Secure and Privacy-preserving Decentralized Identities

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sebastian Friebe

Tag der mündlichen Prüfung: 23. Mai 2022

1. Referentin: Professorin Dr. Martina Zitterbart
   Karlsruher Institut für Technologie (KIT)
2. Referent: Professor Dr. Hannes Hartenstein
   Karlsruher Institut für Technologie (KIT)

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Telematik des Karlsruher Instituts für Technologie (KIT). An erster Stelle möchte ich daher Frau Prof. Dr. Martina Zitterbart danken, die als Doktormutter meine Promotion betreute, und mir auch eine Anstellung am Institut für Telematik ermöglichte. Durch ihre umfangreiche Betreuung und die zahllosen hilfreichen Diskussionen und Anregungen hat sie mich bei meinem Promotionsvorhaben sehr weitergebracht. Ebenso gilt mein Dank Herrn Prof. Dr. Hannes Hartenstein, der sich trotz zahlreicher weiterer Verpflichtungen in Forschung und Lehre sofort dazu bereit erklärt hat, das Korreferat für meine Promotion zu übernehmen und mir wegweisend zur Seite stand.

Mein Dank gilt auch meinen Kolleginnen und Kollegen, die mich während meiner Zeit am Institut für Telematik begleitet haben. Durch zahlreiche wertvolle Diskussionen und Hinweise haben auch sie zu dieser Arbeit beigetragen. Auch abseits der Forschung verdanke ich ihnen viele schöne Momente in meiner Zeit am Institut für Telematik. An dieser Stelle möchte ich mich insbesondere bei Hauke Heseding und Markus Jung bedanken, die mir gerade in der intensiven Zeit der Fertigstellung meiner Promotion trotz eigener Verpflichtungen mit konstruktivem Feedback und moralischer Unterstützung sehr weitergeholfen haben. Nicht zu vergessen ist auch Dr. Martin Florian, der meine Masterarbeit betreute und mir mit seinen Kenntnissen den Einstieg in den Forschungsbereich sehr vereinfachte. Mein Dank gilt auch den Sekretärinnen und dem technischen Personal des Instituts, denen ich eine produktive Arbeitsumgebung verdanke.

Für ihren Beitrag gilt mein Dank auch den Studierenden, die ich als Seminar-, Bachelor- und Masterarbeiter, sowie als wissenschaftliche Hilfskräfte, im Rahmen meiner Tätigkeit betreuten durfte. Im Gespräch mit ihnen sind neue Ideen entstanden, und sowohl die neuen als auch bestehende Ideen und Ansätze wurden reflektiert und weiterentwickelt.

Mein besonderer Dank gilt auch meiner Familie und meinen Freunden. Meine Eltern ermöglichten mir mein Studium, und damit letztlich auch die Promotion. Sowohl meine Familie als auch meine Freunde waren gerade in den anstrengenderen Phasen der Promotion eine wichtige Motivation und Stütze für mich.

Vielen Dank euch allen.

Karlsruhe, im Juli 2022

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 2
- 1.2 Research questions . . . 3
- 1.3 Main Contributions . . . 3
- 1.4 Outline . . . 7

2 Background **9**
- 2.1 Privacy . . . 9
- 2.2 Trust assumptions . . . 12
- 2.3 Cryptography . . . 13
  - 2.3.1 Hash functions . . . 13
  - 2.3.2 Symmetric encryption . . . 14
  - 2.3.3 Asymmetric encryption . . . 15
  - 2.3.4 Asymmetric signatures . . . 15
    - 2.3.4.1 Elliptic Curve Digital Signatures . . . 17
    - 2.3.4.2 Boneh-Lynn-Shacham signatures . . . 17
- 2.4 Blockchains . . . 18
  - 2.4.1 Basics . . . 18
    - 2.4.1.1 Bitcoin . . . 19
  - 2.4.2 Smart Contracts . . . 20
    - 2.4.2.1 Ethereum . . . 20
  - 2.4.3 Storing data in blockchains . . . 21
- 2.5 Online identity management . . . 21
  - 2.5.1 Current Situation . . . 21
  - 2.5.2 Decentralized trusted identities . . . 23
  - 2.5.3 Self-sovereign identities . . . 24
- 2.6 Online social networks and social graphs . . . 27
- 2.7 Sybil attacks . . . 28
- 2.8 Sybil defense . . . 29
  - 2.8.1 Sybil defense based on social graphs . . . 30
  - 2.8.2 Decentralized Sybil defense based on social graphs . . . 31

