noch nicht publiziert, überspringe die Überprüfung. Der Vorgang des Publizierens muss explizit durch die Nutzenden angestoßen werden.
3. Rufe einen zufälligen Netzwerkknoten ab. Sollte kein Knoten erreichbar sein, wird die Überprüfung bis zum nächsten Durchlauf abgebrochen.
4. Stelle die RPC Anfrage **GetContractIndex** mit dem Hash des aktuell betrachteten Vertrags. Sofern die Anfrage fehlschlägt, republiziere den gesamten Vertrag.
5. Prüfe die letzte dem Index bekannte Transaktion, die mit dem Vertrag interagiert hat. Ist jene gleich dem lokalen Zustand, hat sich der Vertrag nicht verändert. Sollte der Vertrag im Zustand *OFFER* sein, pinge dann noch den Vorlagenindex, damit die Partei, welche die Vorlage verwaltet, die Chance hat den neuen Vertrag dort zu finden. Sofern die Ping-Anfrage fehlschlägt, republiziere den Vertrag im Gesamten. Stelle abschließend eine Ping-Anfrage an den Vertragsindex.
6. Unterscheidet sich die letzte Transaktion, wende sie mit der **update** Methode des Vertrags auf das Objekt an. Sollte das Update fehlschlagen, prüfe, ob der Fehler aufgelöst werden kann. Dies ist zum Beispiel der Fall, wenn mehrere Transaktionen nacheinander angewandt werden müssten, wenn der Index mehrere Transaktionen weiter ist als der lokale Zustand.
7. Sofern die Aktualisierung(en) korrekt durchgeführt werden konnten, aktualisiere den Log.

Aktualisieren des Zustands eines Vertrags Bei der aktiven Aktualisierung der Verträge durch Nutzende müssen verschiedene Szenarien un-

terschieden werden. Handelt es sich um die Veröffentlichung eines neuen Vertrags, geht die sendende Partei folgendermaßen vor:

1. Aktualisiere den Vorlagenindex per **UpdateTemplateIndexRequest**, um den Vertrag der anderen Partei zugänglich zu machen.
2. Sofern die Aktualisierung des Vorlagenindex erfolgreich durchgeführt wurde, erstelle den Vertragsindex mittels der RPC Anfrage **UpdateContractIndex**.
3. Wenn beide RPC Anfragen erfolgreich waren, aktualisiere den Log, die Transaktion und den neuen Zustand des Vertrags.

Auf bestimmte Fehler der RPC Anfragen wird auf der Seite der *clients* zusätzlich reagiert. Meldet so der Indexknoten zum Beispiel, dass die Vorlage inaktiv ist (**TEMPLATE_INACTIVE**), so wird die genutzte Vorlage lokal noch aktualisiert, indem die Transaktion, die die Vorlage zurückzog, bezogen und validiert wird.

Nach der Veröffentlichung des Vertrags durch die sendende Partei, reagiert nun die empfangende Partei durch Annahme oder das Ablehnen des Vertrags, indem auch sie den Zustand des Vertrags durch eine Transaktion ändert. In beiden Fällen wird wie folgt vorgegangen:

1. Aktualisiere zunächst den Vorlagenindex, welcher den Eintrag zum jeweiligen Vertrag entfernt, sofern die RPC Anfrage erfolgreich war.
2. Aktualisiere den Vertragsindex und ändere den Zustand des Vertrags hiermit zu *LIVE* oder *REJECTED*.
3. Lokale Aktualisierung, wenn der Vertragsindex erfolgreich aktualisiert wurde (Logs, lokaler Zustand, Transaktion).

Übereinkünfte im Zustand *LIVE* werden dann nur auf dem Vertragsindex aktualisiert, indem Transaktionen des Typs *confirmTask* jeweils via Anfragen des Typs **UpdateContractIndex** an das Netzwerk gestellt werden.

Prüfen des Zustands einer Vorlage Um den Zustand einer Vorlage zu prüfen, wird ähnlich verfahren wie bei einem Vertrag. Entsprechend werden folgende Schritte durch die Komponente *ChordClient* durchgeführt:

1. Lade die Vorlage aus der lokalen Instanz. Der Unterschied zum Vorgehen bei den Verträgen ist hier, dass nur Vorlagen betrachtet werden, die bereits aus der aktuellen Instanz veröffentlicht wurden.
2. Rufe einen zufälligen Netzwerkknoten ab. Auch hier wird die Überprüfung in diesem Durchlauf unterbrochen, sofern kein Netzwerkknoten erreicht werden kann.
3. Führe eine Ping-Anfrage an den Vorlagenindex durch. Sollte die Anfrage fehlschlagen, wird die Vorlage aus dem lokalen Zustand republiciert.
4. Sollten der Vorlagenindex erreichbar und die Ping-Anfrage erfolgreich gewesen sein, beziehe den Vorlagenindex über die RPC Anfrage **Get-TemplateIndex**.
5. Für jeden Eintrag auf dem Index versuche die jeweilige Transaktion zu beziehen und den Vertrag lokal zu importieren.
6. Ergänze den Log nach dem korrekten Import eines Vertrags.

Aktualisieren des Zustands einer Vorlage Da sich Vorlagen entweder im Zustand *ACTIVE* oder im Zustand *INACTIVE* befinden können, werden Vorlagen jeweils über die Transaktionen *templatePublish* bzw. *templateRevoke* auf dem jeweiligen Vorlageindex aktualisiert. Hierbei wird im ersten Fall die neue Vorlage angelegt und im zweiten Fall auf inaktiv gesetzt.

6.6.4 Private Netzwerke

Um die in 5.2.4 beschriebenen Angriffsvektoren auf das Netzwerk reduzieren, wurde zusätzlich die Möglichkeit von privaten Netzwerken eingeführt. Hierbei

soll jedoch weiterhin die Selbstorganisation durch die Nutzenden im Vordergrund stehen und das Projekt oder seine Struktur nicht zentraler (zum Beispiel durch eine Zertifikatsstelle) werden. Ziele von privaten Netzwerken sind es, die Teilnahme an dem *full node* Netzwerk einzuschränken und zusätzlich (optional) beliebige *clients* daran zu hindern, ihre Verträge und Vorlagen in dem jeweiligen Netzwerk abzuwickeln. Grundlegend wird also unterschieden, ob es sich um ein privates Netzwerk handelt und ob hier zusätzlich noch *client*-Zertifikate geprüft werden sollen. Im Folgenden soll ein solches privates Netzwerk exemplarisch aufgebaut werden, um die Änderungen an der normalen Funktionsweise der Anwendung aufzuzeigen.

Zunächst müssen sich die Teilnehmenden im Netzwerk auf einen öffentlichen Schlüssel einigen, welcher die Zertifikate des Netzwerks ausstellen darf (Zertifikationsknoten). Alle Teilnehmende nehmen diesen öffentlichen Schlüssel in ihrer eigenen Konfiguration auf. Hierzu ändern sie das Element *CertNode* in der Sektion *Networkd* (siehe 6.2).

Es ist nun die Aufgabe dieser Partei, für alle *full nodes*, die dem Netzwerk beitreten dürfen, ein Zertifikat auszustellen, welches dann bei jeder weiteren Partei im Konfigurationselement *Cert* hinterlegt wird. Das Zertifikat entspricht der Signatur der jeweiligen ID des Netzwerkknotens, durchgeführt mit dem passenden privaten Schlüssel des Zertifikationsknotens. Dieses Zertifikat wird im Rahmen der RPC Methode **GetNodeInfo** dann auch an andere Parteien im Netzwerk übermittelt.

Entsprechend ändert sich der Beitritt eines Netzwerks als *full node*. Während es in regulären Netzwerken ausreicht, die jeweilige Methode aus *Grond* zu verwenden, um Teil der DHT zu werden, ergänzt Fides den Beitrittsprozess in privaten Netzwerken wie folgt:

1. Baue eine Verbindung zu einem lokal bekannten Netzwerkknoten auf und beziehe dessen ID.
2. Vergleiche die erhaltenen Informationen mit der lokal bekannten ID.

3. Verifiziere das Zertifikat anhand der lokalen Informationen zum Zertifikationsknoten und dem von dem anderen Netzwerkknoten angegebenen Zertifikat.
4. Sofern ein valides Zertifikat vorliegt, führe den regulären Beitrittsprozess über das Modul *Grond* durch.

Für *clients* ergeben sich andere Änderungen. Sollte dort angegeben sein, dass sich die Partei in einem privaten Netzwerk befindet, wird vor der Nutzung eines Netzwerkknotens geprüft, ob sein Zertifikat korrekt ist. Andere Knoten werden ignoriert. Bei *clients* findet hingegen keine Prüfung statt, ob sie sich mit privaten Netzwerken verbinden, wenn sie selbst nicht angeben Teil eines privaten Netzwerkes zu sein. Im Allgemeinen hinterlegen *clients* ihr Zertifikat unter dem Konfigurationselement *ClientCert*. Das Zertifikat entspricht der Signatur von dem öffentlichen Schlüssel der Partei. Dieses Zertifikat wird von Netzwerkknoten geprüft, sofern jene *clients* den Zustand des Index verändern wollen. Andere Anfragen, wie zum Beispiel das Beziehen von Transaktion (sofern der Hash bekannt ist), können weiterhin durchgeführt werden. Somit sind in solchen Netzwerken zwar die Zugänge und Änderungsrechte privat, jedoch die Leserechte nicht. Sofern eine, wenn auch erschwerte, Metadatenanalyse komplett ausgeschlossen werden soll, eignen sich nur Netzwerke, die zusätzlich über einen VPN oder eine Firewall vor Fremdzugriffen abgesichert wurden. Im Allgemeinen verhindern Zertifikate für *full nodes* das Stören der Netzwerkstruktur durch angreifende Parteien, während Zertifikat für *clients* etwaigen Spam einschränken sollen. Zu möglichen Problemen der Zertifikate für *clients* zählt das Erschweren der Republikation von Objekten. Da das Zertifikat nur in der jeweiligen RPC Anfrage und nicht in der Transaktion enthalten ist, müssen Netzwerkknoten die Prüfung pro Anfrage durchführen. Entsprechend können nur eigene Transaktionen, nicht ganze Übereinkünfte, einfach republiziert werden. Es muss also gewartet werden bis die andere Partei aktiv einen Beitrag leistet, um den Zustand der Übereinkunft auf dem Index wiederherzustellen.

Zusätzlich ist es schwierig Accounts zu rotieren, da man von dem Zertifikationsknoten immer zunächst ein neues Zertifikat beziehen müsste. Allgemein ist anzunehmen, dass eine weitere Form der Kommunikation in privaten Netzwerken zwischen den enthaltenen Parteien vorhanden ist (zum Beispiel um die Zertifikate zu übermitteln). Da keine öffentliche Public-Key Infrastruktur (wie zum Beispiel bei PGP [173]) existiert, ist es somit auch nicht direkt möglich, Zertifikate zurückzuziehen oder die jeweiligen Parteien zu sperren. Abhilfe könnte hier eine Sperrliste schaffen, die ggf. mit einer Möglichkeit kombiniert wird, ein Zertifikat zurückzuziehen und die dann an alle *full nodes* verteilt wird. Der Ausblick der Arbeit greift diese Thematik für Fides im Allgemeinen erneut auf.

6.6.5 Isolation der Netzwerke

Innerhalb der Referenzimplementierung werden nicht nur *full nodes* von *clients* isoliert, sondern entsprechend auch verschiedene Fides Netzwerke und deren Teilnehmende, was im Folgenden genauer beleuchtet wird. Hierbei wird unterschieden in Überprüfungen, die die Bibliothek *Grond* durchführt, um die Struktur der DHT zu bewahren, und Überprüfungen, die die Klasse *Fides-GrondNode* zusätzlich ergänzt, indem sie die Methode **check** überschreibt. Folgende Überprüfungen werden von den Netzwerkknoten durchgeführt:

1. Prüfe, ob die angegebene ID des Knotens korrekt ist. Innerhalb der ID wird das Netzwerk nicht genannt, daher prüft der Knoten, ob der SHA-256 Hash von den erhaltenen Informationen in Kombination mit dem eigenen Netzwerknamen gleich ist. Sofern die Gleichheit gegeben ist, gibt der andere Knoten an, im gleichen Netzwerk zu sein.
2. Stelle sicher, dass der Knoten auf der gegebenen IP:Port Kombination erreichbar ist.
3. Sofern es sich um ein privates Netzwerk handelt, prüfe, ob das angegebene Zertifikat korrekt ist.

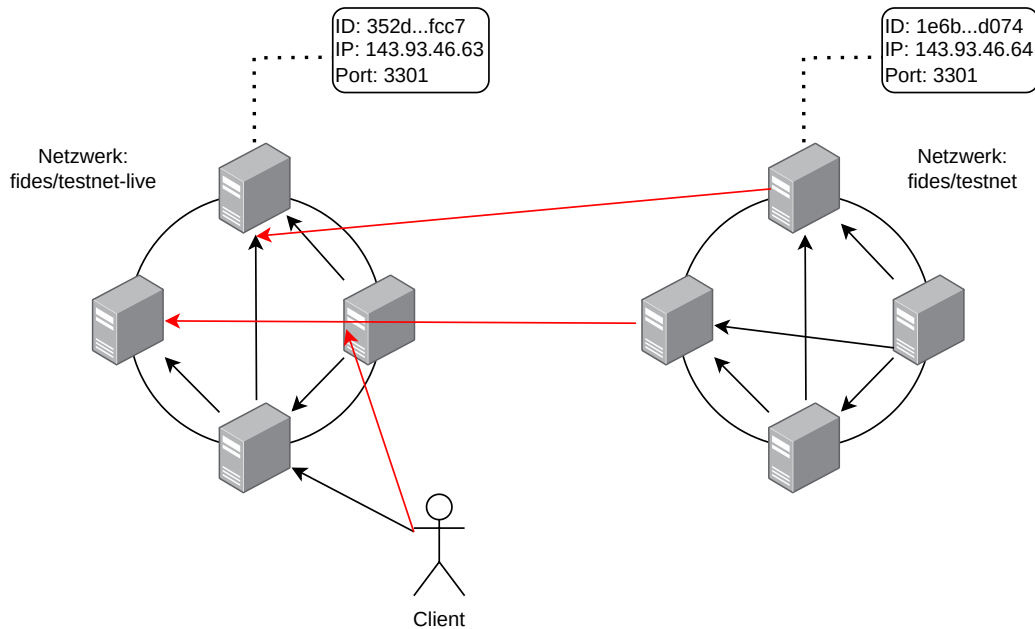


Abbildung 15: Isolation von Netzwerken und zwischen Netzwerkknoten und Nutzenden

Entsprechend ist es nur *full nodes* möglich gewisse RPC Anfragen zu stellen, da nur jene Teil der Ringstruktur sind. Etwaige *clients*, welche keine eingehenden Verbindungen erlauben und keine ID besitzen, könnten jene Anfragen nicht durchführen. Weiterhin können Netzwerkknoten in anderen Netzwerken nicht die eigene Ringstruktur ändern, wodurch die Isolation umgesetzt wird. Abbildung 15 visualisiert das Beschriebene. Rote Pfeile indizieren RPC Aufrufe, welche abgelehnt wurden. Schwarze Pfeile geben valide RPC Anfragen an. Man erkennt, dass es *clients* erlaubt ist ein Subset von Anfragen (außerhalb des Rings) zu stellen, während nur *full nodes*, die Teil des Rings sind, Anfrage innerhalb des Rings stellen können.

Zusätzlich prüfen Nutzende vor dem Senden von Transaktionen die ID von den Knoten, indem sie folgende Schritte durchführen:

1. Prüfe die allgemeine Erreichbarkeit des zufällig ausgewählten Knotens, indem dessen ID über die RPC Methode **GetNodeInfo** abgefragt wird.

2. Sofern der Knoten erreichbar ist, prüfe ob die angegebene ID dem lokalen Datenbestand entspricht (eigene Berechnung der ID beim Hinzufügen des Netzwerkknotens).
3. Handelt es sich um ein privates Netzwerk, prüfe, ob der Netzwerkknoten ein korrektes Zertifikat besitzt.

6.7 Verschlüsselung

Zur Verschlüsselung von Eingabedaten werden einige zuvor beschriebene Konzepte aufgegriffen: Temporäre öffentliche Schlüssel der Nutzenden, AES-256 (siehe 3.1) und HKDF (siehe 3.4).

Da in Fides für jede Übereinkunft neue temporäre Accounts $\mathcal{A}_\mathcal{E}$ für beide Parteien verwendet werden, kann „forward secrecy“ [68] pro Vertrag sichergestellt werden. Dies bedeutet, dass, sofern ein privater, temporärer Schlüssel innerhalb des Vertrags offengelegt wird, man zwar Zugriff auf jene Übereinkunft hat, jedoch nicht auf die vorherigen oder die nachfolgenden. Hier sei erneut erwähnt, dass jene temporären Schlüssel verschlüsselt gespeichert werden. Somit bedeutet ein Verlust des Schlüssels noch nicht unmittelbar den Zugriff für Angreifende. Betrachtet man den Ablauf für die Partei i , welche einen Vertrag mit Partei j eingeht, werden folgende Aktionen durchgeführt, um die Eingabedaten zu verschlüsseln:

1. Erstellen und Veröffentlichen eines Vertrags $\mathcal{C}_{i,j}$. Innerhalb der Transaktion ist der temporäre öffentliche Schlüssel der Partei $\mathcal{K}_{\mathcal{E},i,P}$. Das Passwort zu diesem Account entspricht der Signatur von $\mathcal{H}_{\mathcal{C}_{i,j}}$ mit dem privaten Schlüssel $\mathcal{K}_{i,S}$ des Accounts A_i .
2. Annahme des Vertrags durch die andere Partei j . Nun kennen beide Parteien den temporären öffentlichen Schlüssel der anderen Partei, da die zweite Transaktion den öffentlichen Schlüssel $\mathcal{K}_{\mathcal{E},j,P}$ enthält.

3. Bei der Bestätigung der Vertragsschritte wird nun zunächst ein Diffie-Hellman-Schlüsselaustausch durchgeführt. Diese Geheimnis dient dann als *IKM* innerhalb von HKDF-Extract (2). Als *salt* wird $\mathcal{H}_{C_{i,j}}$ verwendet. Der nun erhaltene pseudozufällige Schlüssel *PRK* wird innerhalb der kompletten Übereinkunft benutzt, um weitere Schlüssel abzuleiten.
4. Aufrufen von HKDF-Expand (3), um den aktuellen Schlüssel abzuleiten. Hierzu dient der Hash der letzten Transaktion \mathcal{H}_{T_l} jeweils als *info* Parameter. Die Länge L entspricht 32, um mit dem durch HKDF-Expand berechneten Schlüssel *OKM* AES-256 [72] zu verwenden.
5. Innerhalb der eigentlichen Verschlüsselung wird nun AES-256 [72] mit dem Modus CBC [75] verwendet. Hierzu wird zunächst ein zufälliger Initialisierungsvektor *iv* berechnet, danach die Eingabedaten aufgefüllt (PKCS #7-Padding [118]) und schließlich werden die Daten derart verschlüsselt.
6. Der zufällige Initialisierungsvektor wird mit den verschlüsselten Daten dann in die jeweilige Transaktion, die einen Vertragsschritt betätigt, eingefügt.

Abbildung 16 verdeutlicht die Verschlüsselung während der Bearbeitung von Verträgen.

6.7.1 Angriffsvektoren auf die Verschlüsselung

Um das in der Referenzimplementierung eingesetzte Verschlüsselungsverfahren AES-CBC sicher benutzen zu können, müssen einige mögliche Angriffsvektoren betrachtet werden, die im Folgenden erläutert werden sollen.

Bit Flip Der in Fides verwendete Modus CBC, welcher in Kombination mit AES in Kapitel 3.1 vorgestellt wurde, ist anfällig für sogenannte Bit Flipping Angriffe [146] [73]. Nachfolgend wird die Art von Angriff an einem

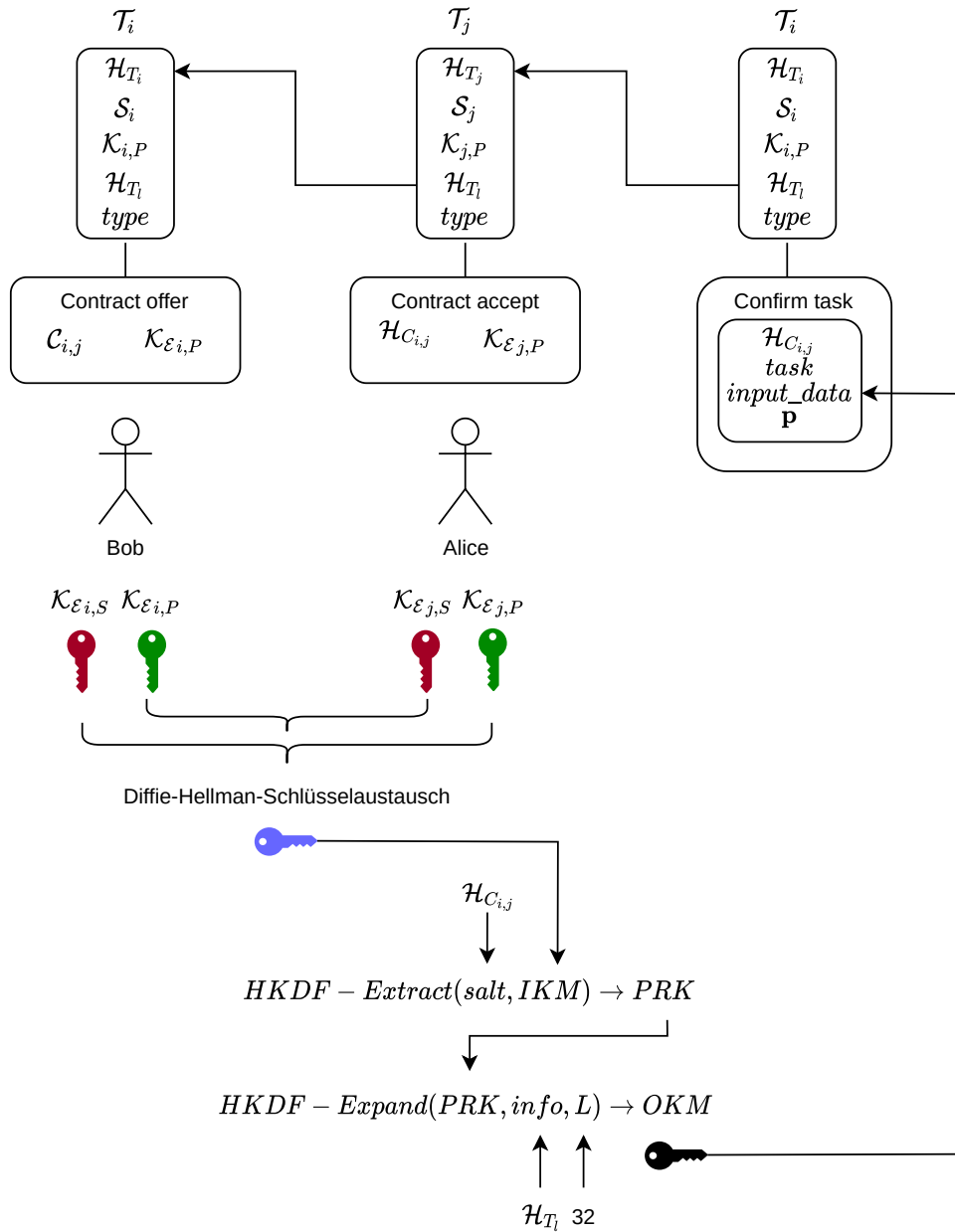


Abbildung 16: Verschlüsselung innerhalb der Bearbeitung von Verträgen in Fides

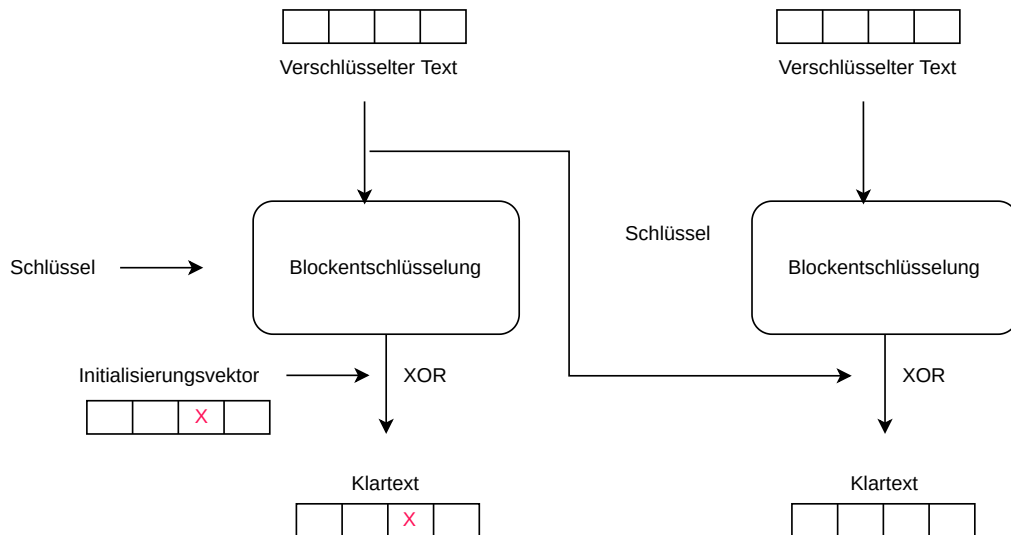


Abbildung 17: Bit Flip Angriff auf AES-CBC Verschlüsselung

praktischen Beispiel erläutert und dann das Gefahrenpotential innerhalb von Fides eingeschätzt. Für das Beispiel nehmen wir folgendes Szenario an: Alice möchte Bob einen gewissen Geldbetrag überweisen. Hierzu sendet sie eine Nachricht an ihre Bank. Die Nachricht wird von Eve abgefangen und manipuliert, ohne dass sie den Schlüssel zwischen Alice und ihrer Bank kennt.

Um Bit Flipping Angriffe erfolgreich durchführen zu können, ist es von Vorteil, Informationen über die Struktur der Nachricht zu bewahren. Nehmen wir weiterhin an, dass Eve selbst Kundin bei der Bank ist und das Format kennt. Sie nutzt jenes Wissen, um den Angriff durchzuführen. Zunächst bereitet Alice ihre Nachricht vor und verschlüsselt jene symmetrisch mit einem Schlüssel, der nur ihr und ihrer Bank bekannt ist. Um das Beispiel anhand der konkreten Implementierung zu unterstreichen, verwenden wir die zuvor eingeführten Implementierungen von Fides:

```

from fides.core.encryption import encrypt
key = "24ffa522cf222afb6c1774a8cd872fac"\
      "d3201fd784598b990bbcbedad633e3f5"
iv, ciphertext = encrypt(bytes.fromhex(key),
                          b"100 Euro -> Bob")

```

Auflistung 9: Bit Flipping Angriff: Verschlüsselung der Eingabe

Nun überträgt Alice die Kombination aus IV und verschlüsseltem Text an ihre Bank. Möchte nun Eve die Nachricht manipulieren, kann sie die Inhalte des Initialisierungsvektors verändern, wodurch sich ebenso die Inhalte des entschlüsselten Texts verändern werden. Sollten sich wichtige Daten im ersten Block der Nachricht befinden, kann ein solcher Angriff problemlos durchgeführt werden, ohne erkannt zu werden. Erst das Verändern von späteren Blocks kann (in Abhängigkeit von etwaigem Padding) bei der Entschlüsselung zu Fehlern führen. Eve passt nun also den IV der Nachricht an, indem sie die jeweiligen Bits „flipt“:

```

new_iv = list(iv)
new_iv[0] = ord(chr(new_iv[0]^3))
new_iv[12] = ord(chr(new_iv[12]^7))
new_iv[13] = ord(chr(new_iv[13]^25))
new_iv[14] = ord(chr(new_iv[14]^7))
new_iv = bytes(new_iv)

```

Auflistung 10: Bit Flipping Angriff: Änderung der verschlüsselten Daten

Wird dieser manipulierte IV nun mit der alten, unveränderten Nachricht an die Bank übertragen, entschlüsselt jene die Nachricht zu:

```
'200 Euro -> Eve'
```

Entsprechend würden nach korrekter Entschlüsselung nicht 100 Euro an Bob, sondern 200 Euro an Eve überwiesen werden.

Die Änderung des IV, welcher nach der Entschlüsselung verworfen wird, ruft somit eine Änderung des ersten Klartext Blocks hervor, ohne dass es zu Fehlern kommt. Das Beispiel sollte auf einfache Weise illustrieren, dass selbst sichere Methoden der Verschlüsselung oft nicht ausreichen, wenn der Nachrichtenaustausch nicht authentifiziert stattfindet. Ein Bit Flipping Angriff wäre in Fides nur mit Kontrolle des lokalen Systems möglich, was bedeuten würde, dass der Rechner bereits kompromittiert wurde und es dadurch entsprechend auch einfachere Methoden gäbe Angriffe durchzuführen. Die verschlüsselten Daten, welche innerhalb von Fides in Transaktionen des Typs „Confirm task“ übermittelt werden, sind im Ganzen signiert. Eine Änderung der Daten bei der Übertragung würde die Signatur ungültig machen, wodurch die Transaktion nicht auf den lokalen Zustand der jeweiligen Partei angewendet werden würde. Der folgende Auszug verdeutlicht das Beschriebene, indem die Funktion `verify_tx` verwendet wird, um anzuzeigen, ob es sich um eine korrekte Transaktion handelt:

```
from fides.core.transaction_impl import db_load_tx, \
                                         verify_tx

tx = db_load_tx("0c47 [...]5557")
tx.confirmTask.input_data
# Ausgabe: 'UKtoZ [...]2DxWH/w==:uKIOHP [...]mVvLQ=='
verify_tx(tx)
# Ausgabe: True
tx.confirmTask.input_data = "Bit flip"
verify_tx(tx)
# Ausgabe: False
```

Auflistung 11: Bit Flipping Angriff: Lokale Validierung einer Transaktion

Transaktionen werden immer auf ihre Gültigkeit geprüft, auch wenn sie bereits Teil der lokalen Instanz sind. Somit sind nachträgliche Manipulationen erkennbar. Dies ist jedoch nur hilfreich, wenn unprivilegierte Nutzende an-

gegriffen werden, die eine systemweite, nicht änderbare Version von Fides verwenden. Hat eine angreifende Partei so die Möglichkeit Transaktionsdaten der lokalen Instanz zu verändern, jedoch keinen Zugriff auf den Code von Fides, so wird die erneute Validierung die Manipulation erkennen. Ist es einer angreifenden Partei jedoch möglich, den Code von Fides zu überschreiben, kann auf einem kompromittierten System keine Abhilfe geschafft werden.

Padded Oracle Ein Padded Oracle Angriff erlaubt das Entschlüsseln einer kryptographischen Nachricht, indem durch ein Orakel Auskünfte über (in)valides Padding bzw. Fehlermeldungen im Allgemeinen bezogen werden können. Die erste Arbeit hierzu wurde von Vaudenay publiziert [166]. Im Folgenden wird ein solcher Angriff exemplarisch dargestellt und die Anwendbarkeit im Kontext der Implementierungen von Fides bewertet. Unser Beispiel orientiert sich an [76] und trifft folgende Annahmen: Gegeben sei ein Server, welchem eine verschlüsselte Nachricht (AES-CBC) übermittelt wird. Diese Nachricht könnte zum Beispiel ein Cookie sein, welcher Daten über die Nutzenden enthält. Jener Server versucht diese Nachricht zu entschlüsseln, reagiert bei Fehlern jedoch mit entsprechender Meldung (zum Beispiel „invalid padding“) auf die Anfrage. Weiterhin nehmen wir für das Beispiel an, dass die Kommunikation lediglich verschlüsselt, jedoch nicht authentifiziert wird. Die Existenz des Orakels liegt unmittelbar auf der Hand: In Abhängigkeit der Nachricht antwortet der Server mit den notwendigen Informationen um einen Padding Oracle Angriff durchführen zu können. Innerhalb des Beispiels wollen wir nun die Nachricht entschlüsseln. Dies könnte, wie bereits erwähnt, unser eigener Cookie sein oder es kann sich um eine abgefangene Nachricht einer anderen Partei handeln, die ebenso mit dem Server kommunizierte. Im ersten Schritt des Angriffs verwenden wir für den Initialisierungsvektor lediglich null Bytes (0x00). Aus Abbildung 2 ergibt sich, dass der Server nun unsere verschlüsselte Nachricht mit dem korrekten Schlüssel entschlüsseln wird und danach mit unserem veränderten Initialisierungsvektor XOR verknüpft, woraus ein anderer

Klartext resultiert. Allgemein wird, zum Beispiel in Fides, zunächst die Entschlüsselung durchgeführt und danach das Padding überprüft. Anhand der Fehlermeldung des Servers können wir unseren Initialisierungsvektor dann Byte für Byte anpassen, bis wir eine korrekte Antwort des Servers erhalten. Im Falle von PKCS #7, dem Padding Schema was auch in Fides verwendet wird, endet unsere entschlüsselte (falsche) Nachricht auf 0x01 (allgemein $n * 0xn$ Byte, hier $1 * 0x01$). Nach [76] kann äquivalent für die nächsten Bytes des Initialisierungsvektors vorgegangen werden, bis die entschlüsselte Nachricht nur aus null Bytes (0x00) besteht. Hierbei werden die gefunden Bytes des manipulierten Initialisierungsvektors mit der im nächsten Schritt gesuchten Padding Länge XOR verknüpft. Durch die Symmetrie von XOR entspricht unser manipulierter Initialisierungsvektor am Ende des Padded Oracle Angriffs der Ausgabe der Entschlüsselungsoperation, die wiederum mit einem verschlüsselten Block oder dem Initialisierungsvektor XOR verknüpft werden kann, um den jeweiligen Klartextblock zu erhalten. Nachdem die Angriffsart anhand eines einfachen Beispiels erklärt wurde, soll nun noch auf die Art der Fehlermeldung des Servers eingegangen werden und wie selbst hier eine Verschleierung des Fehlers trotzdem zu erfolgreichen Padded Oracle Angriffen führen kann. Selbst wenn der Server keine klare Auskunft darüber gibt, ob der Fehler bei der Bearbeitung der Nachricht durch das Entpacken oder durch die Entschlüsselung entstanden ist, kann durch das Beobachten der Anfragen darauf geschlossen werden, welche Operation durchgeführt wurde. Da im Allgemeinen zunächst entschlüsselt wird, bevor das Padding entfernt wird, dauert die Operation das Padding zu prüfen länger als die Entschlüsselung. Ist somit eine angreifende Partei in der Lage die Antwortzeiten genau zu messen, können Padding Oracle Angriffe auch ohne genauere Informationen leicht durchgeführt werden. Abschließend soll die Angriffsart im Kontext der Referenzimplementierung eingeordnet werden. Wie bereits bei dem Bit Flipping Angriff beschrieben, werden die verschlüsselten Daten authentifiziert übertragen. Vor jeder Entschlüsselung wird jene Signatur geprüft, was jede

Manipulation beim Initialisierungsvektor oder bei der verschlüsselten Nachricht erkennbar macht. Weiterhin existieren innerhalb von Fides keine zentralen Systeme, die per se die Schlüssel zur Entschlüsselung kennen und somit auch keine Orakel. Die Entschlüsselung der Daten findet bei beiden Parteien nur lokal statt, wodurch keine Gefahr durch Padded Oracle Angriffe besteht.

6.8 Automatisierung

Nachfolgend werden die Möglichkeiten zur Automatisierung von Anwendungen durch Fides beleuchtet. Bezogen auf die Abbildung 4, die Ricardian mit Smart Contract vergleicht, adressiert die Automatisierung den „smarten“ Bereich.

6.8.1 Allgemeines Vorgehen

Im Allgemeinen enthält die Referenzimplementierung das Modul Hooks, welches die Entwicklung von Anwendungen, die Fides zur Kommunikation nutzen oder die Verträge auf sonstige Arten einbinden, erleichtern soll. Grundlegend sind Hooks gebunden an ein Objekt (Vertrag oder Vertragsvorlage) und rufen eine zuvor definierte, beliebige Funktion einer Python Anwendung auf. Die Methoden, die als Callback verwendet werden können, müssen eine variable Anzahl an Argumenten akzeptieren. Für die Automatisierung stehen fünf verschiedene Hooks zur Verfügung, die von einem einfachen Interface abgeleitet sind, welches die Kernkomponenten einer Hook definiert:

Weiterhin enthalten die verfügbaren Hooks ein Intervall, angegeben in Sekunden, in welchem die Bedingung der jeweiligen Hook geprüft wird. Nachfolgend werden die verfügbaren Hooks, die an das jeweilige Callback übergebenen Parameter und ihr Einsatzbereich beschrieben:

- **TemplateUsedHook:**

Argumente: *contract*

Hook Element	Beschreibung
cb	Auszuführendes Callback (beliebige Methode einer Python Anwendung)
called	Zähler, ob bzw. wie oft das Callback aufgerufen wurde
live_forever	Flag, das manche Hooks „ewig“ ausführt (nachdem das Callback bereits aufgerufen wurde)
**args	Variable Anzahl an Argumenten, die bei der Konstruktion der Hook definiert werden und durch die Hook selbst vor der Ausführung erweitert werden

Tabelle 11: Kernkomponenten des Hook Interfaces

Beschreibung: Sofern ein neuer Vertrag (Zustand *OFFER*) die angegebene Vorlage verwendet, wird das zuvor definierte Callback aufgerufen. Hierbei wird der Hash des Vertrags über das Argument an die Funktion übermittelt. Die Hook kann verwendet werden, um die jeweilige Vorlage komplett zu automatisieren. Entsprechend eignet sich hier das *live_forever* Flag, um nicht nur einen Vertrag pro Ausführung der Hook zu erhalten.

- **ContractAcceptedHook:**

Argumente: *contract*

Beschreibung: Prüfung, ob der angegebene Vertrag angenommen wurde. Sollte der Vertrag abgelehnt werden, wird die Hook beendet ohne das Callback auszuführen. Alternativ wird der Hash des Vertrags an das Callback übergeben, damit die jeweilige Automatisierung die Zuordnung schaffen kann, welcher Vertrag angenommen wurde.

- **ContractRejectedHook:**

Argumente: *contract*

Beschreibung: Äquivalent zur Prüfung, ob der Vertrag angenommen wird mit der Prüfung ob, jener abgelehnt wurde.

- **ContractCompleteHook:**

Argumente: *contract*

Beschreibung: Wird ausgeführt, wenn der Vertrag den Zustand *FINISHED* erreicht. Kann zum Beispiel genutzt werden, um danach einen automatischen Export der Daten anzustoßen oder die jeweiligen Schlüssel zu löschen.

- **ContractStateChangedHook:**

Argumente: *last_tx*

Beschreibung: Sofern das Flag *live_forever* gegeben ist, wird das Callback für jede Änderung des angegebenen Vertrags ausgeführt. Hierbei wird die Transaktion an das Callback übergeben, die für die jeweilige Änderung zuständig ist. Die Hook eignet sich, um den kompletten Ablauf eines Vertrags zu automatisieren, zum Beispiel in Kombination mit der *TemplateUsedHook*.

6.8.2 Wiederverwendbarkeit durch Aufgabendefinition

Durch den Aufbau der Transaktionen und die Definition einer Vorlage ist es möglich, gleiche Aufgaben von verschiedenen Vorlagen äquivalent zu automatisieren. Nimmt man zum Beispiel die folgende Aufgabe einer Vorlage an, die der Partei zugeordnet ist, die den Vertrag erstellt: „Mitteilung der Lieferadresse im Format: PLZ, Straße, Hausnummer“. Nicht nur kann das korrekte Format durch die bereits erörterten Validatoren forciert werden, sondern auch die gleiche Aufgabe Teil von verschiedenen Vertragsvorlagen sein. Die Aufgabe (Definition 8), also der Hash der Beschreibung, wäre für die unterschiedlichen Vorlagen gleich. Somit kann auf eine Änderung innerhalb eines Vertrags gleich reagiert werden.

6.8.3 Import und Export

Um Hooks der Einfachheit halber direkt aus dem Command Line Interface zu starten, existiert innerhalb der Referenzimplementierung sowohl eine Funk-

tion zum dynamischen Laden von Hooks als auch eine Funktion, um eine Hook in dem vorgegebenen Format innerhalb der aktuellen Fides Instanz zu speichern.

Bei dem Export werden die Informationen aus dem Hook Objekt extrahiert und im json Format abgelegt. Um das Callback, welches ggf. dynamisch zur Laufzeit in der Hook festgelegt wurde, korrekt zu speichern, wird die Kernbibliothek *inspect.getfile* [89] der Python Umgebung verwendet, um den genauen Pfad des aufzurufenden Callbacks in der Datei abzulegen. Im Folgenden wird exemplarisch eine Hook vom Typ *TemplateUsedHook* gezeigt, um das Beschriebene und das allgemeine Format zu verdeutlichen:

```
{
  "type": "TemplateUsedHook",
  "hash": "35c89849 [...]"
  "interval": 5,
  "callback": "/home/lars/code/hooks_fides/hook.py",
  "method": "contract_created",
  "live_forever": true,
  "args": {"automation_type": "Custom Argument"}
}
```

Auflistung 12: Dateistruktur einer Hook

Wie bereits beschrieben, enthält die Datei alle notwendigen Informationen, um die Hook zu rekonstruieren. Überwacht wird das Objekt, welches durch den Hash *35c89849[...]* identifiziert wird. Hierbei wird alle 5 Sekunden geprüft, ob neue Verträge vorhanden sind, welche die angegebene Vorlage verwenden. Sollte ein neuer Vertrag gefunden worden sein, wird die Methode *contract_created* aus der Datei, welche in der Sektion *callback* hinterlegt ist, mit den Argumenten der Hook und den in der Hook angegebenen (hier: *automation_type*) aufgerufen.

Für den Import einer exportieren Hook werden die Bibliothek *import-lib.util* [85] und die eingebaute Methode *getattr* [80] verwendet. Allgemein werden die folgenden Schritte in *fides.hooks.parse_hook* durchgeführt:

1. json Datei lesen
2. *ModuleSpec* [86] aus der Datei der Sektion *callback* erstellen
3. Modul aus der Spezifikation erstellen [87]
4. Modul ausführen [88]
5. *getattr* [80] auf Spezifikation ausführen mit in der Hook Datei angegebenen Methode
6. Datentyp der Hook durch *getattr* [80] und der Angabe *type* aus der Hook Datei bestimmen
7. Hook Objekt anlegen und zurückgeben

6.8.4 Beispiel einer Automatisierung

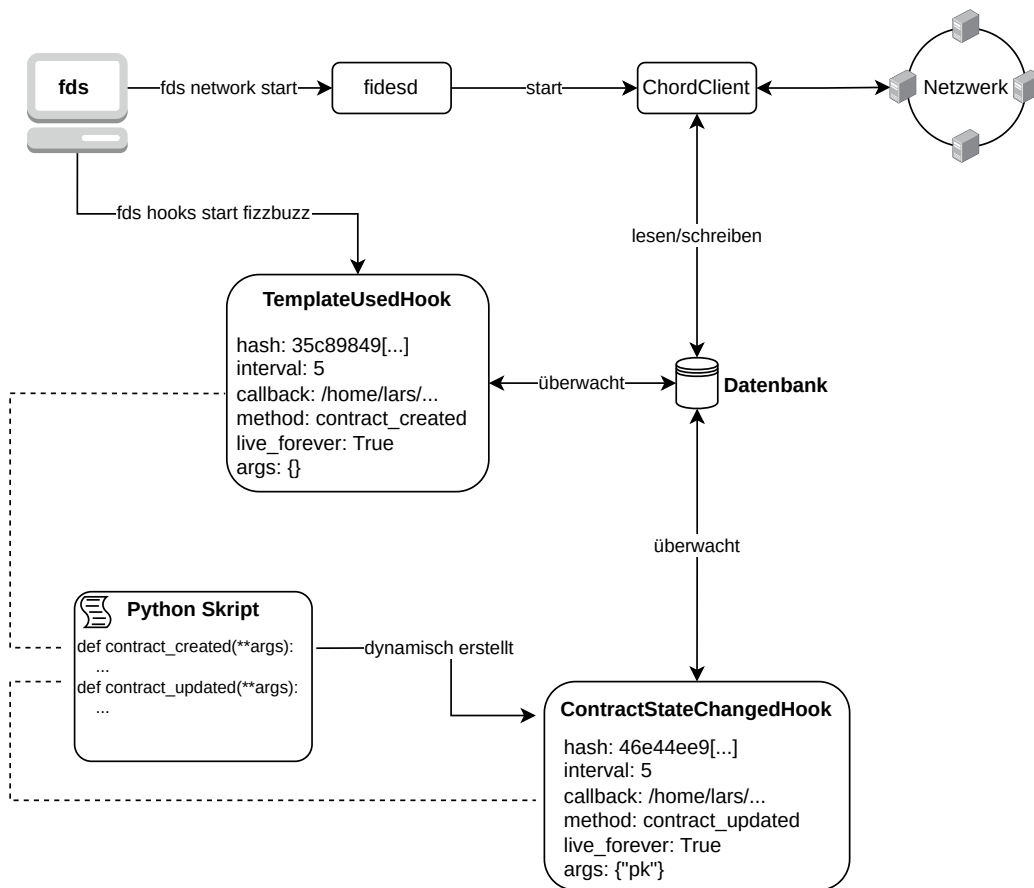
Nachfolgend werden anhand eines Beispiels die Abläufe bei einer möglichen Automatisierung mittels Fides durch Hooks gezeigt. Hierzu wird die *TemplateUsedHook* verwendet, um durchgängig neue Verträge zu überwachen und im Anschluss wird die *ContractStateChangedHook* benutzt, um auf die jeweiligen Zustandsänderungen des Vertrags zu reagieren. Konkret implementiert wird die Programmieraufgabe „FizzBuzz“, bei welcher eine ganzzahlige Eingabe geprüft wird. Bei einer Teilbarkeit durch 3 lautet die Ausgabe „Fizz“, bei einer Teilbarkeit durch 5 „Buzz“ und wenn beide Bedingungen zutreffen „Fizz-Buzz“. Das Vorgehen, welches in Abbildung 18 verdeutlicht ist, besteht exemplarisch aus einem Python Skript, welches die Automatisierung durchführt. Innerhalb dieses Skripts werden zwei Methoden verwendet: *contract_created* dient als Callback für die *TemplateUsedHook* und *contract_updated* wird verwendet, um dynamisch erstellte Hooks des Typs *ContractStateChangedHook*

abzuwickeln. Wird nun das Command Line Interface *fds* verwendet, um die Automatisierung, somit die *TemplateUsedHook*, zu starten, prüft jene den Zustand der lokalen Datenbank, welche unter anderem verändert wird, wenn *fds* bei einer Hintergrundaktualisierung zum Beispiel neue Verträge bezieht, und reagiert bei einer treffenden Änderungen mit der Ausführung des genannten Callbacks. Innerhalb der Methode des Callbacks wird nun zur vollständigen Automatisierung dynamisch eine neue Hook erzeugt, um diesen neuen Vertrag überwachen zu können. Entsprechend einfach ist es somit, die kompletten Abläufe einer Vorlage zu automatisieren. Gleichwohl ist es sehr einfach möglich, den Zugang zu den eigenen Vorlagen zu beschränken. Möchte man zum Beispiel, dass nur eine bestimmte Nutzergruppe die Vorlage verwenden kann, so kann man durch die Automatisierung Verträge von anderen Parteien innerhalb des Callbacks der *TemplateUsedHook* unmittelbar ablehnen.