3 DecentID **33**
3.1 Adversary model . . . 34
3.2 Design Goals . . . 35
3.3 Example Scenario . . . 37
3.4 Important terms and variables . . . 41
3.4.1 Definitions . . . 41
3.4.2 Smart contracts and files . . . 42
3.4.3 Cryptographic keys . . . 42
3.5 Shared Identities . . . 43
3.5.1 Creating and sharing *SharedIdentityContract*s . . . 43
3.5.2 On-Chain Attributes . . . 46
3.5.3 Off-Chain Attributes . . . 50
3.5.4 Granting Attributes . . . 53
3.6 RootIdentityContract . . . 55
3.7 Replacing Cryptographic Keys . . . 56
3.7.1 Keys of permitted users . . . 56
3.7.2 Attribute encryption keys . . . 57
3.7.3 Attribute ownership . . . 58
3.7.4 Keys in the RootIdentityContract . . . 58
3.8 Implementation . . . 59
3.8.1 Demonstrator . . . 59
3.8.1.1 Architecture . . . 60
3.8.1.2 Identity management . . . 61
3.8.1.3 University . . . 62
3.8.1.4 Online shop . . . 63
3.8.1.5 Visualization . . . 64
3.8.2 Smartphone application . . . 65
3.8.2.1 Architecture . . . 65
3.8.2.2 Usage . . . 66
3.9 Evaluation . . . 67
3.9.1 Control over the identities . . . 68
3.9.1.1 Properties of smart contracts . . . 68
3.9.1.2 Authenticity and integrity . . . 69
3.9.1.3 Removals . . . 71
3.9.1.4 Comparison with the state of the art . . . 72
3.9.2 Privacy of identities . . . 74
3.9.2.1 Confidentiality of attributes . . . 74
3.9.2.2 Identifying users . . . 76

3.9.2.3 Passing on data . . . 78
3.9.2.4 Comparison with the state of the art . . . 78
3.9.3 Multiple pseudonymous identities . . . 80
3.9.3.1 Explicit linkability . . . 81
3.9.3.2 Implicit linkability . . . 81
3.9.3.3 Trade-offs . . . 82
3.9.3.4 Comparison with the state of the art . . . 83
3.10 Applicability . . . 85
3.10.1 End user interaction . . . 85
3.10.1.1 Privacy protection . . . 85
3.10.1.2 Interaction with online services . . . 85
3.10.2 Overheads . . . 86
3.10.2.1 Writing to the blockchain . . . 86
3.10.2.2 Reading from the blockchain . . . 86
3.10.2.3 Cryptography . . . 87
3.10.3 Cost considerations . . . 87
3.11 Summary . . . 88

4 Use Cases for DecentID **89**
4.1 Coupling with Palinodia . . . 90
4.1.1 Palinodia . . . 91
4.1.2 Coupling approaches . . . 92
4.1.3 Evaluating the approaches . . . 98
4.1.3.1 Security dependency . . . 98
4.1.3.2 Costs . . . 100
4.1.3.3 Implementation effort . . . 104
4.1.3.4 Interoperability . . . 105
4.1.4 Findings . . . 106
4.2 Voting integration . . . 107
4.2.1 Example Scenario . . . 107
4.2.2 Open Vote Network . . . 109
4.2.2.1 Ethereum implementation . . . 110
4.2.2.2 Adaptions . . . 112
4.2.3 Smart Contracts . . . 113
4.2.3.1 SharedIdentityContract . . . 113
4.2.3.2 VotingContract . . . 115
4.2.3.3 VotingDataContract . . . 116

4.2.4 Poll Execution 118
4.2.4.1 Preparing the poll 118
4.2.4.2 Registering 120
4.2.4.3 Starting the voting 122
4.2.4.4 Voting 123
4.2.4.5 Ending the poll 124
4.2.5 Evaluation 125
4.2.5.1 Security 125
4.2.5.2 Voting for off-chain attributes 126
4.2.5.3 Voting for encrypted attributes 126
4.2.5.4 Privacy analysis 127
4.3 Conclusion 129

5 Sybil defense **131**
5.1 Adversary model 132
5.2 Approach 133
5.3 Core components 134
5.3.1 Ticket Sources 134
5.3.2 Blockchain 136
5.3.3 Authorization Tickets 139
5.3.4 Round-based operation 140
5.4 Design 142
5.4.1 Basics and assumptions 142
5.4.2 Ticket creation 143
5.4.3 Round management 144
5.4.3.1 Counting active nodes 144
5.4.3.2 Selecting new ticket sources 145
5.4.3.3 End of round 148
5.4.4 Gaining authorizations 149
5.5 Further restrictions for Sybil identities 151
5.5.1 Aging Authorizations 152
5.5.2 Edge Values 153
5.6 Evaluation 155
5.6.1 Increase of Sybil strength 155
5.6.1.1 Evaluation environment 155
5.6.1.2 Aging authorizations 156
5.6.1.3 Edge values 161
5.6.2 Sybil controlled ticket sources 163

5.6.3 Overhead estimation . . . 165
5.6.4 Comparison with related work . . . 166
5.7 Integration into DecentID . . . 168
5.7.1 Assumptions . . . 168
5.7.1.1 Existing identities . . . 168
5.7.1.2 Trustworthy ticket sources . . . 168
5.7.1.3 Access to data and systems . . . 169
5.7.2 Requirements . . . 169
5.7.3 Approach . . . 170
5.7.3.1 Sending data to ticket sources . . . 170
5.7.3.2 Handling at ticket sources . . . 171
5.7.3.3 Assembling the link . . . 172
5.7.3.4 Verifying the link . . . 173
5.7.4 Identification of the service . . . 173
5.8 Conclusion . . . 174

6 Conclusion **175**
6.1 Results . . . 176
6.2 Perspectives for Future Work . . . 178

Appendices **179**

A Important terms **181**
A.1 Definitions . . . 181
A.2 Smart contracts and files . . . 182
A.3 Cryptographic keys . . . 183

B Smart Contracts of DecentID **185**
B.1 Mortal.sol . . . 185
B.2 RootIdentityContract.sol . . . 186
B.3 SharedIdentityContract.sol . . . 187
B.4 AttributeContract.sol . . . 191
B.5 VotingDataContract.sol . . . 192
B.6 VotingContract.sol . . . 196

Bibliography **201**

Chapter 1

Introduction

With the increased interactivity of current websites, more and more online services recommend or even require that their users create a user account. This has a variety of reasons, some of those are beneficial and transparent for the user: store user settings, rate posts, or write comments. Other reasons are less visible, but often more profitable for the company running the service: Tracking users and analyzing user behavior, which can be used to suggest interesting articles but also to display personalized advertisements.

This trend has some drawbacks for the users. For instance, they are forced to create multiple user accounts, of with each should be using unique credentials for authentication. However, the increasing number of accounts often leads to reusing the same credentials for multiple accounts. This introduces a security vulnerability, especially if the email address is used as account name and the same password is used for both the service account and the email account. Through the increasing support for Single-Sign-On (SSO) identity providers, e.g., offered by Google or Facebook, the credential management is simplified. With those, only a single user account has to be created and can be used at multiple online services.

A problem of centralized identity providers is that the privacy of the users can be compromised. Especially if a SSO identity provider is used, the identity provider can track which services are visited by the users. While users have the convenience of simplified logins, they pay for it with a loss of privacy. Additionally, the user might lose control over their personal data. They have to trust their identity providers to not modify or sell their data, but can only rarely check whether this is the case. In addition, the stored data poses a tempting target for attackers.