Nachfolgend werden der Vollständigkeit halber die Implementierungen des „FizzBuzz“ Beispiels vorgestellt. Das benutzte Template enthält die folgenden Aufgaben (Auszug aus *fds template show*):

```
1) Provide integer i with 0 <= i < 1000 ---> sender
   Validator: RANGE
   Type: int
   0 <= INPUT < 1000
2) Respond with Fizz, Buzz or FizzBuzz ---> receiver
```

Auflistung 13: Beschreibung einer Vorlage innerhalb der Kommandozeilenanwendung (Auszug)

Abbildung 18: Exemplarische Automatisierung eines Vertrags durch *Hooks*

Die Methode *contract_created* wurde folgendermaßen implementiert:

```

def contract_created(**args):
    c = Contract()
    c.db_load(args["contract"])

    pk = load_key(ACCOUNT_PATH + "default",
                 os.environ.get("FIDES_PW"))

    contract_hook = ContractStateChangedHook(
  
```

```
        hash=c.hash ,
        interval=5,
        live_forever=True ,
        callback=contract_updated ,
        **{"pk": pk, "contract":
        c.hash})
Thread(target=contract_hook.start).start()

while True:
    if not c.accept(pk):
        print("Accept failed, try again")
        sleep(2)
    else:
        break
```

Auflistung 14: Automatisierung: Callback Methode bei neuen Vertragsangeboten

Zunächst wird bei der Methode *contract_created* der für das Template zuständige private Schlüssel des Accounts geladen. Konkret wurde das Passwort des Accounts hier mit einer Umgebungsvariablen übermittelt. Gleichwohl könnte das Passwort beim ersten Start der Hook zur Laufzeit eingegeben oder aus einer Datei gelesen werden. Neben dem privaten Schlüssel wird noch das lokale Contract Objekt geladen, welches den Aufruf des Callbacks auslöste. Der Hash des Vertrags wurde von der *TemplateUsedHook* als variables Argument an die Methode übergeben. Bevor der Vertrag schließlich akzeptiert wird (in diesem Beispiel werden etwaigen Fehlern bei der Übertragung vorgebeugt, indem dies in einer Schleife geschieht), wird eine dynamische Hook des Typs *ContractStateChangedHook* erzeugt, die über die gesamte Lebensdauer des Vertrags (bis zum Status *FINISHED*) ausgeführt wird. Jene Hook prüft alle fünf Sekunden, ob sich der Zustand des Objekts geändert hat und ruft bei einer Änderung die Methode *contract_updated* auf und übergibt jener den

geladenen privaten Schlüssel als variables Argument *pk*. Die Methode *contract_updated* ist folgendermaßen implementiert:

```
def contract_updated(**args):
    tx = db_load_tx(args["last_tx"])
    if tx.sender == get_public_key(args["pk"]):
        return
    c = Contract()
    c.db_load(args["contract"])

    tx_data, valid = c.get_tx_data(args["last_tx"],
                                   args["pk"])

    if not valid:
        while True:
            if not c.confirm(private_key=args["pk"],
                             input_data="Wrong input!"):
                print("Could not confirm, try again")
                sleep(2)
            else:
                break
    else:
        result = check_fizzbuzz(int(tx_data))
        while True:
            if not c.confirm(private_key=args["pk"],
                             input_data=result):
                print("Could not confirm, try again")
                sleep(2)
            else:
                break
```

Auflistung 15: Automatisierung: Callback Methode bei Vertragsaktualisierungen

Zunächst wird die Transaktion geladen, welche die Zustandsänderung ausgelöst und das Callback aufgerufen hat. Danach folgt die Überprüfung ob die Zustandsänderungen von der lokalen Instanz durchgeführt wurden. In unserem konkreten Fall passiert dies zunächst durch die Transaktion, welche den Vertrag akzeptiert. Für unser Beispiel ist es ausreichend, nicht auf eigene Änderungen des Objekts einzugehen. Handelt es sich um eine Transaktion der anderen Partei, so wird der Vertrag mittels des variablen Arguments *contract* geladen. Danach folgt die lokale Überprüfung der Transaktion. Die verwendete Vorlage nutzt einen Validator, welcher sicherstellen soll, dass nur ganzzahlige Eingaben in dem vorgegebenen Bereich gemacht werden, was hier geprüft wird. Sollte es sich um eine bewusst falsche Eingabe handeln (über die API gesendet), findet die Berechnung von „FizzBuzz“ nicht statt und die zweite Aufgabe des Vertrags wird mit einer Fehlermeldung bestätigt. Bei korrekter Eingabe wird jene lokal geprüft und das Ergebnis mit der Bestätigung der Aufgabe übermittelt. In beiden Fällen werden die Bestätigungen in einer Schleife ausgeführt, um etwaigen Übertragungsfehlern entgegenzuwirken und die Automatisierung im Allgemeinen robuster zu machen. Das Beispiel sollte verdeutlichen, wie einfach es ist, mit Fides eine volle Automatisierung mit Templates und Contracts durchzuführen. Weiterhin kann man erkennen, dass in ähnlicher Weise auch Teilautomatisierungen möglich werden, die es erlauben, Prozesse nach und nach zu automatisieren. So können einzelne Vertragsschritte automatisiert, andere durch händische Eingaben den Zustand der Objekte verändern. Die gemachten Implementierungen lassen sich einfach in einem lokalen Testnetzwerk verifizieren, bevor sie durch das Umschalten des Netzwerks innerhalb der Konfiguration in ein reales Fides Netzwerk übertragen werden. Weiterhin sind Änderungen bei der Automatisierung möglich, auch wenn jene schon genutzt werden. Dies bleibt der anderen Partei verborgen und erlaubt die kontinuierliche Verbesserung der eigenen Systeme und Software, was im Falle von regulären Smart Contracts nicht möglich wäre, da jene nach der Veröffentlichung in der Regel nicht mehr

änderbar sind.

6.9 Zusammenfassung

Das zuvor vorgestellte *Cypher Social Contracts* Protokoll wurde innerhalb der Referenzimplementierung Fides vollständig implementiert. Die Referenzimplementierung wird zusätzlich ergänzt um eine Kommandozeilenanwendung, die die reine Verwendung der Software ermöglicht und somit direkt verwendet werden kann. Hier sei anzumerken, dass die Verträge zwar von nicht-technischen Personen verstanden werden, jene jedoch wahrscheinlich nicht vertraut sind mit der Bedienung einer Kommandozeilenanwendung. Dennoch können die gemachten Implementierungen aus dem Command Line Interface unmittelbar z. B. auf grafische Anwendungen, die einfacher zu nutzen sind, übertragen werden, was deren Entwicklung vereinfacht. Neben dem Funktionsumfang und den diversen Einstellungsmöglichkeiten der Software beleuchtet das Kapitel auch vollständig die API zum technischen Verständnis des Systems. Weiterhin werden einige Erweiterungen, die bei der fortlaufenden Verbesserung der Implementierung realisiert wurden, beschrieben. Hierzu zählen unter anderem private, dezentrale Netzwerke, fernab der Annahme einer (offenen) verteilten Hashtabelle.

Zudem werden die Verschlüsselung, Automatisierung und Umsetzung der Indizes, welche jeweils von der Nutzungsart abhängen und daher im Protokoll generisch beschrieben sind, im Detail erläutert.

Abschließend wird der Bereich der IT-Security ganzheitlich über das gesamte Kapitel hinweg betrachtet und es werden in verschiedenen Bereichen mögliche Angriffsvektoren beschrieben, eingeordnet und Designentscheidungen der Implementierung erklärt.

7 Verträge ohne Internetverbindung

Im Folgenden wird die Implementierung zu einer Abstraktion von Fides beschrieben, die es erlaubt, über LoRaWAN in Verbindung mit dem The Things Network [163] mit geringen Einschränkungen an Verträgen teilnehmen zu können. Die Abstraktion erlaubt die Selbstorganisation und digitale Teilhabe auch in Regionen ohne Internetverbindung, sofern eine LoRaWAN Abdeckung gegeben ist.

7.1 Allgemeines

Bei dem „Long Range Wide Area Network“ (LoRaWAN) handelt es sich um eine Spezifikation eines „Low Power, Wide Area“ (LPWA) Netzwerkprotokolls, welches von der LoRa Alliance vorgegeben wird [11]. LoRaWAN wurde insbesondere für den Internet of Things (IoT) Bereich entwickelt und unterstützt sowohl bi-direktionale Kommunikation als auch Ende-zu-Ende Sicherheit [11].

Innerhalb des IoT Bereichs wird LoRaWAN oft verwendet, um Sensordaten zu übermitteln. In [64] untersuchten die Autoren LoRaWAN Anwendungsfälle von 71 Arbeiten. Primär wurde LoRaWAN im Bereich von smarten Städten/Farmen und im Gesundheitsbereich eingesetzt. Allgemein wird LoRaWAN in den untersuchten Arbeiten verwendet, um Sensorwerte aus verschiedenen Kontexten zu überwachen, auf welchen dann weiterführende Anwendungen fußen. Weitere Arbeiten wie [52] und [108] implementieren nachrichtenbasierte Dienste, jedoch wird hier LoRa direkt verwendet und nicht LoRaWAN in Kombination mit dem The Things Network. LoRaWAN wird daher kaum als Abstraktion für eine Anwendung, bzw. als Kernanwendung selbst, verwendet. Wir definieren eine Abstraktion als aktive Partizipation an einem System. Exemplarisch lässt sich das an [43] zeigen, wo LoRaWAN benutzt wurde, um Parkplätze zu überwachen. Die gewonnenen Daten werden dann von einer Webanwendung aufbereitet verwendet, um zum

Beispiel einen Parkplatz reservieren zu können. Eine LoRaWAN Abstraktion für dieses Problem wäre die aktive Teilnahme (z. B. das Reservieren eines Parkplatzes) über ein LoRaWAN Gerät.

Für die meisten Anwendungen ist die direkte Verwendung von LoRaWAN wegen der geringen Datenrate problematisch [132]. Bei der Verwendung des The Things Networks greift zudem noch die sogenannte „Fair Access Policy“ [164], welche die Uplink-Übertragungszeit auf 30 Sekunden und die Anzahl an Downlink Nachrichten auf 10 pro Tag pro Gerät beschränkt. Allgemein gilt innerhalb des The Things Networks, dass die anwendungsspezifischen Nachrichten (uplink) 12 Bytes nicht überschreiten sollten [40].

7.2 Übersicht

Nachfolgend wird eine Übersicht über die LoRaWAN Abstraktion für Fides gegeben, welche die zuvor genannten Limitierungen adressiert und skizziert, wie dennoch mit geringen Einschränkungen an Verträgen ohne Internetverbindung teilgenommen werden kann.

Aufgrund der genannten Limitierungen ist es nicht ratsam komplette Fides Transaktionen über LoRaWAN zu übertragen. Betrachtet man die minimale Größe einer Transaktion zum Erstellen eines Vertrags (bei der Nutzung einer Vorlage mit nur einer Aufgabe), so hat jene Transaktion eine Größe von ungefähr 650 Byte. Nimmt man nun die Datenrate 5 an, welche in der Abstraktion im Standard verwendet wird, so bedeutet dies laut [42], dass der Payload maximal 222 Byte betragen darf. Entsprechend müsste diese Transaktion in 3 Uplink Nachrichten aufgeteilt werden und es könnten maximal 27 Verträge pro Tag (ohne deren Bearbeitung) erstellt werden. Andere Datenraten erlauben zudem noch geringere Payloads. Nach [42] können so zum Beispiel mit Datenrate 3 nur 128 Byte und mit Datenraten 0-2 nur 64 Byte übertragen werden. Ein weitaus größeres Problem bei der Übertragung von ganzen Transaktionen über LoRaWAN ist die Größe von eingehenden

Transaktionen, die durch die andere Partei bestimmt wird. Im Allgemeinen limitiert Fides die Transaktionsgröße auf 1 MB, was bedeuten würde, dass eine Transaktion dieser Größe in fast 5000 Downlink Nachrichten aufgeteilt werden müsste, was entsprechend wenig praktikabel ist bei einem Downlink Limit von 10 Nachrichten pro Tag. Daher verfolgt die LoRaWAN Abstraktion für Fides einen anderen Weg: Die Verwendung eines Protokolls, welches die Verträge und Vorlagen in Fides geschickt adressiert und durch einen zentralen Dienst (Middleware) ergänzt wird, welcher die Nachrichten im Namen der Nutzenden übersetzt. Jene Middleware verwaltet die privaten Schlüssel der Nutzenden und authentifiziert sie über eine einfache Integration über das The Things Network, welches die LoRaWAN Nachrichten und den für das LoRaWAN Gerät generierten API-Key an die Middleware weiterleitet. Weiterhin verwaltet die Middleware pro Account einer Partei Platzhalter, die innerhalb der Abarbeitung von Verträgen durch die von den Nutzenden festgelegten Werte ersetzt werden. An dieser Stelle ist es wichtig zu erwähnen, dass die Spezifikation der Middleware als Teil der Abstraktion veröffentlicht wurde¹¹, wodurch nicht eine zentrale Middleware verwendet werden muss, sondern es jeder Partei möglich ist, eine jeweils eigene zu betreiben. Dies unterstützt das allgemeine Ziel der Selbstorganisation innerhalb der Arbeit. Abbildung 19 verdeutlicht das Vorgehen. Für andere Teilnehmende innerhalb des Fides Netzwerks sehen die übersetzten Transaktionen so aus, als würden sie direkt von der Partei stammen, die sich jedoch in der Region ohne Internetverbindung befindet. Aufgrund des angesprochenen Downlink Limits überträgt die Middleware jedoch keine Eingabedaten der anderen Parteien, sondern nur die Zustände der Objekte (Verträge/Vorlagen) an die LoRaWAN Geräte zurück. Entsprechend sollten die benutzten Vorlagen auf die Benutzung mit LoRaWAN abgestimmt sein und die Übereinkünfte darauf ausgelegt sein, dass die erstellende Partei des Vertrags (mit LoRaWAN) hauptsächlich Daten sendet, während die andere Partei primär die Schritte ohne weitere

¹¹Siehe „fides/abst/lora“ innerhalb des Fides Repository [129]

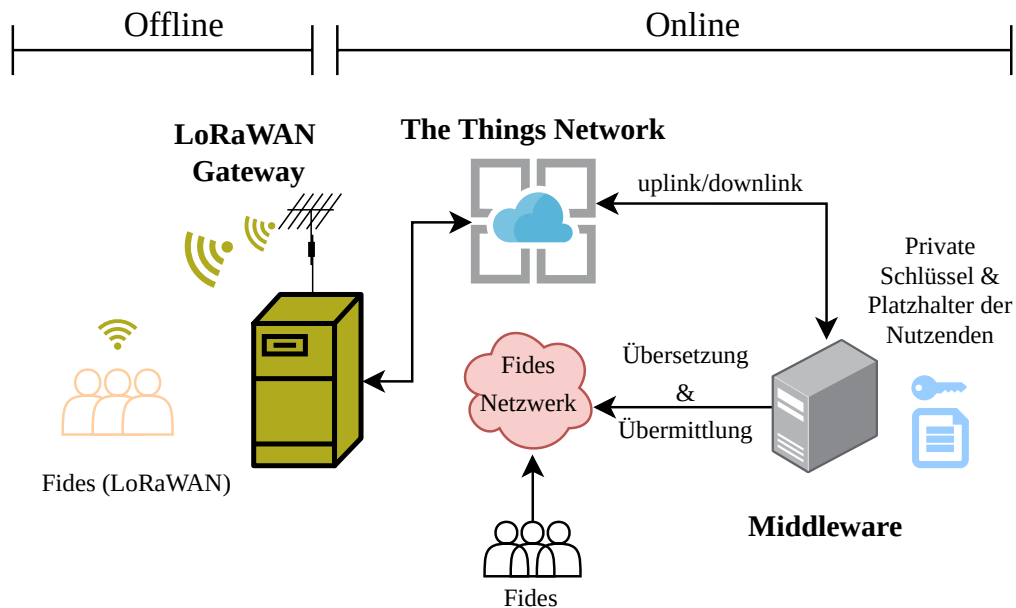


Abbildung 19: Übersicht zur Verwendung von Fides mit LoRaWAN

Eingaben bestätigt.

7.3 Konfiguration

Die LoRaWAN Abstraktion folgt den in 6.2 angegebenen Vorgaben und macht die Konfigurationsparameter über das Modul `fides.abst.lora.core.config` verfügbar. Hierbei werden, wie schon in der regulären Konfiguration beschrieben, die folgenden Parameter aus der Konfigurationsdatei `fides.config` gelesen, welche die LoRaWAN Abstraktion bei ihrer Installation um die Sektion `abst/lora` erweitert.

- **AppEui:** Applikations EUI innerhalb von TTN [162]
- **DevEui:** EUI des LoRaWAN Geräts [131]
- **AppKey:** Schlüssel der LoRaWAN Anwendung, welcher verwendet wird, um die Session Schlüssel abzuleiten. [131]

- **AccountPublicKey**: Öffentlicher Schlüssel zu dem privaten Schlüssel, welcher auf der Middleware verwaltet wird.
- **DataRate**: LoRaWAN Datenrate [132]

Bei der Installation der Abstraktion wird der Ordner *lora* innerhalb der verwendeten Fides Instanz im Ordner *abst* angelegt, in welchem die Abstraktion die zugehörigen Daten speichert.

7.4 Hardware

Die LoRaWAN Abstraktion für Fides wurde für das LoRaWAN Modul „IoT LoRa Raspberry Pi Node pHat“ von PiSupply¹² entwickelt. Die Implementierung verwendet die für das Modul passende Bibliothek¹³, wobei das Modul selbst mit Firmware Version 2 betrieben wird.

Weitere Kombinationen aus Hardware/Software/Firmware können mit leichten Änderungen in die Implementierung integriert werden, sofern ein anderes Hardwaremodul genutzt werden soll.

Das Modul nutzt als Frequenz 868 Mhz (Europa) [132] und innerhalb der Anwendung wird im Standard die Datenrate 5 verwendet, was einem Spreading Factor von 7 entspricht [132]. Nimmt man die größte Protokollnachricht von 11 Byte an, so entspricht dies laut [42] ca. 486 Nachrichten pro 24 Stunden.

7.5 Protokoll

Um trotz der strengen Limits bezogen auf die Up- und Downlink-Datenrate, im Speziellen die Fair Access Policy [164], LoRaWAN für das Erstellen und Bearbeiten von Verträgen verwenden zu können, muss das Protokoll so wenig Daten wie möglich übermitteln und dennoch praktikabel einsatzfähig sein. Nachrichten des Protokolls folgen allgemein dem folgenden Aufbau:

¹²<https://github.com/PiSupply/IoTLoRaRange/> (Abgerufen im Februar 2023)

¹³<https://github.com/AmedeeBulle/pyrak811> (Abgerufen im Februar 2023)

Kommando Argument Objekt

Hierbei wird für die Adressierung des Kommandos der Wertebereich von 1 Byte angenommen. Das Kommando dient nicht nur zur Instruktion für die Middleware, die die Nachricht im Namen der jeweiligen Partei übersetzt, sondern auch als Instruktion von der Middleware an den LoRaWAN Client in Form von Downlink Nachrichten (um so zum Beispiel den Status eines Vertrags zu verändern). In Abhängigkeit der Instruktion kann ab dem zweiten Byte ein Argument folgen. Die Länge des Arguments überschreitet 4 Byte nicht. Letztlich wird das Objekt (Vertrag bzw. Vorlage) mit dem restlichen Teil der Nachricht angesprochen. Zunächst werden hier 6 Byte verwendet, was bedeutet, dass die maximale Nachrichtengröße in der Regel 11 Byte nicht überschreitet. Sollte es bei der Adressierung des Objekts zu Doppeldeutigkeiten kommen, so kann die Nachricht vergrößert werden. Das Vorgehen ist ähnlich zu der Funktionsweise von git bei der Adressierung von *commits* [153].

Tabelle 12 beschreibt die Nachrichten des Protokolls. Abbildung 20 zeigt den Zusammenhang von regulärer Fides Transaktion und einer Fides LoRaWAN Transaktion.

7.6 Datenmodelle

Die Datenmodelle der LoRaWAN Abstraktion orientieren sich an der Kernimplementierung. Entsprechend wird im Folgenden primär auf die Unterschiede bei der Handhabung der Objekte eingegangen.

7.6.1 Datenbankmodelle

Die LoRaWAN Abstraktion ergänzt die Fides Instanz um eine Datenbank speziell zur Speicherung der über LoRaWAN erstellten Verträge. Auch hier handelt es sich um eine einfache Speicherung innerhalb einer sqlite3 Datenbank, fernab von komplexen Fremdschlüsselbeziehungen zu anderen Tabel-

Kommando	Argument	Beschreibung
00	-	Ping Nachricht (keine Aktion durch Middleware)
01	Nonce (4 Byte)	Erstellen eines neuen Vertrags
02	-	Aufgabe bestätigen (ohne Eingabedaten)
03	Platzhalter (1 Byte)	Aufgabe bestätigen (mit Platzhalter)
04	-	Status eines Vertrags abrufen
05	-	Status einer Vorlage abrufen
06	-	Vertrag ist im Zustand <i>OFFER</i> (downlink)
07	-	Hash des Vertrags ist doppeldeutig (downlink)
08	-	Vorlage ist aktiv (downlink)
09	-	Vorlage ist inaktiv (downlink)
0a	-	Hash der Vorlage ist doppeldeutig (downlink)
0b	-	Vertrag innerhalb der Middleware löschen
0c	-	Vertrag nicht gefunden (downlink)
0d	-	Vorlage nicht gefunden (downlink)
0e	Aktuelle Aufgabe (1 Byte)	Vertrag ist im Zustand <i>LIVE</i> (downlink)
0f	-	Vertrag ist im Zustand <i>REJECTED</i> (downlink)
10	-	Vertrag ist im Zustand <i>FINISHED</i> (downlink)

Tabelle 12: LoRaWAN Protokoll

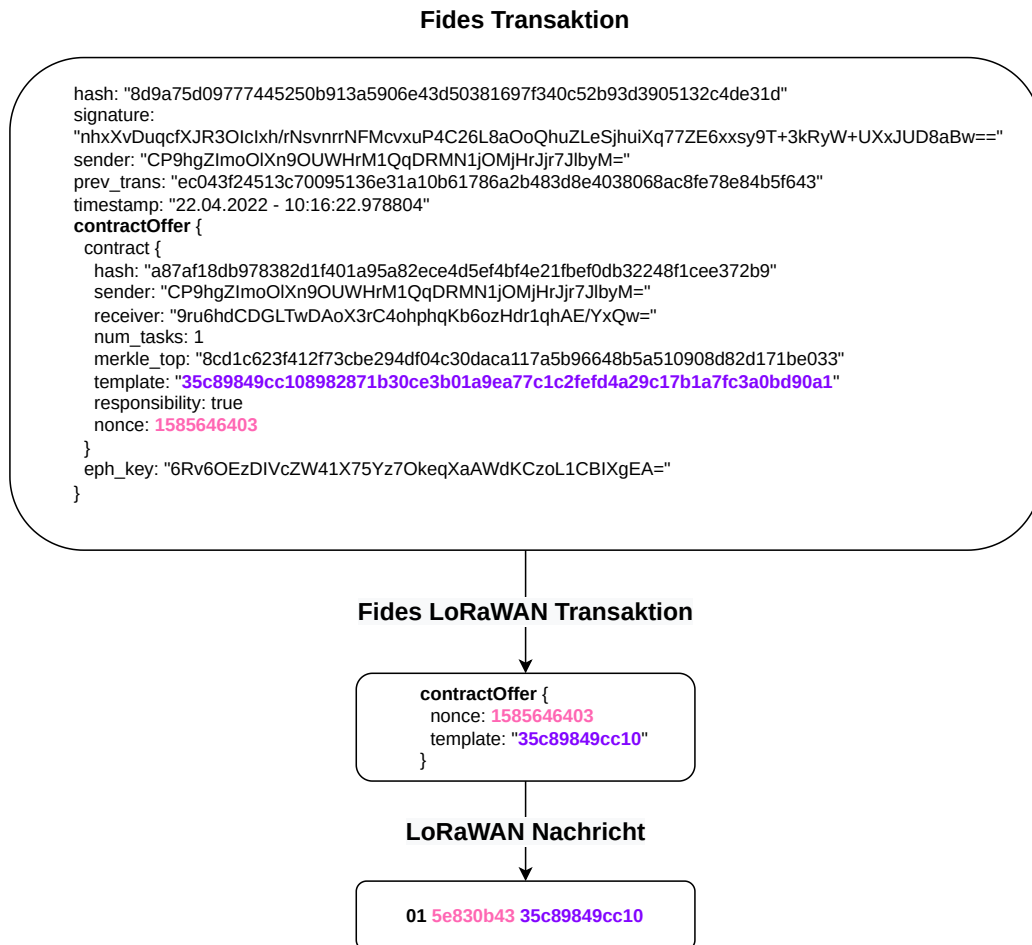


Abbildung 20: Zusammenhang zwischen regulären und LoRaWAN Transaktionen

Python	protobuf	Bemerkung
short_hash (str)	contract (string)	Abkürzung des Hash des Vertrags
lora_storage (LoRaStorage)		Verwaltung der lokalen LoRaWAN Datenbank

Tabelle 13: Erweiterungen des Datentyps *LoRaContract*

len. Für den Zugriff auf die Vorlagen verwendet die Abstraktion die Vorlagen der Kerninstanz, hierdurch soll ein reibungsloser Betrieb und Wechsel zwischen LoRaWAN/Internetverbindung ermöglicht werden, ohne Vorlagen in verschiedenen Tabellen zu speichern. Da es durch die Adressierung der Vorlagen durch das später beschriebene Protokoll (siehe 7.5) unter Umständen zu seltenen Doppeldeutigkeiten bei den Vorlagen kommen kann, werden jene Objekte in einer einfachen Textdatei innerhalb der Instanz dokumentiert.

7.6.2 Datentypen

Im Folgenden werden die Datentypen der LoRaWAN Abstraktion genauer beschrieben. Insbesondere werden die Unterschiede zu den Kerndatentypen aufgezeigt.

Verträge Verträge innerhalb der LoRaWAN Abstraktion sind in der Klasse *LoRaContract* implementiert, welche sich an der Kernklasse *Contract* orientiert. Tabelle 13 beschreibt die zusätzlichen Elemente der Klasse. Da Objekte der Klasse *LoRaContract* nicht über das Internet übertragen werden, entfallen die Felder *tx_ref* und *last_tx*.

Transaktionen Transaktionen der Abstraktion sind ähnlich zur Kernimplementierung von Fides aufgebaut. LoRaWAN Transaktion enthalten hierbei lediglich ein Element *type*, was die Art und den Inhalt der Transaktion bestimmt. Gleichwohl zur Kernimplementierung handelt es sich hierbei um

ein *oneof* Objekt. Tabelle 14 beschreibt die möglichen Nachrichten und deren zusätzliche Elemente.

7.6.3 Instanzen der Abstraktion

Die Abstraktion folgt den Vorgaben der Kernimplementierung und nutzt daher den Ordner *abst* innerhalb der aktuell verwendeten Fides Instanz (siehe Kapitel 6.3.4). Die Daten werden innerhalb des separaten Ordners *lora* gespeichert und enthalten eine separate Datenbank für Verträge, die über LoRaWAN abgewickelt wurden. Zusätzlich werden innerhalb einer Datei doppeldeutige Vorlagen verwaltet, um jene bei späterer Kommunikation korrekt adressieren zu können.

7.7 Blinde Operationen

Zur Einsparung von Bandbreite enthält die LoRaWAN Abstraktion sogenannte blinde Operationen, um lokale Zustände verändern zu können, ohne jene von der Middleware via Downlink Nachricht bezogen zu haben. Dies ist insbesondere hilfreich, wenn:

1. Verträge von der anderen Partei automatisch angenommen werden (zum Beispiel durch eine Automatisierung) oder
2. Vertragszustände im echten Leben beobachtet werden können.

Nutzende des Command Line Interface verwenden hierzu innerhalb der LoRaWAN Abstraktion das Flag *-blind* bei der Bestätigung von Vertragsschritten. Da sich die Abstraktion nur zur Erstellung von Verträgen, nicht zu der Verwaltung von Vorlagen eignet, gilt hier eine Ausnahme beim Akzeptieren eines Vertrags. Allgemein wird außerhalb der LoRaWAN Abstraktion ein Vertrag, identifiziert durch seinen Hash $\mathcal{H}_{c_{i,j}}$, folgendermaßen angenommen:

Typ	Weitere Felder	Bemerkung
Confirm task	type (enum <i>CONFIRM_TYPE</i>), contract (string), input (string)	Adressierung eines Vertrags über die Abkürzung des Hashs. Der Typ ist entweder <i>EMPTY</i> oder <i>PLACEHOLDER</i> . Wenn ein Platzhalter verwendet wird, ist jener im Feld <i>input</i> hinterlegt.
Contract offer	template (string), nonce (fixed32)	Abgekürzter Hash der Vorlage und nonce, die bei der Vertragserstellung auf dem LoRaWAN Gerät verwendet wurde.
Get contract state	contract (string)	Anfrage, um Vertragsstatus via Downlink zu beziehen.
Get template state	template (string)	Anfrage, um Vorlagenstatus via Downlink zu beziehen.
Ping	-	Wird verwendet, um mit einem Byte Uplink Nachricht auf dem Gerät auf neue Downlink Nachrichten prüfen zu können.
Downlink response	type (enum <i>RESPONSE_TYPE</i>), info (int32), object (string)	Das Feld <i>type</i> beschreibt in einer Enumeration den Zustand des Objekts <i>object</i> (siehe Protokoll - Tabelle 12). Bei Verträgen im Zustand <i>LIVE</i> enthält das Feld <i>info</i> die aktuell zu bearbeitende Aufgabe.
Delete contract	contract (string)	Anfrage zum Löschen eines Vertrags über LoRaWAN mittels der Abkürzung des Vertragshashs.

Tabelle 14: Transaktionstypen als Teil eines *oneof* Objekts innerhalb von LoRaWAN Transaktionen (protobuf)

```
fds contract accept  $\mathcal{H}_{c,i,j}$ 
```

Auflistung 16: Akzeptieren eines Vertrags in der Kommandozeilenanwendung

Dieser Befehl, ausgeführt von Partei j , erstellt und übermittelt die jeweilige Transaktion in das Netzwerk, von wo der Index der Vorlage und des Vertrags aktualisiert werden.

Da die LoRaWAN Abstraktion nur die Sichtweise von Partei i betrachtet, wäre eine regulärer *accept* Befehl irreführend. Daher ist es innerhalb der Abstraktion möglich, den Befehl *confirm* auch auf Verträgen im Zustand *OFFER* auszuführen, was innerhalb der regulären Anwendung nicht möglich ist:

```
fds lora contract confirm --blind  $\mathcal{H}_{c,i,j}$ 
```

Auflistung 17: Blinde Operation innerhalb der Kommandozeilenanwendung (LoRaWAN Abstraktion)

Somit kann durch eine blinde Operation lokal der Zustand eines Vertrags von *OFFER* zu *LIVE* verändert werden, ohne dafür Information über Downlink Nachrichten zu beziehen. Einsparpotentiale richten sich stark nach den verwendeten Vorlagen und deren Aufgaben. Allgemein lassen sich jedoch bei dem blinden Akzeptieren eines Vertrages mindestens 7 Bytes an Uplink Nachrichten (eine Nachricht zum Abruf des Status des Vertrags) und 8 Bytes an Downlink Nachrichten (Übermittlung des Zustands mit aktueller Aufgabe) einsparen. Insbesondere das Einsparen von Downlink Nachrichten wirkt dem strikten Limit der Fair Access Policy [164] entgegen.

7.8 API

Nachfolgend werden die Kernelemente der LoRaWAN Implementierung vorgestellt. Im Allgemeinen ist die API der Abstraktion in Anlehnung an die reguläre Implementierung (siehe 6.4) aufgebaut.

7.8.1 Kommandozeilenanwendung

Die Implementierungen der Kommandozeilenanwendung sind innerhalb des Moduls *fides.abst.lora.cli* enthalten. Für die detaillierte Beschreibung des CLI sei auf Kapitel 7.9 verwiesen.

7.8.2 Kernfunktionen

Die Kernfunktionen der LoRaWAN Abstraktion für Fides sind innerhalb des Moduls *fides.abst.lora.core* implementiert und werden im Folgenden beschrieben.

Konfiguration Das Modul *fides.abst.lora.core.config* stellt die in 7.3 genannten Parameter über die gesamte Anwendung zur Verfügung und orientiert sich hierbei an der Kernimplementierung von Fides.

Verträge Innerhalb der LoRaWAN Abstraktion wird die Klasse *LoRaContract* verwendet, um die Übereinkünfte, die über LoRaWAN geschlossen wurden, zu verwalten. Die Klasse enthält die folgenden Methoden:

- **`__init__`**
Beschreibung: Konstruktor der Klasse
Eingabe: -
Ausgabe: LoRaContract Objekt
- **`print`**
Beschreibung: Ausgabe des LoRaContract Objekts
Eingabe: -
Ausgabe: Informationen zum Vertrag (via STDOUT)
- **`db.load`**
Beschreibung: Lädt einen Vertrag aus der lokalen Datenbank.
Eingabe: Hash bzw. Abkürzung des Vertrags
Ausgabe: Boolescher Wert

- **save**
Beschreibung: Speichert das Objekt in der lokalen Datenbank.
Eingabe: -
Ausgabe: -
- **create**
Beschreibung: Neuen Vertrag aus bekannter Vorlage erstellen.
Eingabe: Hash der Vorlage
Ausgabe: Boolescher Wert
- **finalize**
Beschreibung: Finalisiert einen Vertrag, berechnet dessen Hash und prüft das finale Objekt.
Eingabe: Nonce (optional)
Ausgabe: Boolescher Wert
- **check**
Beschreibung: Prüft die allgemeinen Bedingungen an den Vertrag in Kombination mit der zu nutzenden Vorlage.
Eingabe: Template Objekt
Ausgabe: Boolescher Wert
- **_get_rpc_stub**
Beschreibung: Interne Funktion, um eine Verbindung zum lokalen Kommunikations-Daemon herzustellen.
Eingabe: -
Ausgabe: Verbindungsobjekt
- **publish**
Beschreibung: Funktion zur Veröffentlichung eines Vertrags
Eingabe: -
Ausgabe: Boolescher Wert

- **update**

Beschreibung: Lokalen Vertrag anhand einer Transaktion (LoRaWAN downlink) aktualisieren. Die Downlink Informationen gelten hier als ultimative Wahrheit und überschreiben immer den lokalen Zustand. Somit können falsche blinde Operationen korrigiert werden.

Eingabe: Transaktion

Ausgabe: Boolescher Wert

- **_responsible**

Beschreibung: Interne Funktion zur Überprüfung, ob der angegebene öffentliche Schlüssel für die aktuelle Aufgabe des Vertrags zuständig ist.

Eingabe: Öffentlicher Schlüssel

Ausgabe: Boolescher Wert

- **confirm**

Beschreibung: Bestätigung der aktuellen Aufgabe des Vertrags

Eingabe: Platzhalter (optional)

Ausgabe: Boolescher Wert

- **delete**

Beschreibung: Vertrag lokal löschen.

Eingabe: -

Ausgabe: -

- **delete_remote**

Beschreibung: Vertrag auf Middleware löschen.

Eingabe: -

Ausgabe: Boolescher Wert

Speicherung Die LoRaWAN Abstraktion speichert die Daten ähnlich zu der Kernimplementierung von Fides. Als zusätzliche Funktion enthält die *fides.abst.lora.core.storage* Komponente noch das Laden eines Vertrags über

die Abkürzung seines Hashs, da in dieser Weise das Objekt über LoRaWAN adressiert wird. Hierbei wird bei dem Laden noch die etwaige Doppeldeutigkeit des Objekts geprüft und ggf. eine Fehlermeldung ausgegeben.

Transaktionsspezifische Implementierungen Das Modul *fides.abst.lora.core.transaction_impl* implementiert die Übersetzung zwischen protobuf Transaktion (*LoRaTransaction*) und der jeweiligen Nachricht in Bytes, die so über des LoRa Modul übertragen wird. Das Modul enthält pro Nachrichtenart eine private *_parse_{Nachrichtenart}* Methode, die aus den gegebenen Bytes die Transaktion aufbaut. Die öffentlichen Funktionen des Moduls belaufen sich daher auf:

- **bytes_to_tx**
Beschreibung: Aus Bytes eine *LoRaTransaction* erstellen.
Eingabe: Nachricht in Bytes
Ausgabe: *LoRaTransaction* Objekt
- **tx_to_bytes**
Beschreibung: Aus *LoRaTransaction* eine Nachricht in Bytes erstellen.
Eingabe: *LoRaTransaction* Objekt
Ausgabe: Nachricht in Bytes

Netzwerk Innerhalb des Moduls *fides.abst.lora.core.network.lora_client* ist die Klasse *LoRaClient* implementiert, welche die Verbindung zwischen LoRa Modul und Applikation herstellt. Nachfolgend sollen die Methoden der Klasse kurz vorgestellt werden:

- **__init__**
Beschreibung: Konstruktor der Klasse
Eingabe: -
Ausgabe: *LoRaClient* Objekt

- **status**

Beschreibung: Statusmeldung, die von *fidesd* z. B. dem Command Line Interface bereitgestellt wird.

Eingabe: -

Ausgabe: Status der Komponente als String

- **_reset_counter**

Beschreibung: Setzt den Zähler für Up- und Downlinkdaten nach 24 Stunden zurück.

Eingabe: -

Ausgabe: -

- **start**

Beschreibung: Startet den Zähler für Up- und Downlinkdaten und tritt dem LoRaWAN Netzwerk anhand der Konfiguration bei.

Eingabe: -

Ausgabe: Boolescher Wert

- **stop**

Beschreibung: Beendet die LoRaWAN Verbindung.

Eingabe: -

Ausgabe: -

- **_handle_downlink_contract**

Beschreibung: Vertrag anhand der Downlink Informationen aktualisieren.

Eingabe: LoRaTransaction

Ausgabe: Boolescher Wert

- **_handle_downlink_contract_hash_ambiguous**

Beschreibung: Verlängert die Abkürzung des Vertragshashs um 1 Byte, um jenen bei der nächsten Kommunikation anders zu adressieren. Zusätzlich wird der Vertrag als unveröffentlicht markiert.

Eingabe: LoRaTransaction

Ausgabe: -

- **`_handle_downlink_template_inactive`**

Beschreibung: Setzt das lokal genutzte Template auf inaktiv, wenn es nicht für den Vertrag genutzt werden kann.

Eingabe: LoRaTransaction

Ausgabe: -

- **`_handle_downlink_template_hash_ambiguous`**

Beschreibung: Markiert die genutzte Vorlage als doppeldeutig, damit bei der nächsten Anfrage jene Vorlage mit einer größeren Abkürzung des Hashs adressiert wird.

Eingabe: LoRaTransaction

Ausgabe: -

- **`_handle_downlink_contract_not_found`**

Beschreibung: Setzt den lokalen Zustand des Vertrags zurück, wenn jener nicht von der Middleware gefunden werden kann (z. B. wenn die LoRaWAN Nachricht nicht angekommen ist).

Eingabe: LoRaTransaction

Ausgabe: -

- **`_handle_downlink_template_not_found`**

Beschreibung: Setzt alle lokalen Verträge im Zustand *OFFER* zurück, die die angegebene Vorlage benutzen, da jene aktuell nicht im Netzwerk gefunden werden konnte.

Eingabe: LoRaTransaction

Ausgabe: -

- **`_handle_downlink`**

Beschreibung: Allgemeine Methode zur Anwendung einer Downlink

Nachricht. Ruft die vorher gelisteten Methoden je nach Art der Nachricht auf.

Eingabe: `LoRaTransaction`

Ausgabe: -

- **`_check_downlink`**

Beschreibung: Prüft, ob Downlink Nachrichten vorliegen, die bezogen werden können. Wird nach jedem Senden einer Uplink Nachricht aufgerufen.

Eingabe: -

Ausgabe: -

- **`publish`**

Beschreibung: Übersetzen und Veröffentlichen einer Transaktion über das LoRaWAN Modul.

Eingabe: Anfrage/Transaktion

Ausgabe: Tuple aus Booleschem Wert und None bzw. Fehlermeldung

Datentypen Der protobuf Datentyp *LoRaTransaction* wird innerhalb des Moduls *fides.abst.lora.core.types* global bereitgestellt, um auf diese Weise äquivalent zu Verfahren wie die Kernimplementierung.

7.8.3 Zusätzliche Funktionen

Zusätzliche Funktionen, welche von der Abstraktion verwendet werden, wurden im Modul *fides.abst.lora.utils* zusammengefasst. Nachfolgend werden die Teilmodule kurz vorgestellt.

Doppeldeutige Vorlagen Zur korrekten Adressierung von Template wird stets geprüft, ob Doppeldeutigkeiten bei dem Hash der Vorlage existieren. Dies kann entweder lokal oder bei der Middleware auftreten und in Form einer

Downlink Nachricht kommuniziert worden sein. Daher enthält das Modul *fides.abst.lora.utils.ambiguous* die folgenden Methoden:

- **template_ambiguous:** Prüfung, ob eine Abkürzung eines Templates anhand des lokalen Datenbestands doppeldeutig ist oder nicht.
- **add_ambiguous_template:** Eine doppeldeutige Abkürzung eines Hashs dem lokalen Datenbestand hinzufügen.
- **get_unambiguous_template:** Anhand der vorliegenden Informationen und dem Hash einer Vorlage die minimale, nicht doppeldeutige Abkürzung des Hashs zurückgeben, sofern die aktuell genutzte Abkürzung doppeldeutig sein sollte.

Log Die LoRaWAN Abstraktion ergänzt den Log der lokalen Fides Instanz ebenso um Einträge bei Statusänderungen. Hierzu bereitet die Komponente *fides.abst.lora.utils.log* die Nachrichten ähnlich zu der Kernimplementierung auf. Hierbei werden zum Beispiel Informationen zu neu erstellten Übereinkünften, das Bestätigen von Aufgaben (mit Platzhalterinformationen) oder Instruktionen zum Löschen des Vertrags auf der Middleware gespeichert. Weiterhin ergänzt die Abstraktion den Log mit allen Downlink Informationen, die auf dem Gerät empfangen werden.

7.9 Command Line Interface

Die Implementierungen der Abstraktion erweitern das Command Line Interface von Fides und führen ein Subkommando *lora* ein. Nachfolgend werden die verfügbaren Funktionen beschrieben:

init

Initialisiert die LoRaWAN Abstraktion und prüft, ob die notwendige Bibliothek installiert ist, um Daten über LoRaWAN übertragen zu können.

contract

- **fds lora contract check-template:** Prüfen, ob ein Template für den Vertrag verwendet werden kann. Das Template muss auf der Middleware verfügbar und nicht inaktiv sein.
Argument: Hash oder Alias des Template
Optionen: -
- **fds lora contract confirm:** Vertragsschritt bestätigen.
Argument: Hash des Vertrags
Optionen: `-blind` (lokale Annahme, dass der Schritt erledigt wurde),
`-placeholder` (welcher von der Middleware ersetzt werden soll)
- **fds lora contract create:** Vertrag erstellen.
Argument: Hash oder Alias der Vorlage
Optionen: `-publish` (nach der Erstellung direkt veröffentlichen)
- **fds lora contract delete:** Vertrag lokal löschen.
Argument: Hash des Vertrags
Option: `-force` (für die Löschung von Verträgen, die nicht im Zustand *FINISHED* oder *REJECTED* sind)
- **fds lora contract delete-remote:** Instruktion an die Middleware, den Vertrag zu löschen.
Argument: Hash des Vertrags
Optionen: -
- **fds lora contract list:** Alle Verträge auflisten.
- **fds lora contract publish:** Vertrag über LoRaWAN veröffentlichen.
Argument: Hash des Vertrags
Optionen: -

- **fds lora contract show**: Infos über Vertrag anzeigen.
Argument: Hash des Vertrags
Option: `-editor`
- **fds lora contract update**: Vertrag aktualisieren.
Argument: Hash des Vertrags
Optionen: -

update

Senden einer einzelnen Ping Nachricht an die Middleware. Hierdurch ist es möglich, etwaig verspätete Downlink Nachrichten der Middleware beziehen zu können und hierfür nur 1 Byte an Uplink Bandbreite zu verwenden.

7.10 Middleware

Um die LoRaWAN Nachrichten korrekt zu übersetzen, wird eine Middleware benötigt. Die Spezifikation ist Teil des Repository von Fides und beschreibt die Funktionen einer Middleware. Dadurch ist es möglich, unterschiedliche Middlewares zu verwenden und sich auch auf dieser Ebene selbstorganisieren zu können. Für die Erprobung der LoRaWAN Abstraktion wurde hierzu exemplarisch eine Middleware entwickelt, die im Folgenden vorgestellt wird. Die Middleware lässt sich in zwei Gruppen von Funktionen unterteilen:

1. Grundfunktionen zum Übersetzen der LoRaWAN Nachrichten und
2. Zusatzfunktionen, um die Middleware Nutzerinnen und Nutzern in einer einfach bedienbaren Weise zur Verfügung zu stellen.

Die Grundfunktionen wurden als Teil dieser Arbeit implementiert und können dem Anhang (siehe C.2) entnommen werden. Die Zusatzfunktionen wurden hauptsächlich innerhalb der Masterthesis von Kevin Wagner [120] umgesetzt.

7.10.1 Grundfunktionen

Die Grundfunktionen beschreiben die Übersetzung der LoRaWAN Nachrichten anhand des Protokolls und deren Übertragung im Namen der Nutzenden in dem zuvor festgelegten Fides Netzwerk. Hierzu muss die Middleware den HTTP-Endpoint */uplink* bereitstellen und dort *POST* Anfragen akzeptieren. Die Nutzerinnen und Nutzer werden hierbei mittels eines statischen API-Schlüssels authentifiziert, welcher Teil der Intergration innerhalb von TTN ist und über den HTTP-Header *X-API-KEY* mitgesendet wird. Abbildung 21 zeigt eine solche Webhook. Dieser API-Key kann nur verwendet werden, um LoRaWAN Nachrichten an die Webhook zu schicken, nicht, um sich innerhalb der Anwendung einzuloggen oder die dort gespeicherten Daten (temporäre private Schlüssel der Verträge, Platzhalter, vergangene Nachrichten) abzurufen.

Nachfolgend werden die durchgeführten Aktionen der Middleware beim Übersetzen der Uplink Nachrichten beschrieben. Hierbei wird der Zugriff auf die Daten (z. B. privater Schlüssel) der jeweiligen Partei als gegeben angenommen.

Übersetzung von Vertragsanfragen Bei der Übersetzung von Transaktionen des Typs „Contract offer“ muss die Middleware zunächst prüfen, ob das angegebene Template innerhalb der Transaktion identifizierbar ist. Hierzu wird versucht, die Vorlage lokal zu laden. Sollte die Abkürzung des Hash doppeldeutig sein, wird dies über eine separate Downlink Nachricht mitgeteilt. Ebenso falls das Template nicht gefunden werden konnte, weil es nicht Teil der der lokalen Instanz der Middleware ist. Sollte die Vorlage identifizierbar sein, wird jene verwendet, um einen neuen Vertrag zu erstellen. Bei der Finalisierung des Vertrags wird dann die innerhalb der LoRaWAN Transaktion übermittelte *nonce* in Kombination mit dem privaten Schlüssel der Partei verwendet, sodass der gleiche Hash des Vertrags, welcher bereits auf dem LoRaWAN Gerät berechnet wurde, auch auf der Middleware erzeugt wird.

Edit webhook

General settings

Webhook ID *

Webhook format *

Endpoint settings

Base URL *

Downlink API key

The API key will be provided to the endpoint using the "X-Downlink-Apikey" header

Request authentication ⓘ

Use basic access authentication (basic auth)

Additional headers



+ Add header entry

Enabled messages

i For each enabled message type, an optional path can be defined which will be appended to the base URL

Uplink message

Enabled

Abbildung 21: Automatisiert erstellte Integration der Middleware innerhalb des The Things Network

Letztlich wird der Vertrag im Netzwerk publiziert. Sollte die Veröffentlichung scheitern, wird jener direkt lokal gelöscht und von dem LoRaWAN Gerät (bei nicht Auffindbarkeit) eine erneutes Erstellen angewiesen.