To avoid the drawbacks of centralized SSO providers, an alternative can be offered by decentralized identity providers. With those, trust into the company operating the centralized identity provider is replaced with trust into a verifiable system. This way, users can maintain self-sovereign identities, completely under their own control.

1.1 Problem statement

Important properties of all identity management systems are to ensure the privacy, authenticity, and availability of the users data. These systems store large quantities of personal, privacy-relevant data, to allow their users to display parts of the stored data towards online services when required. If the authenticity of the data is not ensured, services cannot trust the received data. When the system is unavailable, access to dependent services is no longer possible. These properties have to be ensured for decentralized identity management systems as well. Another property is that users should have full control about their pseudonymous online identities. This includes adding new attributes to an identity, modifying or deleting attributes, and sharing the identity, by allowing online services to access the identity or parts of it. For example, most services are permitted to retrieve the users email address to send notifications to the user, while the birthday of the user is rarely required and consequently should not be known by all services. With existing identity providers it only seems as if the user is in control. If a centralized identity provider wants to modify or share personal data, e.g., by selling the personal data to advertising agencies, it is able to do so. To ensure that the user is in control and to allow the user to maintain self-sovereign identities, the dependency on centralized trust anchors should be avoided.

A problem when permitting users to create and use pseudonymous identities for an online service is that this also permits Sybil attacks on the service. These attacks allow a single user, called a Sybil adversary, to create many digital identities in order to manipulate or take over a system. If the identities of the Sybil adversary outnumber the identities of honest users within a system, the adversary is able to subvert majority votes, e.g. by single-handedly deciding on the results of polls. As such, Sybil adversaries have to be prevented from creating a large number of Sybil identities. The easiest way to restrict Sybil identities is to force users to present unique personal identifiers towards the service, e.g., presenting their passport. To protect the privacy of the users and allow integration into decentralized systems, other approaches are preferable. While excluding Sybil adversaries, honest users should not be stopped from creating identities. To prove towards online services that an identity is no Sybil identity, a proof of “non-sybilness” should be generated.

1.2 Research questions

In this thesis an identity management system is designed that allows its users to maintain self-sovereign digital identities. To avoid that the system is abused by Sybil adversaries, a Sybil defense system is designed that works together with the identity management system to thwart the creation of Sybil identities. As such, this thesis addresses two major research questions:

(1) How can the creation and sharing of digital identities be designed for a decentralized identity provider, while providing privacy of the stored data. Access and modification of the stored data has to be designed without permitting unauthorized access to the data. Additionally, the system has to be trustworthy in the sense that modifications of and granting access permissions to the identities have to be accountable for and permitted by the user.

(2) How can such a system be secured against the creation of large numbers of Sybil identities. The presence of numerous Sybil identities would reduce the trust other services have in the system, impeding its acceptance and widespread deployment. At the same time, the usability of the system for honest users must be maintained.

1.3 Main Contributions

The approach to answering the research questions is threefold: the design and evaluation of a self-sovereign identity management system, an analysis of its interactions with other smart contracts, and the design and evaluation of an accompanying Sybil defense system. The designed solutions are based on existing infrastructure and, thus, can be readily deployed. The main contributions of this thesis are the following:

Self-sovereign identity management
To improve the control users have about their digital identities, the decentralized identity management system DecentID was designed [FSZ18]. An important part of that is to ensure the trust into DecentID, both from the perspectives of users and online services. Towards this goal, DecentID was designed based on smart contracts on the blockchain, with the blockchain acting as a decentralized trust anchor. Due to the used smart contracts, the user creating an identity is the only one able to permit other users or services to access it. Attributes, which can be stored on the blockchain or in off-chain storage, can be attached to the identities by permitted users. Additionally, certifying attributes can be granted to other users, i.e., claims

that can be verified by third parties. Adversary models are defined and used for evaluations and privacy analyses. It is analyzed which information is available to the adversaries and whether a breach of privacy occurred. Furthermore, practical considerations, e.g., costs of operation and computational overheads, are discussed.

Interactions between smart contracts
Currently, most deployed smart contracts implement all required functionalities, e.g., identity management, themselves, resulting in higher implementation efforts and possible introduction of additional security vulnerabilities. To improve interactions between smart contracts, a case study was conducted with DecentID being used as identity provider for another smart contract [Fri+21]. To solve the challenge of mismatching function interfaces, different coupling approaches were designed and evaluated. For the evaluations, multiple criteria were considered, e.g., security, implementation effort, and financial costs. The findings where that the approaches vary regarding their costs and capabilities, with the best approach depending on the specific use case. When writing attributes to DecentID identities, a major challenge is that the private key of a permitted user is required to do so. This was solved by instead accepting a verifiable majority decision of authorized voters, with the decision being ascertained by executing a poll on the blockchain. While not requiring a private key, the security of the system is still upheld. The findings of a privacy analysis indicated that, while the individual vote can be kept secret, the participation of a user in the poll is always visible on the blockchain.

Decentralized Sybil defense
Using the trust relationships of online social graphs is an established approach to defend against Sybil attacks. However, existing approaches are either unable to authorize many nodes as honest at once, or are unable to deal with frequently joining nodes. This thesis introduces a novel approach for decentralized Sybil defense, Detasyr [FMZ19]. Besides filling the aforementioned gap in functionality, Detasyr provides its users with proofs of authorization, which can be used in other contexts, e.g., in the identities of DecentID. This is not possible with existing approaches. Simulations were used to evaluate Detasyrs effectiveness when restricting Sybil adversaries. The results show that honest identities become authorized with only a small delay, while the Sybil adversary is bounded to an absolute number of identities. For example, when evaluating on a static graph consisting of 10,000 honest users, the Sybil adversary was only able to authorize around 15 Sybil identities, limiting its presence in the social graph to under 0.2 percent. An evaluation of the overhead for participating in Detasyr found that the number of exchanged network messages is mostly negligible, with an average of only three messages exchanged per minute.

In this thesis two systems, DecentID and Detasyr, are designed and evaluated. They work together to provide self-sovereign but Sybil-resistant digital identities.