Übersetzung bei der Bestätigung von Aufgaben Sofern eine Aufgabe bestätigt werden soll, muss zunächst der Vertrag lokal identifiziert werden. Hierbei muss zunächst geprüft werden, ob überhaupt ein Vertrag bekannt ist, welcher mit der angegebenen Abkürzung des Hashs übereinstimmt. Danach muss für jeden Kandidaten geprüft werden, ob der jeweilige Vertrag von der Partei stammt, welche auch die LoRaWAN Transaktion gesendet hat. Erst dann kann geprüft werden, ob der gesuchte Vertrag gefunden oder doppeldeutig ist. Entsprechendes Feedback passiert jeweils pro Downlink Nachricht. Ist der korrekte, nicht doppeldeutige Vertrag gefunden, wird in Abhängigkeit des *type* Parameters der Transaktion (*EMPTY* oder *PLACEHOLDER*) entweder die Aufgabe ohne Eingabe oder mit dem Inhalt des gegebenen Platzhalters als Eingabe bestätigt. Bei der Bestätigung wird entsprechend der private Schlüssel der Partei verwendet, um eine korrekte Transaktion innerhalb des Fides Netzwerks zu erzeugen.

Übersetzung beim Abfragen des Vertragsstatus Wenn eine Partei den Zustand eines Vertrags aktualisieren möchte, geschieht dies über eine manuelle Anfrage. Die Middleware muss entsprechend mit einer Downlink Nachricht antworten. Hierzu wird gleich wie bei der Bestätigung einer Aufgabe verfahren, um den Vertrag zu identifizieren. Wenn der Vertrag gefunden wurde, wird dessen Zustand geprüft und in dessen Abhängigkeit nach den Vorgaben des Protokolls geantwortet (siehe Tabelle 12).

Übersetzung beim Abfragen des Vorlagenstatus Bei der Prüfung des Zustands einer Vorlage muss zunächst durch die Middleware versucht werden, jene eindeutig zu identifizieren. Bei Doppeldeutigkeiten wird entsprechend mit einer Downlink Nachricht an das LoRaWAN Gerät reagiert. Sofern die

Vorlage gefunden wurde, wird dann innerhalb des Netzwerks zusätzlich validiert, ob sich jene noch im Zustand *ACTIVE* befindet. Sollte die Vorlage bereits lokal inaktiv sein oder durch die Anfrage im Netzwerk ihren Zustand ändern, wird das an das LoRaWAN Gerät weitergegeben. Sollte sich die Vorlage aktuell nicht im Netzwerk auffinden lassen, wird auch dies der anfragenden Partei mitgeteilt. Lediglich, wenn die Vorlage aktiv und in dem Moment der Anfrage im Netzwerk auffindbar ist, wird via Downlink mitgeteilt, dass die Vorlage sich im Zustand *ACTIVE* befindet.

Übersetzung beim Löschen eines Vertrags Sofern eine Partei die Löschung eines Vertrags via LoRaWAN Uplink Nachricht instruiert, wird zur Identifikation des korrekten Vertrags wie bei der Bestätigung einer Aufgabe vorgegangen. Sofern der Vertrag gefunden werden konnte, wird jener über die Fides API gelöscht.

7.10.2 Zusatzfunktionen

Die Zusatzfunktionen werden hier der Vollständigkeit halber nur kurz angesprochen. Innerhalb der Middleware ist es möglich, die Platzhalter dynamisch festzulegen. Hierbei können ganzzahlige Werte des 1 Byte Wertebereich auf eine beliebige Zeichenkette festgelegt werden, welche dann bei der Übersetzung der LoRaWAN Nachricht ersetzt werden, sofern der jeweilige Platzhalter übermittelt wurde. Abbildung 22 zeigt diese Übersicht exemplarisch.

Weiterhin enthält die Middleware als Zusatzfunktion eine Übersicht über die bisher erhaltenen Nachrichten, welche dauerhaft gespeichert wird. Innerhalb des The Things Network werden die übertragenen Daten nicht dauerhaft dargestellt. Abbildung 23 zeigt die Übersicht der gesendeten und übersetzten Nachrichten.

Int	Value	Actions
0	Familie Muster, Musterstraße 4, 55765 Birkenfeld	
1	06782 171951	

Rows per page: 10 1-2 of 2

Abbildung 22: Platzhalter innerhalb der Middleware

Uplink data	Fides LoRaWAN transaction
01f2e4e79fc40164f1c121	<pre>contractOffer { nonce: 4075087775 template: "c40164f1c121" }</pre>
03001f9bdd63faed	<pre>confirmTask { type: PLACEHOLDER contract: "1f9bdd63faed" input: "0" }</pre>
041f9bdd63faed	<pre>getContractState { contract: "1f9bdd63faed" }</pre>
041f9bdd63faed	<pre>getContractState { contract: "1f9bdd63faed" }</pre>
0b1f9bdd63faed	<pre>deleteContract { contract: "1f9bdd63faed" }</pre>
00	<pre>ping { }</pre>

Rows per page: 10 1-6 of 6

Abbildung 23: Gesendete LoRaWAN Nachrichten und Übersetzung innerhalb der Middleware

7.11 Exemplarischer Ablauf

Nachfolgend wird die LoRaWAN Abstraktion anhand eines Beispielszenarios erklärt. Hierzu wird eine Vorlage verwendet, die zwei Aufgaben enthält. Die erste Aufgabe ist der Partei i , also jener, die den Vertrag erstellt, zugeordnet und für die zweite Aufgabe ist die Partei j , welche die Vorlage verwaltet, zuständig. Damit die Middleware korrekt operieren kann, muss jene Vorlage innerhalb der lokalen Fides Instanzen (Middleware und LoRaWAN Abstraktion) bekannt sein. Für LoRaWAN Geräte, die diese Vorlage zum Beispiel nicht über das Internet beziehen können, kann jene über das Command Line Interface auch offline importiert werden. Für unser Beispiel nehmen wir für die ersten 6 Byte der Vorlage $\mathcal{H}_{\mathcal{O}_j} = c40164f1c121[\dots]$ an.

Erstellen und Publizieren eines Vertrags Bevor ein Vertrag erstellt werden kann, muss auf dem LoRaWAN Gerät, welches die Fides Abstraktion ausführt, zunächst das Netzwerk gestartet werden. Nach dem korrekten Start des Netzwerks kann dann ein Vertrag erstellt und publiziert werden:

```
fds lora contract create -p  $\mathcal{H}_{\mathcal{O}_j}$ 
```

Auflistung 18: LoRaWAN Beispiel: Erstellen und Publizieren eines Vertrags

Für unser Beispiel nehmen wir an, dass hierbei der zufällige Wert des Elements *nonce* von 4075087775 generiert wurde, was im Hex Format (big endian) f2e4e79f entspricht.

Lokal wurde in unserem Beispiel $\mathcal{H}_{\mathcal{C}_{i,j}} = 1f9bdd63faed[\dots]$ berechnet. Bevor die Informationen über LoRaWAN übermittelt werden, werden jene zunächst an den lokalen Kommunikations-Daemon *fidesd* innerhalb einer *LoRaTransaction* übermittelt. Enthalten sind der Wert für *nonce* sowie der abgekürzte Hash (ohne Doppeldeutigkeit) für die Vorlage. Die Netzwerkkomponente *LoRaClient* übersetzt diese Transaktion anhand des Protokolls und überträgt die folgenden Bytes über LoRaWAN:

```
01 f2e4e79f c40164f1c121
```

Da eine automatisierte Vorlage verwendet wurde, welche den Vertrag für die Partei automatisch annimmt (sofern jener korrekt an die Middleware übertragen wurde), sparen wir uns Up- und Downlink Nachrichten und nutzen eine blinde Operation, um den Zustand lokal von *OFFER* zu *LIVE* zu verändern:

```
fds lora contract confirm -b 1f9bdd63faed
```

Auflistung 19: LoRaWAN Beispiel: Blinde Operation zur Annahme des Vertrags

Aufgabe bestätigen Um nun die erste Aufgabe des Vertrags via LoRaWAN zu bestätigen, wird folgende Nachricht über LoRaWAN gesendet:

```
03 00 1f9bdd63faed
```

Bei unserem Beispiel handelt es sich um einen nicht-doppeldeutigen Vertragshash, womit die Middleware die Nachricht zuordnen kann. Zusätzlich verwendet die Middleware den von uns festgelegten Platzhalter bei der Bestätigung der Aufgabe und übermittelt jenen somit verschlüsselt in unserem Namen an die andere Partei. Um zu prüfen, ob unsere Nachricht die Middleware erreicht hat, bzw. ob unsere blinde Operation erfolgreich war, müssen wir manuell den Zustand des Vertrags prüfen.

Vertrag aktualisieren Um den Vertragszustand zu prüfen, teilen wir dies der Middleware über die folgende LoRaWAN Nachricht mit:

```
04 1f9bdd63faed
```

Da der Vertrag klar uns zuzuordnen ist, sendet die Middleware über das The Things Network eine Downlink Nachricht an unser Gerät. In der Regel (je nach Verbindung zum aktuellen Gateway) können wir diese Nachricht direkt abrufen, also nachdem wir den Zustand angefragt haben. Sollte sich

die Downlink Nachricht verzögern, reicht es eine einfache Ping Nachricht (1 Byte) zu senden, um danach lokal auf neue Downlink Nachrichten zu prüfen.

In unserem Beispiel nehmen wir die folgende Downlink Nachricht von der Middleware an:

```
0e 02 1f9bdd63faed
```

Wir können erkennen, dass der Vertrag im Zustand *LIVE* ist und aktuell die zweite Aufgabe bearbeitet wird. Folglich wissen wir, dass (1) unsere blinde Operation lokal erfolgreich war, da die andere Partei den Vertrag angenommen hat, und dass (2) unsere Bestätigung der Aufgabe korrekt durch die Middleware erfolgt ist.

Sofern eine blinde Operation nicht gelingt, wird die Downlink Nachricht der Middleware als ultimative Wahrheit angenommen und überschreibt unseren lokalen Zustand. In unserem Beispiel wäre es möglich, dass die automatische Annahme des Vertrags durch die andere Partei erst nach unserer Bestätigung der Aufgabe durchgeführt wird. Entsprechend könnte unsere Bestätigung nicht durchgeführt werden, da zu dem Zeitpunkt der Vertrag noch im Zustand *OFFER* war. Dies würden wir über die Downlink Nachricht in Erfahrung bringen und könnten dann erneut versuchen, die erste Aufgabe des Vertrags über die Middleware zu bestätigen.

Da es sich in unserem Beispiel bei der zweiten Aufgabe um eine Aufgabe für die andere Partei handelt, gehen wir der Vollständigkeit halber noch auf ein weiteres Feedback der Middleware ein, sofern wir unsere Zustandsanfrage wiederholen und die andere Partei zu jenem Zeitpunkt bereits den Vertrag aktualisiert hat. Entsprechend ergibt sich die Downlink Nachricht zu:

```
10 1f9bdd63faed
```

Lokal wissen wir nun, dass sich der Vertrag im Zustand *FINISHED* befindet. Handelt es sich bei der letzten Aufgabe um eine Aktion in der realen Welt, die wir beobachten können, so wäre hier auch die Verwendung einer blinden Operation möglich.

Time	Type	Data preview
09:51:24	Accept join-request	
↑ 09:50:47	Forward uplink data message	MAC payload: 0B 1F 9B DD 63 FA ED FPort: 1 Data rate: SF7BW125 SNR: 7.25 RSSI: -59
↓ 09:49:37	Forward downlink data mess...	FPort: 1 Payload: 10 1F 9B DD 63 FA ED
↑ 09:49:37	Forward uplink data message	MAC payload: 04 1F 9B DD 63 FA ED FPort: 1 Data rate: SF7BW125 SNR: 9.75 RSSI: -59
↓ 09:48:11	Forward downlink data mess...	FPort: 1 Payload: 0E 02 1F 9B DD 63 FA ED
↑ 09:48:11	Forward uplink data message	MAC payload: 04 1F 9B DD 63 FA ED FPort: 1 Data rate: SF7BW125 SNR: 10.75 RSSI: -61
↑ 09:47:29	Forward uplink data message	MAC payload: 03 00 1F 9B DD 63 FA ED FPort: 1 Data rate: SF7BW125 SNR: 7.5 RSSI: -58
↑ 09:43:52	Forward uplink data message	MAC payload: 01 F2 E4 E7 9F C4 01 64 ... FPort: 1 Data rate: SF7BW125 SNR: 7.75 RSSI: -58
09:42:16	Accept join-request	

Abbildung 24: LoRaWAN Up- und Downlink Nachrichten innerhalb des The Things Network

Vertrag löschen Sofern ein Vertrag aus einer offline Region mittels LoRaWAN gelöscht werden soll, kann die Middleware über eine LoRaWAN Nachricht instruiert werden. In unserem Beispiel ergäbe sich die Nachricht zu:

0b 1f9bdd63faed

Abbildung 24 zeigt die angesprochenen Nachrichten (Up- und Downlink) als Screenshot innerhalb der Live Datenübersicht aus dem The Things Network.

7.12 Zusammenfassung

Das Kapitel behandelte die LoRaWAN Abstraktion für Fides zur Teilnahme an Verträgen innerhalb von Regionen ohne direkte Internetverbindung. Zunächst wurde die allgemeine Idee der Abstraktion beschrieben, welche sich von der regulären Verwendung von LoRaWAN unterscheidet. Die Nutzung von LoRaWAN zur aktiven Teilnahme an einem System grenzt sich von einem Großteil der Arbeiten ab, die LoRaWAN lediglich nutzen, um auf Grundlage

jener Daten (z. B. Sensorwerte) Anwendungen bereitzustellen.

Die Unterschiede zur Kernimplementierung, insbesondere bei den Abläufen und Datentypen, wurden aufgezeigt. Neben der LoRaWAN API und den Ergänzungen zur Kommandozeilenanwendung von Fides wurde zusätzlich noch das Vorgehen der Middleware, welche die LoRaWAN Nachrichten im Namen der Nutzenden übersetzt, vorgestellt.

8 Evaluation

Im Folgenden werden die Resultate der Arbeit mit Blick auf verschiedene Gesichtspunkte evaluiert. Hierzu wird zunächst auf die rechtliche Bewertung des Protokolls, der Referenzimplementierung Fides und deren LoRa-WAN Abstraktion eingegangen. Zusätzlich werden verschiedene Messungen samt Szenarien vorgestellt, die die Performanz der Implementierung nach Nutzungsart zeigen. Betrachtet werden hierbei Netzwerkknoten und reguläre Nutzende. Abschließend wird das Protokoll in Teilen als Ethereum Smart Contract implementiert, um zu zeigen, dass das Konzept zwar mit regulären Smart Contracts umsetzbar ist, es jedoch zum aktuellen Zeitpunkt im Bezug auf die entstehenden Kosten sinnvoller ist, separat zu betreiben.

8.1 Rechtliche Bewertung

Im Folgenden wird die detaillierte rechtliche Bewertung der Konzepte der Arbeit und ihrer Referenzimplementierung, welche in Form eines unabhängigen Gutachtens des Lehrstuhls Öffentliches Recht, Informationsrecht, Umweltrecht, Verwaltungswissenschaft der Universität Frankfurt [95] vorliegt, vorgestellt. Hierbei wird zwischen zivil- und datenschutzrechtlicher Betrachtung unterschieden.

8.1.1 Terminologie

Zur klaren Trennung zwischen rechtlich bindendem Vertrag und *Cypher Social Contract* unterscheidet das Rechtsgutachten [95] folgende Begriffsbedeutungen:

- **Vertrag:** Rechtswirksamer Vertrag
- **Contract:** Gleichzusetzen mit *Cypher Social Contract*. Eine Verpflichtung zur Kommunikation zwischen *clients*.

8.1.2 Zivilrechtliche Betrachtung

Zunächst werden die in der Arbeit entwickelten *Cypher Social Contracts* und ihre dezentrale Implementierung innerhalb der Software Fides zivilrechtlich betrachtet. Hierzu werden die Ergebnisse der Rechtsstudie aus [95] wiedergegeben. Allgemeine zivilrechtliche Anforderungen an Verträge werden nicht gesondert erläutert. Die zivilrechtliche Betrachtung in [95] betrachtet verschiedene Fallgestaltungen, die nachfolgend aufgegriffen werden:

Erste Fallgestaltung

„In der ersten Fallgestaltung will ein Client, ohne dass es zuvor zu einem Kontakt zwischen den Parteien gekommen ist, einen Vertrag auf der Basis eines von ihm erstellten Templates eingehen.“
[95]

Hier stellt die Vorlage rein rechtlich noch kein Angebot dar, da jene dem anderen *client* (noch) nicht bekannt ist. Ein Angebot entsteht vielmehr durch das Erstellen des Contracts durch die andere Partei. Bei der Erstellung wird die Vorlage mit dem öffentlichen Schlüssel der zweiten Partei verbunden, wodurch eine Willensäußerung in Form einer bewussten Handlung entsteht.
[95]

„Angesichts des Ziels der Erstellung eines Contracts, nämlich einen Vertrag mit dem anderen Client einzugehen, handelt der Ersteller des Contracts auch in dem Bewusstsein, dass sein Verhalten als rechtserhebliche Erklärung aufgefasst werden kann. Es ist sogar sein Ziel, dass sein an den anderen Client gerichteter Contract als solcher aufgefasst wird, beabsichtigt er doch die Eingehung eines Vertrages.“ [95]

Weiterhin lässt sich der Contract verwenden, um „den eigenen Handlungs-, Erklärungs- und auch Geschäftswillen zum Ausdruck zu bringen“, um so den

„Willen zur Herbeiführung einer bestimmten Rechtsfolge, d.h. eines Vertrags“ darzulegen, sofern die Parameter des Vertrages im Contract enthalten sind. [95]

Die Veröffentlichung des Contracts über das Netzwerk kann nach [95] als „Willensakt“ verstanden werden, durch welchen die empfangsbedürftige Willenserklärung korrekt übertragen werden kann und somit die andere Partei „unter normalen Umständen die Möglichkeit zur Kenntnisnahme hat“.

Entsprechend handelt es sich bei einem über einen *full node* veröffentlichten Contract um eine Willenserklärung. In Abhängigkeit der Vorlage kann es sich auch um ein Angebot handeln, was durch die generische Natur der *Cypher Social Contracts* per Einzelfallbetrachtung geprüft werden muss.

Damit ein Vertrag zustande kommen kann, muss die andere Partei jenen annehmen. Hierbei müssen zwei Situationen unterschieden werden:

- **Annahme des Contracts entspricht Annahme des Angebots:** Hierbei handelt es sich laut [95] um einen Ausnahmefall der nur dann gilt, sofern die Vorlage alle notwendigen Informationen über den zu schließenden Vertrag enthält und somit durch die erstellende Partei des Vertrags keine Parameter (zum Beispiel Liefermenge bei einer Bestellung) verändert werden können. Weiterhin müsste der öffentliche Schlüssel der erstellenden Partei der anderen bekannt sein, sodass rein durch den Contract (im Zustand *OFFER*) alle notwendigen Informationen vorliegen.
- **Annahme des Contracts entspricht Willen zur Kommunikation:** Hierbei wird durch die Partei, die die Vorlage verwaltet, die Kommunikation angenommen. Laut [95] ist dies vergleichbar mit dem „Öffnen eines Briefes, der ein zugesandtes Angebot enthält“. Erst durch die weiteren Schritte innerhalb der Abarbeitung des Contracts (je nach Ausgestaltung der Vorlage) werden so alle notwendigen Parameter bekannt und es kann entschieden werden, ob ein Vertrag zustande kommen soll. In der Regel bedeutet dies für Vorlagen, dass zunächst über

die Aufgaben die Parameter und Informationen ausgetauscht werden und es dann separat über die Bestätigung weiterer Aufgaben zur Annahme/Ablehnung des eigentlichen Angebots kommt.

Zweite Fallgestaltung

„Den häufigsten Fall stellt die Einbindung von Fides in ein Verifikationssystem, wie zum Beispiel einen Onlinemarktplatz oder eine andere Onlineplattform dar, über die ein Client seine vertraglichen Leistungen anbietet und über die er seinen Public Key sowie unter Umständen ein eigenes Template für einen Contract bei Fides bereitstellt.“ [95]

Die zweite Fallgestaltung untersucht, ob die Veröffentlichung einer Vorlage auf einer Plattform bereits als Angebot verstanden werden kann. Weshalb dies nicht der Fall ist und somit die Terminologie gleich zu der in dem Protokoll gewählten bleibt, wird nachfolgend anhand der Punkte aus [95] aufgegriffen. Die Vorlage enthält nur die Partei, die den Vertrag empfängt und somit noch nicht beide Parteien des Vertrags, wobei dies in der Regel notwendig ist. Weiterhin muss beachtet werden, ob eine Leistung überhaupt noch verfügbar ist. Würde man die Vorlage selbst als Angebot verstehen, so könnten unbegrenzt viele Verträge geschlossen werden und ggf. Schadensersatzforderungen gestellt werden, wenn jene Leistungen nicht erbracht werden können. [95] Entsprechend ist nach [95] davon auszugehen, dass es sich bei der Vorlage/dem Template um eine „*invitatio ad offerendum*“ und kein Angebot handelt. Dies deckt sich mit der Idee des Protokolls und den allgemeinen Abläufen. Wird über einen Contract nun das Angebot abgegeben, so sind die Abläufe gleich der ersten Fallgestaltung.

Dritte Fallgestaltung

„Zuletzt besteht die Möglichkeit, dass im Rahmen einer vorhergehenden Kommunikation beider Seiten ein Austausch von Tem-

platehashes oder der Public Keys erfolgt ist und sogar Vertragsverhandlungen stattgefunden haben.“ [95]

Hier ändert sich nach [95] die rechtliche Bewertung nur, sofern bereits unabhängig von Fides Willenserklärungen ausgetauscht wurden, die man als Angebot und dessen Annahme verstehen kann. Hier kann, ähnlich zu den vorherigen Fallgestaltungen, bei der Annahme des Contracts noch nicht von der Annahme des Vertrags ausgegangen werden, da die Parameter des Vertrags in der Regel erst nach Aufnahme der Kommunikation vorliegen bzw. erst dann geprüft werden können. [95]

Neben den Kernfallgestaltungen betrachtet das Rechtsgutachten weiterhin noch mögliche Auswirkungen auf den Vertragsschluss, welche nachfolgend aufgegriffen werden.

Verwendung von AGB Nach [95] ergeben sich zwei Fragestellungen bezogen auf die Nutzung von AGB:

1. **Integration existierender AGB in einen Vertrag, der über Fides geschlossen wird:** Existierende AGB können in Verträge einbezogen werden. Dies kann zum Beispiel bei der Bestätigung von Aufgaben der Fall sein.
2. **AGB durch Bestimmungen des durch Fides geschlossenen Vertrags:** Die Vorlage selbst kann auch die Bedingungen an eine AGB erfüllen. Nach [95] ist hier eine Inhaltskontrolle notwendig.

AGB sind somit in Fides unmittelbar integrier- bzw. umsetzbar.

Rechtshindernde Einwendungen Fides kann nach [95] die Textform einhalten und ermöglicht die Speicherung der Übereinkunft auf „dauerhaften Datenträgern“, da *clients* die jeweiligen Transaktionen über das Netzwerk in ihren lokalen Datenbestand aufnehmen. Die genutzten öffentlichen Schlüssel

reichen für die jeweiligen Erklärungen aus, da hierfür nach [95] auch „ein Wahl- oder Spitzname“, jedoch nicht der bürgerliche Name, genügt. Weiterhin kann sowohl ein Contract im Ganzen, als auch abgeschlossene Aufgaben in Form von Transaktionen als abgeschlossene Erklärung verstanden werden [95]. Konträr zur Textform können nach [95] andere Formen nicht eingehalten werden. Hierzu zählen:

- Schriftform
- Elektronische Schriftform
- Öffentliche Beglaubigung
- Notarielle Beurkundung

Rechtsvernichtende Einwendungen Das Rechtsgutachten betrachtet bei der Einordnung etwaiger rechtsvernichtender Einwendungen unterschiedliche Klassifizierungen möglicher Verträge, welche über Fides geschlossen werden. Hierbei kann Fides zunächst als Fernkommunikationsmittel verstanden werden, welches es erlaubt Fernabsatzverträge zu erstellen. Dies ist abhängig von der Nutzung des Systems durch die involvierten Parteien. Hierbei müssten beide Parteien ausschließlich Fides verwenden und kein sonstiger persönlicher Kontakt vorhanden sein. [95]

Für Verbraucherverträge, die über Fides geschlossen werden, ergeben sich keine Besonderheiten [95].

Ferner besteht theoretisch die Möglichkeit, dass Fides verwendet wird, um Verträge außerhalb von Geschäftsräumen zu schließen, wenn die Parteien die Software zusammen an einem Ort, welcher kein Geschäftsraum ist, verwenden. [95]

Nach [95] ist es zudem möglich, die über Fides geschlossenen Verträge als Verträge im elektronischen Geschäftsverkehr zu klassifizieren. Hierbei kann Fides als Informations- und Kommunikationsdienst verstanden werden, je-

doch nicht, um ein Telekommunikationsdienst, da Fides primär darauf abzielt Verträge zu schließen und somit keine direkte Messengerfunktion enthält [95]. Weiterhin ist Fides weder ein telekommunikationsgestützter Dienst noch Rundfunk, sondern nach [95] ein Telemedium.

Abschließend betrachtet das Gutachten noch etwaige Pflichten für die Betreuung eines Onlinemarktplatzes, insbesondere im Bezug auf Informationspflichten. Hierbei hat die Nutzung von Fides keine Auswirkungen. Weiterhin hat Fides keine vermittelnde Eigenschaft und ist lediglich das Medium zum Vertragsschluss [95].

Rechtshemmende Einreden Laut [95] ergeben sich keine rechtshemmende Einreden für Ansprüche an Verträge durch die Verwendung von Fides. Selbiges gilt auch für Besonderheiten des Handelsrechts.

Vergleich mit Stand der Forschung Das Rechtsgutachten macht ferner Angaben dazu, wie Fides im Vergleich zu anderen Systemen des Forschungsfelds einzuordnen ist. Hierbei nehmen die Autoren in [95] Bezug auf die bereits in Kapitel 4.3 beschriebenen Punkte zur rechtlichen Lage in Deutschland. Das Gutachten hebt positiv hervor, dass die rechtlichen Fragestellungen auf blockchainbasierten Smart Contracts, welche eine Programmiersprache und nicht natürliche Sprache verwenden, nicht auf Fides übertragen werden können. Fides kann wie bereits erwähnt die Textform erfüllen, worin bei der Nutzung von anderen Systemen Uneinigkeit herrscht. Keine Unterschiede ergeben sich bei der Betrachtung von AGB, Fernabsatzverträgen und Verträgen im elektronischen Geschäftsverkehr zwischen Fides und anderen Smart Contract Systemen. [95]

„Aufgrund der grundlegend unterschiedlichen Funktionsweise von Fides im Vergleich zum Einsatz blockchainbasierter Smart Contracts ergeben sich beim Einsatz des Systems aus rechtlicher Sicht an einigen Stellen Unterschiede. Aufgrund der Unterschiede wer-

den jedoch in der Literatur bestehende Zweifels- und Streitfragen eher ausgeräumt, was die Einsatzfähigkeit von Fides rechtssicherer gestaltet.“ [95]

Nutzung von LoRaWAN Neben der allgemeinen zivilrechtlichen Betrachtung ordnet das Rechtsgutachten [95] auch die Verwendung der in dieser Arbeit entwickelten LoRaWAN Abstraktion in Kombination mit der Middleware, beides vorgestellt in Kapitel 7, ein. Nach [95] hat die Verwendung von LoRaWAN keine Auswirkungen auf das Schließen von Verträgen. Hierbei spielt LoRaWAN nur eine Rolle bei der Übermittlung, um das System in Regionen ohne direkte Internetverbindung nutzbar zu machen:

„LoRaWAN kommt vielmehr die rechtliche Rolle eines Boten zu, der die Willenserklärung des Clients transportiert, übersetzt und an der vom Client vorgegeben Stelle im Contract platziert.“[95]

Die Willenserklärungen werden so auf andere Weise übermittelt, sind jedoch dem *client* unmittelbar zuzuordnen. Weiterhin stellt LoRaWAN im Allgemeinen sicher, dass die Nachricht sich über die Middleware dem korrekten *client* zuordnen lässt. Weiterhin ist nach [95] die Nutzung von variablen Platzhaltern innerhalb eines Contract ebenfalls unproblematisch. Zu beachten sei die Notwendigkeit der Kenntnisnahme von AGB oder anderen Informationen (z. B. Widerrufsrecht) [95]. Diese Informationen lassen sich unmittelbar in die Vorlagen integrieren und können somit auch offline zugänglich sein. Lediglich kann die asynchrone Aktualisierung der Contracts für den Zeitpunkt von Abgabe und Zugang der Willenserklärungen relevant werden [95]. Dies hängt entsprechend davon ab, wie die LoRaWAN Abstraktion verwendet wird und wie oft anhand der die durch LoRaWAN gegebenen Limitierungen der Bandbreite geprüft werden kann, ob sich ein Contract verändert hat. In der Regel sollte dies nicht zu Problemen führen.

8.1.3 Datenschutzrechtliche Betrachtung

Nachfolgend werden die datenschutzrechtlichen Erkenntnisse des Rechtsgutachtens [95] zusammengefasst. Nach [95] greift die DSGVO (sofern beteiligte *full node* oder *client* aus der EU sind), da bei der Nutzung von Fides personenbezogene Daten verarbeitet werden, auch wenn jene z. B. *full nodes* lediglich pseudonym vorliegen.

Datenschutzrechtliche Verantwortlichkeiten Das Gutachten beschreibt in [95] die Problematik der Bestimmung der datenschutzrechtlichen Verantwortlichkeit durch die dezentrale Struktur in Kombination mit der Offenheit des Systems. Entsprechend gehen die Autoren davon aus, dass „gemeinsame Verantwortlichkeit der beteiligten Akteure“ besteht. Entsprechend sind nach [95] sowohl die Betreiber von *full nodes*, als auch *clients* gemeinsam verantwortlich. Selbiges gilt, sofern de LoRaWAN Abstraktion verwendet wird.

Rechtmäßigkeitsprinzip Nach [95] ist die Verarbeitung von personenbezogene Daten nur dann rechtmäßig, „wenn eine Einwilligung oder ein gesetzlicher Erlaubnistatbestand dies rechtfertigen“. Innerhalb von Fides (inklusive LoRaWAN Abstraktion) kann die Datenverarbeitung zur Vertragserfüllung laut [95] eine Rechtsgrundlage sein.

Pflichten des Datenverarbeiters Dadurch, dass in Fides personenbezogene Daten verwaltet werden, geht dies laut [95] mit besonderen Pflichten der DSGVO entsprechend einher. Innerhalb der rechtlichen Beurteilung der Software wurden hier die für Fides relevantesten Anforderungen der DSGVO betrachtet:

1. **Datenschutz durch Technikgestaltung und durch datenschutzfreundliche Voreinstellungen (Art. 25)**¹⁴: Nach [95] verhalten sich offene und dezentrale Netzwerke widersprüchlich zu den in der DSGVO

¹⁴vgl. <https://dejure.org/gesetze/DSGVO/25.html> (Abgerufen im Februar 2023)

geforderten „datenschutzfreundliche Voreinstellungen“. Die eingesetzte Verschlüsselung und Pseudonymisierung werden nach den Autoren werden zum Teil bereits als „privacy by design“ bzw. „privacy by default“ angesehen [95]. Problematisch wird es, da nur Teile der Datenverarbeitung und nicht der gesamte Prozess so abgewickelt werden, was durch die offene Struktur des Netzwerks entsteht und unter anderem Metadatenanalysen (in der Regel durch *full nodes*) theoretisch möglich macht. In der Praxis hängt die konkrete Ausgestaltung des Netzwerks bzw. notwendige Änderungen von den Verträgen, welche über Fides geschlossen werden sollen, ab [95]. Das Gutachten hebt positiv hervor, dass die verschlüsselten Daten innerhalb von Contracts nur von den involvierten Parteien gelesen und verarbeitet werden können und es nur dem ersten *full node*, welche die Transaktion zuerst empfängt, möglich ist, die IP-Adresse des *client* in Erfahrung zu bringen¹⁵.

2. **Sicherheit der Verarbeitung (Art. 32)**¹⁶: Bezogen auf die Sicherheit der Verarbeitung der Daten wird in [95] erneut die Pseudonymisierung, Verschlüsselung und die Möglichkeit für private Netzwerke positiv hervorgehoben. Dennoch wird auf etwaige Konflikte bei den eingesetzten kryptographischen Verfahren hingewiesen. Dies gilt insbesondere, wenn die Integrität der Daten (durch die Verfahren) in Konflikt steht mit dem Recht auf Berichtigung oder Löschung [95]. Für die Nutzung der LoRaWAN Abstraktion muss sichergestellt werden, dass die Funkübertragung und die eingesetzte Hardware (Raspberry Pi, LoRaWAN Gateway/Modul) die Anforderungen an die Sicherheit im Bezug auf die Übertragung der Daten gewährleisten kann [95].

Betroffenenrechte Das Gutachten analysiert in [95] ferner die Betroffenenrechte für Personen, deren Daten verarbeitet werden. Die Autoren be-

¹⁵Dies wäre nur durch die Veränderung der Software möglich, da Fides selbst keine Daten zu den IP-Adressen speichert

¹⁶vgl. <https://dejure.org/gesetze/DSGVO/32.html> (Abgerufen im Februar 2023)

leuchten die Komplexität bei der Bestimmung einer verantwortlichen Partei innerhalb des dezentralen und offenen Netzwerks. Sofern man die Rechte gegenüber einem *full node* geltend machen möchte, muss unterschieden werden, ob jener Akteur identifiziert werden kann oder nicht. Dies ist insbesondere abhängig von der Struktur des Netzwerks und dem Speicherort der Objekte. Ist es einem *client* nicht möglich, den zuständigen *full node* zu identifizieren, so kann nach [95] dem „normativen Transparenzverständnis der DSGVO grundsätzlich nicht entsprochen werden“.

Sieht man *clients* als verantwortlich an, so können etwaige Rechte dort geltend gemacht werden [95].

8.1.4 Zusammenfassung

Zusammenfassend schafft die rechtliche Beurteilung der Ideen und Implementierungen dieser Arbeit Klarheit über die weiteren Einsatzmöglichkeiten der Resultate. Zunächst soll hier erwähnt werden, dass das sehr umfangreiche Rechtsgutachten hier nur knapp umrissen wurde. Das Gutachten [95], welches frei verfügbar verwendet werden kann, detailliert die einzelnen angesprochenen Aspekte und gibt zudem weitere Hinweise bei der Nutzung/Einbindung der in der Arbeit entwickelten Software.

Wichtig für den möglichen Erfolg des Protokolls ist dessen zivilrechtliche Beurteilung. Die Möglichkeit rechtlich bindende Verträge abzubilden fördert dessen Adaption in breiteren Kontexten. So könnte die Software oder Ansätze zum Beispiel von Unternehmen genutzt werden, wohingegen die weiterhin unklaren rechtlichen Rahmenbedingungen für herkömmliche Smart Contract Systeme hier ein Hindernis darstellen.

Durch die dezentrale, offene Struktur und Anonymität innerhalb des Netzwerks ist es hingegen schwierig, allen Anforderungen an den Datenschutz (aus rechtlicher Sicht) gerecht zu werden. Eine zentrale Implementierung, welche die Auflagen einfacher erfüllen könnte, ist zukünftig denkbar. Hierdurch kann die Dezentralität in der Anwendung des Protokolls verstanden werden, bei

welcher diverse verschiedene Dienste auf die gleiche Art und Weise angeboten werden. Dies würde für Nutzende bedeuten, die Abläufe nur einmal verstehen zu müssen, da das mentale Modell für die Erstellung und Bearbeitung von Verträgen, unabhängig von dem jeweiligen zentralen Dienst, gleich bleiben würde. Wichtig ist jedoch, dass das Gutachten in [95] die in der Implementierung gewählten Methoden zur Verschlüsselung/Verifikation dennoch als positiv hervorhebt.

8.2 Messungen der Performanz

Nachfolgend erfolgt eine Darstellung der gemessenen Performanz der Referenzimplementierung Fides. Hierzu wurde die Software in Version 0.5.1 verwendet. Für die Rohdaten und Messergebnisse sei auf den Anhang D.2 der Arbeit verwiesen.

8.2.1 Werkzeuge zur Messung

Zur Wahrung der Reproduzierbarkeit der Messergebnisse wurde *collectl* [3] verwendet, um den Ressourcenverbrauch der Systeme zu messen. Innerhalb der Rohdaten wurden folgende Subsysteme aufgezeichnet: CPU, Disk, Memory, Networks und TCP [4]. Weiterhin wurde bei der Aufzeichnung das *Plot* Format benutzt, um die Daten einfacher darstellen oder in anderen Anwendungen auswerten zu können [4]. Zur Messung von *clients* wurde zusätzlich das Flag *-netfilt* verwendet, um die Daten des korrekten Netzwerkinterfaces aufzuzeichnen [4]. Am Messintervall wurde nichts verändert, somit wurden die genannten Subsysteme einmal pro Sekunde aufgezeichnet [4].

8.2.2 Darstellung der Messergebnisse

Um die von *collectl* aufgezeichneten Daten zu visualisieren, wurde *colplot*, welches als Teil der Anwendung ausgeliefert wird, verwendet. Da das Standardformat der Ausgabe die Grafiken nur sehr schwer lesbar macht, wurde

die Anwendung leicht verändert, um andere Plotformate zu ermöglichen. Die gemachten Änderungen sind in Anhang D.1 erläutert.

8.2.3 Auslastung bei Netzwerkknoten

Nachfolgend werden Messungen verschiedener Szenarien vorgestellt, welche die Auslastung bei der Verwendung von Fides in Abhängigkeit von der Nutzungsart zeigen.

Simulation von Clients Zunächst soll auf die Simulation von *clients* innerhalb des Netzwerks eingegangen werden. Hierzu werden *clients* (Betriebsart *REGULAR*) wie folgt simuliert. Das Python Skript zur Simulation befindet sich in Anhang D.2. Für die Konfiguration des Skripts stehen die folgenden Parameter zur Verfügung:

- **num_clients**: Anzahl an *clients*, die parallel simuliert werden sollen. Für jeden *client* wird eine separate Fides Instanz erzeugt.
- **test_num_iterations**: Iterationen des Messdurchlaufs
- **test_num_of_contracts**: Anzahl der zu erstellenden Verträge pro Iteration
- **test_iteration_sleep**: Wartezeit zwischen Iterationen. Die Verträge können jedoch noch nicht beendet worden sein, da jene im Hintergrund automatisiert werden.

Die Simulation wurde derart konzipiert, dass die Resultate „reale“ Werte liefern sollen und nicht per se darauf ausgelegt sind, den zu messenden Knoten zu überlasten. Entsprechend sollen periodisch Hintergrundaktualisierungen der jeweiligen Instanzen stattfinden, unabhängig davon, ob jene gerade mit der Abwicklung eines Vertrages beschäftigt sind. Gleichwohl wurde sich für separate Instanzen entschieden, da jene die Objekte entsprechend

über das Netzwerk beziehen müssen, um jeweils einen eigenen Datenstand zu erreichen. Allgemein führt das Simulationsskript folgende Aktionen durch:

1. Callback Python Skripts anlegen, welche von allen Instanzen verwendet werden, um Vorlagen und Verträge zu automatisieren.
2. Pro *client* eine separate Fides Instanz anlegen. Hierbei wird der zu messende Knoten als Endpoint hinzugefügt und die jeweilige Instanz konfiguriert. Für das Hintergrundaktualisierungsintervall werden pro Instanz ein zufälliger Wert im Bereich von 1-60 Sekunden gewählt, damit die jeweiligen Hintergrundaktualisierungen zu verschiedenen Zeiten durchgeführt werden. Weiterhin wird ein Account für die jeweilige Instanz angelegt und das Netzwerk gestartet.
3. Erstellen einer Vorlage, die für alle *clients* gleich ist.
4. Speicherung der Instanz für einfachen Zugriff (ID, Pfad zur Instanz, privater Schlüssel des Accounts, Hash der Vorlage).
5. Hook zur Automatisierung der erstellten Vorlage generieren und in der Instanz des *clients* speichern.
6. Hook für Vorlage starten.
7. Innerhalb der verschachtelten Schleife (Iterationen/Anzahl Verträge) jeweils einzelnen Test durchführen:
 - (a) Bestimmen von zwei verschiedenen *clients* c1, c2. Hierbei erstellt c1 den Vertrag, c2 verwaltet die Vorlage.
 - (b) Abrufen des Templates aus dem Netzwerk.
 - (c) Account laden.
 - (d) Vertrag erstellen und finalisieren.
 - (e) Hook für erstellten Vertrag erzeugen und starten.

(f) Vertrag publizieren.

Bevor die Automatisierung der Vorlage erläutert wird, wird die Vorlage selbst noch vorgestellt. Jene enthält sechs Aufgaben, deren Beschreibung „This is a task to the template. Could be anything in the real world“ lautet. Innerhalb der Vorlage werden ausschließlich Validatoren des Typs *PLAIN-TEXT* verwendet. Die Zuordnung zu den Parteien wurde festgelegt als $R = \{False, True, False, False, True, True\}$.

Nachfolgend wird die Automatisierung der Verträge und Vorlagen durch das Modul *Hooks* kurz beschrieben. Für das Template, welches für jede Instanz verwendet wird, wird jeweils folgende *Hook* angelegt:

```
{
  "type": "TemplateUsedHook",
  "hash": "{Hash der Vorlage}",
  "interval": 2,
  "callback": "/tmp/fides_simulation//hooks/callback_template.py",
  "method": "callback_fn_template",
  "live_forever": true,
  "args": {
    "instance": "/tmp/fides_simulation/0/.fides/"
  }
}
```

Entsprechend wird alle zwei Sekunden geprüft, ob neue Verträge im Netzwerk existieren, welche die Vorlage verwenden. Sofern dies der Fall ist, wird das Callback *callback_fn_template* aufgerufen, welchem die Instanz als Argument übergeben wird. Hierdurch ist es möglich, die gleiche Automatisierung für beliebig viele Instanzen zu benutzen. Der genaue Ablauf ist wie folgt:

1. Lade den Vertrag, welcher die Vorlage benutzt.

2. Lade den privaten Schlüssel des Accounts innerhalb der übergebenen Instanz.
3. Nehme den Vertrag an.
4. Solange sich der Vertrag noch nicht im Zustand *FINISHED* befindet, prüfe, ob die aktuelle Instanz für die nächste Aufgabe zuständig ist und bestätige den Vertragsschritt. Dieser Schritt wird in einen separaten Thread ausgelagert, um die Parallelität zu gewährleisten.

Allgemein wird bei der Bestätigung von den Vertragsschritten eine zufällige Zahl i zwischen 100 und 50000 bestimmt und somit $i \cdot a$ übertragen. Dies stellt sicher, dass die empfangenen Transaktionen des zu messenden Knotens unterschiedliche Größen aufweisen.

Für die Automatisierung der Verträge werden bei der erstellenden Partei ebenfalls *Hooks* verwendet. Jene werden wie folgt dynamisch angelegt:

```
{
  "type": "ContractStateChangedHook",
  "hash": "{Hash des Vertrags}",
  "interval": 2,
  "callback": "/tmp/fides_simulation//hooks/callback_contract.py",
  "method": "callback_fn_contract",
  "live_forever": true,
  "args": {
    "instance": "/tmp/fides_simulation/0/.fides/",
    "contract_hash": "{Hash des Vertrags}"
  }
}
```

Auch hier wird alle zwei Sekunden die Bedingung geprüft. Neben der Instanz wird hier noch zusätzlich der Hash des Vertrags übergeben, damit

man auch hier leichter die Zuordnung vornehmen kann und für alle Instanzen die gleiche Automatisierung verwenden kann. Der Ablauf der Methode des Callbacks ist ähnlich zur Automatisierung der Vorlage:

1. Lade den Vertrag, welcher als Argument übermittelt wurde.
2. Prüfe die Zuständigkeit der aktuellen Aufgabe.
3. Sofern zuständig, bestätige die Aufgabe.

Bei der Bestätigung der Aufgabe wird hier ebenso eine zufällige Zahl i bestimmt, welche hier zwischen 50 und 70000 liegen kann. Entsprechend wird $i*b$ innerhalb der Transaktion verschlüsselt übertragen.

Spezifikation der Hardware Für die Messung wurden virtuelle Server mit gleicher Leistung, dem Betriebssystem Ubuntu 18.04 und Python in Version 3.8 verwendet. Die Spezifikation der Prozessorleistung wurde exemplarisch für einen Server in Tabelle 15 angegeben. Die Informationen entstammen der *lscpu* Anwendung [12]. Weiterhin verfügt der virtuelle Server über 4GB an Arbeitsspeicher.

Modellname	Intel(R) Xeon(R) CPU E5-2640 v3
Socket	1
Kern(e) pro Socket	2
Thread(s) pro Kern	1
CPU MHz	2600
L1d Cache	32K
L1i Cache	32K
L2 Cache	256K
L3 Cache	20480K

Tabelle 15: Prozessorinformationen eines Netzwerkknotens

Durchführung der Messung Bei der Messung wurde sich nicht auf den einzelnen Prozess des Indexknotens fokussiert, sondern das Gesamtsystem gemessen, um so die Mehrbelastung auf das System während der Messung zu verdeutlichen, um damit einschätzen zu können, was bei der zusätzlichen Verwendung von Fides auf einem bestehenden System zu erwarten ist. Während der Messung war weiterhin eine SSH Verbindung über ein VPN zum Simulationsrechner hergestellt, um etwaige Fehler/Ausfälle des Indexknotens direkt feststellen zu können. Da ein solcher Server im Allgemeinen einen gewissen Ressourcenverbrauch aufweist, dient die Zeit vor und nach der Messung zur deutlichen Abgrenzung davon und dazu, den ungefähren Ressourcenverbrauch im Leerlauf zu verdeutlichen.

Szenario 1 Das erste Szenario soll die pure Leistungsfähigkeit von Fides beleuchten. Daher werden die Anfragen an ein Netzwerk gerichtet, was aus lediglich einem Knoten besteht, um Netzwerkeffekte bei der Herstellung und Wartung der Distributed Hash Table (DHT) in Teilen ausblenden zu können. Für die Messungen kontaktieren *clients* einen einzelnen im Internet erreichbaren *full node*. Entsprechend kann nicht ausgeschlossen werden, dass andere *clients* (fernab der Simulation) im gleichen Zeitraum Anfragen an den Indexknoten stellen. Daher gibt die Messung Auskünfte über die Mehrbelastung innerhalb eines konkreten Zeitraums.

Versuch 1:

- Anzahl an *clients*: 20
- Anzahl an Iterationen: 5
- Verträge pro Iteration: 20
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten

- Dauer der Simulation: 12 Minuten
- Leerlauf nach Messung: 5 Minuten

Bei dem ersten Versuch traten weder lokal noch auf dem Indexknoten Fehler auf. Die CPU Auslastung des Indexknotens übersteigt während der Simulation nicht 20 % und beläuft sich über den kompletten Messzeitraum im Durchschnitt auf 6,1 % (CPU User). Die Speicherauslastung des Systems ist nahezu konstant und durch die Last der Simulation unverändert. Aus früheren Messungen [61] ist zu entnehmen, dass sich der prozentuale Speicherverbrauch des Fides-Prozesses um den Bereich von 1 % bewegt. Der Zeitraum der Simulation lässt sich ebenso an den Festplattenzugriffen ablesen, wo primär Schreiboperationen durch das Sichern der Transaktionen bei der Abarbeitung der Übereinkünfte ausgeführt werden. Ebenso deutlich sieht man den Simulationszeitraum bei der Betrachtung des Netzwerkverkehrs. Da der Knoten alleine im Netzwerk ist, ist der ausgehende Netzwerkverkehr deutlich größer als der eingehende, da Transaktionen nicht im Netzwerk verteilt werden. Weiterhin ist anzunehmen, dass die Auslastung der Festplatte mit mehr Indexknoten kleiner wird, da entsprechend weniger pro Knoten gespeichert werden muss.

Versuch 2:

- Anzahl an *clients*: 40
- Anzahl an Iterationen: 5
- Verträge pro Iteration: 20
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 13 Minuten
- Leerlauf nach Messung: 5 Minuten

Im zweiten Versuch wurden die Anzahl der *clients* verdoppelt. Dies bewirkt somit mehr Hintergrundaktualisierungen der jeweiligen Instanzen, da jede simulierte Partei die erstellte Vorlage aktuell hält. Weder auf den simulierten Instanzen noch auf dem Indexknoten traten während der Simulation Fehler auf. Die Auslastung der CPU ist ähnlich zum ersten Versuch und beläuft sich im Durchschnitt auf 6,8 % (CPU User). Weiterhin ist die Speicherauslastung des gemessenen Systems über den gesamten Messzeitraum nahezu konstant. Bei den Festplattenzugriffen lässt sich eine leicht höhere Auslastung erkennen. Dies ist auch sichtbar an der Prozessorauslastung (CPU Wait). Im Allgemeinen führt die durch das Protokoll selbst limitierte Transaktionsgröße (1MB im Maximum) doch hierbei nicht zu nennenswerten Auslastungen. Die Netzwerkauslastung zeigt erneut das ungleiche Verhältnis zwischen ein- und ausgehenden Daten, was an dem einzelnen Indexknoten im Netzwerk liegt.

Versuch 3:

- Anzahl an *clients*: 20
- Anzahl an Iterationen: 5
- Verträge pro Iteration: 40
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 13 Minuten
- Leerlauf nach Messung: 5 Minuten

Dadurch, dass sich im zweiten Versuch kaum eine Änderung bei der Anzahl an *clients* zeigte, sollen im dritten Versuch mehr Verträge pro Iteration erstellt werden. Hier konnten zum ersten Mal Fehler bei der Messung beobachtet werden. Durch die Parallelität der Anfragen und die geringe Rechenleistung traten beim Indexknoten Fehler auf, als jener versuchte sich selbst zu kontaktieren. Dies beruht darauf, dass der Knoten zwar allein im Netzwerk ist,

jedoch die DHT Funktionen in dieser Version der Software nicht in Gänze abgeschaltet sind. Entsprechend versucht der Knoten bei jeder Anfrage durch die simulierten *clients* zu bestimmen, wer für diese Anfrage verantwortlich ist. Um dies umzusetzen, stellt der Indexknoten eine RPC Anfrage an sich selbst, um den richtigen Ort zur Speicherung des Objekts zu finden. Da diese Anfrage (durch die Überlastung bei der Simulation) nicht schnell genug bearbeitet werden kann (Timeout 1 Sekunde), erreicht der Knoten sich selbst nicht schnell genug, was zu Problemen bei der Bearbeitung der Übereinkünfte führt. Zwar gibt er sich selbst zurück, um die Speicherung der Daten zu übernehmen, jedoch beeinflussen sich die sich überlappenden Anfrage gegenseitig, da für jede erneute Anfragen immer wieder die Zuständigkeit geprüft wird. Für die simulierten *clients* ergaben sich hieraus kleinere Verzögerungen beim Erreichen des Knotens, die jedoch nicht zu Fehlern bei der Abarbeitung der Übereinkünfte führten. Die durchschnittliche Prozessorauslastung betrug 7,2 % (CPU User). Weiterhin lässt sich im Vergleich zu den vorherigen Versuchen eine leicht höhere wartende Auslastung der CPU ablesen. Die Speicherauslastung ist erneut ähnlich zu den vorherigen Versuchen und somit nahezu konstant. Auch die Festplattenzugriffe und der Netzwerkverkehr weisen keine deutlichen Unterschiede zu den vorherigen Versuchen auf.

Einordnung der Messergebnisse Die Versuche vom ersten Szenario zeigen, dass die Verwendung von Fides als Netzwerkknoten nur eine geringe Belastung des Systems auslöst und auch auf schwächerer Hardware problemlos möglich ist. Die Probleme aus Versuch 3 beziehen sich auf den seltenen Fall, dass ein Knoten alleine das Netzwerk darstellt, also eine zentralistische Verwendung von Fides. Da es als eher unüblich anzunehmen ist eine DHT langfristig mit einem Knoten zu verwenden, verzichtete die Software bisher auf die Optimierung dieser Überprüfung. Abhilfe schaffen würde hier, wenn jeder Indexknoten vor Prüfung der Zuständigkeit validiert, ob er bereits mit anderen Knoten verbunden ist. Sollten diese Verbindungen nicht bestehen,

so ist die Zuständigkeit bei dem Knoten selbst und man braucht die RPC Anfrage (Knoten kontaktiert sich selbst) nicht durchzuführen. Sofern Fides in zentralistischen Anwendungen Verwendung finden sollte, kann auch der gesamte DHT Code einfach entfernt werden, um so nur den Index-Service zu bearbeiten (siehe Abbildung 14).

Szenario 2 Im zweiten Szenario wird ein Fides Netzwerk bestehend aus insgesamt vier *full nodes* aufgebaut. *clients* kontaktieren dennoch erneut einen einzelnen Indexknoten, der mit den drei weiteren *full nodes* ein Netzwerk bildet. Die anderen drei *full nodes* sind nicht direkt über das Internet erreichbar (VPN), daher wird jeder Request direkt an den zu messenden Knoten gestellt. Der zu messende Knoten wird über das Internet kontaktiert. Hierbei befindet sich der Rechner, welcher die *clients* simuliert, ebenso im VPN. Da es sich um einen öffentlichen Knoten handelt, kann nicht ausgeschlossen werden, dass parallel zur Simulation noch Anfragen von anderen Akteuren verarbeitet wurden. Die Simulation gibt daher nur Aufschlüsse über die Mehrbelastung des Knotens durch die Simulation. Nachfolgend werden die Versuche aus dem ersten Szenario wiederholt, um die Unterschiede zwischen einem einzelnen Indexknoten und dem Zusammenschluss von Rechnern zu einem Netzwerk zu zeigen.

Versuch 1:

- Anzahl an *clients*: 20
- Anzahl an Iterationen: 5
- Verträge pro Iteration: 20
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 12 Minuten

- Leerlauf nach Messung: 5 Minuten

Die Simulation wurde erfolgreich beendet. Weder lokal noch auf dem Indexknoten traten Fehler auf. Alle Verträge wurden korrekt durch die Automatisierung bearbeitet. Allgemein lassen sich bereits im Leerlauf vor und nach der Messung Unterschiede erkennen, welche auf die Stabilisierung der DHT zurückzuführen sind. Somit ist der allgemeine Ressourcenverbrauch als Teil eines größeren Netzwerks leicht erhöht. Der Anstieg der Prozessorauslastung während der Simulation ist zudem erkennbar, verhält sich im Durchschnitt aber auch gering (6,6 % CPU User). Ein besonderer Unterschied zum ersten Szenario ist im Bereich der CPU Wait Auslastung des Systems, welche deutlich geringer ist als im gleichen Versuch des ersten Szenarios. Dies ist zurückzuführen auf das Weiterleiten der Transaktionen an die anderen drei *full node* Knoten. Da jener Wert ein Indikator für Ein- und Ausgabeoperationen ist, unterstreicht dies auch die deutliche Reduktion der Auslastung der Festplatte während des Versuchs. Während im ersten Szenario Spitzen von ca. 3 MB in der Messung auftraten, liegt der Maximalwert bei dem zweiten Szenario ca. bei 0.65 MB. Bei der Speicherauslastung zeigt sich erneut ein etwa konstanter Verbrauch, der ähnlich zum ersten Szenario ist. Weitere Unterschiede sind bei der Netzwerkauslastung erkennbar. Hier sieht man nicht nur kleinere Ausschläge vor und nach der Simulation, welche auf die Wartung der DHT zurückzuführen sind und zeitgleich zu den Ausschlängen bei der Prozessorauslastung auftreten, sondern auch, dass der ein- und ausgehende Netzwerkverkehr in etwa gleich ist. Die gleiche Auslastung beruht auf dem Weiterleiten der Transaktionen an andere Indexknoten und dem Weiterleiten der Antworten der jeweiligen Anfragen zurück an die simulierten *clients*.

Versuch 2:

- Anzahl an *clients*: 40
- Anzahl an Iterationen: 5

- Verträge pro Iteration: 20
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 12 Minuten
- Leerlauf nach Messung: 5 Minuten

Die Ergebnisse des zweiten Versuchs unterscheiden sich nur unwesentlich von denen des ersten Versuchs. Bei der Simulation kam es weder lokal noch im Netzwerk zu Fehlern. Anhand der etwas höheren CPU-Auslastung während der Simulation lässt sich der Messzeitraum deutlich erkennen. Insgesamt belief sich die durchschnittliche Prozessorauslastung (CPU User) auf 7,8 %. Bei der Speicher- und Netzwerkauslastung zeigen sich im Vergleich zum ersten Versuch des Szenarios kaum Unterschiede. Die Schreibzugriffe auf die Festplatte nehmen in der zweiten Hälfte des Tests zwar zu, bleiben jedoch deutlich geringer als im ersten Szenario und sind ähnlich zu dem vorherigen Versuch. Die Mehrbelastung lässt sich mit der Zuordnung der dort gesendeten Objekte erklären, also, dass der gemessene Knoten dafür verantwortlich war genau diese Objekte zu speichern.

Versuch 3:

- Anzahl an *clients*: 20
- Anzahl an Iterationen: 5
- Verträge pro Iteration: 40
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 37 Minuten

- Leerlauf nach Messung: 5 Minuten

Beim dritten Versuch ließen sich Verzögerungen auf dem Simulationsrechner beobachten, die auf die Parallelität der Tests, den damit verbundenen Kommunikationsaufwand und die allgemein für die Anzahl an Verbindungen zu geringe Rechenleistung der verwendeten (virtualisierten) Hardware der Indexknoten zurückzuführen sind.

Auf dem Indexknoten lassen sich nach ca. 8 Minuten nach Beginn der Simulation Ausfälle beobachten. Der Knoten kann sich lokal selbst nicht mehr erreichen, um die DHT Funktionen aufrecht zu erhalten. Dies geschieht, da in der verwendeten Fides Version innerhalb des externen Moduls *Grond* pro RPC Aufruf ein Timeout von einer Sekunde festgelegt wurde. Das Timeout soll langsame Knoten aus dem Netzwerk entfernen, um die DHT Struktur robust umzusetzen. Durch die Überlastung bei den parallelen Anfragen durch die separaten Instanzen und deren Automatisierungen, verursacht dieses Timeout die Störungen der DHT. Nicht nur kann sich der Knoten nicht mehr selbst erreichen (vergleiche Szenario 1/Versuch 3), sondern er kann von den anderen Indexknoten aufgrund des gleichen Timeouts ebenfalls nicht erreicht werden. Somit entsteht eine Situation des Netzwerks wie in Abbildung 9 (DoS Angriff) dargestellt. Im konkreten Fall wird bei diesem Fehler der selbe Knoten zur Speicherung der Objekte verwendet, da andere nicht erreicht werden können. Die Kommunikationsversuche verzögern entsprechend die Bearbeitung der Anfragen und erklären die Dauer der Simulation. Da die simulierten *clients* jedoch nur den einen Server als Einstiegspunkt in das Netzwerk verwenden, gab es insgesamt bei der Bearbeitung der Übereinkünfte nur geringe Probleme. So konnten Objekte, die zuvor auf anderen Knoten gespeichert wurden, ab dem Zeitpunkt der Überlastung des Indexknotens nicht mehr gefunden werden, da die jeweiligen Anfragen durch das Timeout nicht bearbeitet wurden. Entsprechend teilte der gemessene Indexknoten den *clients* mit, dass die Objekte nicht gefunden wurden, was zum Teil dazu führte, dass jene Objekte republiciert werden mussten. Durch das Republi-

zieren entstanden somit verschiedene Zustände bei den jeweiligen Parteien, insbesondere wenn ein *client* einen nicht aktuellen Indexzustand auf den lokalen Stand anwenden wollte und hierdurch Fehler entstanden (siehe 5.2.4). Diese Fehler wurden nach und nach durch die Hintergrundaktualisierungen aufgelöst, wodurch die Übereinkünfte insgesamt (auf dem einen Indexknoten) korrekt bearbeitet werden konnten.

Die Ausfälle, die nach Eintreten über einen längeren Zeitpunkt stattfanden, wurden auch in der Messung des Versuchs deutlich.

Man kann erkennen, dass die Speichernutzung weiterhin sehr gering war. Die Prozessorauslastung stieg im Vergleich zu den vorherigen Versuchen stärker an und ist vergleichbar mit der Prozessorauslastung gegen Ende der anderen Versuche, wo deutlich mehr Objekte von den simulierten *clients* aktuell gehalten wurden und somit durch die Hintergrundaktualisierungen entsprechende Auslastung entstanden ist. Auch hier lässt sich der Zeitpunkt erkennen, ab dem die DHT nicht mehr funktionierte. Hier bricht die CPU-Auslastung (User) unmittelbar ein und der Anteil von CPU Wait vergrößert sich.

Dass der Knoten die Dokumentation der Objekte in großen Teilen alleine übernommen hat, erkennt man auch an der Messung zur Festplatte und der Netzwerkauslastung. Die Netzwerkauslastung zeigt zudem die Reduktion des Durchsatzes zum Zeitpunkt der DHT Fehler.

Die einzelnen Spitzen bei CPU und Festplattenzugriffen deuten auf einzelne erfolgreiche Kommunikationsversuche hin, also dem sporadischen Bearbeiten von einzelnen Anfragen trotz der allgemeinen Überlastung des Gesamtsystems.

Einordnung der Messergebnisse Die Messergebnisse der Versuche zeigen, dass die Anfragen korrekt verteilt werden, wenn sich mehrere Indexknoten zu einem Netzwerk zusammenschließen. Man erkennt, dass die einzelne Auslastung pro Knoten bei der Bearbeitung von Anfragen geringer ist (ins-

besondere Festplattenzugriff und CPU Wait). Bei dem dritten Versuch der Simulation wurde die verwendete Hardware schlichtweg überlastet, was zur Störung des Netzwerks führte. Positiv hervorzuheben ist, dass Fides trotz der Störung nicht abgestürzt ist (weder lokal noch auf dem Simulationsrechner) und die Übereinkünfte trotzdem in eingeschränkter Form bearbeitet wurden. Nimmt man für reale Anwendungsszenarien nun an, dass verschiedene *clients* verschiedene Indexknoten für ihre Anfragen wählen, so wird nicht nur die Last von den Indexknoten verteilt, sondern auch durch die *clients*, da innerhalb der Software (sofern verfügbar) immer verschiedene Knoten für die jeweilige Anfrage ausgewählt werden. Nachfolgend wird noch auf die technischen Aspekte der Überlastung eingegangen werden und Abhilfen diskutiert, um jenen Problemen entgegenwirken zu können. Der gRPC Server der DHT (Grond) verwendet den Standardwert für die Anzahl an Threads (innerhalb eines Pools) zur Bearbeitung der Anfragen. In der verwendeten Python Version 3.8 ergibt sich dies zu $\{\text{Anzahl der CPUs}\} + 4$ [82], wodurch bei der verwendeten Hardware somit sechs Threads für die Bearbeitung der Anfragen zur Verfügung stehen. Weiterhin können beliebig viele RPC Anfragen an den Server gerichtet werden (Konfigurationselement *maximum_concurrent_rpcs*), ohne dass der Server vorher zurückmeldet, dass seine Ressourcen erschöpft sind [102]. Würde man diesen Wert limitieren (zum Beispiel gleiche Anzahl wie der verwendete Threadpool), würde sich lediglich das Problem verlagern, da die RPC zwischen den *full nodes* und *clients* kombiniert sind (siehe Abbildung 14) und es somit zu den gleichen Fehlern kommt, wenn RPC Anfragen anderer *full nodes* abgelehnt statt ausreichend schnell bearbeitet werden. Da *clients* immer neue Verbindungen zu einem zufälligen, bekannten Knoten aufbauen, werden jene Verbindungen nicht wiederverwendet, was erkennen lässt, wie ein einzelner, schwacher Server schnell überlastet werden kann. Bei der Festlegung der Parameter (z. B. 1 Sekunden Timeout für Anfragen von Indexknoten) muss somit zwischen Stabilität des Netzwerks und dem schnelleren Ausschließen einzelner, langsamer Netzwerkknoten abgewogen werden. Das

festgelegte Timeout erlaubt der DHT sich schnell zu stabilisieren und somit das Routing korrekt durchzuführen. Eine Abhilfe zur Behebung der Fehler auf schwächerer Hardware wäre unter anderem das Einführen des gleichen Intervalls für *clients* bei dem Kontaktieren eines Indexknotens. Entsprechend würden *clients* den jeweiligen Knoten ignorieren, wenn er nicht schnell genug antwortet. In der aktuellen Version von Fides wird dies jedoch nicht durchgeführt, damit auch langsame *clients* mit schlechter Internetverbindung die Möglichkeit haben, das System zu verwenden. Ein optionales Timeout wäre somit ein Weg, den oben genannten Fehlern vorzubeugen und zu verhindern, dass sich die Netzwerke trennen, ohne dass dies von den *clients* bemerkt wird. Weiterhin möglich wäre die Unterscheidung, ob eine RPC Anfrage von einem *full node* kommt und jene Anfragen zu priorisieren. Dies würde Fehler bei den *clients* hervorrufen, jedoch das Netzwerk etwas stabiler machen. Das Problem bei diesem Vorgehen ist dennoch die Angreifbarkeit durch andere *full nodes* und eine damit einhergehende Überlastung durch jene RPC Anfragen. Ein zielführender Ansatz wäre das Betreiben von Index Knoten hinter Load Balancern oder Reverse Proxys, um die Anzahl und Häufigkeit der Anfragen pro IP-Adresse einzuschränken und somit Überlastungen entgegenzuwirken. Die dabei entstehenden Fehler viel-sender *clients* müssten jene dann durch das Verwenden verschiedener Indexknoten umgehen, was sich zudem positiv auf die Lösung des ursprünglichen Problems auswirken würde. Eine andere Lösung könnte ein variables Timeout je nach aktueller Auslastung des jeweiligen *full node* sein, um so den Mittelweg zwischen robustem Routing und korrekter Funktionsweise des Index zu gehen. Allgemein können zukünftige Verbesserung des Systems erst dann gemacht werden, wenn mehr Einblicke existieren, wie eine solche Software in realen Anwendungsszenarien verwendet wird. Hierbei muss nicht nur das Verhalten der Nutzerinnen und Nutzer, sondern auch die eingesetzte Hardware betrachtet werden. Zusätzlich muss je nach Anwendungsfall analysiert werden, in welcher Beziehung die *full nodes* zueinander stehen. Hierbei spielen insbesondere deren geografische Position

zueinander und die damit verbundene Wahl der Netzwerkparameter wie das angesprochene Timeout eine wichtige Rolle.

Weitere Messungen Eine ähnliche Messung wurde mit anderen Parametern in [62] durchgeführt. Hierbei wurde für die bessere Reproduzierbarkeit der Arbeit ausschließlich lokal kommuniziert. Hierzu wurden vier Raspberry Pi (Model B Rev. 1.2) als *full nodes* über einen Switch verbunden und die Simulation innerhalb des gleichen Netzwerks ausgeführt. Die Resultate der Versuche sind in den Tabellen 16 und 17 aufgeführt.

# Contracts	CPU user %		CPU wait %		NET Rx KB		NET Tx KB	
	avg	max	avg	max	avg	max	avg	max
5	3.7	18	1.1	15	9.6	339	38.8	504
10	3.9	21	1.1	14	12.9	294	38.2	521
20	4.9	26	1.9	18	19.6	414	61.1	485
40	5.4	22	2.5	20	27.3	373	87.6	613
80	5.4	24	2.8	20	31.6	461	85.5	595
160	6.8	22	4.1	21	38.6	453	112.9	750

Tabelle 16: Simulation von zwei Clients in fünf Iterationen mit einer Wartezeit von 120s

# Clients	CPU user %		CPU wait %		NET Rx KB		NET Tx KB	
	avg	max	avg	max	avg	max	avg	max
2	5.4	24	2.8	20	31.6	461	85.5	595
4	6.4	26	3.4	19	33.3	552	100.8	666
8	4.3	20	1.5	14	15.9	260	50.6	623

Tabelle 17: Simulation von 80 Contracts mit zwei, vier und acht clients in fünf Iterationen mit einer Wartezeit von 120s

Die Resultate zeigen, dass die allgemeine Auslastung des Systems gering ist und somit auch *full nodes* mit schwacher Hardware umgesetzt werden können. Weiterhin zeigt Tabelle 16 die geringen Unterschiede der Auslastung, wenn zwei *clients* eine verschiedene Anzahl an Contracts abwickeln.

Die gleichen Fehler wie bei den vorherigen Versuchen mit den virtuellen Servern konnten auch bei der Simulation von acht *clients* und der Abwicklung von 80 Contracts beobachtet werden und sind in Tabelle 17 dargestellt. Hier sieht man, wie die CPU-Auslastung fällt (vgl. Szenario 2 - Versuch 3) und die Netzwerkauslastung aufgrund der bereits beschriebenen Probleme zurückgeht. Durch die Messungen konnte also bestätigt werden, dass der limitierende Faktor für die Performanz des Systems die Anzahl an parallelen Verbindungen ist, da es erst zu Fehlern kam, als die Anzahl parallel simulierter *clients* in Kombinationen mit den anderen Hintergrundthreads die Anzahl an verfügbaren Threads der genutzten Hardware überschritten haben und es zu den erwähnten Verzögerungen (Timeouts) innerhalb des Netzwerks kam.

8.2.4 Auslastung bei regulärer Nutzung

Nachfolgend soll die Performanz von Fides bei regulärer Nutzung, also von *clients*, untersucht werden. Da es anzunehmen ist, dass deutlich weniger Nutzende einen Beitrag zum Netzwerk leisten werden (*full nodes*), sondern primär als *client* agieren, gibt die Evaluation der Performanz aus Sicht der *clients* zudem wichtige Einblicke wenn Fides bzw. Cypher Social Contracts in zentralistischen Anwendungsszenarien verwendet werden sollen.

Spezifikation der Hardware Die Evaluation betrachtet die Auslastung auf das Gesamtsystem zum einen für ein aktuelles und leistungsfähiges Notebook und zum anderen für einen leistungsschwächeren Raspberry Pi. Innerhalb des ersten Versuch wird das Notebook HP Omen 15-en1177ng¹⁷ verwendet, welches nicht nur leistungsstärker als der im zweiten Versuch verwendete Raspberry Pi 4 ist, sondern auch deutlich mehr Leistung aufweist, als die zuvor eingeführten virtuellen Server, wie Tabelle 18 zeigt. Die Werte wurden wie zuvor anhand der Anwendung *lscpu* [12] bestimmt. Auf dem Notebook

¹⁷<https://support.hp.com/za-en/document/c07538406> (Abgerufen im Februar 2023)

wurde während der Messungen das Betriebssystem „Ubuntu MATE 22.04“ genutzt. Das Notebook verfügt über 16GB Arbeitsspeicher.

Modellname	AMD Ryzen 7 5800H
Sockel	1
Kern(e) pro Sockel	8
Thread(s) pro Kern	2
CPU MHz	4400 (im Maximum)
L1d Cache	256K (8 Instanzen)
L1i Cache	256K (8 Instanzen)
L2 Cache	4M (8 Instanzen)
L3 Cache	16M

Tabelle 18: Prozessorinformationen des Notebooks

Bei dem Raspberry Pi 4, welcher im zweiten Versuch verwendet wurde, handelt es sich um die Version mit 4 GB Arbeitsspeicher. Da das Betriebssystem Raspbian verwendet wurde, wird hierbei jedoch eine andere Version der gRPC Bibliothek verwendet, da die in Fides 0.5.1 verwendete Version nicht mit der in Raspbian verwendeten glic Bibliothek kompatibel ist. Dieses Problem betrifft andere Betriebssysteme für den Raspberry Pi (z. B. Ubuntu) nicht.

Durchführung der Messung Die Simulation nutzt das bereits eingeführte Simulationsskript und bearbeitet die Übereinkünfte über das *full node* Netzwerk, welches zuvor untersucht wurde und insgesamt aus vier Knoten besteht. Die Simulation wurde lokal ausgeführt und die Übereinkünfte wurden über das Internet geschlossen. Hierbei wurde keine VPN Verbindung verwendet. Innerhalb der Messungen wurden immer zwei Fides Instanzen simuliert, um die Übereinkünfte automatisiert bearbeiten zu können. Wird Fides entsprechend in nur einer Instanz auf der Seite der *clients* genutzt, ist der Ressourcenverbrauch noch geringer.

Versuch 1:

- Verwendete Hardware: HP Omen Notebook
- Anzahl an *clients*: 2
- Anzahl an Iterationen: 20
- Verträge pro Iteration: 5
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 42 Minuten
- Leerlauf nach Messung: 5 Minuten

Bei dem Versuch wurden insgesamt 100 Übereinkünfte in 20 Iterationen bearbeitet. Die Parameter wurden bewusst so gewählt, dass die Dauer der Simulation höher ist als zuvor. Die Werte sollten in Teilen reale Interaktionen simulieren, also somit auch das händische Bearbeiten der Übereinkünfte durch Nutzende. Auf dem leistungsstarken Notebook zeigte sich eine sehr geringe allgemeine CPU Auslastung. Im Durchschnitt lag der Verbrauch (CPU User) bei 0,1 % während des Testzeitraums. Einzelne Ausschläge stehen im Zusammenhang mit den Hintergrundaktualisierungen der jeweiligen Instanz und waren nur von kurzer Dauer. Weder bei den Zugriffen auf die Festplatte noch bei dem entstandenen Netzwerkverkehr waren Verzögerungen zu erkennen. Lokal traten während der Simulation keine Fehler auf. Die Speicherauslastung nahm im Zeitraum des Tests leicht zu. Im regulären Fall, fernab dieser Simulation, wäre auch hier eine konstante Speicherauslastung messbar, jedoch erhöhten die verwendeten Hooks zur Automatisierung der Übereinkünfte, welche pro Vorlage und Vertrag erstellt werden, den Speicherbedarf minimal. Bei der manuellen Bearbeitung der Verträge würde diese Auslastung entsprechend nicht entstehen. Weiterhin gibt die geringe Mehrbelastung wichtige

Einblicke in die Kosten für die Automatisierung der Übereinkünfte aus der Sicht von *clients*. So ist den Messungen zu entnehmen, dass Entwicklerinnen und Entwickler die jeweiligen Vorlagen/Verträge ressourceneffizient automatisieren können.

Versuch 2:

- Verwendete Hardware: Raspberry Pi 4
- Anzahl an *clients*: 2
- Anzahl an Iterationen: 20
- Verträge pro Iteration: 5
- Wartezeit nach Iteration: 120 Sekunden
- Leerlauf vor Messung: 5 Minuten
- Dauer der Simulation: 45 Minuten
- Leerlauf nach Messung: 5 Minuten

Der zweite Versuch ist von den Parameter gleich zu dem ersten, verwendet als Hardware nur den zuvor vorgestellten Raspberry Pi. Bei der Messung wurden von `collectl` keine Daten zur Festplattenzugriffen aufgezeichnet, da in dem Computer keine richtige Festplatte sondern nur eine SD-Karte verbaut ist. Innerhalb der Resultate liefert daher CPU Wait Einblicke in aktuelle Eingabe/Ausgabe Operationen. Allgemein war die Auslastung im Vergleich zu dem leistungsstarken Notebook erhöht. Dies zeigte sich besonders in den Ausschlägen bei den Hintergrundaktualisierungen, die zum Teil für CPU Auslastungen (User) von $> 50\%$ verantwortlich waren. Da diese Ausschläge nur kurz auftraten, belief sich die durchschnittliche CPU Auslastung (User) auf $3,7\%$. Die Netzwerkauslastung war ähnlich zum ersten Versuch. Auch hier konnten keine Fehler bei der Bearbeitung der Übereinkünfte festgestellt werden. Weiterhin verhielt sich die Speicherauslastung ähnlich zu der im ersten

Versuch. Den Anstieg des Ressourcenverbrauchs durch die Automatisierung der Übereinkünfte war auch hier zu erkennen.

Einordnung der Messergebnisse Die Resultate der Messungen zeigen, dass Fides für Anwenderinnen und Anwender nur wenig Anforderungen an die Hardware stellt. So ist es möglich, die Anwendung auch problemlos auf kostengünstigen Einplatinenrechnern (z. B. Raspberry Pi) zu verwenden, da selbst dort die Auslastungen für CPU und Speicher gering sind.

Die Messungen lassen auch Rückschlüsse auf die Performanz der LoRaWAN Abstraktion zu, bei welcher noch weniger Ressourcen benötigt werden, da die gesamte Netzwerkkommunikation wegfällt. Innerhalb der LoRaWAN Abstraktion werden lediglich lokal mittels RPC Methoden Transaktionen an den Kommunikations-Daemon *fidesd* gesendet, welche dann übersetzt und über LoRaWAN übertragen werden. In [58] wurde dies in einer anderen Messung bestätigt.

8.3 Vergleich mit Ethereum

Nachfolgend soll das *Cypher Social Contracts* Konzept rudimentär als Ethereum Smart Contract implementiert werden. Hierbei gilt es nicht den vollständigen Funktionsumfang von Fides zu replizieren, sondern es geht lediglich um die Erkenntnisse darüber, welche Kosten die Verwendung des Protokolls auf der Ethereum Blockchain verursachen würde. Die exemplarische Umsetzung vereinfacht daher einige Prozesse und nutzt leicht veränderte Datentypen im Vergleich zur Referenzimplementierung. Nachfolgend wird angenommen, dass das System als einzelner Smart Contract realisiert wird, welcher von beliebigen Parteien genutzt werden kann. Jener Smart Contract soll somit alle Vorlagen und Verträge dokumentieren. Im Folgenden werden zunächst die Funktionen des Smart Contracts beschrieben, danach auf die Kosten (Gaskosten) bei der Interaktion eingegangen.

Der Smart Contract wurde innerhalb der Remix IDE [25] umgesetzt und

dort in Testnetzwerken erprobt. Der dazugehörige Quelltext findet sich im Anhang D.3 der Arbeit.

8.3.1 Datentypen und Funktionen

Zunächst werden die Datentypen und Funktionen des Smart Contracts allgemein eingeführt. Hierbei wird zusätzlich die Funktionsweise der jeweiligen Methoden kurz beschrieben.

Datentypen Innerhalb des Smart Contracts „Fds“ werden Datentypen für Templates und Contracts festgelegt. Allgemein wurde der Smart Contract derart programmiert, dass er nach seiner Veröffentlichung nicht änderbar ist. Grundlegend werden Vorlagen/Verträge nicht über ihren Hash adressiert, sondern über einen Zähler, der durch den Smart Contract vorgegeben wird. Da im Smart Contract kein Index implementiert wurde, entspricht der Index des Arrays dem angesprochenen Objekt.

Um Vorlagen abzubilden, werden innerhalb des *struct* „Template“ nach Definition 7 folgende Elemente gespeichert: $\mathcal{K}_{j,P}$, B , T , R . Somit enthält die Vorlage innerhalb des Ethereum Smart Contracts einen öffentlichen Schlüssel der Partei (entnommen aus der Ethereum Transaktion), eine Beschreibung der Vorlage, die Aufgaben und die Zugehörigkeiten zwischen den Parteien. Die Implementierung verzichtet auf Validatoren V , den Merkle Tree Hash r_M und auf den Hash des Templates \mathcal{H}_{O_j} .

Verträge werden innerhalb des *struct* „Contract“ realisiert und enthalten nach Definition 12: $\mathcal{K}_{i,P}$, $\mathcal{K}_{j,P}$, N , R . Die Referenzen zu der Vorlage werden unmittelbar aus dem innerhalb des Smart Contracts gespeicherten Template bezogen. Die Referenz auf die Vorlage \mathcal{H}_{O_j} wird innerhalb des Smart Contracts als Referenz auf die Nummer des Templates umgesetzt. Weiterhin ergänzt der Smart Contract den Datentyp „Contract“ um ein Feld *tx_inputs*, um die Eingabedaten bei der Bestätigung von Aufgaben direkt im Vertrag

speichern zu können. Wie auch in der Referenzimplementierung enthält das Objekt weiterhin das Feld *current_task* zur Dokumentation der Bearbeitung. Die Zustände der Objekte werden weiterhin als Teil des jeweiligen *structs* umgesetzt. Abschließend dient ein *mapping* innerhalb des Smart Contracts zur Schlüssel-Wert Zuordnung von Zählernummer des Objekts (Template/-Contract) zu den jeweiligen Objektdaten, die im Smart Contract gespeichert sind.

Vorlagen erstellen Um eine neue Vorlage zu erstellen, verfügt der Smart Contract über die öffentliche Funktion *create_template*. Nutzende rufen die Funktion über die Adresse des publizierten Smart Contracts „Fds“ auf und übergeben folgende Parameter:

- **tasks:** Aufgabendefinitionen T
- **responsibility:** Zugehörigkeiten R
- **description:** Beschreibung der Vorlage B

Nach Aufnahme der Vorlage in den Smart Contract antwortet jener mit dem aktuellen Zähler, mit welchem sich in unserem Beispiel die Vorlage verwenden lässt.

Vorlage zurückziehen Sofern der Zustand einer Vorlage von *ACTIVE* auf *INACTIVE* geändert werden soll, kann dies nur von dem Account durchgeführt werden, welcher die Vorlage publiziert hat.

Die Funktion *revoke_template* ändert jenen Zustand, sofern das Template existiert und die zuvor genannte Bedingung erfüllt ist.

Vertrag erstellen Nutzende können die Funktion *create_contract* verwenden, um einen neuen Vertrag innerhalb des Smart Contracts zu erstellen. Hierbei wird der Zähler der Vorlage als Parameter an den Smart Contract innerhalb der Ethereum Transaktion übergeben. Der Smart Contract prüft

danach, ob die zu nutzende Vorlage existiert und ob sie aktiv ist, da nur dann ein Vertrag erstellt werden kann. Zusätzlich übernimmt der Smart Contract die Werte aus der Vorlage, welche im Vertrag referenziert wurden und speichert den Vertrag mit Zustand *OFFER*.

Vertrag annehmen/ablehnen Verträge im Zustand *OFFER* können von der Partei, die die Vorlage veröffentlicht hat, über die Funktionen *accept_contract* und *decline_contract* entsprechend angenommen oder abgelehnt werden. Hierzu prüft der Smart Contract, ob der Vertrag existiert und ob die Vorlage von der sendenden Partei der Transaktion publiziert wurde. Je nach Funktion ändert sich dann der Zustand des Vertrags zu *LIVE* oder *REJECTED*.

Aufgabe bestätigen Zur Bestätigung einer Aufgabe eines Vertrags im Zustand *LIVE* kann die Funktion *confirm_task* genutzt werden. Hierbei prüft der Smart Contract, ob sich der Vertrag im korrekten Zustand befindet und ob die Partei für die aktuelle Aufgabe zuständig ist. Übermittelt wird der Methode der Zähler des Vertrags und Eingabedaten. Im Gegensatz zu Fides werden die Eingabedaten nicht verschlüsselt und könnten somit von jedem auf der Ethereum Blockchain eingesehen werden. Inwiefern man mit verschlüsselten Daten arbeiten könnte, wird später erneut aufgegriffen.

Vorlage/Vertrag abrufen Um den aktuellen Zustand einer Vorlage oder eines Vertrags abrufen zu können, enthält der Smart Contract ferner die Methoden *get_template* und *get_contract*. Das ledigliche Beziehen des Status des jeweiligen Objekts verursacht für Nutzende keine Kosten. Innerhalb des Smart Contracts wird lediglich anhand des übergebenen Zählers geprüft, ob das jeweilige Objekte existiert und entsprechend zurückgegeben.

Aktion	Gas
Smart Contract Veröffentlichung	2480328
Vorlage erstellen	291023
Vertrag erstellen	260767
Vertrag annehmen	58124
Aufgabe bestätigen	69773

Tabelle 19: Gaskosten des Smart Contract

8.3.2 Ausführungskosten

Nachfolgend werden die Ausführungskosten (Gaskosten) zur Umsetzung der zuvor beschriebenen Implementierung aufgeführt. Die Angaben wurden der Remix IDE [25] (Feld „transaction cost“ der Transaktion) entnommen.

Die Vorlage enthält vier Aufgaben, deren Beschreibung „Ethereum Test-aufgabe“ entspricht. Die Beschreibung der Vorlage selbst entspricht „Testbeschreibung“. Bei der Bestätigung einer Aufgabe wird die Eingabe „Testeingabe“ verwendet.

Die Berechnung der Kosten wurde nach [10] durchgeführt.

Tabelle 19 zeigt die Gaskosten der Veröffentlichung und Benutzung des Smart Contracts auf der Ethereum Blockchain. Tabelle 20 setzt die jeweilige Aktion mit historischen Gaspreisen innerhalb des Netzwerks in Verbindung. Hierbei wurden die durchschnittlichen Gaspreise und Ethereum Kurse in den Monaten Oktober-November 2022 via Etherscan [8] bezogen.

Man erkennt, dass die Verwendung eines solchen Smart Contracts hohe Kosten verursachen kann. Insbesondere wenn Übereinkünfte mit vielen Aufgaben und detaillierten Aufgabenbeschreibungen abgebildet werden, wird die Speicherung der Objekte schnell teuer. Auch innerhalb der Ausführung entstehen für beide Parteien Kosten, die unverhältnismäßig zu dem Konzept der *Cypher Social Contracts* und der darunterliegenden generischen (sozialen) Interaktion sind.

Aktion	Datum	Ether Preis	\$
Smart Contract Veröffentlichung	10.10.2022	1290 \$	121 \$
	30.10.2022	1590 \$	47 \$
	09.11.2022	1104 \$	167 \$
Vorlage erstellen	10.10.2022	1290 \$	14 \$
	30.10.2022	1590 \$	5 \$
	09.11.2022	1104 \$	19 \$
Vertrag erstellen	10.10.2022	1290 \$	13 \$
	30.10.2022	1590 \$	5 \$
	09.11.2022	1104 \$	17 \$
Vertrag annehmen	10.10.2022	1290 \$	3 \$
	30.10.2022	1590 \$	1 \$
	09.11.2022	1104 \$	4 \$
Aufgabe bestätigen	10.10.2022	1290 \$	3 \$
	30.10.2022	1590 \$	1 \$
	09.11.2022	1104 \$	5 \$

Tabelle 20: Ungefähre Kosten (USD) des Smart Contracts bei unterschiedlichen Preisen pro Ether.

8.3.3 Offene Fragen

Nachfolgend werden noch einige offene Fragen für die Nutzung des *Cypher Social Contracts* Konzept innerhalb von Ethereum betrachtet.

Eingabedaten von Transaktionen Wie bereits in der Arbeit erwähnt, findet jede Interaktion mit einem Smart Contract transparent statt. Dies bedeutet, dass die Transaktionen, die mit Verträgen interagieren, von jedem einsehbar sind. Um einen *Cypher Social Contract* somit gestützt durch den Ethereum Smart Contract abwickeln zu können, müssten die beiden Parteien die Eingabedaten bei der Bestätigung von Aufgaben zunächst lokal verschlüsseln. Dies wird in der Regel umgesetzt durch eine separate Implementierung für Nutzende. Sofern pro Übereinkunft wie in Fides separate Schlüssel genutzt werden sollen, müssen jene lokal erstellt und jeweils bei den Schritten des Angebots und jener Annahme, äquivalent zu Fides, übermittelt

werden. Dies erhöht die Kosten der jeweiligen Transaktion, da die öffentlichen Schlüssel übermittelt werden müssen. Weiterhin sind keine Berechnungen auf den verschlüsselten Eingabedaten durch den Smart Contract möglich, was bedeutet, dass etwaig implementierte Validatoren, genau wie in Fides, bei den Parteien lokal geprüft werden müssten.

Integration von Zahlungen Die Nutzung von *Cypher Social Contracts* innerhalb eines Smart Contracts eröffnet die Möglichkeit zur Inklusion direkter Bezahlung mit der darunterliegenden Kryptowährung Ether. Unabhängig der unverschlüsselten Transaktionen könnten die Übereinkünfte so automatisch an die Zahlung gebunden werden. Unklar ist hier, inwiefern in der realen Welt geprüft wird, ob die jeweiligen Angaben der Parteien korrekt sind und ob die Aktion durchgeführt wurde. Etwaige Probleme mit Orakeln, die bereits in der Arbeit thematisiert wurden, gelten somit auch hier.

Allgemeine Verbesserung des Smart Contracts Der entwickelte Smart Contract zur Umsetzung des *Cypher Social Contract* Konzeptes dient lediglich zur Einschätzung der Kosten und dem Vergleich der Konzepte. Eine reale Implementierung sollte sich daher einer Sicherheitsprüfung unterziehen und ggf. Änderungen an der Implementierung vornehmen.

9 Ausblick

Nachfolgend werden weitere Anwendungsfelder für die Resultate der Arbeit diskutiert. Hierbei werden zwei Bereiche separat betrachtet: die Verbesserung der Referenzimplementierung und die Übertragung des Konzeptes der *Cypher Social Contracts* auf weitere Anwendungsfelder.

9.1 Referenzimplementierung

Die Referenzimplementierung wurde im Kontext der Arbeit kontinuierlich verbessert und um Funktionen erweitert. Zudem wurde die Implementierung über einen langen Zeitraum getestet und in diversen Projekten und Feldversuchen erprobt. Die folgenden Punkte adressieren je nach Verwendung der Anwendung dennoch weitere Verbesserungsmöglichkeiten.

Parallele Netzwerke Sofern Fides über eine Vielzahl von Domänen genutzt werden soll, so ist anzunehmen, dass sich verschiedene Nutzende mit unterschiedlichen Netzwerken verbinden. Entsprechend ist der Wechsel zwischen Netzwerken notwendig, was aktuell entweder über eine separate Fides Instanz, oder durch die Änderung der lokalen Konfiguration geschieht. In letzterem Fall werden beim Start des Netzwerks die eigenen Objekte dann in das umgestellte Netzwerk übertragen, was in der Regel nicht notwendig ist. Abhilfe würde hier schaffen, wenn Objekte wie Verträge oder Vorlagen direkt lokal einem Netzwerk zugeordnet werden und sich somit die Netzwerkknoten innerhalb der lokalen Instanz nicht vermischen können. Je nach Objekt kann bei seiner Aktualisierung dann ein Knoten im richtigen Netzwerk gewählt werden, um so parallel als *client* in verschiedenen Netzwerken teilnehmen zu können. Für *full nodes* ergäbe sich hier keine Änderung an der Implementierung, sofern die Netzwerkinformationen nur lokal gehalten werden und nicht Teil der Objekte oder Transaktionen werden.

Stabilität des Netzwerks Wie die Messungen innerhalb der Evaluation zeigten, ist es möglich schwache Netzwerkknoten unter Umständen durch parallele Anfragen zu überlasten. In jenem Kapitel wurden einige technische Aspekte genannt, die es zukünftig erlauben diesen Angriffsvektor in Abhängigkeit der Nutzung abzuschwächen. Neben der Überlastung, welche vorrangig von *clients* ausgelöst werden, bieten sich noch weitere Ansätze an, inwiefern sich das Verhalten von *full nodes* überprüfen lässt. Hier wäre es möglich, dass *full nodes* einige Vorlagen und Verträge erstellen und hiermit prüfen, ob sich andere Knoten korrekt verhalten. Durch die geschickte Wahl des Parameters *nonce* können so Objekte erzeugt werden, welche auf den zu prüfenden Knoten gespeichert werden. Verhalten sich jene Knoten falsch (z. B. speichern die Transaktionen nicht), so können jene aus dem Netzwerk ausgeschlossen werden. Ehrliche Knoten könnten das Fehlverhalten über das Netzwerk kommunizieren, wodurch andere Knoten das Fehlverhalten ebenfalls erkennen können.

Ein weiterer Ansatz wäre das Beschränken von *clients* direkt durch *full nodes*, zum Beispiel durch Sperrlisten oder zuvor festgelegte Limitierungen der Anfragen.

Desktoanwendung/mobile Applikationen Die in der Arbeit umgesetzte Referenzimplementierung des Protokolls stellt neben der API ein Kommandozeilenwerkzeug zur Verfügung. Jene Art von Anwendung ist eher geeignet für Nutzende mit IT Hintergrund. Für die breite Verwendung von Fides sollte daher die Implementierungen zum Beispiel auf eine separate grafische Desktoanwendung oder gar auf mobile Anwendungen übertragen werden, um so zugänglicher zu sein. Die Resultate dieser Arbeit wurden und werden in Forschung und Lehre verwertet, wodurch bereits erste Ansätze von grafischen Anwendungen in unterschiedlichen Kontexten umgesetzt wurden.

Änderungen im Bezug auf die rechtliche Bewertung Änderungen an der Referenzimplementierung im Bezug auf das in [95] durchgeführte Rechts-

gutachten ergeben sich primär aus datenschutzrechtlicher Sicht. Gerade im Bezug auf die DSGVO kann die Nutzung von offenen, dezentralen Netzwerken problematisch werden. In voll privaten Netzwerken, wo sowohl *full nodes* als auch *clients* über ein Zertifikat verfügen müssen, lassen sich Verantwortlichkeiten einfacher bestimmen. Die privaten Netzwerke lassen sich unmittelbar durch die Konfiguration der Anwendung in der aktuellen Softwareversion bereits umsetzen. Eine mögliche technische Änderung wäre das weitere Einführen eines Kontaktfeldes als Parameter der Knoteninformationen von *full nodes* um hierdurch unmittelbar innerhalb von Fides eine Möglichkeit zur Kontaktaufnahme erhalten zu können. Da man hier dennoch ohne eine zentrale Stelle die Richtigkeit dieser Angaben nicht gewährleisten kann, besteht weiterhin das ursprüngliche Problem. Entsprechend ist es wahrscheinlich, dass zukünftige Anwendungen der Software Fides selbst als Blaupause nutzen, um dann zentrale Dienste/Netzwerke/Foren auf Basis des Protokolls anzubieten. Diese zentralisierten Dienste stützen sich dann nicht nur auf die positive zivilrechtliche Bewertung der *Cypher Social Contracts*, sondern können auch datenschutzkonform (aus rechtlicher Sicht) ausgestaltet werden.

9.2 Übertragung des Konzepts

Die generische Natur des Protokolls erlaubt seine breite Anwendung in diversen Feldern. Ein Kernziel der Arbeit, die Herstellung von Transparenz trotz sicherer Kommunikation, könnte somit in diversen Anwendungen stattfinden. Intuitiv ist es wahrscheinlicher, dass das Protokoll selbst mehr Traktion erhalten kann als die generische Implementierung Fides. So könnten verschiedene Dienste *Cypher Social Contracts* verwenden, um die Kommunikation bzw. Abläufe zu beschreiben. Für Anwenderinnen und Anwender bleibt das einfache mentale Modell gleich. Entsprechend einfach wäre das Verständnis bei der Verwendung eines Dienstes. Wichtig zu erwähnen ist hier die Bereitstellung von zentralen Diensten, die auf die verteilte Hashtabelle und Indexfunktionen verzichten können und somit ggf. effizienter implementiert werden können.