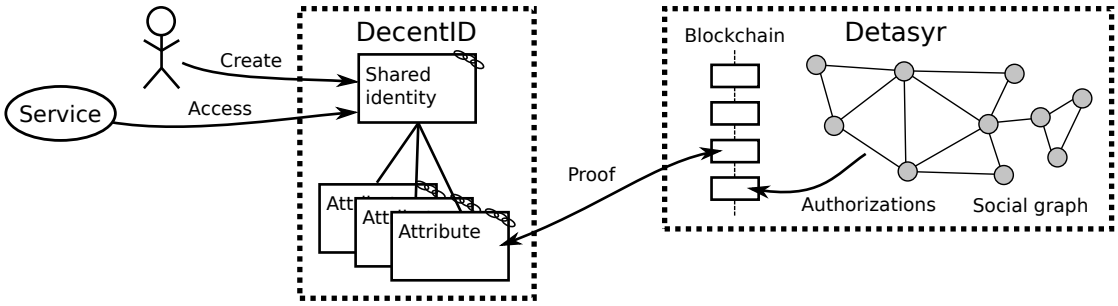

Figure 1.1: *DecentID and Detasyr and their interaction.*

DecentID, depicted in the middle of figure 1.1, can be used to create digital identities. Technically, DecentID is implemented as smart contracts on the blockchain Ethereum. The identities can have attached attributes which describe properties of the user, e.g., its username within an online service. These attributes can be stored either on the blockchain or in off-chain storage, to reduce the financial costs of maintaining the identity. Access to the identity can be granted to other users, which for example might represent online services the user interacts with. This service is then able to read the attached attributes and attach attributes of its own to the shared identity. The design of DecentID is described in detail in chapter 3.
Detasyr, depicted on the right in figure 1.1, is a Sybil defense system based on an online social graph. The graph represents trust relationships between users of the system. To get authorized as an honest user, i.e., as not being a Sybil identity, a user has to collect a certain number of authorization tickets that are flooded through the graph. After enough tickets have been collected by a node, the public key of this node is added to the next block of a blockchain that is part of Detasyr. Due to the structure of the social graph and the distribution strategy of the tickets, only few of the flooded authorization tickets reach the Sybil adversary, restricting the number of nodes it can authorize. Two extensions were evaluated to further restrict the number of Sybil identities: keeping authorizations valid only for a limited amount of time, and restricting the number of authorizations per neighbor. Detasyr and its extensions are explained in depth in chapter 5.
In principle, these two systems can operate independently of each other. However, since users are able to create digital identities by themselves, DecentID is in principle vulnerable to Sybil attacks. To restrict the creation of identities by Sybil adversaries, the two systems can be linked to each other. With Detasyr, a proof of authorization can be created by a user. Later on, the user can use this proof to convince others,

e.g., an online service they want to access, that their identity belongs to an honest user and not to a Sybil adversary. For this, a cryptographic link between the service and the proof of the user is established. This link is then stored in an attribute of the identity used for the service. Afterwards, the service is able to verify the validity of the link, without identifying which proof belongs to the user. Due to this unlinkability, the privacy of the user is protected, since colluding services are unable to determine whether two cryptographic links have been created by the same user. However, due to Detasyr, each user is only able to generate a single link per service, preventing Sybil attacks. Section 5.7 describes how this link has been designed.

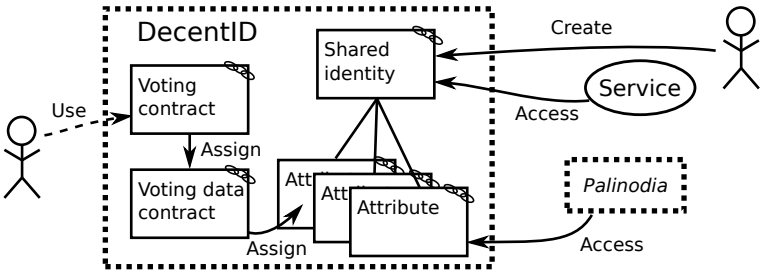

Figure 1.2: *DecentID and its use cases.*

Two use cases for DecentID where evaluated, as depicted in figure 1.2. For once, DecentID was used to replace the integrated identity management of the already existing smart contract-based system Palinodia. Instead of using the integrated special purpose identity management, the general purpose identities of DecentID were used to decide whether a user is permitted to access Palinodia. It was shown that the attributes attached to the identities of DecentID can be read and processed by other smart contracts, without depending on support by off-chain software. As part of this case study different coupling approaches were designed, to evaluate how DecentID can be accessed by other smart contracts. The coupling approaches and their evaluation are discussed in section 4.1.

Furthermore, to evaluate how other smart contracts can write attributes to DecentID, a voting system was integrated. It consists of two smart contracts: the voting contract contains the program code to execute the polls, while the voting data contract stores the temporary data for the current execution of one poll. To interact with DecentID, the used smart contracts have to read existing identity attributes to determine whether a user is permitted to vote, and add new attributes to existing identities. As was determined in this case study, writing attributes was not possible without modifying the smart contracts of DecentID. A detailed description of the voting integration is found in section 4.2.

The focus of this work is on the technical design of the systems, and explicitly does not investigate some considerations regarding their real-world deployment. No insurmountable obstacles to their practical use are known, but some aspects still have to be looked into. One aspect is the practical use of created identities by online services, including both the standardization of a format to store the attribute data, and also how users are presenting their identity towards the services. For Detasyr, the success in restricting Sybil adversaries depends on the assumption that users will not accept everyone as a friend. While this is a common assumption in related work, it does not apply for current online social networks.