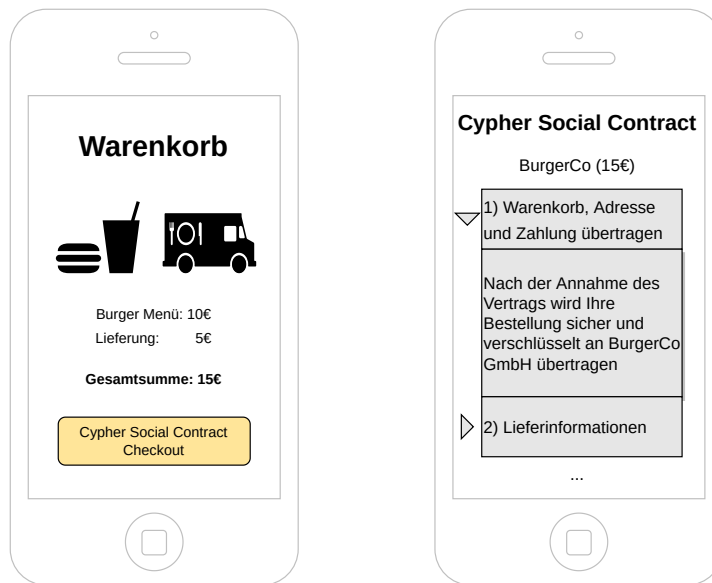


Abbildung 25: Mockup mobiler Bestellprozess

Insbesondere bei der Heterogenität der aktuell verfügbaren Onlineangebote könnte ein einfaches mentales Modell bei der Nutzung nicht nur die Nutzbarkeit steigern, sondern auch durch ein höheres Maß an Transparenz den Nutzenden vermitteln, welche Daten für die Übereinkunft benötigt und übermittelt werden. Die Homogenität in Kombination mit der dezentralen Bereitstellung von Informationen schützt so auch die Privatsphäre der Nutzenden. Dies soll exemplarisch an einigen Mock-Ups gezeigt werden, welche eine Onlinebestellung über ein modernes Smartphone abwickeln. Abbildung 25 zeigt hierbei die Sicht der Nutzenden, die eine einfache Übersicht über die Übereinkunft und die genauen Details enthalten. Abbildung 26 skizziert die Infrastruktur zwischen der fiktiven Firma und des Smartphone-Herstellers. Etwaige Zahlarten können so direkt in das jeweilige Ökosystem integriert werden (z. B. Apple Pay). Über eine domänenspezifische Sprache können so vorlagenübergreifend die benötigten Nutzerdaten abgefragt werden oder Vorgaben für die Auslegung der Vorlage gemacht werden. Verträge können dann über den Server des Providers bearbeitet werden, wobei die Daten der

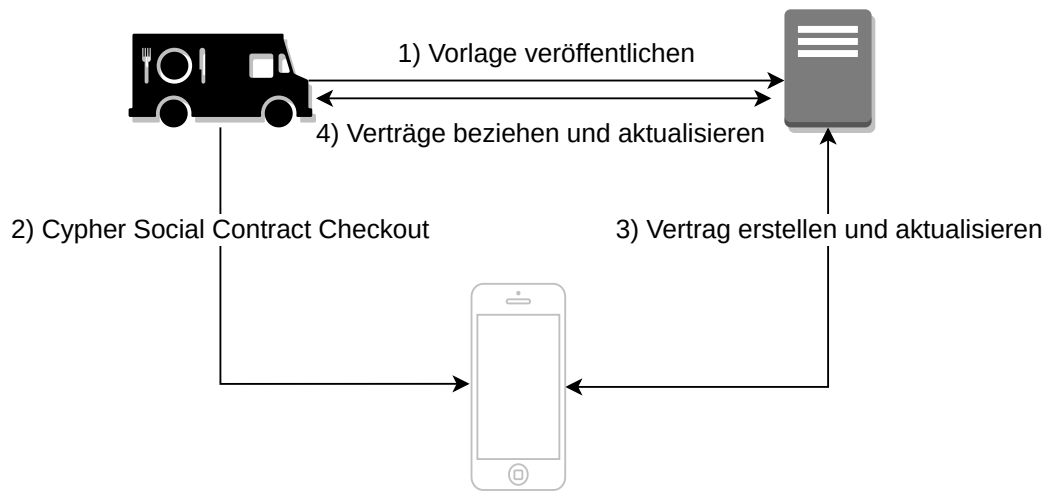


Abbildung 26: Mockup Kommunikation der Parteien

Eingabe nur für die beiden Parteien lesbar sind. Der Server übernimmt hier die Aufgaben eines Indexknotens mit der zusätzlichen vorherigen Prüfung, ob die Vorlage innerhalb des jeweiligen Ökosystems genutzt werden kann.

10 Fazit

Die Arbeit thematisierte die Spezifikation des *Cypher Social Contracts* Protokolls in Kombination mit dessen Implementierung und der Umsetzung der Teilhabe an digitalen Verträgen in Regionen ohne Internetverbindung durch datenbegrenzte Netzwerke.

Zunächst wurden verwandte Arbeiten, im speziellen Smart und Ricardian Contracts, beschrieben und offene Forschungsfragen und Ziele formuliert, die durch die Arbeit beantwortet werden sollten. Hierzu wurde zunächst das angesprochene Protokoll im Detail in Kapitel 5 eingeführt. Anschließend widmete sich Kapitel 6 dem Funktionsumfang der Referenzimplementierung, welche als Basis für die Verwendung bzw. Weiterentwicklung der Resultate dieser Arbeit diente. Kapitel 7 stellte die LoRaWAN Abstraktion der Implementierung vor, welche es erlaubt in Regionen ohne direkt Internetverbindung ebenfalls Verträge zu erstellen und an jenen teilnehmen zu können. Kapitel 8 evaluierte die Arbeit im Bezug auf die rechtliche Bewertung des Protokolls und seiner Implementierung, der allgemeinen Performanz der Software und führte zudem einen Vergleich mit Ethereum Smart Contracts durch, in welchem das Protokoll exemplarisch als Smart Contract umgesetzt wurde.

Die Kernergebnisse werden anhand der in Kapitel 2 definierten Ziele hier erneut aufgegriffen:

- **Transparenz:** Die Vorlagen innerhalb des Protokolls geben die darin definierten Aufgaben unmittelbar transparent an. Jede Partei, die diese Vorlage nutzen möchte, um ein Vertragsangebot zu erstellen, kann direkt und vor jeder Interaktion sehen, was Teil der Übereinkunft ist. Somit kann die Partei, welche die Vorlage benutzt, unmittelbar einschätzen, welche Daten in welchem Umfang übermittelt werden müssen und sich für oder gegen die Interaktion entscheiden. Da die Aufgaben generisch in natürlicher Sprache formuliert werden, können sie somit auch von Anwenderinnen und Anwendern verstanden werden,

die keinen direkten IT-Bezug haben. Dies ist im Kontext von Smart Contracts, deren Implementierung nur von Experten verstanden wird, so nicht möglich.

- **Privatsphäre:** Für jede Übereinkunft werden separate kryptographische Schlüssel verwendet, um die Eingabedaten bei der Bearbeitung des Vertrags sicher zu verschlüsseln und nur den beiden Parteien zugänglich zu machen. Verliert eine Partei einen solchen Schlüssel, wirkt sich dies entsprechend nur auf die jeweilige Übereinkunft aus und betrifft weder vorherige noch nachfolgende Verträge. Zusätzlich erlaubt das Vorgehen einzelne Daten der Übereinkunft offenzulegen. So entstehen unter anderem neue Möglichkeiten der Kooperation zwischen weiteren Parteien. Selbiges erlaubt die Streibarkeit eines Vertrages vor Gericht. Machen die beiden Parteien unterschiedliche Angaben, so kann jeder seinen Stand glaubhaft darlegen und durch die jeweiligen Signaturen Absichten klar belegen. Es reicht daher aus, das Minimum an Daten zu teilen, um einen etwaigen Streitfall aufzulösen. Ein Gericht würde im Kontext eines solchen Streitfalls keinen Zugriff auf die gesamten Systeme der Parteien benötigen.
- **Dezentralisierung und Selbstorganisation:** Die Referenzimplementierung nutzt ein dezentrales Netzwerk zur Dokumentation der Verträge und Vorlagen. Verschiedene Netzwerke können hierbei parallel existieren und von den Nutzerinnen und Nutzern verwendet werden, da ein universelles Fides Netzwerk nicht forciert wird. Weiterhin können private Netzwerke erzeugt werden, um so Zugänge zu den jeweiligen Netzwerken/Foren einzuschränken. Das Protokoll und die Implementierung lassen sich zudem auf zentralen Servern mit geringen technischen Änderungen nutzen. Hierbei kann entweder lediglich die Konfiguration angepasst werden, um ein privates, zentrales Netzwerk zu realisieren oder auch die Implementierung leicht verändert werden, um

die verteilte Hashtabelle im Ganzen zu entfernen. Die Übertragbarkeit des Protokolls in andere Domänen ist daher einfach realisierbar und die Nutzbarkeit auch in unterschiedlichen Kontexten gewährt. Es steht den Nutzenden frei Fides direkt zu verwenden oder Anpassungen für den jeweils eigenen Verwendungszweck zu machen, um sich so digital selbst zu organisieren.

- **Referenzimplementierung:** Die Referenzimplementierung wurde unter der MIT Lizenz veröffentlicht und kann auf Linux, MacOS und Windows mit dem Ausführen eines einzelnen Befehls installiert werden. Anwendungen auf Basis von Fides wurden bereits in einigen Feldversuchen des Projekts „LandLeuchten“ umgesetzt, wodurch ein Transfer in die Gesellschaft bereits exemplarisch stattgefunden hat. Weiterhin finden die Resultate der Arbeit aktiv Verwendung in der Lehre.
- **Digitale Teilhabe in datenbegrenzten Netzwerken:** Die LoRaWAN Abstraktion von Fides ist unmittelbar Teil der Implementierung und ebenfalls unter der MIT Lizenz nutz- und erweiterbar. Das speziell entwickelte LoRaWAN Protokoll wurde anhand der Abstraktion ebenfalls erfolgreich in Feldversuchen verwendet. Weiterhin wurde das Protokoll in Teilen auf Microcontroller übertragen, die zwar nicht Fides selbst, aber das Protokoll verwenden können, um so einfache und energieeffiziente Mobility-on-Demand Anwendungen erproben zu können.
- **Ressourceneffizienz:** Die Anforderungen an die Hardware sind gering, was unter anderem durch die Messungen in Kapitel 8 gezeigt wurde. Somit ist die Anwendung zum Beispiel problemlos verwendbar auf günstiger, leistungsschwacher Hardware wie einem Raspberry Pi. Entsprechend ist es auch Nutzergruppen, welche nicht die finanziellen Mittel für leistungsstarke Hardware verfügen, möglich, sich digital selbstorganisieren zu können.

Zusammenfassend entstand durch die Arbeit ein neuartiger Ansatz für digitale Verträge im Forschungsfeld der Ricardian Contracts/Smart Contracts. Die Verwendung von natürlicher Sprache erlaubt es, ebenjene Übereinkünfte auch verständlich für nicht-technikversierte Personen zu gestalten und eröffnet daher breite Verwendungsmöglichkeiten. Besonders herauszustellen ist nicht nur, dass die Verträge die Privatsphäre der Nutzenden schützen, sondern dass jene Verträge auch rechtlich bindend verwendbar sein können, was durch ein unabhängiges Rechtsgutachten bestätigt wurde.

A Grundlagen

A.1 Betriebsmodi von Blockchiffren

Um eine Blockchiffre wie AES für Datenmengen zu verwenden, die größer sind als ein Block, werden die Chiffren in sogenannten Betriebsmodi verwendet [138]. Nachfolgend werden einige Betriebsmodi, welche mit AES genutzt werden können, vorgestellt.

Electronic Codebook (ECB) Bei der Betriebsart ECB werden identische Blöcke an Klartext äquivalent verschlüsselt. Weiterhin führt das Vertauschen von Blöcken im verschlüsselten Text bei der Entschlüsselung zu der gleichen Verschiebung im Resultat. Zusätzlich betreffen Fehler bei der Entschlüsselung eines Blocks nur jenen und beziehen sich nicht auf die gesamte Operation. [138] Von der Verwendung von ECB für Datenmengen, die größer als einen Block sind, wird abgeraten [138], da Muster zwischen verschlüsseltem Text und Klartext erkennbar sind. Sollten die Eingabedaten nicht einem Vielfachen der Blockgröße entsprechen, müssen jene durch Padding aufgefüllt werden [138].

Cipher feedback (CFB) Die Betriebsart CFB kann benutzt werden, um Daten, deren Länge kein Vielfaches der Blockgröße ist, durch eine Stromchiffre zu verschlüsseln. Für die Verwendung wird ein Initialisierungsvektor (IV) benutzt, dessen Wiederverwendung in dem gleichen verschlüsselten Text resultiert. Entsprechend sollte jener Wert nicht vorhersehbar sein. Durch die Verkettung hängen die unterschiedlichen verschlüsselten Blöcke voneinander ab, was bedeutet, dass Fehler in verschlüsselten Blöcken sich auf die Entschlüsselung im Allgemeinen auswirken. Für die Nutzung des Modus CFB wird kein Padding benötigt. [138]

Output feedback (OFB) Im Vergleich zu CFB wird bei OFB nicht ein verschlüsselter Block, sondern die Ausgabe der Verschlüsselungsfunktion als Feedback für die Stromchiffre benutzt. OFB kann in Anwendungen benutzt werden, wo die Fehlerfortplanzung vermieden werden muss. Die Unabhängigkeit gegenüber der Nachricht bedeutet, dass ein Initialisierungsvektor (IV) nie mehrfach benutzt werden darf. Ähnlich zum Modus CFB wird auch bei dem Modus OFB kein Padding benötigt. [138]

A.2 Distributed Hash Table

Neben der in der Arbeit genutzten Distributed Hash Table (DHT) Chord [155] sollen nachfolgend noch weitere verteilte Hashtabellen vorgestellt werden.

Kademlia Die verteilte Hashtabelle Kademlia [136] verwendet die XOR Metrik zur Bestimmung der Distanz von Knoten zu anderen Knoten/Schlüsseln. Hierbei kann die ID des Knotens ähnlich wie in Chord festgelegt werden, auch wenn in [136] zufällige IDs angenommen werden. Ein Kernaspekt von Kademlia ist die asynchrone Kommunikation zur Abwicklung paralleler Routinganfragen. Dies verhindert Timeouts und machen das System nicht nur effizienter, sondern auch robuster gegenüber Ausfällen anderer Knoten [136]. Kademlia wird zum Beispiel innerhalb von Ethereum verwendet, um andere Knoten im Netzwerk aufzufinden [15].

Pastry Innerhalb der DHT Pastry [151] wird versucht die Distanz der zu übertragenden Nachrichten zu minimieren, indem die Netzwerklokalität einbezogen wird [151]. Hierbei wird als skalare Näherungsmetrik zum Beispiel die Anzahl an Hops pro Routing genutzt. Durch die Nutzung von zufällig generierten IDs für Knoten wird heuristisch angenommen, dass eine Route zum Zielknoten aus der eigenen Menge benachbarter Knoten gefunden werden kann, welche näher zu dem ursprünglichen Knoten sind, von dem die

Nachricht stammte. Beim Routing wird die Nachricht also entweder an einen Knoten weitergeleitet, dessen ID näher an K ist, oder, sofern dies aus der gegebenen Menge an Knoten nicht der Fall sein sollte, die räumliche Nähe benutzt, um die Anfrage mit K an einen anderen Knoten zu leiten, der sich näher an dem Ursprung der Nachricht befindet. Verwendet wird Pastry zum Beispiel in PAST [71], einer peer-to-peer Anwendung zur Speicherung von Daten oder SCRIBE [152], einem Publish/Subscribe Benachrichtigungssystem.

Tapestry Bei Tapestry [172] werden [151] die Routingtabellen ähnlich zu Pastry zu Beginn lokal optimiert und gepflegt, um das Routing von Anfragen möglichst effizient zu machen. Dies unterscheidet sich zum Beispiel von dem angeführten Verhalten von Chord [155]. Innerhalb des Netzwerks sind Zeiger bekannt, die jenes Routing unterstützen. Die Effizienz dieses Verfahrens bestimmt sich hierbei aus dem Verhältnis der Entfernung, die eine Nachricht zu einem Knoten zurücklegen muss und der minimalen Entfernung der Quelle zu diesem Knoten.

Innerhalb von Tapestry wird zwischen Knoten des Overlays und anwendungsspezifischen Endpoints unterschieden, deren jeweilige ID mit der gleichen Hashfunktion bestimmt werden. Dadurch, dass Tapestry mit der Anzahl an Knoten skaliert, bietet sich die Kombination unterschiedlicher Anwendungen innerhalb eines Tapestry Netzwerkes an. Entsprechend können Tapestry und dessen Grundfunktionen als übergreifende Schnittstelle zur Umsetzung diverser Anwendungen verstanden werden. [172]

B Stand der Forschung

B.1 Smart Contract Anwendungen

Bitcoin Bitcoin kann als erste Blockchain bezeichnet werden und setzt eine gänzlich offene Blockchain in Kombination mit einer dezentralen Kryp-

towährung um [141].

Bitcoin verfügt mit „Bitcoin Script“ über die Möglichkeit, Transaktionen über Skripte auszuführen, welche jedoch im Vergleich zu Ethereum nicht Turing-vollständig sind. Jene Skripte sind implizit Teil der regulären Verwendung von Bitcoin, so werden zum Beispiel einfache Werttransfere via eines Skripts verwaltet, dessen Operationen aussagen, dass die nächste Partei, die die Werte ausgeben möchte, über den privaten Schlüssel, welcher in der Transaktion (bzw. in dem Skript) angegeben wurde, verfügen muss. Äquivalent lassen sich so auch Transaktionen abbilden, die von mehreren privaten Schlüsseln signiert werden müssen. [26]

Corda Bei Corda handelt es sich um eine zugangsbeschränkte Blockchain, die insbesondere auf die Abwicklung von Geschäftsprozessen ausgelegt ist [148].

Corda unterstützt Smart Contract Anwendungen in Form von Java Anwendungen. Transaktionen der Smart Contracts enthalten daher Java Bytecode, welcher vom Netzwerk ausgeführt wird. Um diesen Bytecode deterministisch interpretieren zu können, nimmt Corda einige Einschränkungen, zum Beispiel bei Zufallsgeneratoren, vor. [104]

Hyperledger Fabric Ähnlich zu Corda entspricht Fabric auch einer zugangsbeschränkten Blockchain. Hyperledger Fabric ist modular aufgebaut und erlaubt das Ausführen von Smart Contracts in verschiedenen Programmiersprachen (Go, Java, Javascript). Auch hier besteht das Ziel darin, Geschäftsprozesse abzubilden und so Kooperationen zwischen verschiedenen Parteien über die eingeschränkte Blockchain zu realisieren. Hierbei entstehen Netzwerke aus Netzwerken zwischen den Parteien, zum Teil mit unterschiedlichen Berechtigungen. Im Gegensatz zu öffentlichen Netzwerken erlaubt Fabric somit auch private Transaktionen oder vertrauliche Übereinkünfte, die nicht für alle Parteien lesbar sind. [78]

Hawk Bei Hawk [121] handelt es sich um ein dezentaales Smart-Contract System, was Transaktionsdaten verschlüsselt auf einer Blockchain speichern soll. In der Arbeit beschreiben die Autoren einen Compiler, welcher C-Code in drei Teile übersetzt: ein öffentliches Blockchain Programm, ein Programm für Nutzende und ein Manager-Programm. Dem Manager muss während der Übereinkunft vertraut werden und er kann alle verschlüsselten Eingaben (Transaktionen) sehen, hat hierbei jedoch keinen Einfluss auf die Bearbeitung der Smart Contracts. Hierzu verwenden die Autoren Zero-Knowledge-Proofs und nutzen eine veränderte Version des Zerocash-Protokolls [44]. Die Arbeit wird hier der Vollständigkeit halber erwähnt, da jene vielzitiert ist und selbst in diversen Patenten referenziert wird.¹⁸ Dennoch wurde, obwohl dies innerhalb der Arbeit genannt wird, bisher keine Version dieses Compilers open source veröffentlicht. Die zugehörige Webseite ist zum aktuellen Zeitpunkt (Oktober 2022) nicht erreichbar und ältere Snapshots (2015-2021) der Seite verweisen lediglich auf eine spätere Veröffentlichung der Anwendung.¹⁹

Weitere Anwendungen Neben den zuvor genannten Projekten gibt es noch eine Vielzahl weiterer Projekte in dem Bereich, von denen einige der Vollständigkeit halber noch kurz angesprochen werden sollen.

Algorand [55] ist eine Proof-of-Stake Blockchain, die ebenfalls Smart Contracts unterstützt. Programmiert werden jene in Python mit der hierfür entwickelten Bibliothek PyTeal [24], welche die Generation der TEAL Programme erleichtert. TEAL beschreibt eine Assemblersprache, die später in Bytecode umgewandelt wird, der von der virtuellen Maschine von Algorand, ähnlich wie bei Ethereum, interpretiert wird [31].

Solana ist eine weitere Blockchain, die neben Proof-of-Stake Proof-of-History verwendet [170]. Smart Contracts innerhalb des Solana Netzwerks speichern

¹⁸[vgl.https://ieeexplore.ieee.org/document/7546538](https://ieeexplore.ieee.org/document/7546538) (Abgerufen im Februar 2023)

¹⁹<https://web.archive.org/web/20210923210230/http://oblivm.com/hawk/download.html> (Abgerufen im Februar 2023)

keine Zustände, sondern nur die Programmlogik [35]. (Dezentrale) Anwendungen, die auf den Smart Contracts aufbauen, rufen deren Funktionen über Schnittstellen auf. Smart Contracts werden mit dem LLVM Compiler in ein ausführbares und verknüpfbares Format (ELF - Executable and Linkable Format) kompiliert, das eine Variante des Berkeley Packet Filter (BPF) Bytecodes enthält [21]. Entgegen anderer Blockchain-Projekte sind die bereits veröffentlichten Smart Contracts innerhalb von Solana änderbar [35]. Aktuell unterstützt Solana zur Entwicklung die Programmiersprachen Rust und C/C++ [21].

Cardano ist eine weitere Proof-of-Stake Blockchain mit Smart Contract Unterstützung [2]. Hierbei wird die Entwicklung von Smart Contracts in zwei Bereiche aufgeteilt: Marlowe [13], eine domainspezifische Sprache, um Smart Contracts mit Fokus auf Finanzen umzusetzen, und Plutus [22], um ganze Applikationen in der Programmiersprache Haskell zu entwickeln, die mit der Blockchain interagieren.

Tezos [97] ist ebenfalls eine Proof-of-Stake Blockchain mit Smart Contract Funktionalitäten. Allgemein wird eine domainspezifische Sprache für die Entwicklung von Smart Contracts verwendet [14].

B.2 Rechtliche Bewertung Smart/Ricardian Contracts

Rechtliche Lage in Deutschland Im Rahmen des Forschungsprojekts „LandLeuchten“ wurde für die Resultate dieser Arbeit ein unabhängiges Rechtsgutachten durch den Lehrstuhl für Öffentliches Recht, Informationsrecht, Umweltrecht, Verwaltungswissenschaft der Universität Frankfurt erstellt, in welchem auch die allgemeinen zivilrechtlichen Rahmenbedingungen für Smart Contracts thematisiert wurden [95]. Die allgemeine rechtliche Lage für Smart Contracts in Deutschland wird im Folgenden zusammengefasst. Die detailliertere Analyse der Konzepte und Implementierung dieser Arbeit werden im Bezug auf zivil- und datenschutzrechtliche Aspekte in Kapitel 8 vorgestellt.

Innerhalb der Literatur herrschen Uneinigkeiten, ob ein Smart Contract ein rechtlich bindender Vertrag sein kann und ob die Transaktionen, die in der Regel mit einer Blockchain interagieren, in der Lage sind die Willenserklärungen der jeweiligen Parteien abzubilden. [95] Arbeiten, die den Standpunkt vertreten, dass Smart Contracts keine Verträge im rechtlichen Sinn darstellen, thematisieren die Verwendung von Programmiersprachen als „Blackbox“ für Nutzende. Kann der Programmcode nicht vollständig verstanden werden, so können auch keine Willenserklärungen abgegeben werden. Eine Ausnahme wäre hier die Verwendung einer Programmiersprache als Vertragssprache, zuvor festgelegt durch die involvierten Parteien. [95]

Gegensätzlich hierzu geben andere Arbeiten laut [95] an, dass Smart Contracts als Verträge im Rechtssinn angesehen werden können. Hierbei können sie entweder in der realen Welt geschlossene Verträge wiedergeben und ausführen oder auch ein Mittel zum Vertragsschluss sein. Entsprechend wird die Meinung vertreten, dass es möglich sei, Willenserklärungen abgeben zu können.

Ein allgemeines Problem besteht in der Definition des Begriffs Smart Contract, woraus sich weiterhin Unstimmigkeiten im Bezug auf die rechtliche Bewertung ergeben [95]. Selbiges wurde bereits zuvor bei der Einführung von Smart Contracts aufgezeigt und betrifft zusätzlich auch die Anwendung von Ricardian Contracts.

Smart Legal Contracts in England/Wales Der Begriff „Smart Legal Contract“, welcher die Autoren in [56] verwenden, beschreibt einen Smart Contract, welcher dazu verwendet wird, Verpflichtungen eines rechtsverbindlichen Vertrags zu definieren und zu erfüllen. Die Autoren geben die Definition als rechtsverbindlichen Vertrag, bei dem einige oder alle vertraglichen Verpflichtungen in einem Programm (Smart Contract) definiert bzw. von diesem automatisch ausgeführt werden, an [56].

In ihrer Analyse beschreiben die Autoren, dass eine Blockchain (Distri-

buted Ledger Technologie) nicht unbedingt notwendig sei für Smart Legal Contracts, sondern sich die automatische Ausführung eher darauf bezieht, dass keine menschliche Aktion benötigt wird, um das jeweilige Programm auszuführen. [56]

Mit dieser Annahme können auch komplett zentrale Anwendungen als Smart Legal Contracts verstanden werden, was kontraintuitiv zu den gängigen Smart Contract Systemen ist, jedoch auch bedeutet, dass Ricardian Contracts nach dieser Definition auch als Smart Legal Contract verstanden werden können.

In der Arbeit werden drei Formen von Smart Legal Contracts unterschieden, die jedoch nicht unbedingt für alle Anwendungsszenarien ausreichen bzw. nicht notwendigerweise alle Übereinkünfte damit abgebildet werden können: [56]

1. **Natürliche Sprache mit automatischer Codeausführung:** Hier wird der Smart Contract als Werkzeug verwendet, welches von einer der beiden Vertragsparteien genutzt wird und die vertraglichen Verpflichtungen ganz oder in Teilen automatisiert ausführt. Der eigentliche Vertrag liegt hierbei in natürlicher Sprache vor und das Programm definiert keine direkten Bedingungen des Vertrags. Der Smart Contract wird hier als „externer“ Vertrag, unabhängig vom rechtlich bindenden Vertrag in natürlicher Sprache, bezeichnet und stellt laut den Autoren die häufigste Form von Smart Legal Contracts dar.
2. **Hybrider Vertrag:** Einige Teile des Vertrags werden in Form von Programmcode abgebildet und sind somit Teil des Smart Contracts, die anderen werden in natürlicher Sprache formuliert.
3. **Ausschließlich in Code formulierter Vertrag:** In dieser Ausführung existiert kein Vertrag in natürlicher Sprache und lediglich ein Smart Contract Programm. Dadurch, dass auf einen regulären Vertrag verzichtet wird, ist es in diesem Fall schwieriger zu bewerten, wann und

wie sich ein Vertrag zwischen den Parteien formt. Entsprechend gehen die Autoren davon aus, dass sich bei dieser Ausführung um eine Seltenheit in der Praxis handeln wird.

Die Autoren sind insgesamt der Auffassung, dass Smart (Legal) Contracts ohne Änderungen der bestehenden Gesetze verwendet werden können und beschreiben im Detail die einzelnen Schritte bei Verträgen in dem betrachteten Rechtssystem im Vergleich mit den Aktionen bei der Verwendung von Smart Contracts [56].

Smart Contracts im schweizer Recht Die Dissertation [103] untersucht die Fragestellungen, ob und wie Smart Contracts im schweizer Recht eingesetzt werden können. Hierzu werden beim Vertragsabschluss die folgenden Unterscheidungen gemacht: [103]

- **Fachkundige Personen:** Sofern ein Smart Contract von fachkundigen Personen verwendet wird, welche die Implementierung kennen und nachvollziehen können, so regelt jener Smart Contract die Bedingungen des Vertrags. Hierbei müssen jedoch die klassischen Vertragsbedingungen, wie zum Beispiel Willenserklärungen, innerhalb des Smart Contracts abgebildet werden.
- **Fachunkundige Personen:** Es ist sehr unwahrscheinlich, dass eine fachunkundige Person „aus Versehen“ einen Smart Contract veröffentlicht. Daher ist jede Publikation eines Smart Contracts mit einem Handlungswillen vergleichbar, da die Transaktion aktiv ausgelöst und mit dem privaten Schlüssel signiert werden muss. Sofern fachunkundige Personen über einen Smart Contract kollaborieren und die Handlungen nicht verstehen, so kann nicht von einem Vertrag gesprochen werden.
- **Fachkundige und fachunkundige Personen:** Unterscheiden sich die Wissensstände der beteiligten Parteien, so muss jeder Fall einzeln betrachtet werden. Die Autorin gibt das Beispiel eines Smart Contracts

zwischen Unternehmen (fachkundig) und Konsumenten (fachunkundig) an. Existiert hier eine Übersetzung der Vertragsbedingungen des Smart Contract in natürliche Sprache, so kann jener einen Vertrag darstellen, sofern die allgemeinen Bedingungen für Verträge erfüllt sind. Fehlt jene Übersetzung komplett oder ist nicht durch die fachunkundige Partei bei der fachkundigen Partei in Erfahrung zu bringen, so stellt der Smart Contract in der Regel keinen Vertrag dar.

Die Resultate sind ähnlich zu der zuvor betrachteten Einschätzung aus England [56]. Allgemein ist es dennoch unklar, zu welchem Zeitpunkt ein Smart Contract in seiner konkreten Ausführung rechtlich bindend ist und was allgemein notwendig ist, um so zum Beispiel sichere Geschäfte durchführen zu können.

Smart Contracts in einigen US-Bundesstaaten In den Vereinigten Staaten von Amerika wurden von einigen Bundesstaaten verschiedene Gesetze erlassen, die die Verwendung von Smart Contracts und Blockchain-Technologien anerkennen.

Iowa beschreibt die Verwendung von Smart Contracts zur Verwahrung und Übertragung von Vermögenswerten und gibt zusätzlich an, dass die Rechtswirkung/Durchsetzbarkeit eines Vertrags nur wegen seiner Form als Smart Contract nicht versagt werden darf [27].

Arizona erkennt in [1] an, dass Smart Contracts im Handelsverkehr existieren können und Verträge auf Transaktionen Bezug nehmen können. Weiterhin darf einem solchen Vertrag die Rechtswirkung/Durchsetzbarkeit ebenso nicht abgesprochen werden.

Nevada regelt im „Uniform Electronic Transactions Act“ [33] wie Blockchains im dortigen Rechtssystem zu verstehen sind und inwiefern automatische Transaktionen, die ohne menschliche Kontrolle ausgeführt werden, Teil von Verträgen und deren Bearbeitung sein können. Weiterhin werden digitale Verträge, ähnlich wie in den anderen Bundesstaaten, beschrieben, was sich

somit auch auf Smart Contracts anwenden lässt.

Die angeführten Beispiele bilden nur einen Teil der US-Bundesstaaten ab und sollen einen Einblick liefern, inwiefern Smart Contracts innerhalb der Vereinigten Staaten verstanden und angewandt werden können.

C Code

C.1 Fides

Die in der Arbeit genutzte Version von Fides (0.5.1) kann aus dem offiziellen Repository [129] bezogen werden.

Um den Stand der Arbeit festzuhalten, wurde der Code zusätzlich in [59] veröffentlicht.

C.2 LoRaWAN Middleware

Die nachfolgenden Codeauszüge der implementierten LoRaWAN Middleware dienen der Übersetzung von LoRaWAN Nachrichten zu Fides Transaktionen. Zusatzfunktionen der Webanwendung (z. B. Nutzerauthentifizierung) werden nicht aufgelistet.

Hierbei wird von folgendem, selbsterklärendem Datenbankmodell für Nutzende ausgegangen: `user_id`, `username`, `email`, `password_hash`, `ttn_api_key`, `ttn_application_name`, `lora_device`, `private_key`, `api_key`. Der öffentliche Schlüssel der Partei kann entweder über die API von Fides oder über die Funktion `to_dict` abgerufen werden.

Weiterhin seien folgende Funktionen gegeben:

- **post_downlink**: Senden von Downlink Nachrichten an das Gerät einer Partei
- **decrypt_message**: Entschlüsseln von verschlüsselten Datenbankelementen (z. B. private Schlüssel)

Route für Uplink Nachrichten:

```

@api.route('/ttn/webhook/uplink', methods=['POST'])
@apikey_required
def webhook_uplink(current_user):
    data = request.json
    if data['end_device_ids']['device_id'] == current_user.lora_device:
        global DEBUG_STRING
        decoded = base64.b64decode(data['uplink_message']['frm_payload'])
        tx = bytes_to_tx(decoded)
        if tx is None:
            return jsonify({'message': 'tx error'})
        tx.type = tx.WhichOneof("type")

        if tx.type == "contractOffer":
            _handle_contractOffer(tx, current_user)
        if tx.type == "confirmTask":
            _handle_confirmTask(tx, current_user)
        if tx.type == "getContractState":
            _handle_getContractState(tx, current_user)
        if tx.type == "getTemplateState":
            _handle_getTemplateState(tx, current_user)
        if tx.type == "deleteContract":
            _handle_deleteContract(tx, current_user)
        if tx.type == "ping":
            _handle_ping(tx, current_user)

        message_log = LorawanMessageLog(decoded.hex(), tx._str_(), current_user.user_id)
        db.session.add(message_log)
        db.session.commit()

    return jsonify({'message': 'success'})
else:
    return jsonify({'message': 'wrong device'})

```

Aufistung 20: LoRaWAN Middleware: Uplink Nachrichten

Übersetzung von Vertragsanfragen:

```

def _handle_contractOffer(tx, current_user):
    t = Template()
    s = Storage()
    cur_t = s.template_db.cursor()
    cur_t.execute("SELECT hash from Templates WHERE hash LIKE ?", (tx.contractOffer.
        template + "%", ))
    data = cur_t.fetchall()
    if len(data) > 1:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_HASH_AMBIGUOUS, object
            =tx.contractOffer.template))

        post_downlink(current_user, resp)

        return jsonify({'message': 'template ambiguous (offer)'})

    if len(data) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_NOT_FOUND, object=tx.
            contractOffer.template))
        post_downlink(current_user, resp)
        return jsonify({'message': 'template not found (offer)'})

```

```

t.db_load(data[0]["hash"])
user_private_key = load_private_key_raw(encrypt_message(current_user.private_key))

c = Contract()
c.create(t.hash)

c.finalize(private_key=user_private_key, nonce=tx.contractOffer.nonce)
if not c.publish(private_key=user_private_key):
    c.delete()
return jsonify({'message': 'contract created'})

```

Auflistung 21: LoRaWAN Middleware: Neue Vertragsanfragen

Übersetzung bei der Bestätigung von Aufgaben:

```

def _handle_confirmTask(tx, current_user):
    s = Storage()
    cur_c = s.contracts_db.cursor()
    cur_c.execute("SELECT hash, serialized_data from Contracts WHERE hash LIKE ?", (tx.
        confirmTask.contract + "%", ))
    data = cur_c.fetchall()

    if len(data) == 0:
        return jsonify({'message': 'contract not found (confirm)'})

    ud = current_user.to_dict()
    user_pub_key = ud["public_key"]

    user_contracts = []
    for d in data:
        c = Contract()
        c.db_load(d["hash"])
        if c.sender == user_pub_key:
            user_contracts.append(c)

    if len(user_contracts) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_NOT_FOUND, object=tx.
            confirmTask.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'contract not found (confirm)'})

    if len(user_contracts) > 1:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_HASH_AMBIGUOUS, object=
            tx.confirmTask.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'contract is ambiguous (confirm)'})

    user_private_key = load_private_key_raw(encrypt_message(current_user.private_key))

    if tx.confirmTask.type == LoRaTransaction.ConfirmTask.CONFIRM_TYPE.EMPTY:
        user_contracts[0].confirm(private_key=user_private_key)
        return jsonify({'message': 'empty confirm ok (confirm)'})
    else:
        placeholder = Placeholder.query.filter_by(user=current_user.user_id,
            placeholder_digit=int(tx.confirmTask.input)).first()
        if placeholder:
            user_contracts[0].confirm(private_key=user_private_key, input_data=placeholder.
                placeholder_value)

```

```

else:
    user_contracts[0].confirm(private_key=user_private_key)
return jsonify({'message': 'placeholder confirm ok (confirm)'})

```

Auflistung 22: LoRaWAN Middleware: Bestätigung von Aufgaben

Übersetzung beim Abfragen des Vertragsstatus:

```

def _handle_getContractState(tx, current_user):
    s = Storage()
    cur_c = s.contracts_db.cursor()
    cur_c.execute("SELECT hash, serialized_data from Contracts WHERE hash LIKE ?", (tx.
        getContractState.contract + "%", ))
    data = cur_c.fetchall()

    if len(data) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_NOT_FOUND, object=tx.
            getContractState.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'contract not found (getContractState)'})

    ud = current_user.to_dict()
    user_pub_key = ud["public_key"]

    user_contracts = []
    for d in data:
        c = Contract()
        c.db.load(d["hash"])
        if c.sender == user_pub_key:
            user_contracts.append(c)

    if len(user_contracts) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_NOT_FOUND, object=tx.
            getContractState.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'contract not found (getContractState)'})

    if len(user_contracts) > 1:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_HASH_AMBIGUOUS, object=
            tx.getContractState.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'contract is ambiguous (getContractState)'})

    if user_contracts[0].state == "FINISHED":
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_FINISHED, object=tx.
            getContractState.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'get state ok (getContractState finished)'})

    if user_contracts[0].state == "OFFER":
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_OFFER, object=tx.
            getContractState.contract))
        post_downlink(current_user, resp)
        return jsonify({'message': 'get state ok (getContractState offer)'})

```

```

if user_contracts[0].state == "REJECTED":
    resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
        LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_REJECTED, object=tx.
        getContractState.contract))
    post_downlink(current_user, resp)
    return jsonify({'message': 'get state ok (getContractState rejected)'})

if user_contracts[0].state == "LIVE":
    resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
        LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_LIVE, object=tx.
        getContractState.contract, info=user_contracts[0].current_task))
    post_downlink(current_user, resp)
    return jsonify({'message': 'get state ok (getContractState live)'})

return jsonify({'message': 'fail (getContractState wrong type)'})

```

Auflistung 23: LoRaWAN Middleware: Vertragsstatus abrufen

Übersetzung beim Abfragen des Vorlagenstatus:

```

def _handle_getTemplateState(tx, current_user):
    s = Storage()
    cur_t = s.template_db.cursor()
    cur_t.execute("SELECT hash from Templates WHERE hash LIKE ?", (tx.getTemplateState.
        template + "%", ))
    data = cur_t.fetchall()
    if len(data) > 1:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_HASH_AMBIGUOUS, object=
            tx.getTemplateState.template))
        post_downlink(current_user, resp)

        return jsonify({'message': 'template ambiguous (get template state)'})

    if len(data) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_NOT_FOUND, object=tx.
            getTemplateState.template))
        post_downlink(current_user, resp)
        return jsonify({'message': 'template not found (get template state)'})

    t = Template()
    t.db.load(data[0]["hash"])
    if t.state == "ACTIVE":
        cc = ChordClient()
        res, network_tx = cc.fetch_template(t.hash)
        if res:
            if network_tx.WhichOneof("type") == "templateRevoke":
                if t.validate(network_tx):
                    t.state = "INACTIVE"
                    t.tx_ref = network_tx.hash
                    t.save()
                    save_tx(network_tx)
                    resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
                        LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_INACTIVE, object=
                        tx.getTemplateState.template))
                    post_downlink(current_user, resp)
            else:
                resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
                    LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_ACTIVE, object=tx.
                    getTemplateState.template))

```

```

        post_downlink(current_user , resp)
    else :
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_NOT_FOUND, object=tx.
            getTemplateState.template))
        post_downlink(current_user , resp)

    else :
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.TEMPLATE_INACTIVE, object=tx.
            getTemplateState.template))
        post_downlink(current_user , resp)
    return jsonify({'message': 'get state ok (getTemplateState)'})

```

Auflistung 24: LoRaWAN Middleware: Vorlagenstatus abrufen

Übersetzung beim Löschen eines Vertrags:

```

def _handle_deleteContract(tx , current_user):
    s = Storage()
    cur_c = s.contracts_db.cursor()
    cur_c.execute("SELECT hash, serialized_data from Contracts WHERE hash LIKE ?", (tx.
        deleteContract.contract + "%", ))
    data = cur_c.fetchall()

    if len(data) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_NOT_FOUND, object=tx.
            deleteContract.contract))
        post_downlink(current_user , resp)
        return jsonify({'message': 'contract not found (deleteContract)'})

    ud = current_user.to_dict()
    user_pub_key = ud["public_key"]

    user_contracts = []
    for d in data:
        c = Contract()
        c.db_load(d["hash"])
        if c.sender == user_pub_key:
            user_contracts.append(c)

    if len(user_contracts) == 0:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_NOT_FOUND, object=tx.
            deleteContract.contract))
        post_downlink(current_user , resp)
        return jsonify({'message': 'contract not found (deleteContract)'})

    if len(user_contracts) > 1:
        resp = LoRaTransaction(downlinkResponse=LoRaTransaction.DownlinkResponse(type=
            LoRaTransaction.DownlinkResponse.RESPONSE_TYPE.CONTRACT_HASH_AMBIGUOUS, object=
            tx.deleteContract.contract))
        post_downlink(current_user , resp)
        return jsonify({'message': 'contract is ambiguous (deleteContract)'})

    user_contracts[0].delete()
    return jsonify({'message': 'contract deleted (deleteContract)'})

```

Auflistung 25: LoRaWAN Middleware: Vertrag löschen

Übersetzung von Ping Nachrichten:

Ping Nachrichten geben nur eine Bestätigung an die TTN Integration zurück. Im Hinblick auf die Nutzenden oder den Status der lokalen Fides Instanz werden keine Änderungen gemacht.

D Evaluation

D.1 Darstellung der Messergebnisse

Prozessorauslastung *collectl* verwendet in der genutzten Version das folgende Format für das CPU Subsystem:

```
cpu{title=Cpu-Util mask=3 yname=[CPU]User%+[CPU]Nice
%, [CPU]Sys%+[CPU]Irq%+[CPU]Soft%+[CPU]Steal%, [CPU
]Irq%+[CPU]Soft%+[CPU]Steal%, [CPU]Wait% clabels=
User, Sys, More, Wait }
```

Auflistung 26: Standardkonfiguration zur Messung der Systemlast (CPU)

Entsprechend wird die prozentuale Auslastung von User, Sys(tem), More und Wait dargestellt, was keine gute Lesbarkeit und zum Teil nur geringe Aussagekraft aufweist. Durch erste Messungen konnte Folgendes festgestellt werden: Die systemische Auslastung (Kernel) ist vor dem Beginn und nach dem Ende der Simulation höher als die Benutzerauslastung (Userspace). Während der Tests ist entsprechend die User-Auslastung der CPU höher und gibt eine bessere Aussage über die Last des Systems. Das angepasste Format für die Prozessorauslastung wurde daher wie folgt definiert:

```
customcpu {title=Cpu mask=1 yname=[CPU]User%, [CPU]
Wait% ymax=50}
customcpuraspi {title=Cpu mask=1 yname=[CPU]User%, [
CPU]Wait% ymax=60}
customcpushort { title=Cpu mask=1 yname=[CPU]User%, [
CPU]Wait% ymax=25}
```

Auflistung 27: Angepasste Konfiguration zur Messung der Systemlast (CPU)

Wir stellen somit die Auslastung von User in Prozent (ohne Nice, da bei jeder Messung 0) dar und verwenden hierzu zwei verschiedene Skalenbereiche: 0-50% zur Abdeckung aller Messpunkte und 0-25%, wo sich die Auslastung in

der Regel bewegt. Eine Ausnahme besteht in dem später genutzten Raspberry Pi, der zum Teil Auslastungen $> 50\%$ aufweist. Entsprechend wurde jene Skala angepasst.

Speicherauslastung Die Speicherauslastung der Anwendung wurde bereits in früheren Messungen (siehe [61] oder [58]) als nahezu konstant bei 1% festgestellt. Das Standardformat zur Darstellung wurde daher zur besseren Lesbarkeit wie folgt angepasst:

```
mem {title=MemGB mask=6 yname=[MEM]Buf,[MEM]Cached,[MEM]Slab,[MEM]Map,[MEM]Anon,[MEM]Inactive ydivisor=1048576 }
custommem {title=MemGB mask=6 yname=[MEM]Buf,[MEM]Cached,[MEM]Map,[MEM]Inactive ydivisor=1048576 }
```

Auflistung 28: Konfiguration zur Messung der Systemlast (RAM)

Entsprechend werden nur Systembuffer, Cache zwischen Kernel/Festplatte, Mapped Memory von Prozessen und inaktiver Speicher dokumentiert [34].

Festplattenzugriff Für die Messungen, bezogen auf die Festplatte der Systeme, wurde das Standardformat verwendet, was Lese- und Schreiboperationen in MB dokumentiert.

Netzwerkauslastung Bei der Netzwerkauslastung wird auch das Standardformat verwendet, welches ein- und ausgehende Daten in MB darstellt.

D.2 Messungen der Performanz

Messdaten

Die Daten zur Messung wurden in [60] veröffentlicht und sind öffentlich zugänglich. Das Repository enthält neben den Rohdaten auch die ausgewerteten Messungen (dargestellt unten) und der Vollständigkeit halber das

Simulationsskript.

Simulationsskript

```
# client configuration

num_clients = 20
client_path = "/tmp/fides_simulation/"
first_fidesd_port = 50000

# full node information - change here if you run the measurement on your own servers

full_node_ip = "143.93.46.63"
full_node_port = 3301
full_node_network = "fides/testnet-live"

class User():
    def __init__(self, user_id, instance, private_key, template):
        self.user_id = user_id
        self.instance = instance
        self.private_key = private_key
        self.template = template

users = [] # list of users

# create hook path and callbacks
from os import mkdir

# create base path for simulation
mkdir(client_path)
mkdir(client_path + "/hooks")

# the hooks will load the unencrypted pk through **args
import json
def create_contract_hook(user, hash):
    jh = {}
    jh["type"] = "ContractStateChangedHook"
    jh["hash"] = hash
    jh["interval"] = 2
    jh["callback"] = client_path + "/hooks/callback_contract.py"
    jh["method"] = "callback_fn_contract"
    jh["live_forever"] = True
    jh["args"] = {"instance": user.instance, "contract_hash": hash}
    with open(user.instance + "/hooks/" + hash, "w") as f:
        f.write(json.dumps(jh, indent=2))

# the hooks will load the unencrypted pk through **args
import json
def create_template_hook(user, hash):
    jh = {}
    jh["type"] = "TemplateUsedHook"
    jh["hash"] = hash
    jh["interval"] = 2
    jh["callback"] = client_path + "/hooks/callback_template.py"
    jh["method"] = "callback_fn_template"
    jh["live_forever"] = True
    jh["args"] = {"instance": user.instance}
    with open(user.instance + "/hooks/template", "w") as f:
        f.write(json.dumps(jh, indent=2))
```

```

callback_file = """
#/usr/bin/env python3
from fides.core.account import get_public_key, load_key
from time import sleep
from fides.core.contract import Contract
from threading import Thread
import random
def finish_contract(contract_hash, user_private_key, user_public_key):
    while True:
        c = Contract()
        c.db_load(contract_hash)
        if c.state == "FINISHED":
            print("Contract finished: ", contract_hash)
            break
        if c._responsible(user_public_key):
            c.confirm(user_private_key, random.randint(100, 50000) * "a")
            sleep(1)

def callback_fn_template(**args):
    c = Contract()
    c.db_load(args["contract"])

    user_private_key = load_key(args["instance"] + "/account/main")
    user_public_key = get_public_key(user_private_key)

    while True:
        if c.accept(user_private_key):
            break
        sleep(1)

    Thread(target=finish_contract, args=(c.hash, user_private_key,
                                         user_public_key), daemon=True).start()

"""

with open(client_path + "/hooks/callback_template.py", "w") as f:
    f.write(callback_file)

# write callback_contract.py to file
# will be used by the user that created the contract

callback_file = """
#/usr/bin/env python3
from fides.core.account import get_public_key, load_key
from time import sleep
from fides.core.contract import Contract
import random
def callback_fn_contract(**args):
    c = Contract()
    c.db_load(args["contract_hash"])

    user_private_key = load_key(args["instance"] + "/account/main")
    user_public_key = get_public_key(user_private_key)

    if not c._responsible(user_public_key):
        return

    while True:
        if c.confirm(user_private_key, random.randint(50, 70000) * "b"):
            break
        sleep(1)

```

```

"""
with open(client_path + "/hooks/callback_contract.py", "w") as f:
    f.write(callback_file)

from os import mkdir, environ, system
import importlib
import fides.core.storage
import fides.core.template
import fides.core.account
import fides.core.types
import fides.core.contract
import fides.core.config
from time import sleep
from grpc import insecure_channel
from fides.core.types.fidesd_pb2_grpc import fidesdStub
from fides.core.types import Empty
import random

for i in range(0, num_clients):

    # create the user directory first
    user_path = client_path + str(i)
    mkdir(user_path)

    system("fds init " + user_path + " --default")

    # set severel config parameters that differ from the default config

    environ["FIDES_PATH"] = user_path + "/.fides/"

    random_refresh_interval = random.randint(1, 60)

    system("fds config set Networkd Network " + full_node_network)
    system("fds config set Networkd Refresh " + str(random_refresh_interval))
    system("fds config set Fidesd Port " + str(first_fidesd_port + i))
    system("fds config set Logging Level WARNING")

    importlib.reload(fides.core.config)
    importlib.reload(fides.core.template)
    importlib.reload(fides.core.storage)
    importlib.reload(fides.core.account)
    importlib.reload(fides.core.types)
    importlib.reload(fides.core.contract)
    from fides.core.config import BASE_PATH, FIDESD_PORT

    # add full node endpoint
    system("fds endpoint add " + full_node_ip + " " + str(full_node_port) +
           " " + full_node_network)

    # create an account for the user
    system("fds account create main --password-file /dev/null")

    # start the network in the background

    system("fds network start &")

    # wait until fidesd is started
    while True:
        try:
            chan = insecure_channel("localhost:" + FIDESD_PORT)

```

```

    stub = fidesdStub(chan)
    res = stub.GetStatus(Empty())
    if res:
        break
    except:
        sleep(0.2)

# load the users private key, finalize and publish the template
from fides.core.account import load_key
private_key = load_key(user_path + "/.fides/account/main", None)
from fides.core.template import Template

from fides.core.types import Validator

t = Template()
t.description = "Fides simulation template"
t.tasks = ["This is a task to the template. Could be anything in the real world",
           "This is a task to the template. Could be anything in the real world",
           "This is a task to the template. Could be anything in the real world",
           "This is a task to the template. Could be anything in the real world",
           "This is a task to the template. Could be anything in the real world",
           "This is a task to the template. Could be anything in the real world"]
t.responsibility = [False, True, False, False, True, True]
t.validators = 6*[Validator()]
assert(t.finalize(private_key))

# create the User for our tests

u = User(i, user_path + "/.fides/", private_key, t.hash)
users.append(u)

# create the hook for the template and start it

create_template_hook(u, t.hash)

system("fds hooks start template -v &")

#system("fds template publish " + t.hash + " --password-file /dev/null")
assert(t.publish(private_key))

# number of contract creation loops

test_num_iterations = 5

test_num_of_contracts = 20

test_iteration_sleep = 120

from time import sleep
def run_test(u1, u2):
    print("Contract: ", u1.user_id, " —> ", u2.user_id)

    system("FIDES_PATH='" + u1.instance + "' fds template fetch " + u2.template)

# let u1 create a contract

environ["FIDES_PATH"] = u1.instance

importlib.reload(fides.core.config)
importlib.reload(fides.core.template)
importlib.reload(fides.core.storage)
importlib.reload(fides.core.account)

```

```
importlib.reload(fides.core.types)
importlib.reload(fides.core.contract)

from fides.core.config import BASE_PATH, ACCOUNT_PATH
from fides.core.contract import Contract

c = Contract()
c.create(u2.template)
c.finalize(u1.private_key)

create_contract_hook(u1, c.hash)
print("CONTRACT: ", c.hash)

system("FIDES_PATH='" + u1.instance + "' fds hooks start " + c.hash + " &")

c.publish(u1.private_key)

from time import sleep
import random

for i in range(0, test_num_iterations):
    for j in range(0, test_num_of_contracts):

        # get a random user sample
        sample = random.sample(users, 2)
        u1 = sample[0]
        u2 = sample[1]

        run_test(u1, u2)
        sleep(test_iteration_sleep)
```

Aufistung 29: Skript zur Simulation von Nutzenden

Messergebnisse

Auslastung bei Netzwerkknoten - Szenario 1

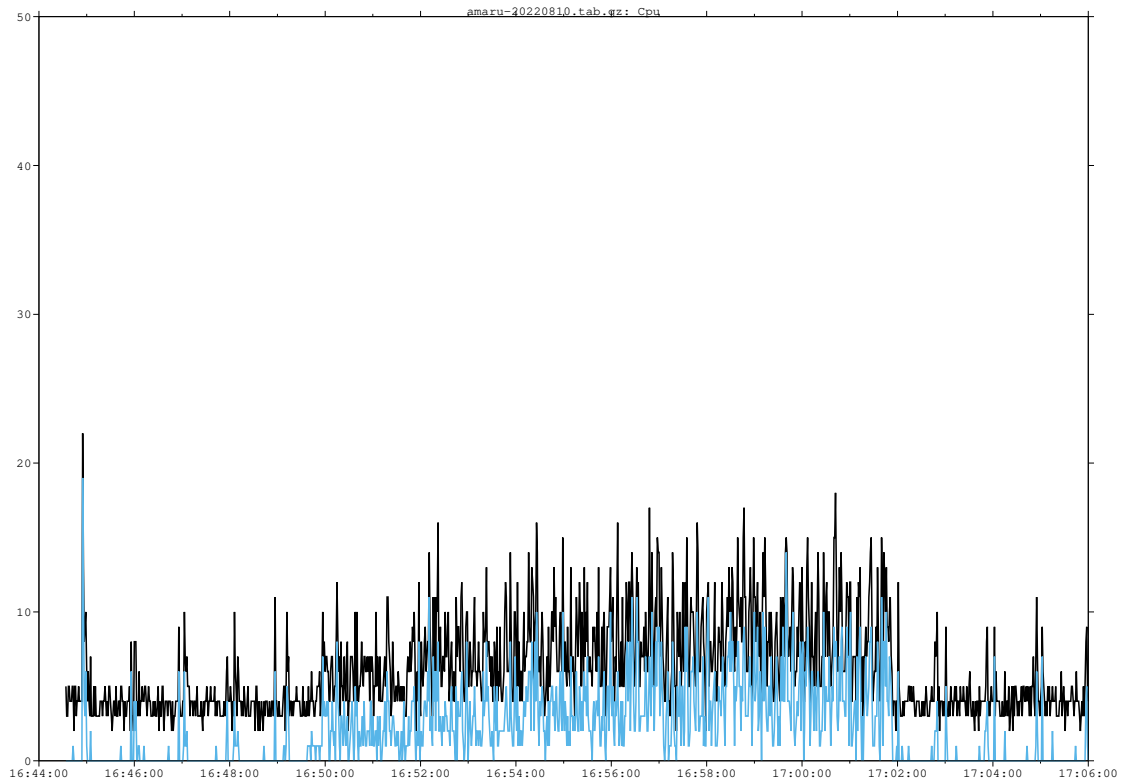


Abbildung 27: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 1

■ CPU User ■ CPU Wait

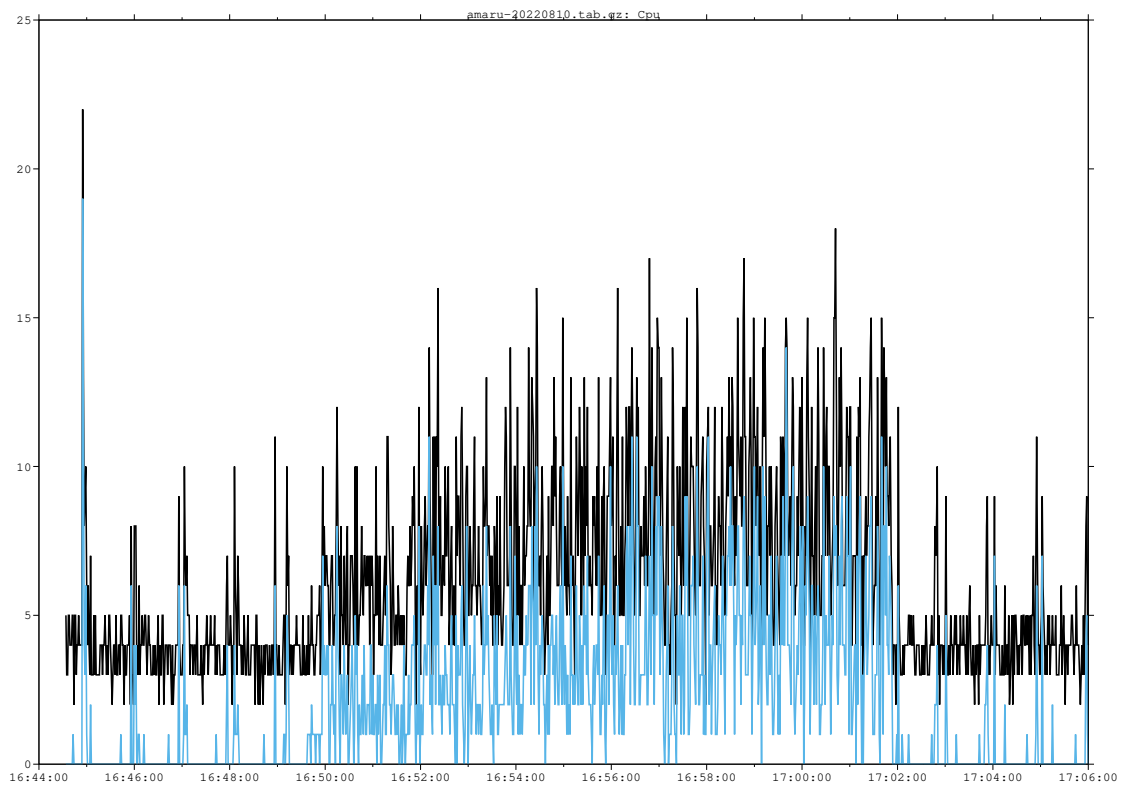


Abbildung 28: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 1

■ CPU User ■ CPU Wait

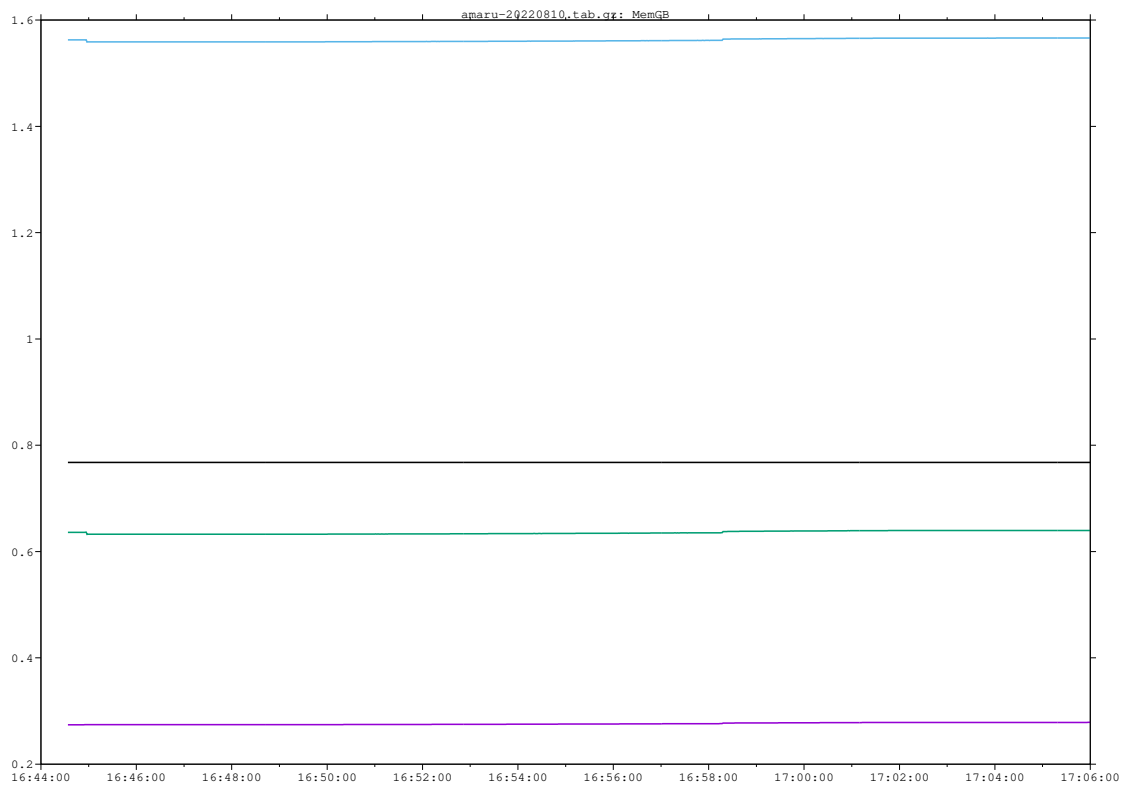


Abbildung 29: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 1

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

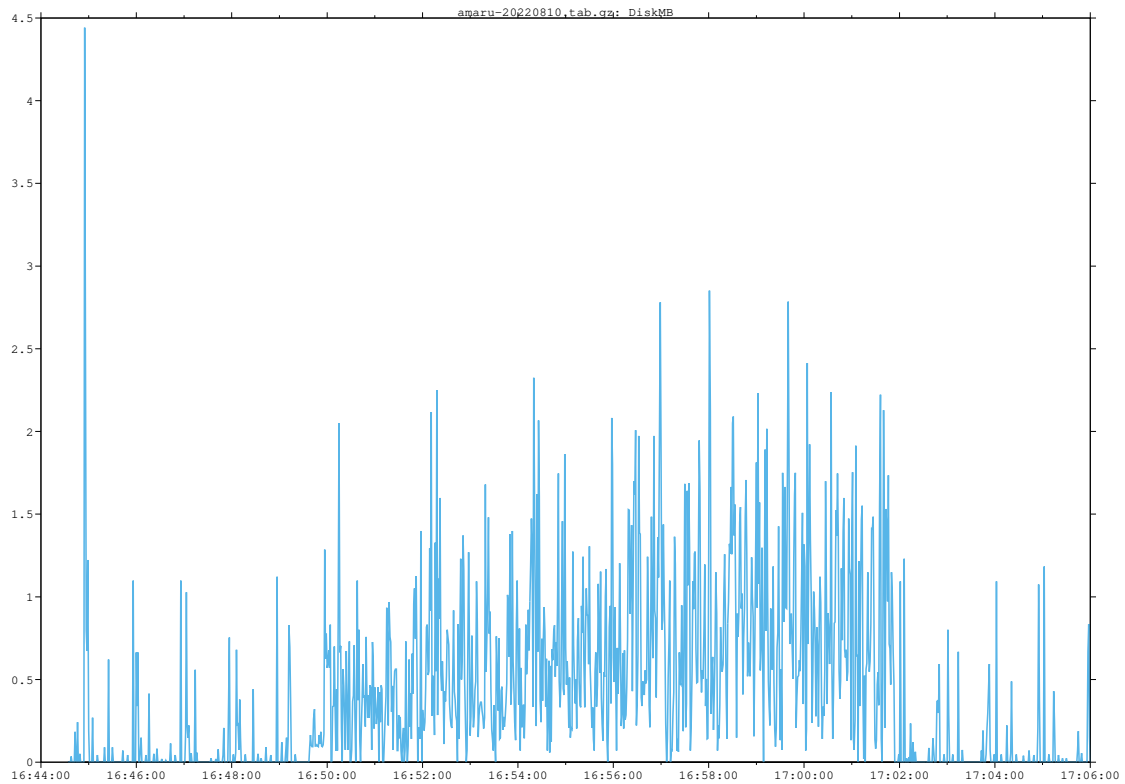


Abbildung 30: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Versuch 1
■ Write ■ Read

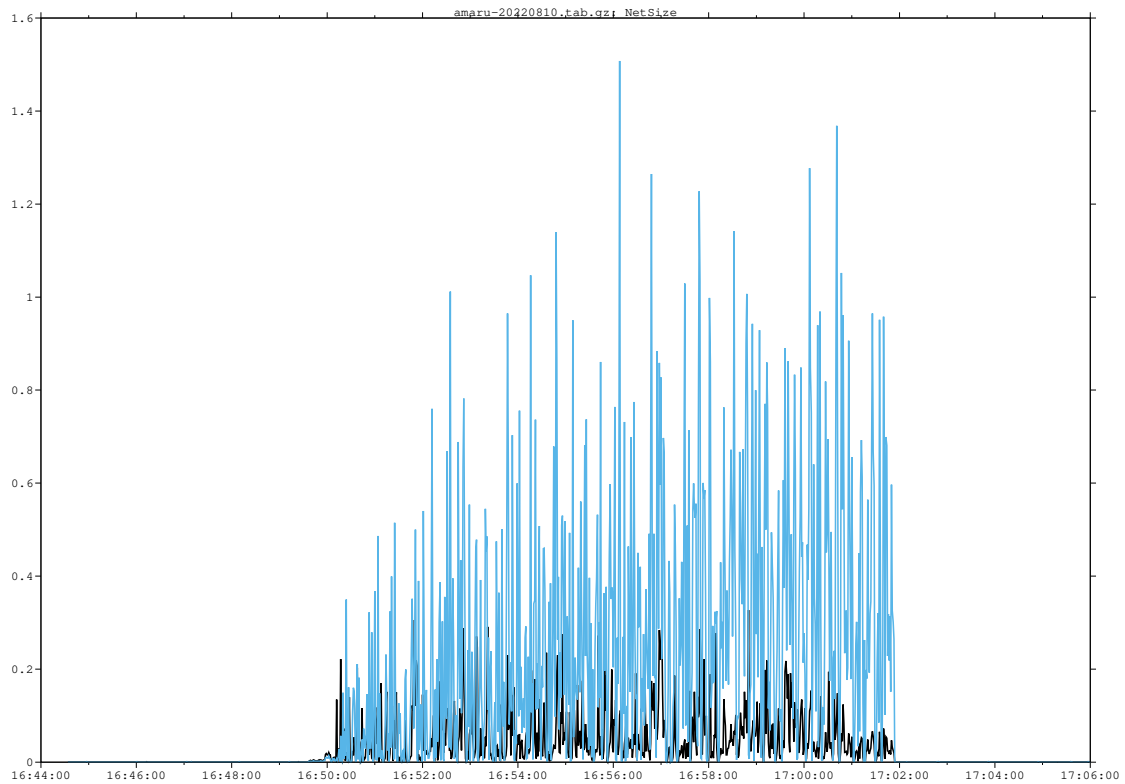


Abbildung 31: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Versuch 1

■ Ausgehend ■ Eingehend

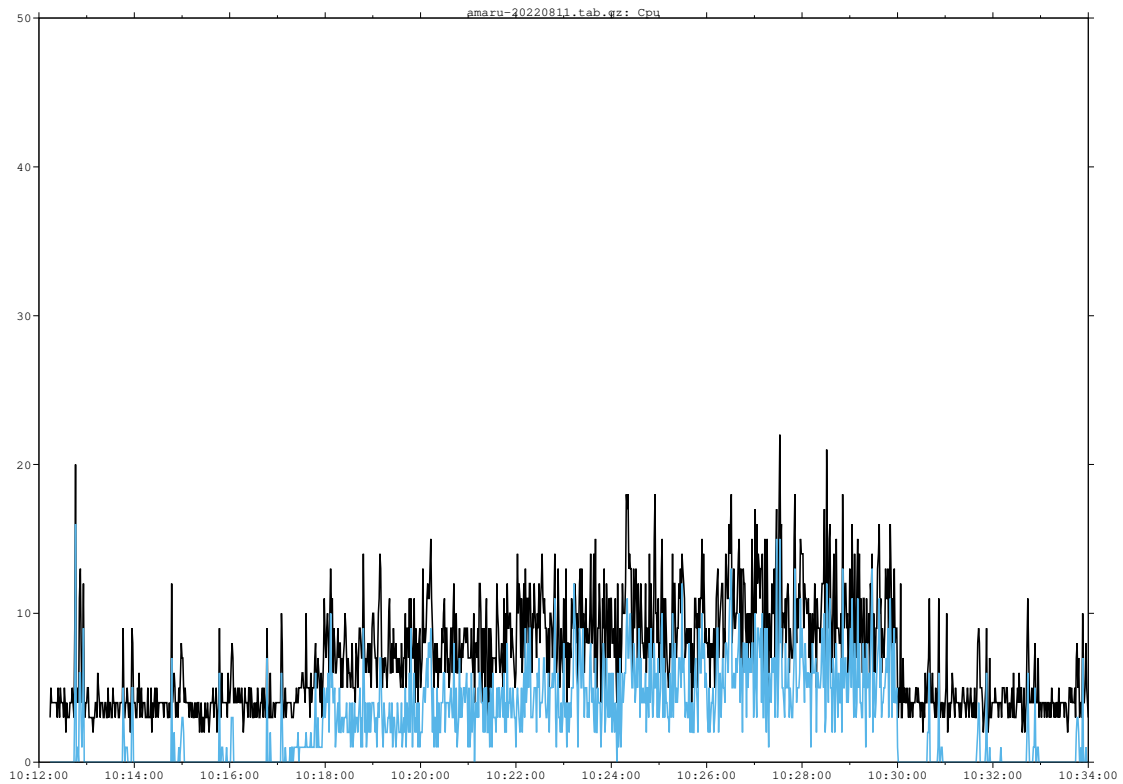


Abbildung 32: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 2

■ CPU User ■ CPU Wait

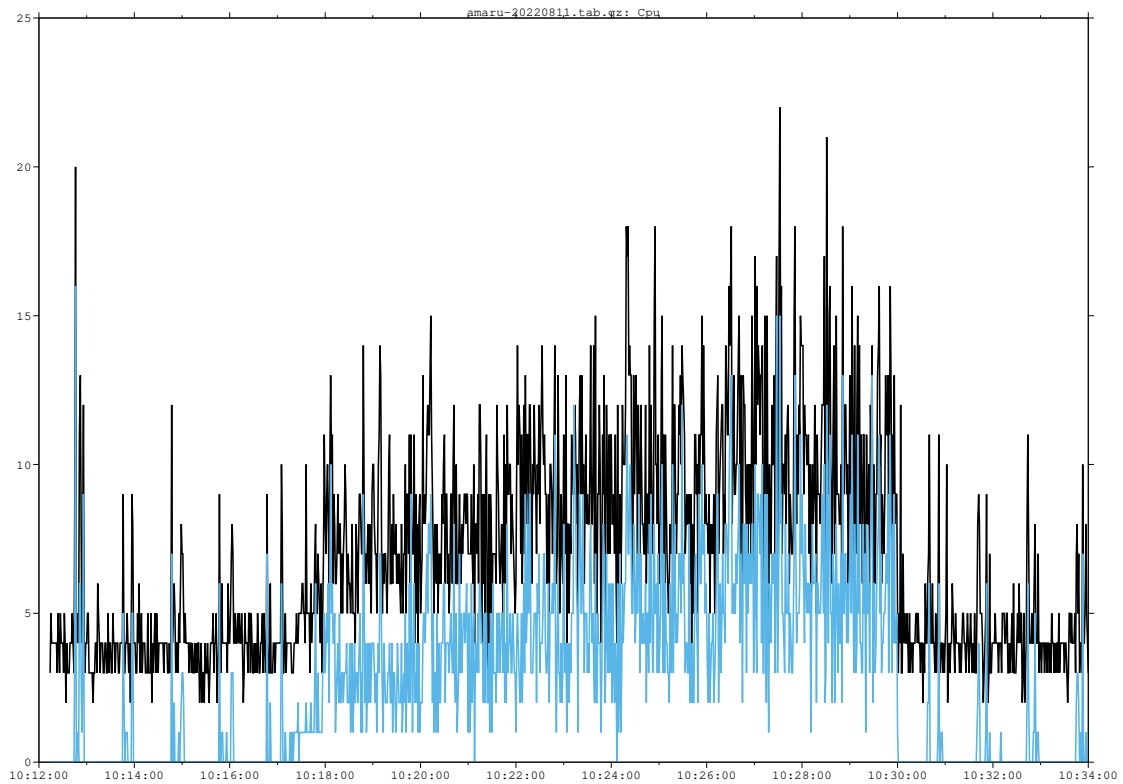


Abbildung 33: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 2

■ CPU User ■ CPU Wait

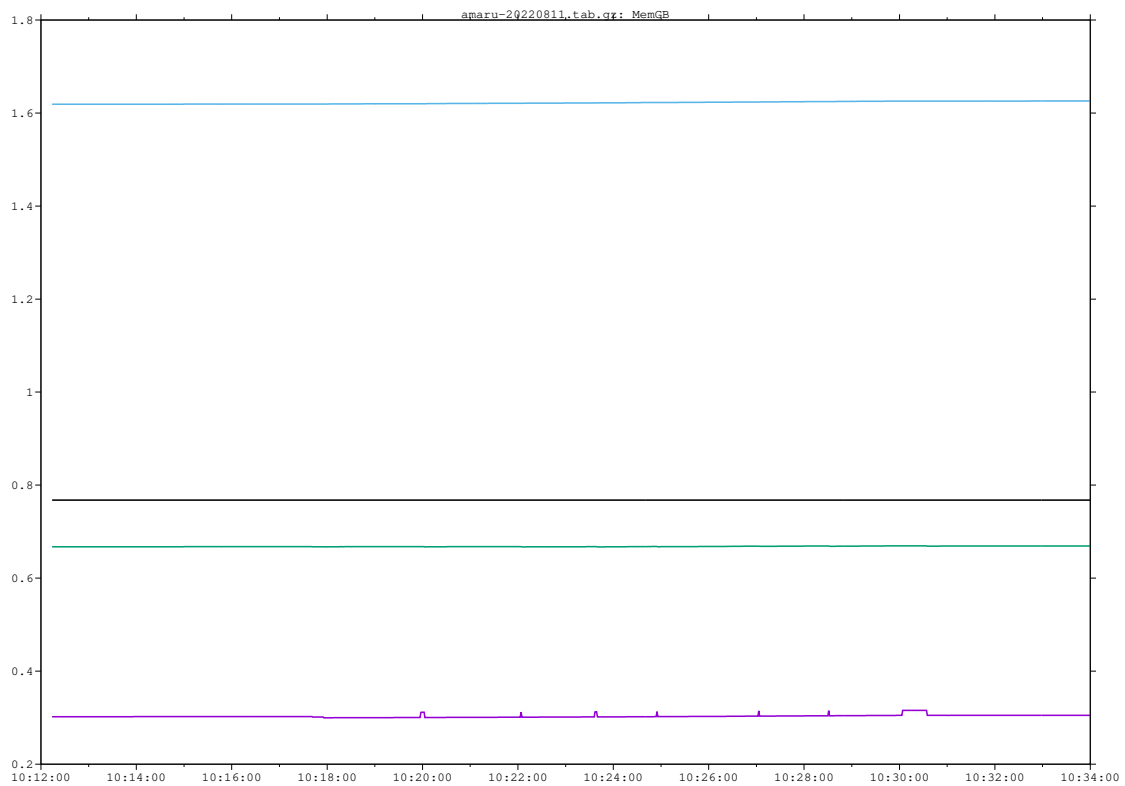


Abbildung 34: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 2

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

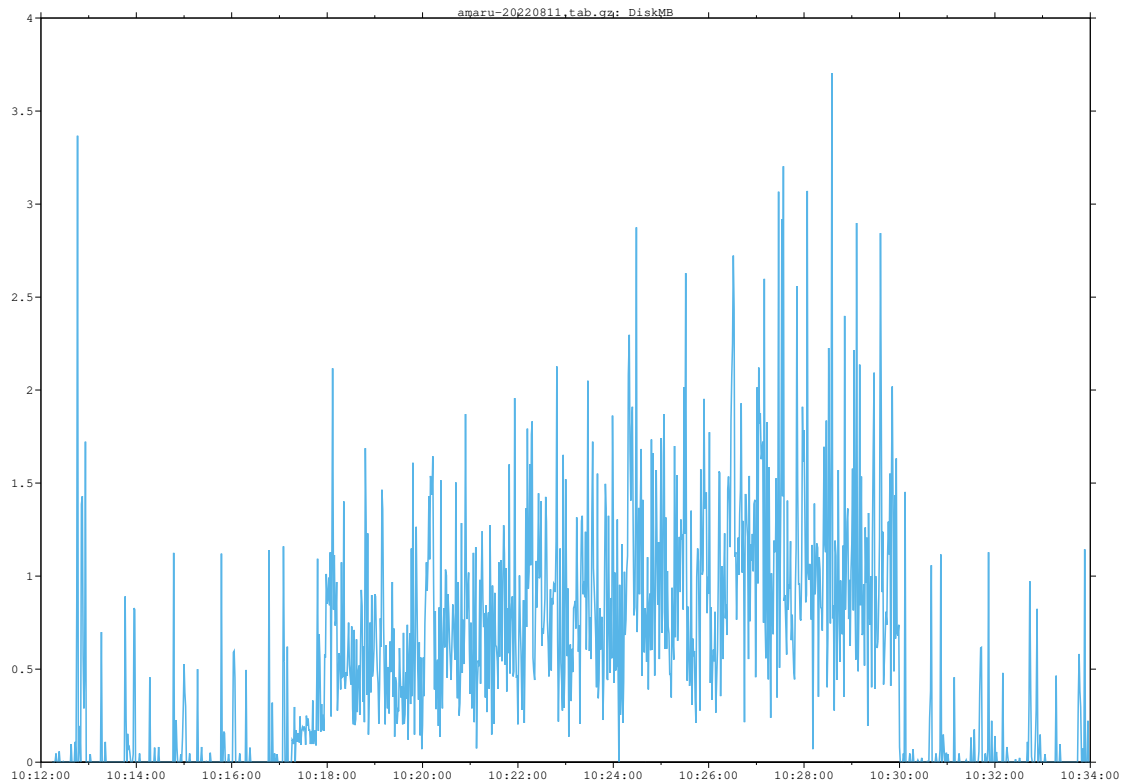


Abbildung 35: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Versuch 2
■ Write ■ Read

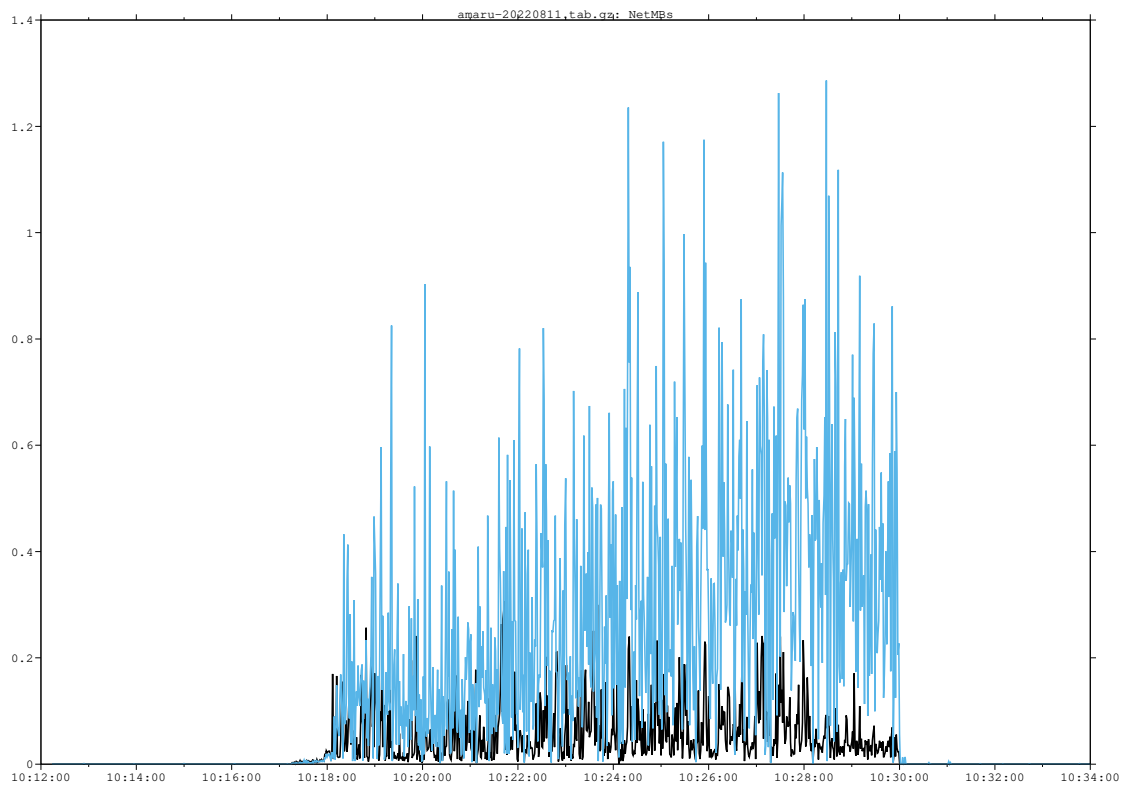


Abbildung 36: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Versuch 2

■ Ausgehend ■ Eingehend

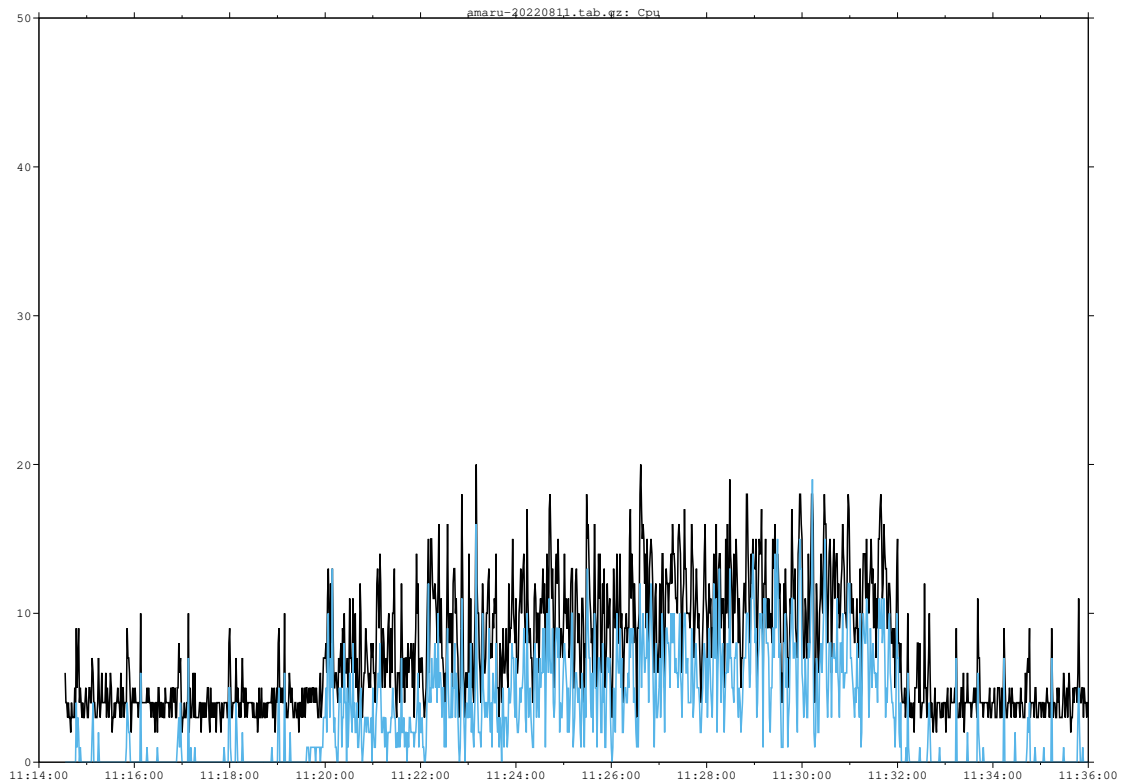


Abbildung 37: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 1/Versuch 3

■ CPU User ■ CPU Wait

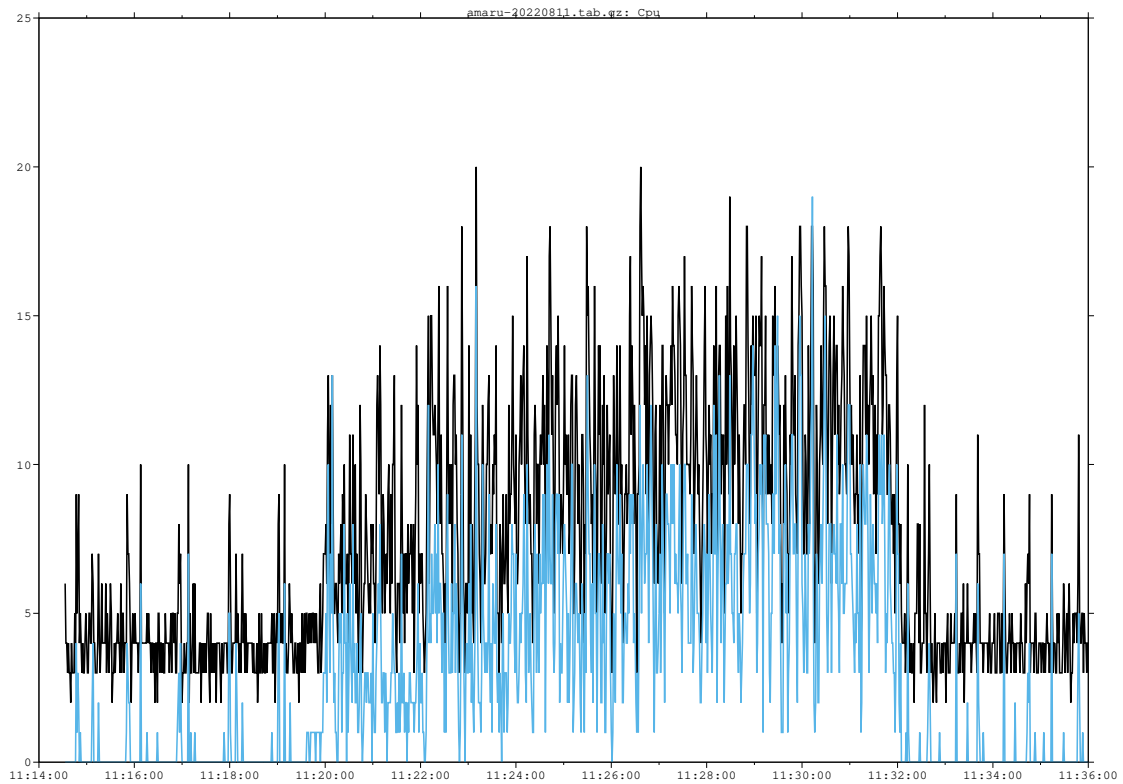


Abbildung 38: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 1/Versuch 3

■ CPU User ■ CPU Wait

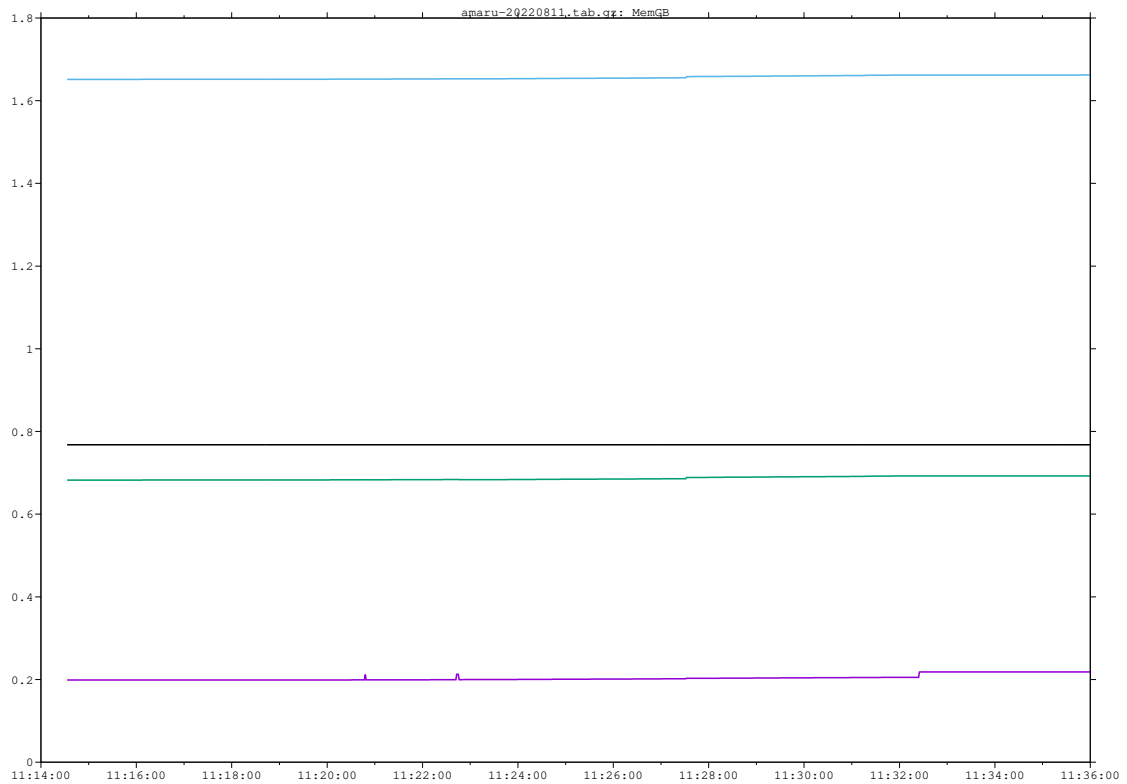


Abbildung 39: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 1/Versuch 3

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

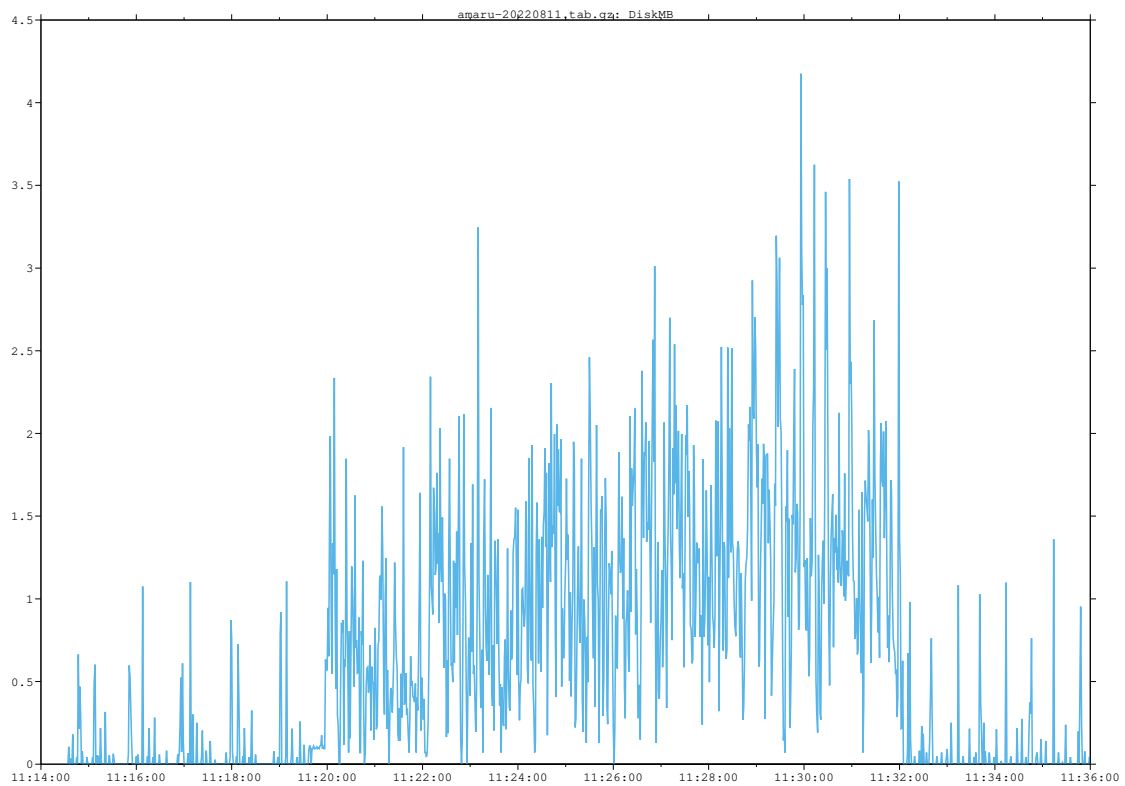


Abbildung 40: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 1/Versuch 3

■ Write ■ Read

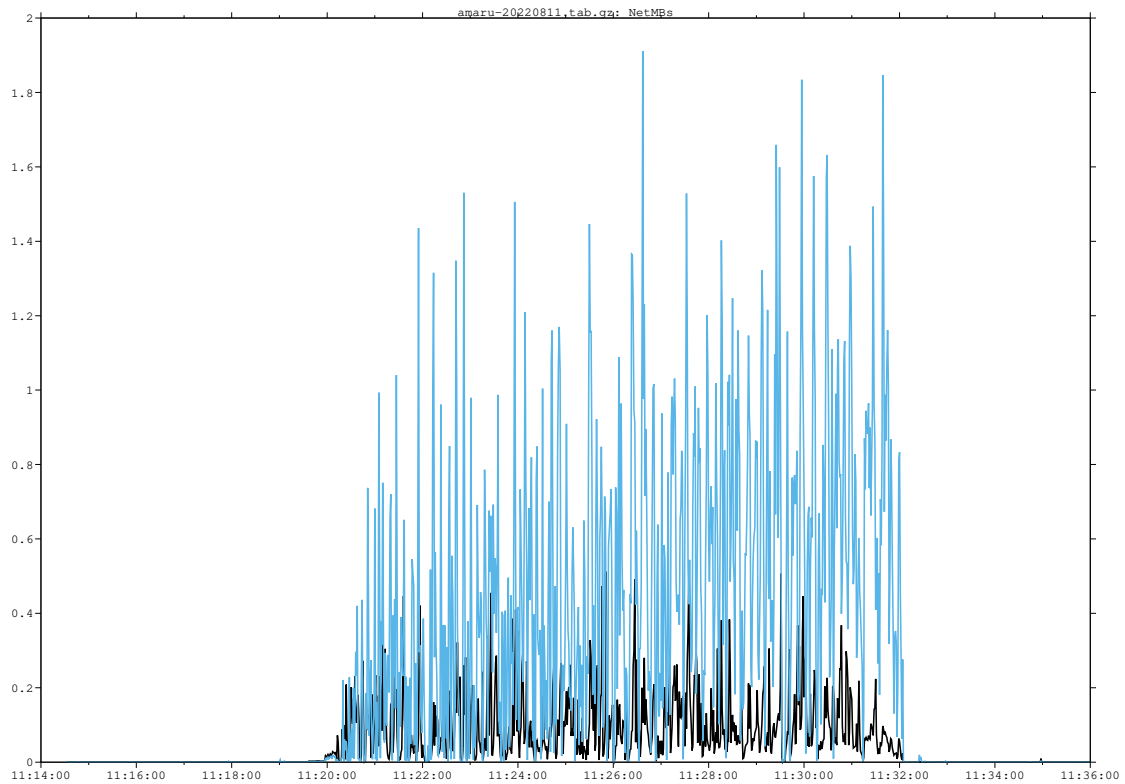


Abbildung 41: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 1/Versuch 3

■ Ausgehend ■ Eingehend

Auslastung bei Netzwerkknoten - Szenario 2

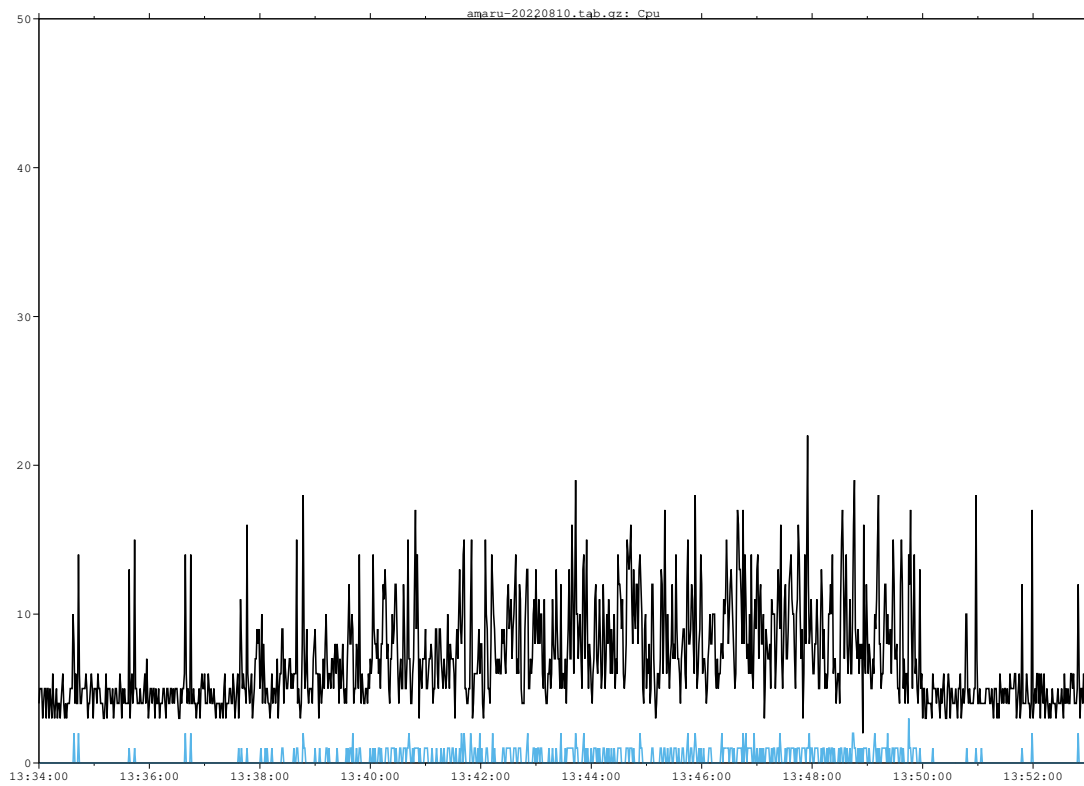


Abbildung 42: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 1

■ CPU User ■ CPU Wait

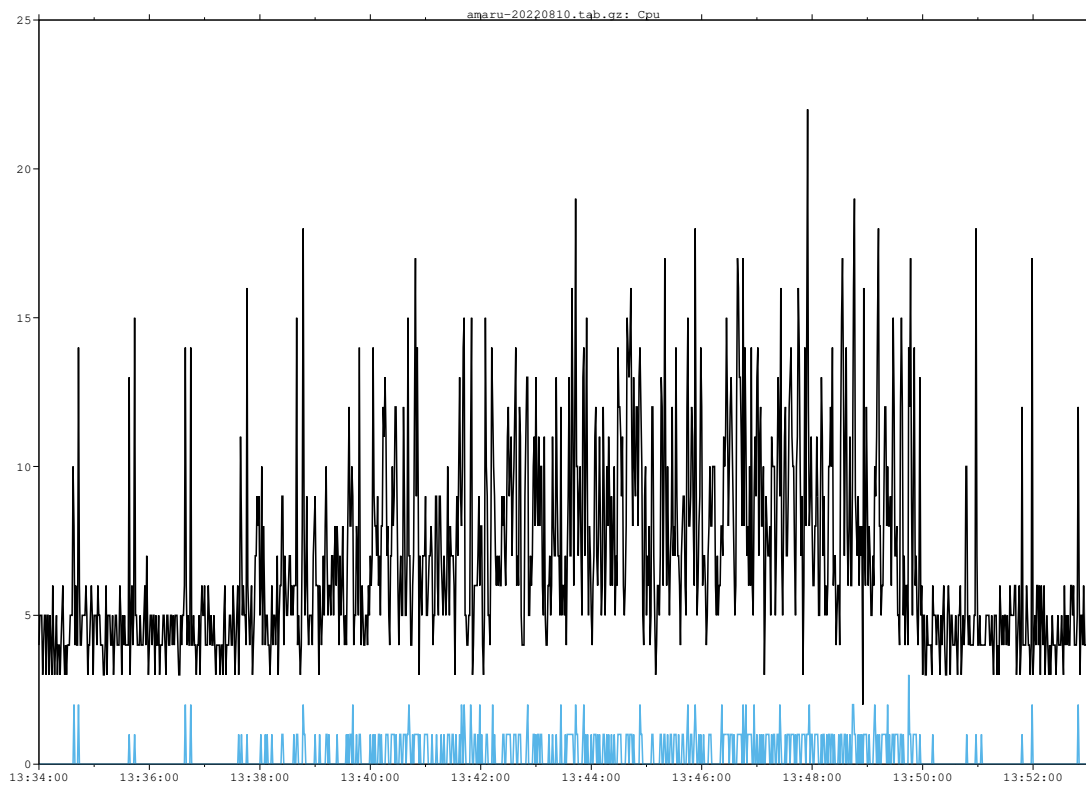


Abbildung 43: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 1

■ CPU User ■ CPU Wait

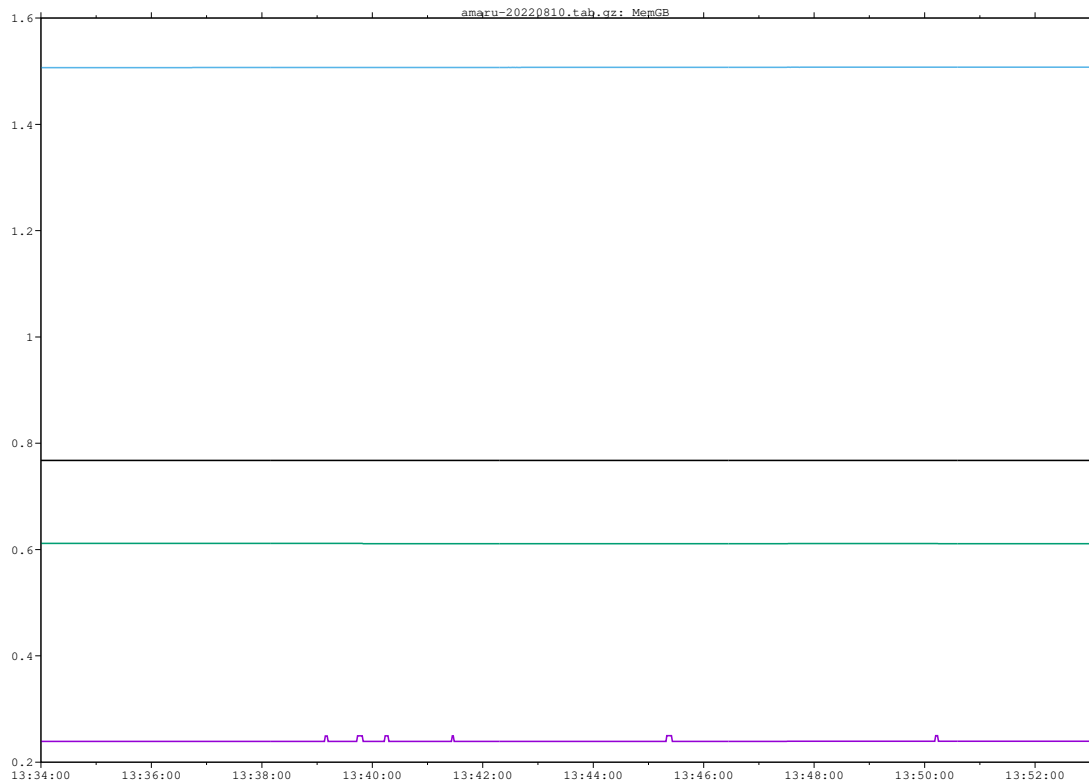


Abbildung 44: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 1

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

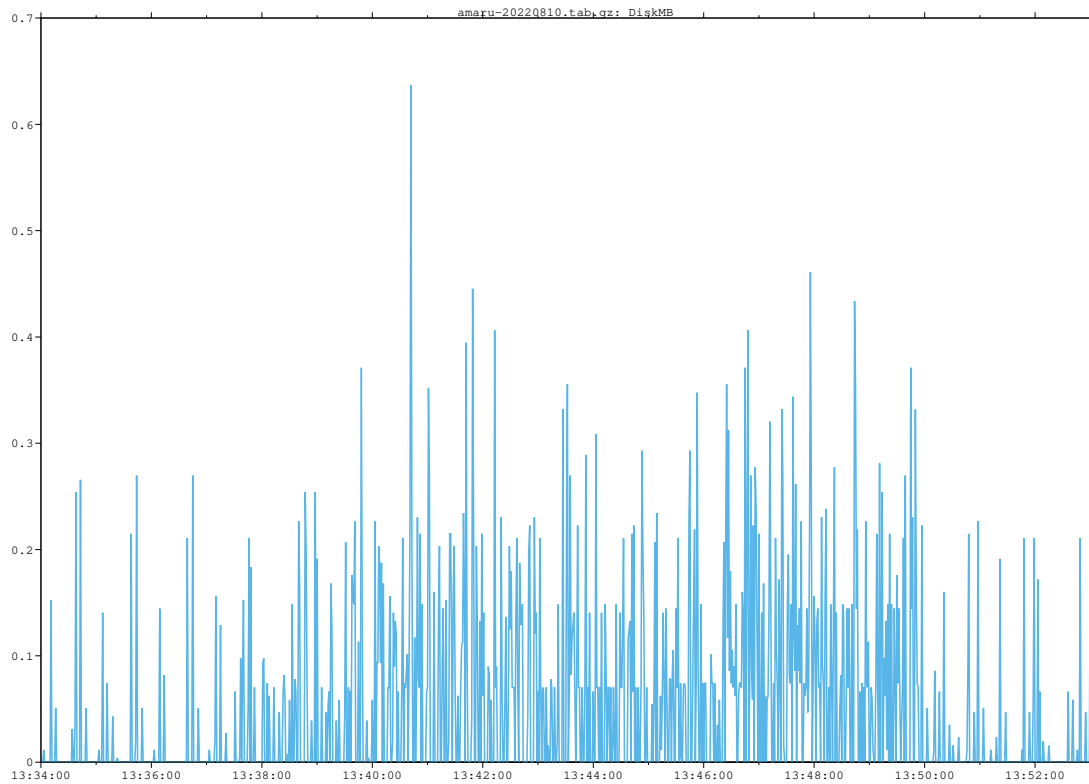


Abbildung 45: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Versuch 1

■ Write ■ Read

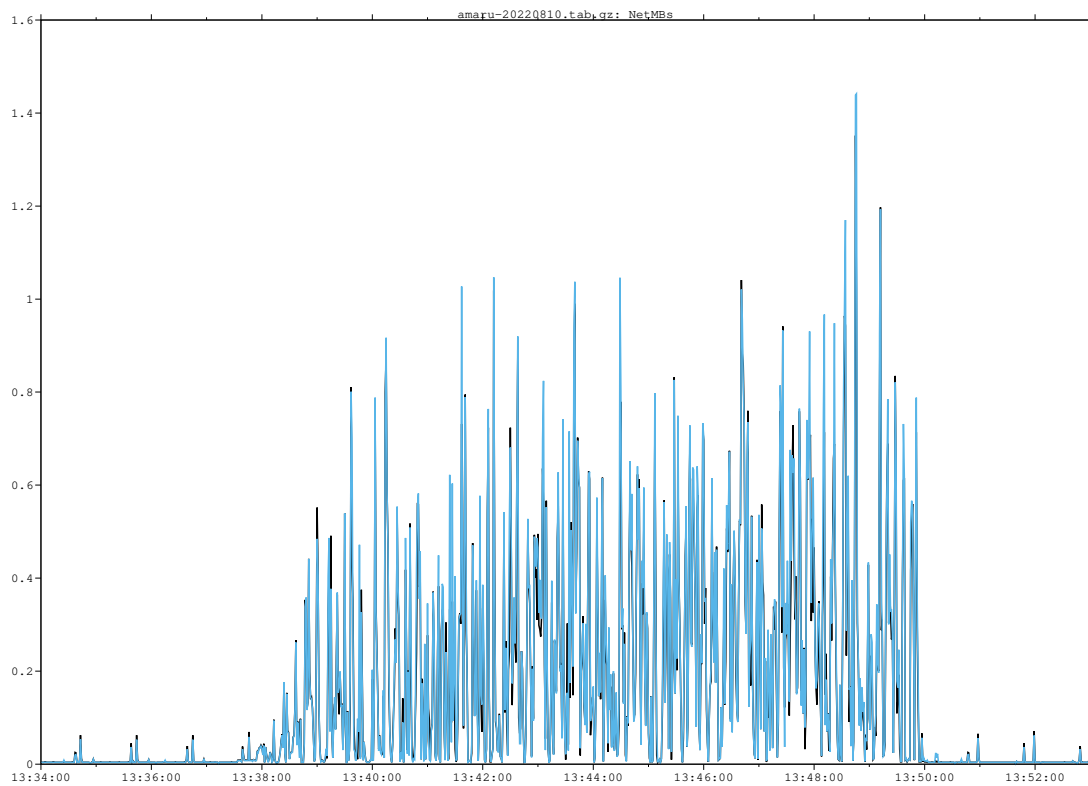


Abbildung 46: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Versuch 1

■ Ausgehend ■ Eingehend

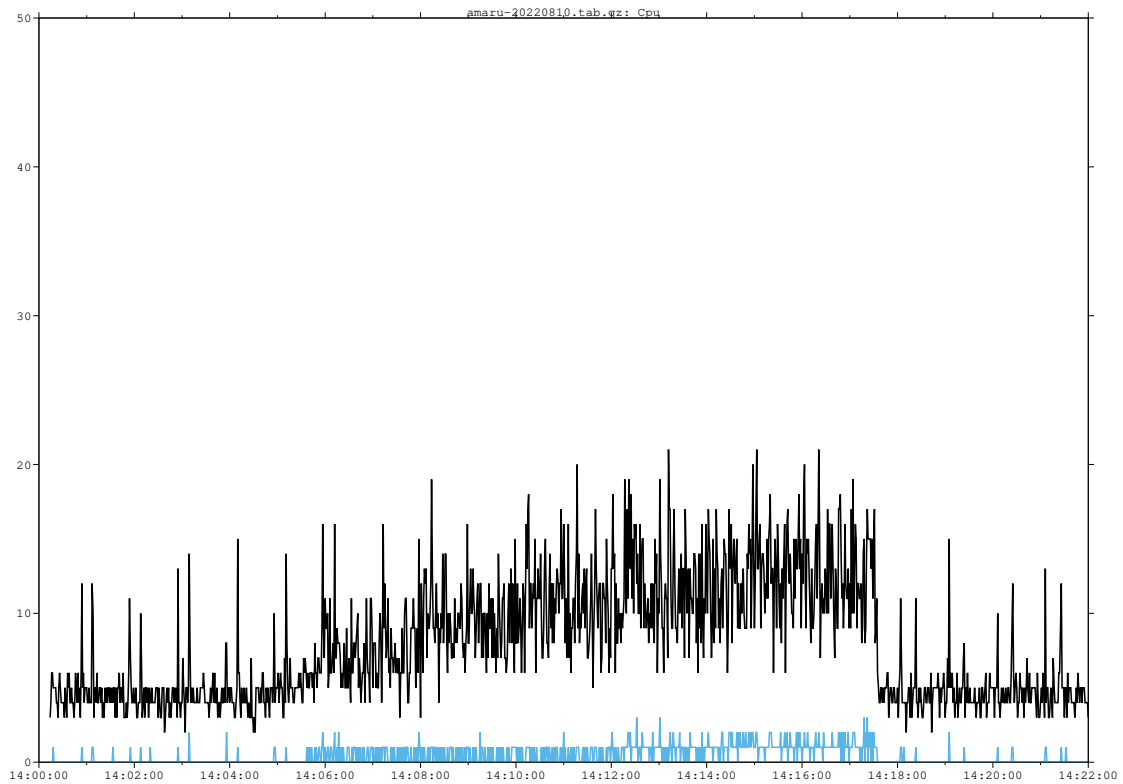


Abbildung 47: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 2

■ CPU User ■ CPU Wait

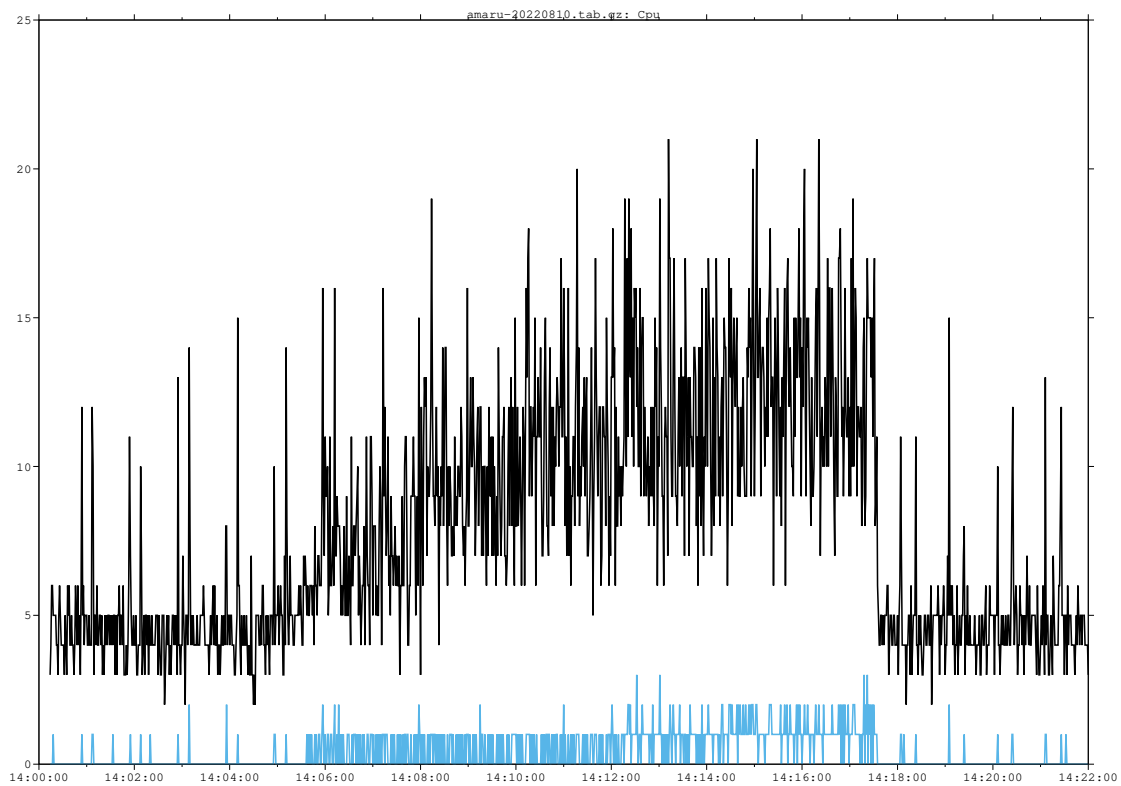


Abbildung 48: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 2

■ CPU User ■ CPU Wait

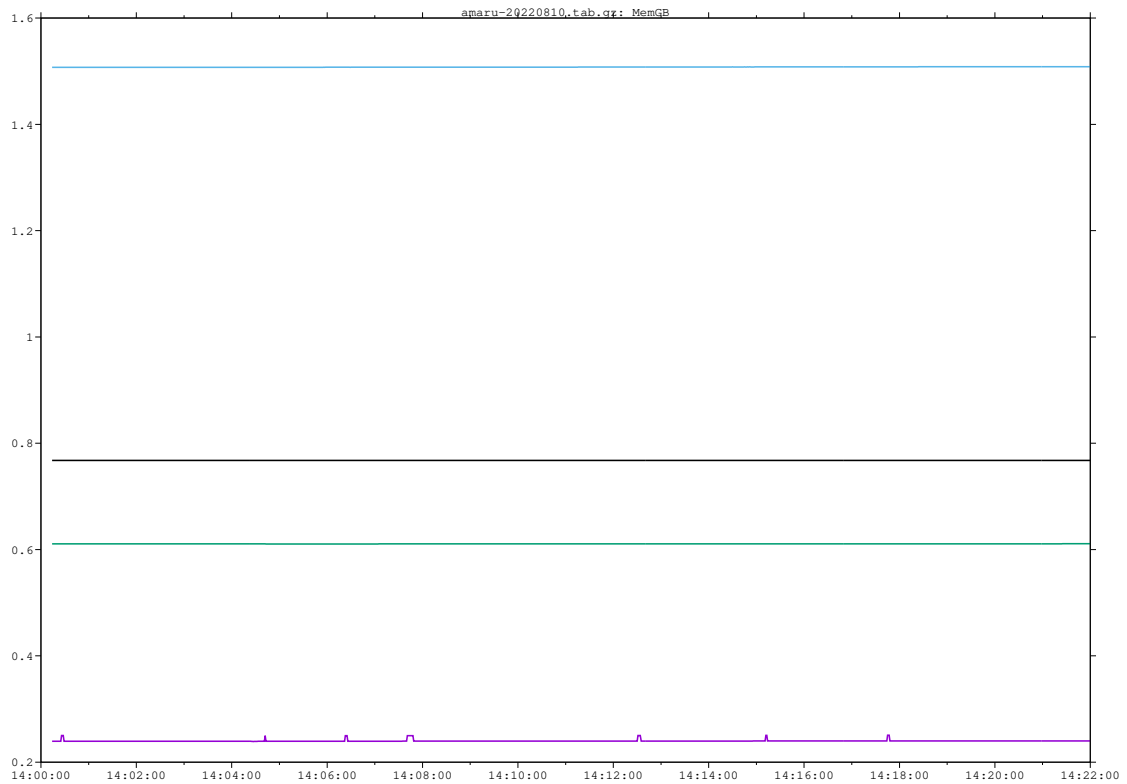


Abbildung 49: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 2

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

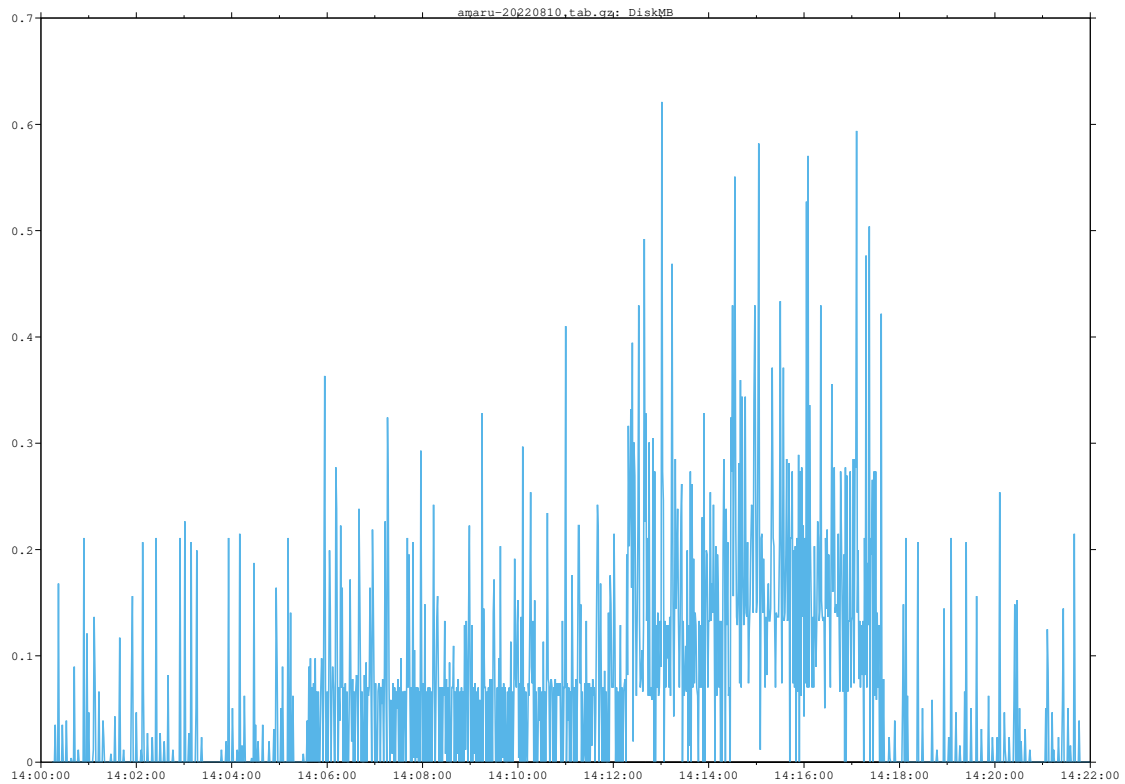


Abbildung 50: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Versuch 2

■ Write ■ Read

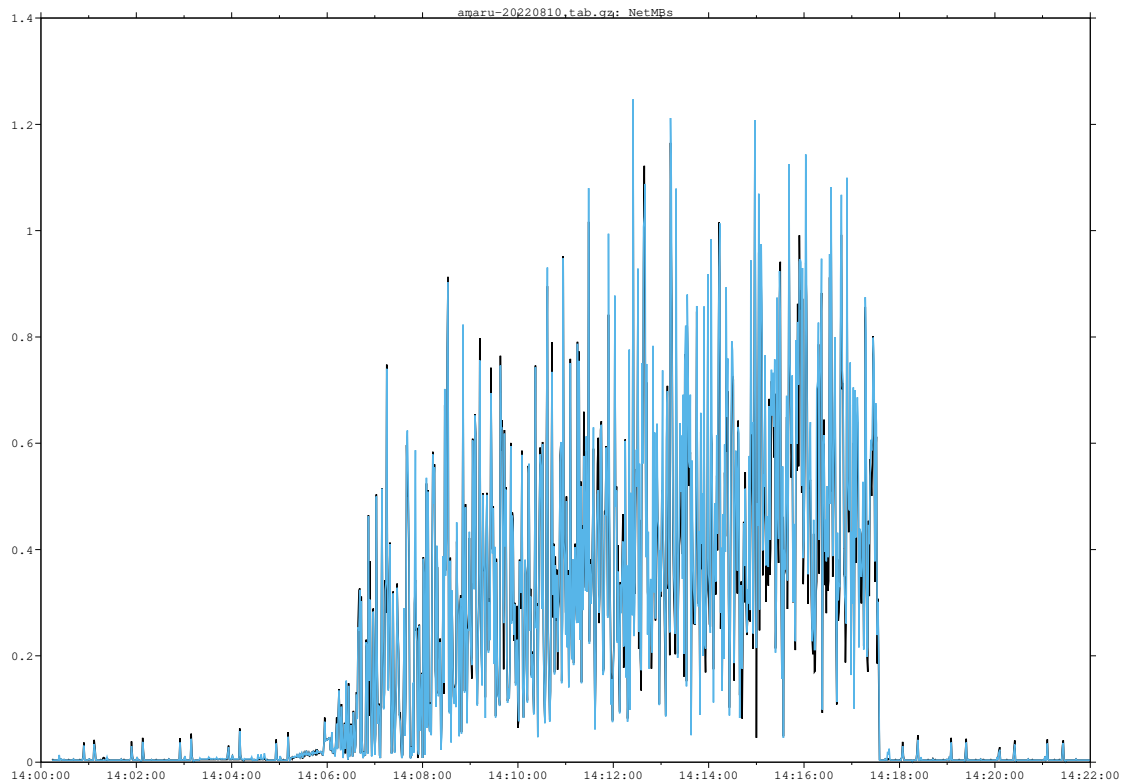


Abbildung 51: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Versuch 2

■ Ausgehend ■ Eingehend

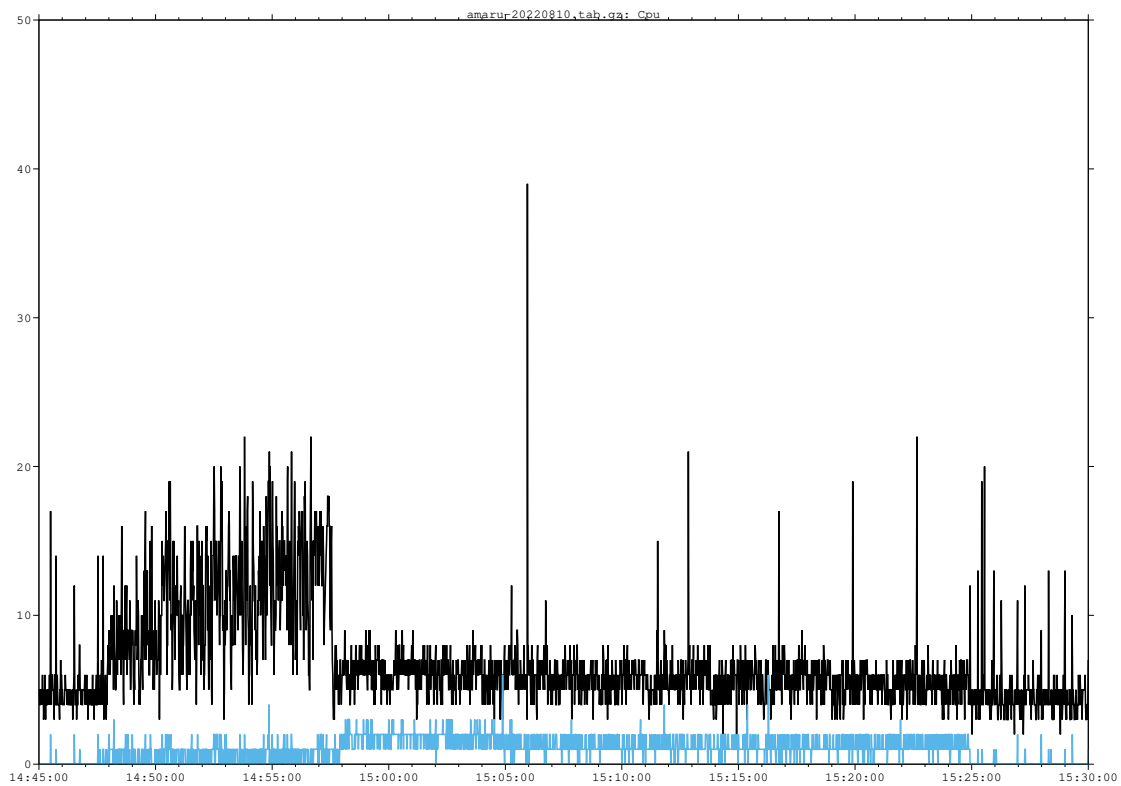


Abbildung 52: CPU Auslastung (0-50 %) des Netzwerkknotens bei Szenario 2/Versuch 3

■ CPU User ■ CPU Wait

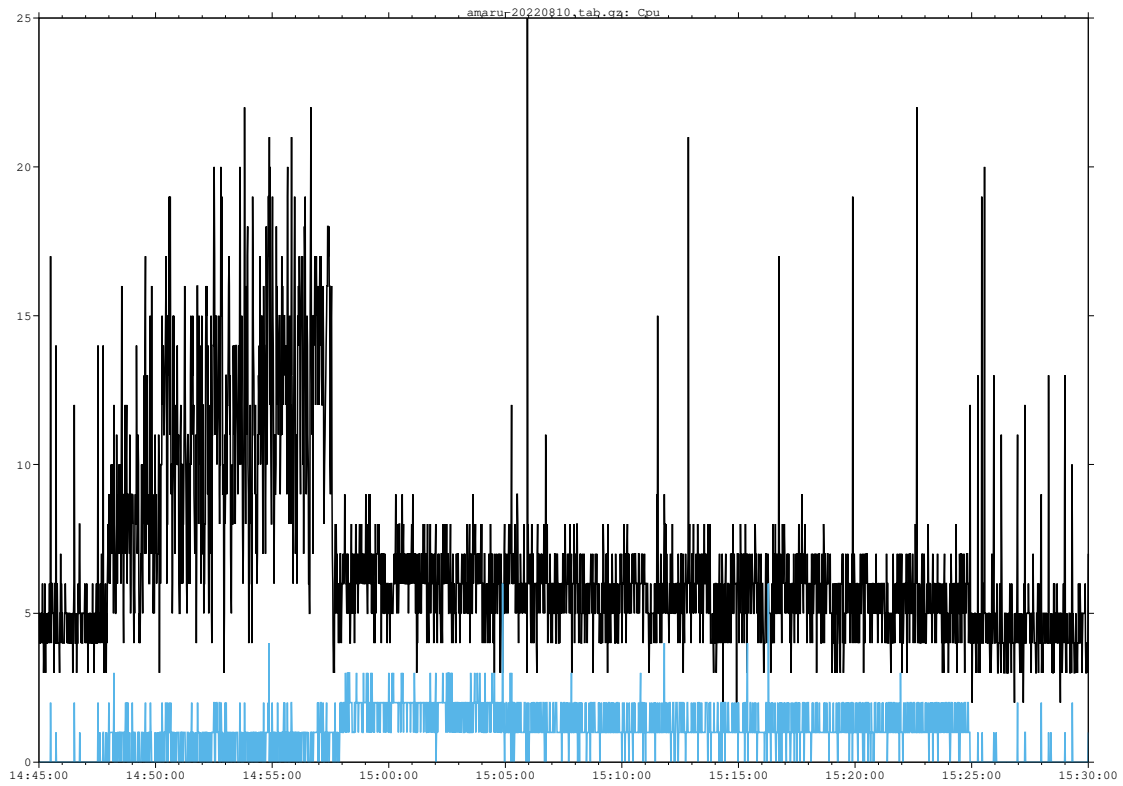


Abbildung 53: CPU Auslastung (0-25 %) des Netzwerkknotens bei Szenario 2/Versuch 3
 ■ CPU User ■ CPU Wait

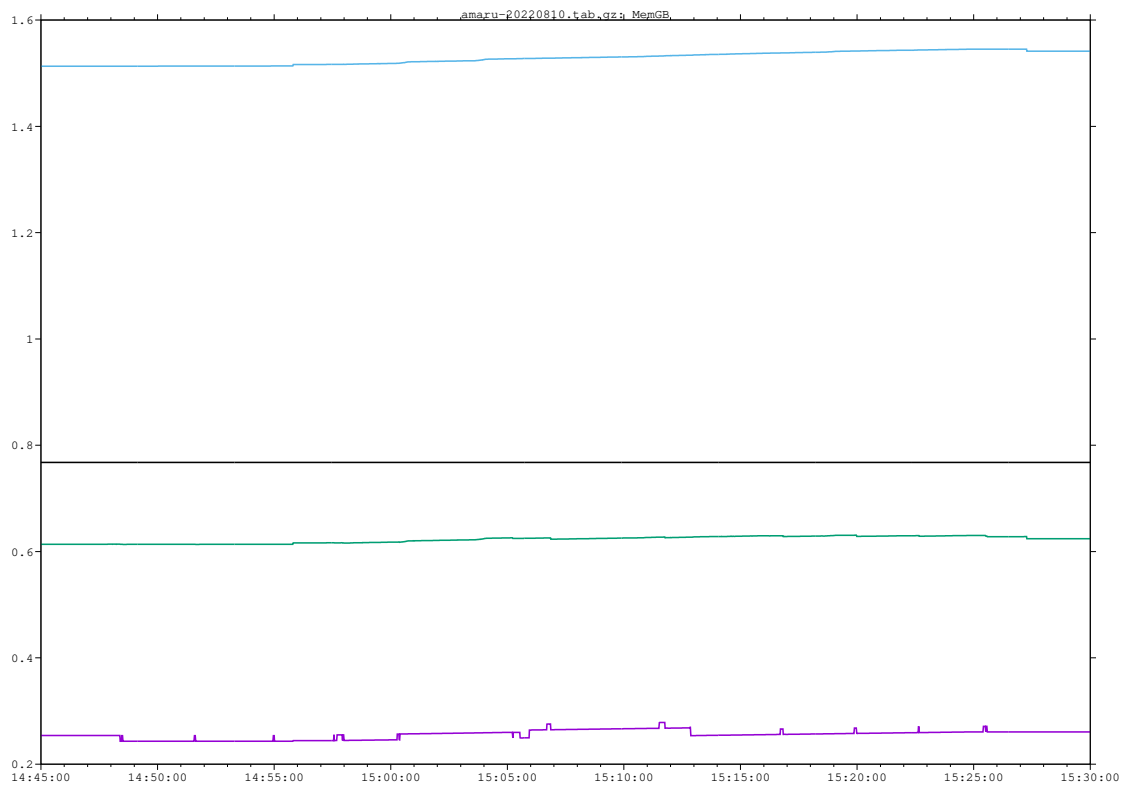


Abbildung 54: Speicherauslastung (GB) des Netzwerkknotens bei Szenario 2/Versuch 3

■ Cached
 ■ Buffered
 ■ Inactive
 ■ Mapped

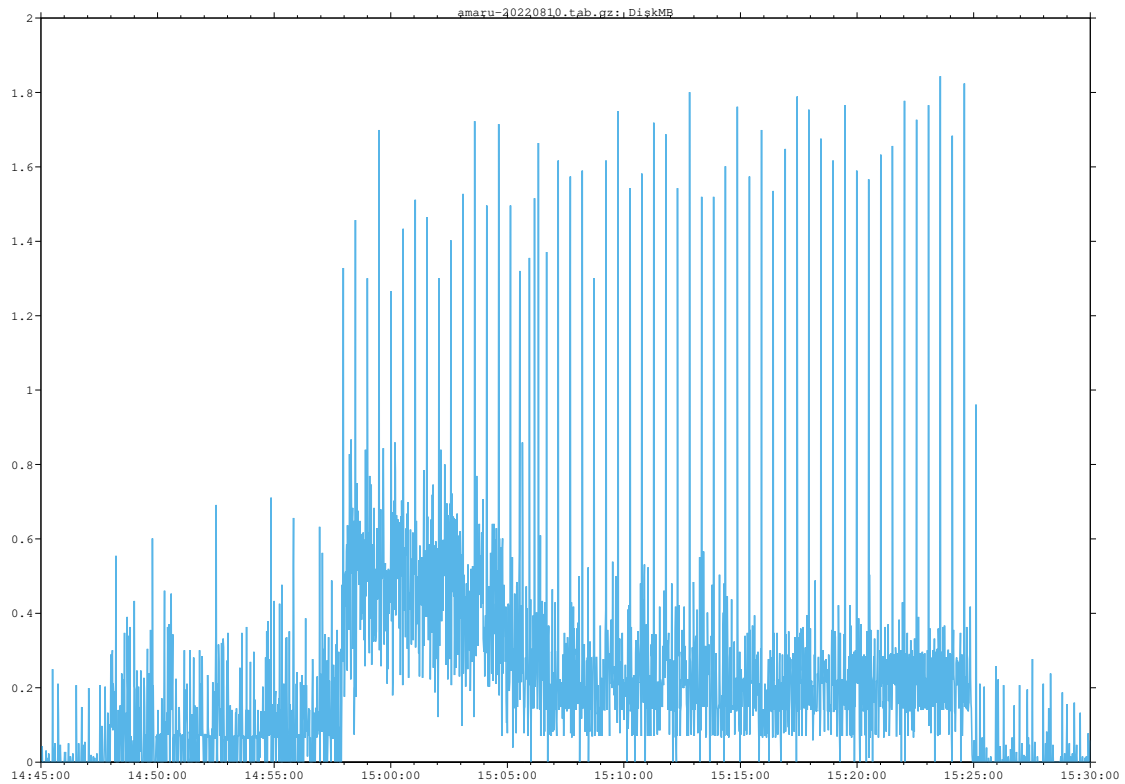


Abbildung 55: Festplattenzugriff (MB) des Netzwerkknotens bei Szenario 2/Versuch 3

■ Write ■ Read

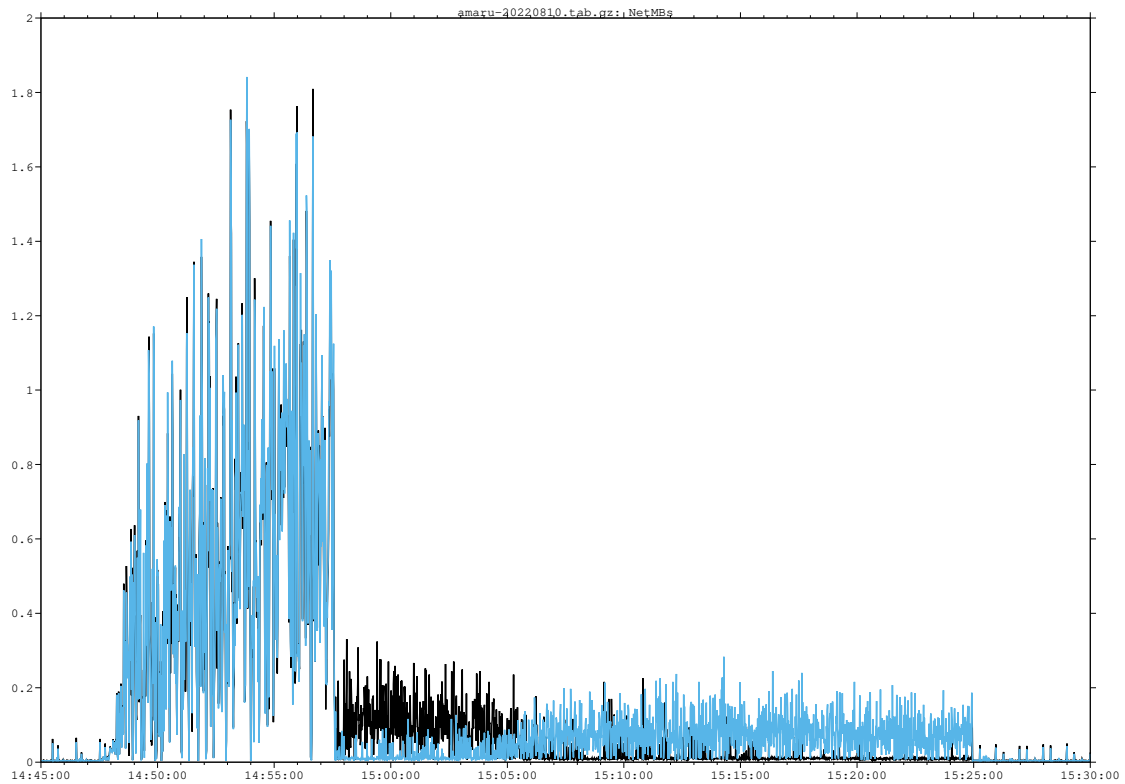


Abbildung 56: Netzwerkverkehr (MB) des Netzwerkknotens bei Szenario 2/Versuch 3

■ Ausgehend ■ Eingehend

Auslastung bei regulärer Nutzung

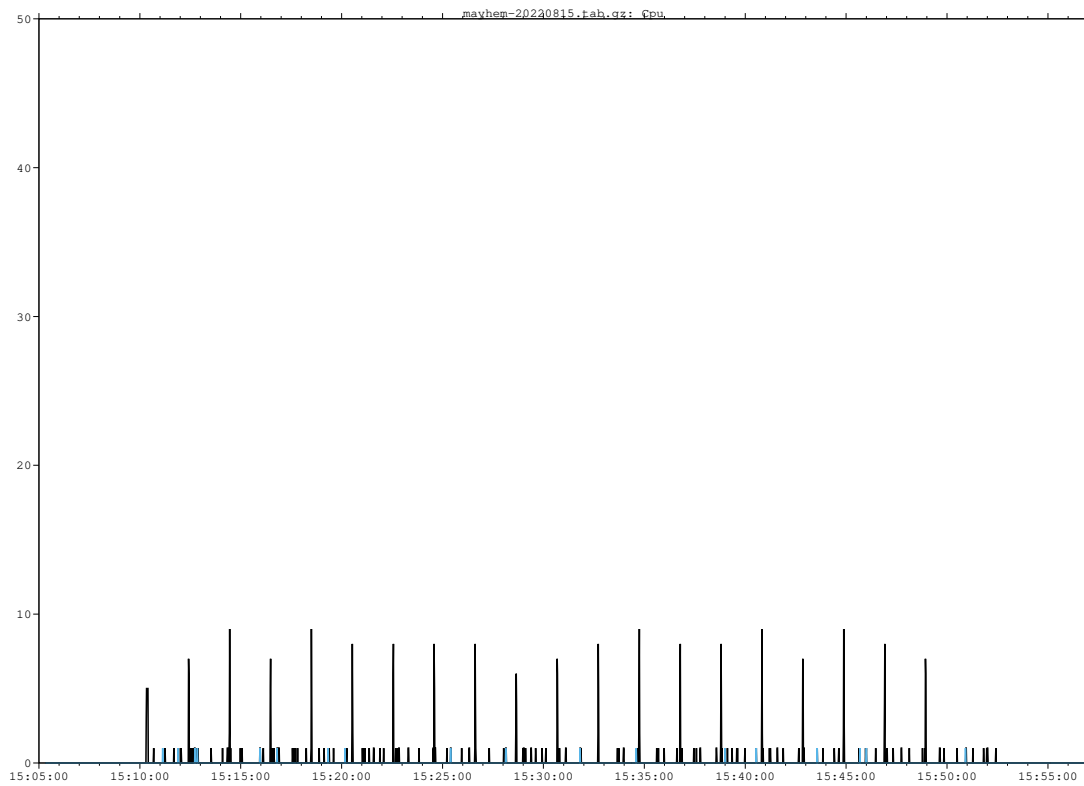


Abbildung 57: CPU Auslastung (0-50 %) bei regulärer Nutzung bei Versuch 1

■ CPU User ■ CPU Wait

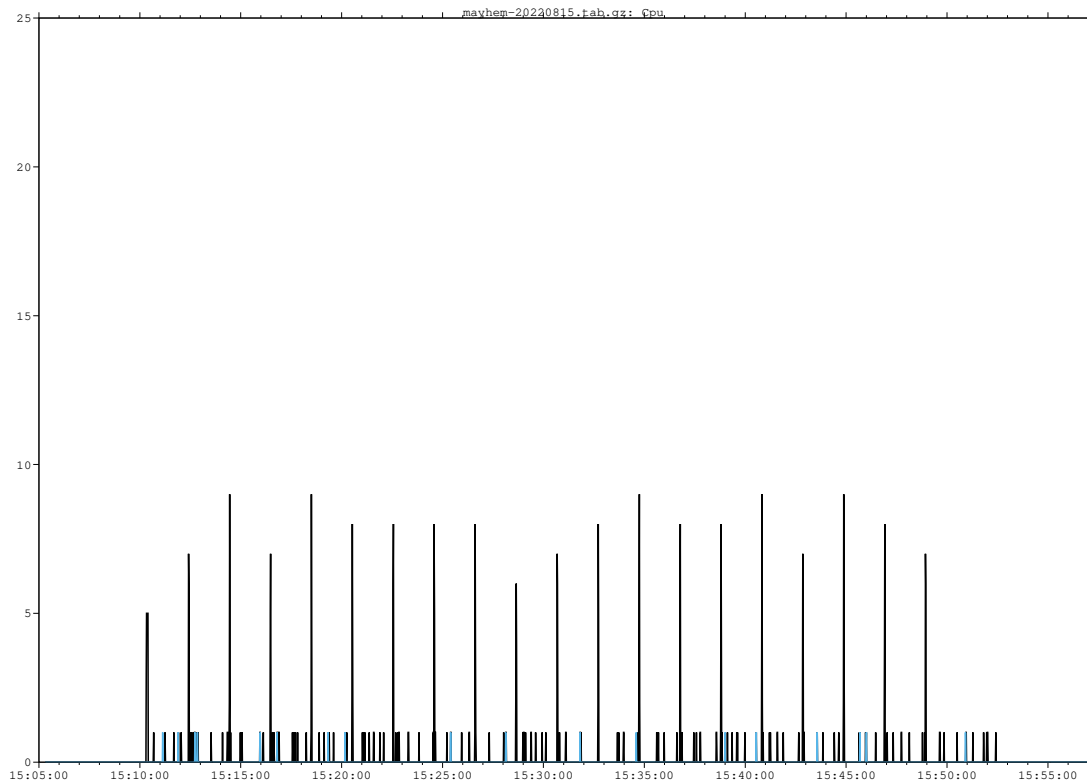


Abbildung 58: CPU Auslastung (0-25 %) bei regulärer Nutzung bei Versuch 1

■ CPU User ■ CPU Wait

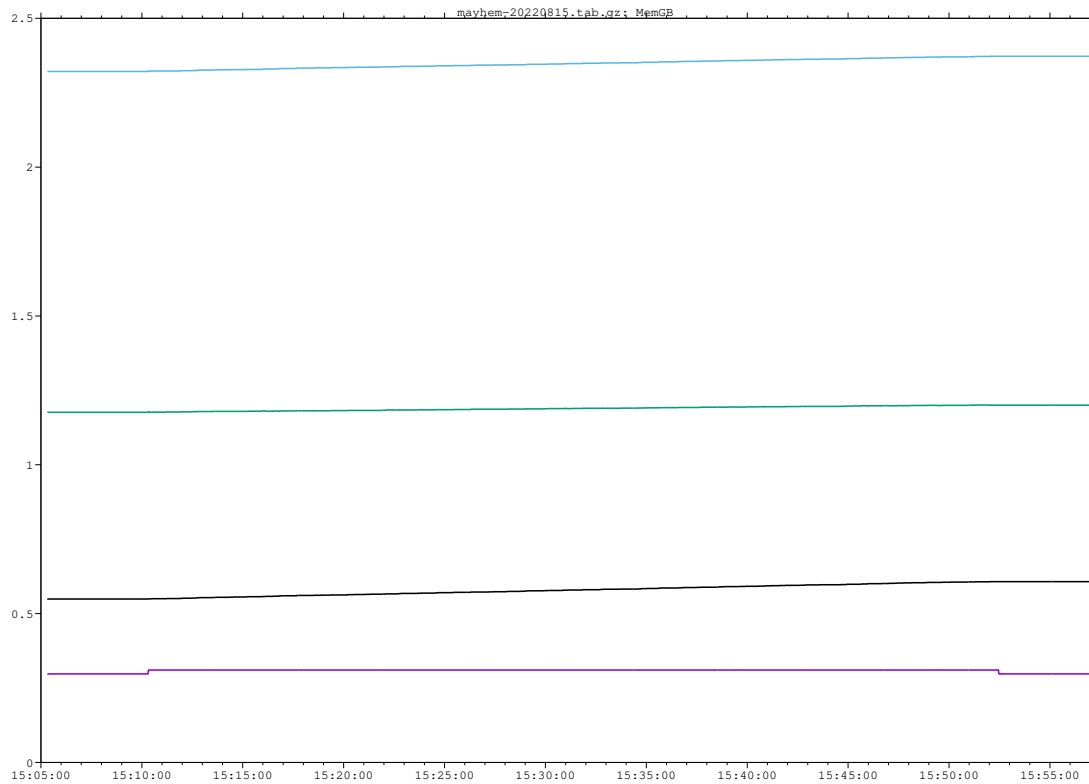


Abbildung 59: Speicherauslastung (GB) bei regulärer Nutzung bei Versuch 1
 ■ Cached ■ Buffered ■ Inactive ■ Mapped

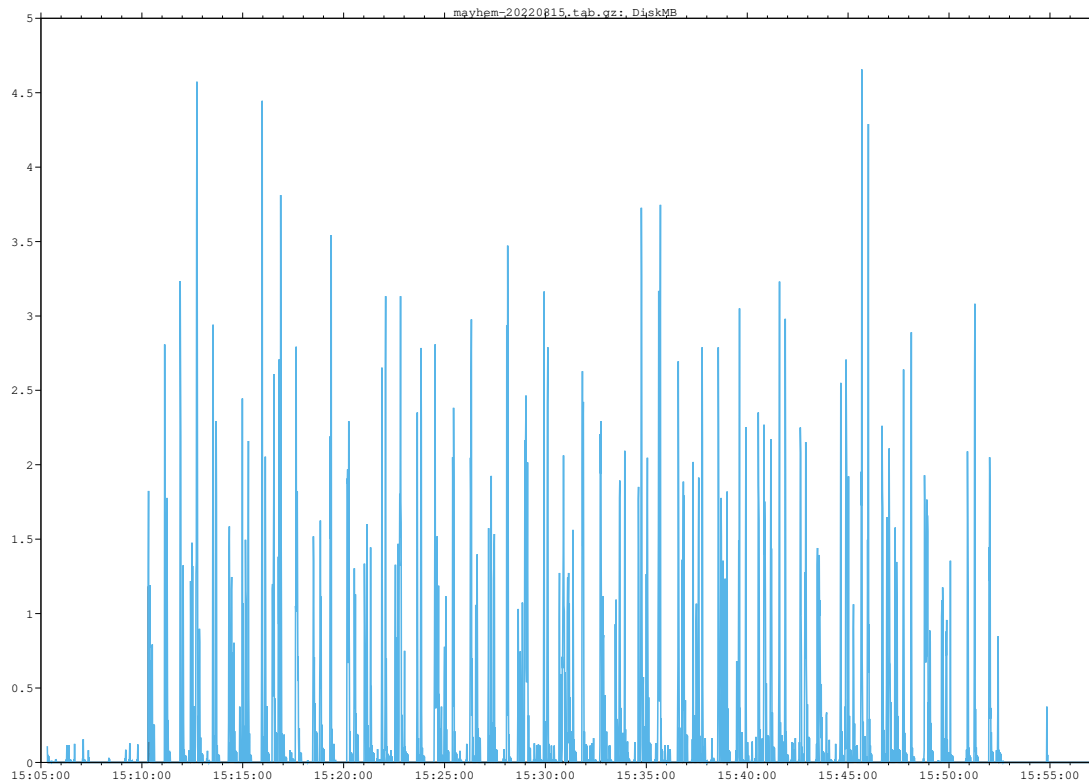


Abbildung 60: Festplattenzugriff (MB) bei regulärer Nutzung bei Versuch 1
■ Write ■ Read

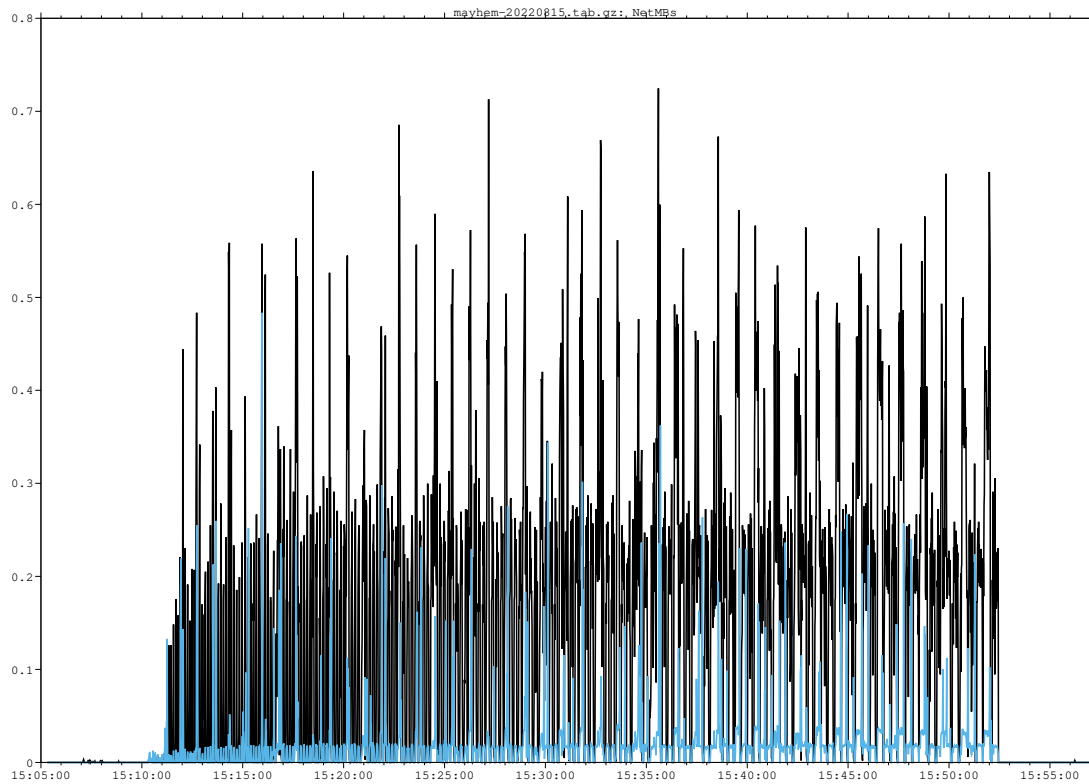


Abbildung 61: Netzwerkverkehr (MB) bei regulärer Nutzung bei Versuch 1
■ Ausgehend ■ Eingehend

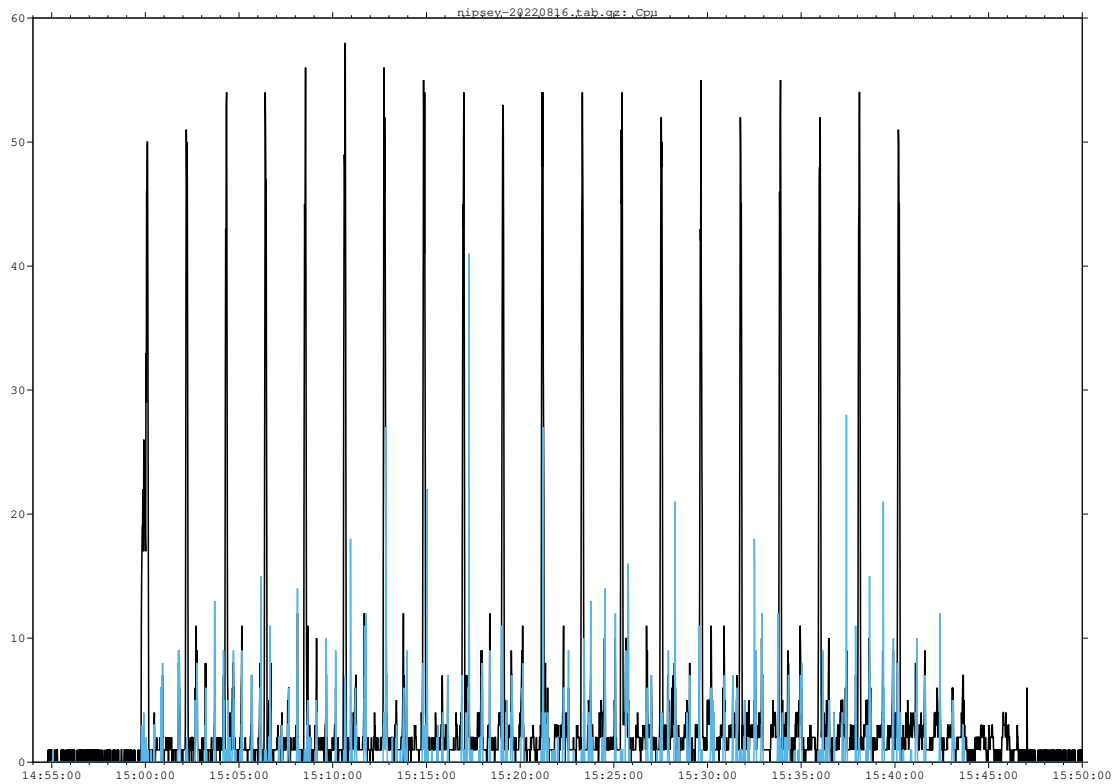


Abbildung 62: CPU Auslastung (0-60 %) bei regulärer Nutzung bei Versuch 2

■ CPU User ■ CPU Wait

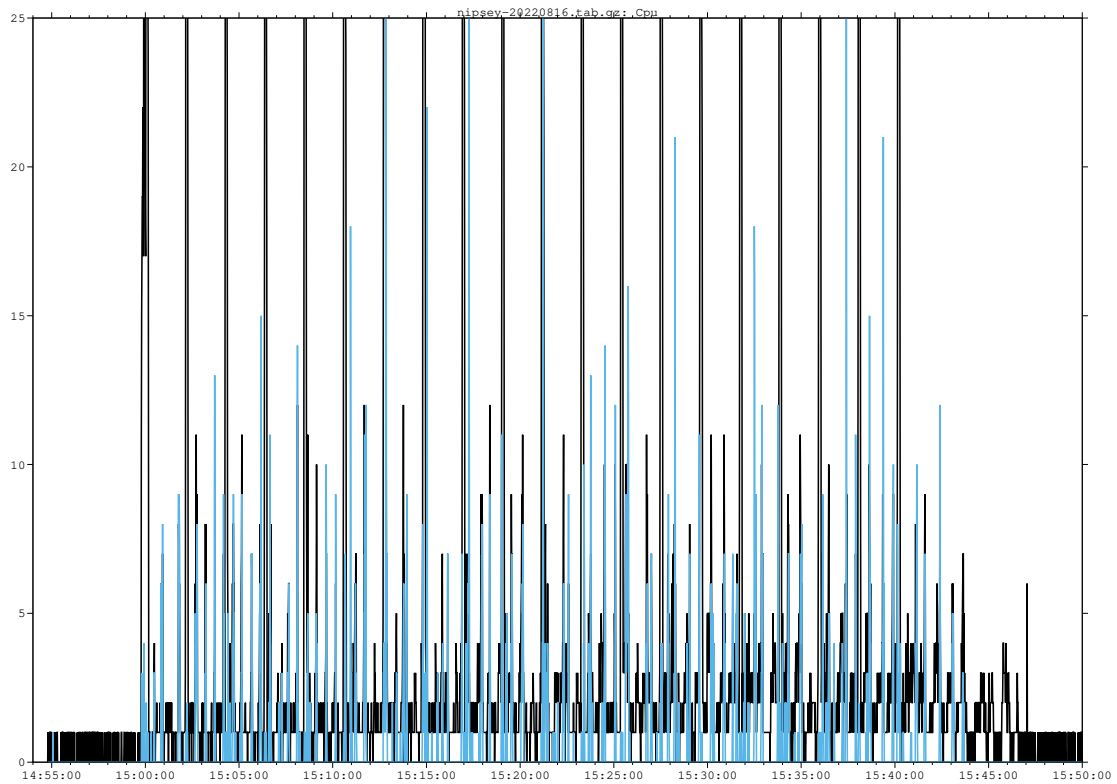


Abbildung 63: CPU Auslastung (0-25 %) bei regulärer Nutzung bei Versuch 2

■ CPU User ■ CPU Wait

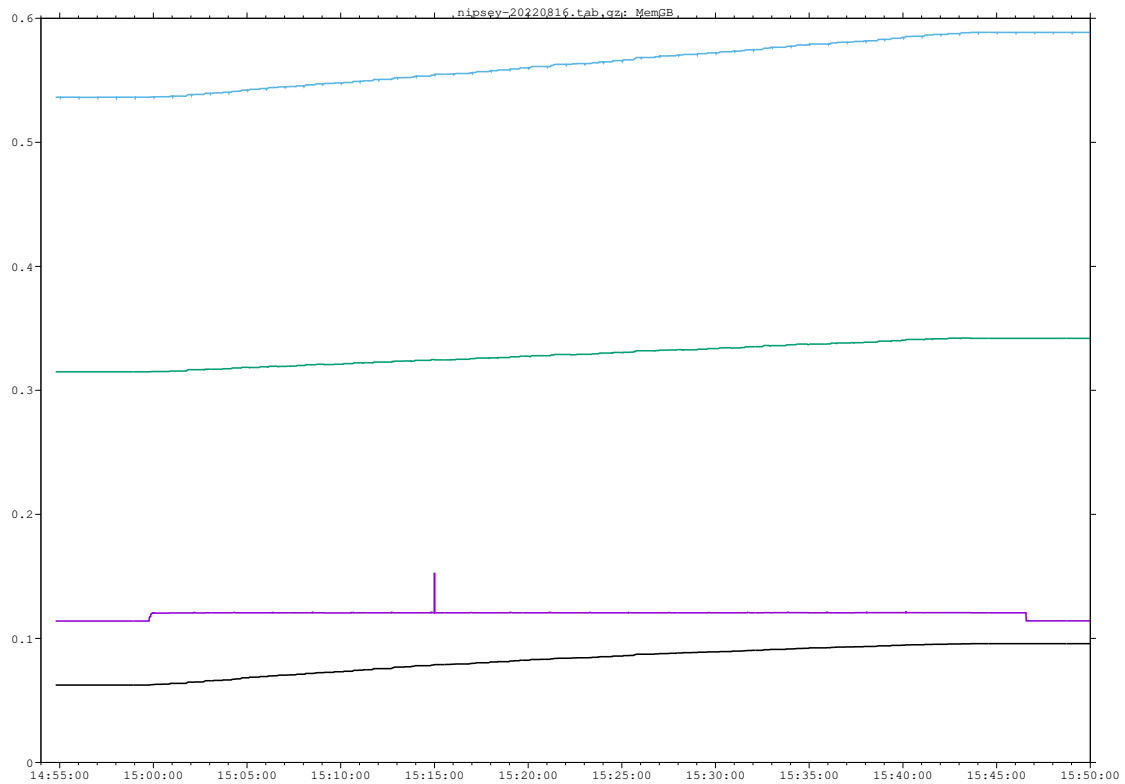


Abbildung 64: Speicherauslastung (GB) bei regulärer Nutzung bei Versuch 2
 ■ Cached ■ Buffered ■ Inactive ■ Mapped

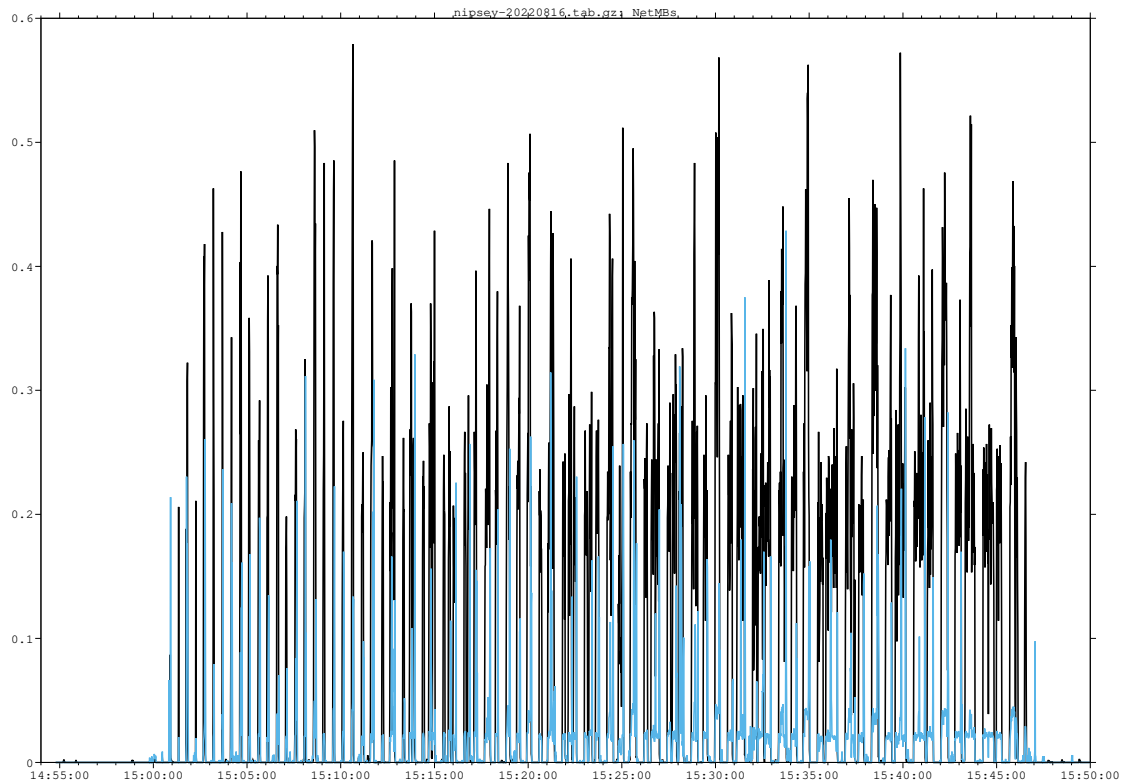


Abbildung 65: Netzwerkverkehr (MB) bei regulärer Nutzung bei Versuch 2
■ Ausgehend ■ Eingehend

D.3 Fides als Smart Contract

```

//”SPDX-License-Identifier: MIT”
pragma solidity >=0.7.0 <0.9.0;

contract Fds {

    uint256 template_index = 0;
    struct Template{
        address receiver;
        string [] tasks;
        bool [] responsibility;
        string description;
        string state;
    }
    mapping (uint256 => Template) templates;

    uint256 contract_index = 0;
    struct Contract{
        address sender;
        address receiver;
        uint256 template;
        uint256 current_task;
        uint256 num_tasks;
        bool [] responsibility;
        string state;
        string [] tx_inputs;
    }
    mapping (uint256 => Contract) contracts;

    function create_template(string [] memory tasks, bool [] memory responsibility, string
        memory description) public returns (uint256){
        require(tasks.length > 0, "Template must have tasks");
        require(tasks.length == responsibility.length, "Tasks and Responsibility must
            match");
        template_index++;
        templates[template_index] = Template(msg.sender, tasks, responsibility,
            description, "ACTIVE");
        return template_index;
    }

    function get_template(uint256 num) public view returns (Template memory){
        require(num <= template_index && num > 0, "Template not found");
        return templates[num];
    }

    function revoke_template(uint256 num) public{
        require(num <= template_index && num > 0, "Template not found");

        Template storage t = templates[num];
        require(msg.sender == t.receiver, "Only the owner of the Template can revoke the
            template");
        t.state = "INACTIVE";
    }

    function create_contract(uint256 template) public returns (uint256){
        require(template > 0 && template <= template_index, "Template not found");
        contract_index++;
        Template storage t = templates[template];
        require(keccak256(abi.encodePacked(t.state)) != keccak256(abi.encodePacked("
            INACTIVE")), "Template is inactive");
        require(t.receiver != msg.sender, "Can't create contract with yourself");
    }
}

```

```

        contracts[contract_index] = Contract(msg.sender, t.receiver, template, 0, t.
            responsibility.length, t.responsibility, "OFFER", new string[](t.
                responsibility.length));
        return contract_index;
    }

    function get_contract(uint256 num) public view returns (Contract memory){
        require(num <= contract_index && num > 0, "Contract not found");
        return contracts[num];
    }

    function accept_contract(uint256 num) public{
        require(num <= contract_index && num > 0, "Contract not found");
        Contract storage c = contracts[num];
        Template storage t = templates[c.template];

        require(msg.sender == t.receiver, "Only the owner of the Template can accept the
            contract");
        c.state = "LIVE";
        c.current_task = 1;
    }

    function decline_contract(uint256 num) public{
        require(num <= contract_index && num > 0, "Contract not found");
        Contract storage c = contracts[num];
        Template storage t = templates[c.template];

        require(msg.sender == t.receiver, "Only the owner of the Template can decline
            the contract");
        c.state = "REJECTED";
    }

    function confirm_task(uint256 num, string memory input_data) public{
        require(num <= contract_index && num > 0, "Contract not found");

        Contract storage c = contracts[num];

        require(keccak256(abi.encodePacked(c.state)) != keccak256(abi.encodePacked("
            OFFER")), "Contract is in a wrong state (OFFER)");
        require(keccak256(abi.encodePacked(c.state)) != keccak256(abi.encodePacked("
            REJECTED")), "Contract is in a wrong state (REJECTED)");
        require(keccak256(abi.encodePacked(c.state)) != keccak256(abi.encodePacked("
            FINISHED")), "Contract is in a wrong state (FINISHED)");

        if (!c.responsibility[c.current_task - 1]){
            require(msg.sender == c.sender);
        }else{
            require(msg.sender == c.receiver);
        }
    }

    c.tx_inputs.push(input_data);
    if (c.current_task == c.responsibility.length){
        c.state = "FINISHED";
    }else{
        c.current_task++;
    }
}
}
}

```

Literatur

- [1] Arizona House Bill 2417. <https://legiscan.com/AZ/text/HB2417/id/1588180> (Abgerufen im Februar 2023).
- [2] Cardano — Discover Cardano. <https://cardano.org/discover-cardano/> (Abgerufen im Februar 2023).
- [3] collectl. <http://collectl.sourceforge.net/> (Abgerufen im Februar 2023).
- [4] collectl(1) - Linux man page. <https://linux.die.net/man/1/collectl> (Abgerufen im Februar 2023).
- [5] Contract ABI Specification. <https://docs.soliditylang.org/en/v0.8.17/abi-spec.html> (Abgerufen im Februar 2023).
- [6] Decentralized Reputation in OpenBazaar (Wayback Machine). <https://web.archive.org/web/20210308130752/https://openbazaar.org/blog/decentralized-reputation-in-openbazaar/> (Abgerufen im Februar 2023).
- [7] Ethereum accounts. <https://ethereum.org/en/developers/docs/accounts/> (Abgerufen im Februar 2023).
- [8] Ethereum Charts and Statistics — Etherscan. <https://etherscan.io/charts> (Abgerufen im Februar 2023).
- [9] Ethereum Virtual Machine (EVM). <https://ethereum.org/en/developers/docs/evm/> (Abgerufen im Februar 2023).
- [10] Gas and fees. <https://ethereum.org/en/developers/docs/gas/> (Abgerufen im Februar 2023).
- [11] LoRa Alliance. <https://lora-alliance.org/> (Abgerufen im Februar 2023).

- [12] lscpu(1) - Linux man page. <https://linux.die.net/man/1/lscpu> (Abgerufen im Februar 2023).
- [13] Marlowe — Cardano Developer Portal. <https://developers.cardano.org/docs/smart-contracts/marlowe/> (Abgerufen im Februar 2023).
- [14] Michelson: the language of Smart Contracts in Tezos. <https://tezos.gitlab.io/active/michelson.html> (Abgerufen im Februar 2023).
- [15] Networking layer. <https://ethereum.org/en/developers/docs/networking-layer/> (Abgerufen im Februar 2023).
- [16] Opcodes for the EVM. <https://ethereum.org/en/developers/docs/evm/opcodes/> (Abgerufen im Februar 2023).
- [17] OpenBazaar - GitHub. <https://github.com/OpenBazaar> (Abgerufen im Februar 2023).
- [18] openbazaar-go/contracts.proto. <https://github.com/OpenBazaar/openbazaar-go/blob/74075f74a22fff3e24ee0104212b1010b84efe96/pb/protocols/contracts.proto> (Abgerufen im Februar 2023).
- [19] OpenBazaar (Wayback Machine). <https://web.archive.org/web/20211228065459/https://openbazaar.org/> (Abgerufen im Februar 2023).
- [20] Oracle. <https://ethereum.org/en/developers/docs/oracles/#applications-of-oracles-in-smart-contracts> (Abgerufen im Februar 2023).
- [21] Overview — Solana Docs. <https://docs.solana.com/developing/on-chain-programs/overview> (Abgerufen im Februar 2023).

- [22] Plutus — Cardano Developer Portal. <https://developers.cardano.org/docs/smart-contracts/plutus/> (Abgerufen im Februar 2023).
- [23] Proof-of-stake (PoS). <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/> (Abgerufen im Februar 2023).
- [24] PyTeal - Algorand Developer Portal. <https://developer.algorand.org/docs/get-details/dapps/pyteal/> (Abgerufen im Februar 2023).
- [25] Remix - Ethereum IDE. <https://remix.ethereum.org/> (Abgerufen im Februar 2023).
- [26] Script - Bitcoin Wiki. <https://en.bitcoin.it/wiki/Script> (Abgerufen im Februar 2023).
- [27] Senate File 541. <https://www.legis.iowa.gov/legislation/BillBook?ga=89&ba=SF541> (Abgerufen im Februar 2023).
- [28] Smart contracts. <https://ethereum.org/en/smart-contracts/> (Abgerufen im Februar 2023).
- [29] Smart contracts. <https://ethereum.org/en/developers/docs/smart-contracts/> (Abgerufen im Februar 2023).
- [30] Solidity Programming Language. <https://soliditylang.org/> (Abgerufen im Februar 2023).
- [31] The Algorand Virtual Machine (AVM) and TEAL. - Algorand Developer Portal. <https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/> (Abgerufen im Februar 2023).
- [32] Transactions. <https://ethereum.org/en/developers/docs/transactions/> (Abgerufen im Februar 2023).

- [33] Uniform Electronic Transactions Act. <https://www.leg.state.nv.us/nrs/nrs-719.html> (Abgerufen im Februar 2023).
- [34] Verbose Data. <http://collect1.sourceforge.net/Data-verbose.html> (Abgerufen im Februar 2023).
- [35] What are Solana Programs? — Solana Docs. <https://docs.solana.com/developing/intro/programs#on-chain-programs> (Abgerufen im Februar 2023).
- [36] What is Ethereum? <https://ethereum.org/en/what-is-ethereum/> (Abgerufen im Februar 2023).
- [37] Zero-knowledge proofs. <https://ethereum.org/en/zero-knowledge-proofs/> (Abgerufen im Februar 2023).
- [38] Ben Finney. python-daemon - PyPI. <https://pypi.org/project/python-daemon/> (Abgerufen im Februar 2023).
- [39] Aanand Prasad et al. Command Line Interface Guidelines. <https://clig.dev/> (Abgerufen im Februar 2023).
- [40] arjanvanb. Fair Use Policy explained. <https://www.thethingsnetwork.org/forum/t/fair-use-policy-explained/1300> (Abgerufen im Februar 2023).
- [41] Armin Ronacher. click - PyPI. <https://pypi.org/project/click/> (Abgerufen im Februar 2023).
- [42] avbentem. Airtime calculator for LoRaWAN. <https://avbentem.github.io/airtime-calculator/> (Abgerufen im Februar 2023).
- [43] Jhonattan J. Barriga, Juan Sulca, José León, Alejandro Ulloa, Diego Portero, José García, and Sang Guun Yoo. A smart parking solution architecture based on lorawan and kubernetes. *Applied Sciences*, 10(13), 2020.

- [44] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.
- [45] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [46] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012.
- [47] A. Beutelspacher, J. Schwenk, and K.D. Wolfenstetter. *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge*. Vieweg+Teubner Verlag, 2013.
- [48] Bitdefender. Supply Chain Attack Detected in PyPI Library. <https://www.bitdefender.com/blog/hotforsecurity/supply-chain-attack-detected-in-pypi-library/> (Abgerufen im Februar 2023).
- [49] Bundesministerium für Wirtschaft und Energie, Bundesministerium der Finanzen. Blockchain-Strategie der Bundesregierung. <https://www.bmwk.de/Redaktion/DE/Publikationen/Digitale-Welt/blockchain-strategie.pdf> (Abgerufen im Februar 2023).
- [50] Vitalik Buterin et al. Ethereum white paper. <https://ethereum.org/en/whitepaper/> (Abgerufen im Februar 2023), 2013.
- [51] C.H. Cap. Theoretische Grundlagen der Informatik. page 107. Springer Vienna, 2013.

- [52] Angelica Cardenas, Kiyoshy Nakamura, Ermanno Pietrosemoli, Marco Zennaro, Marco Rainone, and Pietro Manzoni. A low-cost and low-power messaging system based on the lora wireless technology. *Mobile Networks and Applications*, 25, 06 2020.
- [53] Abdelberi Chaabane, Mohamed Ali Kaafar, and Roksana Boreli. Big friend is watching you: Analyzing online social networks tracking capabilities. In *Proceedings of the 2012 ACM Workshop on Workshop on Online Social Networks, WOSN '12*, page 7–12, New York, NY, USA, 2012. Association for Computing Machinery.
- [54] David Chaum. Blind signatures for untraceable payments. In D. Chaum, R.L. Rivest, and A.T. Sherman, editors, *Advances in Cryptology Proceedings of Crypto 82*, pages 199–203, 1983.
- [55] Jing Chen and Silvio Micali. Algorand. <https://arxiv.org/abs/1607.01341>, 2016.
- [56] Law Commission. Smart legal contracts Advice to Government. <https://www.lawcom.gov.uk/project/smart-contracts/> (Abgerufen im Februar 2023).
- [57] L. Creutz and G. Dartmann. Cypher social contracts a novel protocol specification for cyber physical smart contracts. In *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, pages 440–447, 2020.
- [58] L. Creutz, K. Wagner, and G. Dartmann. Cyber-physical contracts in offline regions. In *2022 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and*

- IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, pages 461–469, 2022.
- [59] Lars Creutz. Fides 0.5.1. <https://doi.org/10.5281/zenodo.7331483> (Abgerufen im Februar 2023).
- [60] Lars Creutz. Fides Simulation/Evaluation. <https://doi.org/10.5281/zenodo.7331320> (Abgerufen im Februar 2023).
- [61] Lars Creutz, Jens Schneider, and Guido Dartmann. Fides: Distributed cyber-physical contracts. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 51–60, 2021.
- [62] Lars Creutz, Jens Schneider, and Guido Dartmann. Distributed hash table with extensible remote procedure calls. In *2022 5th International Conference on Computational Intelligence and Networks (CINE)*, pages 1–6, 2022.
- [63] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*. Springer, 1 edition, 2002.
- [64] Lucas R de Oliveira, Poliana de Moraes, Lauro PS Neto, and Arlindo F da Conceição. Review of lorawan applications. *arXiv preprint arXiv:2004.05871*, 2020.
- [65] Google Developers. Protocol Buffers. <https://developers.google.com/protocol-buffers> (Abgerufen im Februar 2023).
- [66] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22:644–654, 1976.

- [67] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.
- [68] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges, 1992.
- [69] Nicolas Dorier. open-assets-protocol - GitHub. <https://github.com/NicolasDorier/open-assets-protocol> (Abgerufen im Februar 2023).
- [70] John R. Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- [71] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss elmau, Germany, IEEECompSoc. Citeseer*, 2001.
- [72] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.
- [73] Morris J. Dworkin. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, Gaithersburg, MD, USA, 2001.
- [74] Alexander Egberts. The oracle problem - an analysis of how blockchain oracles undermine the advantages of decentralized ledger systems. 2017.
- [75] William F. Ehrsam, Carl H. W. Meyer, L. Smith, and Walter L. Tuchman. Message verification and transmission error detection by block chaining, 4 1976. US Patent US4074066A.

- [76] Eli Sohl. Cryptopals: Exploiting CBC Padding Oracles. <https://research.nccgroup.com/2021/02/17/cryptopals-exploiting-cbc-padding-oracles/> (Abgerufen im Februar 2023).
- [77] Fabian Froehlich. fides - PyPI. <https://pypi.org/project/fides/#history> (Abgerufen im Februar 2023).
- [78] Hyperledger Fabric. Open, Proven, Enterprise-grade DLT. https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf (Abgerufen im Februar 2023), 2019. Version 1.0.
- [79] The Python Software Foundation. Built-in Functions. <https://docs.python.org/3.5/library/functions.html#open> (Abgerufen im Februar 2023).
- [80] The Python Software Foundation. Built-in Functions. <https://docs.python.org/3/library/functions.html#getattr> (Abgerufen im Februar 2023).
- [81] The Python Software Foundation. Common pathname manipulations. <https://docs.python.org/3/library/os.path.html> (Abgerufen im Februar 2023).
- [82] The Python Software Foundation. concurrent.futures - ThreadPoolExecutor. <https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor> (Abgerufen im Februar 2023).
- [83] The Python Software Foundation. Configuration file parser. <https://docs.python.org/3/library/configparser.html> (Abgerufen im Februar 2023).

- [84] The Python Software Foundation. `getpass` — Portable password input. <https://docs.python.org/3/library/getpass.html> (Abgerufen im Februar 2023).
- [85] The Python Software Foundation. `importlib` -The implementation of import. <https://docs.python.org/3/library/importlib.html> (Abgerufen im Februar 2023).
- [86] The Python Software Foundation. `importlib` -The implementation of import. <https://docs.python.org/3/library/importlib.html#importlib.machinery.ModuleSpec> (Abgerufen im Februar 2023).
- [87] The Python Software Foundation. `importlib` -The implementation of import. https://docs.python.org/3/library/importlib.html#importlib.util.module_from_spec (Abgerufen im Februar 2023).
- [88] The Python Software Foundation. `importlib` -The implementation of import. https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.exec_module (Abgerufen im Februar 2023).
- [89] The Python Software Foundation. `inspect` — Inspect live objects. <https://docs.python.org/3/library/inspect.html#inspect.getfile> (Abgerufen im Februar 2023).
- [90] The Python Software Foundation. `sqlite3` — DB-API 2.0 interface for SQLite databases. <https://docs.python.org/3/library/sqlite3.html> (Abgerufen im Februar 2023).
- [91] The Python Software Foundation. `sqlite3` — DB-API 2.0 interface for SQLite databases. <https://docs.python.org/3/library/sqlite3.html#sqlite3.connect> (Abgerufen im Februar 2023).
- [92] The Python Software Foundation. `threading` — Thread-based parallelism. <https://docs.python.org/3/library/threading.html#threading.Lock> (Abgerufen im Februar 2023).

- [93] C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016.
- [94] Mark Friedenbach and Jorge Timón. Freemarkets: extending bitcoin protocol with user-specified bearer instruments, peer-to-peer exchange, off-chain accounting, auctions, derivatives and transitive transactions. <http://freico.in/docs/freemarkets.pdf> (Abgerufen im Februar 2023), 2013.
- [95] Indra Spiecker gen. Döhm, Sebastian Bretthauer, and Dirk Müllmann. Rechtliche Studie zum Cypher Social Contracts Konzept „Fides“. <https://doi.org/10.5281/zenodo.7680213>, January 2023.
- [96] Henri Gilbert and Helena Handschuh. Security analysis of sha-256 and sisters. In *ACM Symposium on Applied Computing*, 2003.
- [97] L.M Goodman. Tezos: A Self-Amending Crypto-Ledger. <https://tezos.com/position-paper.pdf> (Abgerufen im Februar 2023).
- [98] Ian Grigg. On the intersection of Ricardian and Smart Contracts. https://iang.org/papers/intersection_ricardian_smart.html (Abgerufen im Februar 2023).
- [99] Ian Grigg. The Ricardian Contract. https://iang.org/papers/ricardian_contract.html (Abgerufen im Februar 2023).
- [100] Ian Grigg. Financial cryptography in 7 layers. In *Proceedings of the 4th International Conference on Financial Cryptography, FC '00*, page 332–348, Berlin, Heidelberg, 2000. Springer-Verlag.
- [101] gRPC Authors. gRPC. <https://grpc.io/> (Abgerufen im Februar 2023).

- [102] The gRPC Authors. gRPC Python documentation. <https://grpc.github.io/grpc/python/grpc.html#grpc.server> (Abgerufen im Februar 2023).
- [103] E. Gyr. *Blockchain und Smart Contracts: die vertragsrechtlichen Implikationen einer neuen Technologie*. Rechtswissenschaftliche Fakultät der Universität Bern, 2019.
- [104] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. <https://www.r3.com/blog/corda-technical-whitepaper/> (Abgerufen im Februar 2023), 2019. Version 1.0.
- [105] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin’s Peer-to-Peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, Washington, D.C., August 2015. USENIX Association.
- [106] Gary Howland. Development of an Open and Flexible Payment System. <http://www.systemics.com/docs/sox/overview.html> (Abgerufen im Februar 2023), 1996.
- [107] Eric Hughes. A Cypherpunk’s Manifesto. <http://www.activism.net/cypherpunk/manifesto.html> (Abgerufen im Februar 2023), 1993.
- [108] Jonas Höchst, Lars Baumgärtner, Franz Kuntke, Alvar Penning, Artur Sterz, and Bernd Freisleben. LoRa-based Device-to-Device Smartphone Communication for Crisis Scenarios. In *17th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2020)*, Blacksburg, Virginia, USA, 2020.
- [109] Systemics Inc. Contracts. <http://webfunds.org/ricardo/contracts/> (Abgerufen im Februar 2023).

- [110] Systemics Inc. Implementations of Ricardian Contracts. http://www.webfunds.org/guide/ricardian_implementations.html (Abgerufen im Februar 2023).
- [111] Systemics Inc. Ricardian contracts. <http://www.webfunds.org/guide/ricardian.html> (Abgerufen im Februar 2023).
- [112] Systemics Inc. Ricardo - FAQ. http://www.systemics.com/docs/ricardo/issuer/faq_security.html#sec_crypto (Abgerufen im Februar 2023).
- [113] Systemics Inc. Ricardo by Systemics. <http://www.systemics.com/docs/ricardo/> (Abgerufen im Februar 2023).
- [114] Systemics Inc. The Ricardian Financial Instrument Contract. <http://www.systemics.com/docs/ricardo/issuer/contract.html> (Abgerufen im Februar 2023).
- [115] Systemics Inc. WebFunds. <https://webfunds.org/> (Abgerufen im Februar 2023).
- [116] James Henry Ellis. The possibility of non-secret digital encryption. CESG Research Report 3006, 1970.
- [117] Jens Schneider. Lars Creutz / Fides · GitLab. <https://gitlab.rlp.net/1.creutz/fides/-/tree/master/fides/external/grond> (Abgerufen im Februar 2023).
- [118] Burt Kaliski. Pkcs #7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. <http://www.rfc-editor.org/rfc/rfc2315.txt> (Abgerufen im Februar 2023).
- [119] Christian Karpfinger and Kurt Meyberg. *Zyklische Gruppen*, pages 59–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

- [120] Kevin Wagner. Webanwendung zur Inklusion und Verwaltung von LoRaWAN-Geräten in übergeordneten Anwendungen mit Hilfe von The Things Network. Masterthesis, Hochschule Trier, Umwelt-Campus Birkenfeld, 2022.
- [121] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016.
- [122] Kostas Christidis. Notes on protocol buffers and deterministic serialization (or lack thereof). <https://gist.github.com/kchristidis/39c8b310fd9da43d515c4394c3cd9510> (Abgerufen im Februar 2023).
- [123] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.
- [124] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.
- [125] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Report 2010/264, 2010. <https://ia.cr/2010/264> (Abgerufen im Februar 2023).
- [126] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, 2010.
- [127] Katharina Krombholz, Aljosha Judmayer, Matthias Gusenbauer, and Edgar Weippl. The other side of the coin: User experiences with bitcoin security and privacy. In *International conference on financial cryptography and data security*, pages 555–580. Springer, 2016.

- [128] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2), apr 2020.
- [129] Lars Creutz. Lars Creutz / Fides · GitLab. <https://gitlab.rlp.net/1.creutz/fides> (Abgerufen im Februar 2023).
- [130] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, February 1993.
- [131] LoRa Alliance. LoRaWAN Sicherheit. https://lora-alliance.org/wp-content/uploads/2020/11/la_whitepaper_security-german_0519.pdf (Abgerufen im Februar 2023).
- [132] LoRa Alliance, Inc. LoRaWAN 1.0.3 Regional Parameters. <https://lora-alliance.org/wp-content/uploads/2021/05/RP002-1.0.3-FINAL-1.pdf> (Abgerufen im Februar 2023).
- [133] Aspose Pty Ltd. INI - Initiation File Format. <https://docs.fileformat.com/system/ini/> (Abgerufen im Februar 2023).
- [134] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum’s peer-to-peer network. Cryptology ePrint Archive, Paper 2018/236, 2018. <https://eprint.iacr.org/2018/236>.
- [135] O. Markowitch, A. Bilas, J.H. Hoepman, C.J. Mitchell, and J.J. Quisquater. *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks: Third IFIP WG 11.2 International Workshop, WISTP 2009 Brussels, Belgium, September 1-4, 2009 Proceedings Proceedings*, page 117. LNCS sublibrary: Security and cryptology. Springer, 2009.

- [136] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- [137] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. SpringerLink: Springer e-Books. Springer Berlin Heidelberg, 2008.
- [138] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
- [139] Ralph Merkle. Protocols for Public Key Cryptosystems. pages 122–134, 04 1980.
- [140] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, apr 1978.
- [141] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.
- [142] National Institute of Standards and Technology. Data encryption standard (des). FIPS Publication 46-3, October 1999.
- [143] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.
- [144] Open Source Initiative. The MIT License. <https://opensource.org/licenses/MIT> (Abgerufen im Februar 2023).
- [145] OpenAssets. open-assets-protocol - GitHub. <https://github.com/OpenAssets/open-assets-protocol> (Abgerufen im Februar 2023).
- [146] Kenneth G. Paterson and Arnold K. L. Yau. Cryptography in theory and practice: The case of encryption in ipsec. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 12–29, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

- [147] Harry Thurston Peck. Harper’s Dictionary of Classical Literature and Antiquities. <http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.04.0062%3Aalphabetic+letter%3DF%3Aentry+group%3D3%3Aentry%3Dfides2-harpers> (Abgerufen im Februar 2023).
- [148] R3. Corda — Leading DLT Platform for Regulated Industries. <https://www.corda.net/> (Abgerufen im Februar 2023).
- [149] E. Regnath and S. Steinhorst. Smaconat: Smart contracts in natural language. In *2018 Forum on Specification Design Languages (FDL)*, pages 5–16, 2018.
- [150] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.
- [151] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.
- [152] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *International workshop on networked group communication*, pages 30–43. Springer, 2001.
- [153] Scott Chacon et al. Git - gitrevisions Documentation. [<https://git-scm.com/docs/gitrevisions>] (Abgerufen im Februar 2023).
- [154] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979.

- [155] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, page 149–160, New York, NY, USA, 2001. Association for Computing Machinery.
- [156] Nick Szabo. Smart contracts. <http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html>, 1994. (Abgerufen im Februar 2023).
- [157] Nick Szabo. Smart Contracts: Building Blocks for Digital Markets. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996. (Abgerufen im Februar 2023).
- [158] Nick Szabo. The Idea of Smart Contracts. <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html>, 1997. (Abgerufen im Februar 2023).
- [159] The gRPC Authors. grpcio-tools- PyPI. <https://pypi.org/project/grpcio-tools/> (Abgerufen im Februar 2023).
- [160] The Python Cryptographic Authority and individual contributors. cryptography - PyPI. <https://pypi.org/project/click/> (Abgerufen im Februar 2023).
- [161] The Python Software Foundation. PyPI - The Python Package Index. <https://pypi.org/> (Abgerufen im Februar 2023).

- [162] The Things Industries. Addressing & Activation. <https://www.thethingsnetwork.org/docs/lorawan/addressing/> (Abgerufen im Februar 2023).
- [163] The Things Industries. The Things Network. <https://www.thethingsnetwork.org/> (Abgerufen im Februar 2023).
- [164] The Things Network. Duty Cycle. <https://www.thethingsnetwork.org/docs/lorawan/duty-cycle/> (Abgerufen im Februar 2023).
- [165] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Comput. Surv.*, 43(2), feb 2011.
- [166] Serge Vaudenay. Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ... In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology, EUROCRYPT '02*, page 534–546, Berlin, Heidelberg, 2002. Springer-Verlag.
- [167] F Wittenberger. BALL - Askemos (Wayback Machine). <https://web.archive.org/web/20200222102234/http://ball.askemos.org/> (Abgerufen im Februar 2023).
- [168] F Wittenberger. Askemos-a distributed settlement. 2002.
- [169] Maximilian Wöhrer and Uwe Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.
- [170] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain. <https://solana.com/solana-whitepaper.pdf> (Abgerufen im Februar 2023). v0.8.13.

- [171] H. Zhang, Y. Wen, H. Xie, and N. Yu. *Distributed Hash Table: Theory, Platforms and Applications*. SpringerBriefs in Computer Science. Springer New York, 2013.
- [172] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.
- [173] Philip R Zimmermann. *The official PGP user's guide*. MIT press, 1995.