1.4 Outline

Chapter 2 presents required basics for the rest of the thesis. The chapter provides definitions of privacy and trust, outlines the importance of them for the following chapters, and states the goals of adversaries attacking theses properties. Introductions into relevant cryptography schemes, the basics of blockchain networks, and to online identity management are given. After explaining the structure of online social graphs, the goals, abilities, and restrictions of Sybil adversaries are discussed.
The following three chapters present the design and technical details of DecentID and Detasyr. In chapter 3, DecentID is explained in depth. Following the definition of the considered adversaries, the design goals of DecentID are explained. Based on a presented example scenario, the design is explained in detail. For evaluating DecentID, an extensive analysis of its security and privacy is conducted and further practical considerations are outlined regarding its real-world use.
How DecentID can be used together with other blockchain-based systems is explored in chapter 4. Two use cases are explained, implemented, and evaluated. In the first use case, DecentID is used as an identity provider for another application based on smart contracts. The second use case considers the integration of a voting system into DecentID to evaluate how attributes can be written by other smart contracts.
Detasyr is presented in chapter 5. After discussing the adversary and its restrictions, the general approach of Detasyr and its core components are explained. Following, its design is discussed in detail. Based on the presented design, two extensions for further improved Sybil defense are presented and their advantages and disadvantages analyzed. After discussing the results of a simulation-based evaluation, the integration of the Detasyr authorizations into DecentID is explained.
A conclusion in chapter 6 summarizes the achieved results and outlines perspectives for further work.

Chapter 2

Background

In this chapter, a number of required fundamentals for the rest of the thesis are presented. The chapter starts with a definition of the concepts of privacy and trust. Afterwards, relevant cryptographic schemes and the used algorithms are introduced. The following introduction in blockchains contains an introduction into the smart contract technology used extensively for the design of DecentID. Different approaches for managing digital identities are presented afterwards. After defining online social networks, Sybil attacks are explained as they pose a frequent problem in these networks, before presenting approaches for Sybil defense.

2.1 Privacy

In the past years, privacy, and the related confidentiality of users data, has become increasingly important to users of online services. However, most people are unable to give a clear definition what "privacy" actually means, even though it has been declared a fundamental human right [HC48]. Different approaches try to specify what is meant by privacy, often by defining multiple aspects of it. Many definitions are based on legal considerations, for example [Dan+14]. In their paper [PH10], Pfitzmann and Hansen give definitions and explanations for various privacy related terms, e.g., anonymity and unlinkability. Additionally, they define some terms relating to digital identities. Some of these definitions, which are used in this thesis, are given in the following.

Anonymity A user is anonymous if it cannot be identified within a group of users, its anonymity set. Given a certain group of users, e.g., all users of an online service, anonymity means that it is impossible to differentiate between the users within this group. For example, if one user sends a message to the service, an adversary is unable to find out which of the users has send this message. Furthermore, anonymity means that an adversary is unable to find out which human controls a digital user identity.

Pseudonymity While anonymity is ideal to protect the privacy of the anonymous users, it is not practical for most use cases. For example, a two-way communication between a user and a service is not sensible possible if the service cannot identify the user. A pseudonym, which is an identity other than the real identity of the user, can help in this case. The service only sees the pseudonym of the user, but does not know the real identity of the user and should be unable to discover it given only the pseudonym. Pseudonymity is especially important in this work, where both the presented approaches DecentID and Detasyr are using pseudonyms to identify their users. Both approaches aim to hide the real identity of their users, while using only pseudonyms to address and interact between the participating users and services.

Unlinkability When two users are unlinkable, an adversary is unable to find out whether these users are related. Such a relationship might exist if two users know each other, but also when a single human controls both digital user identities. In the context of this work, unlinkability is important since it stops adversaries from tracking the actions of a user, e.g., which services are interacted with. If the same (potentially pseudonymous) identity is used for all services this might easily be possible, resulting in a requirement for multiple identities used by a single user.

To evaluate the privacy protection of a system, a list of seven privacy threats is given and discussed by the LINDDUN methodology [Den+11]. These privacy threats and subsequent protection goals were developed based on the definitions of Pfitzmann and Hansen. While not all of these threats are completely applicable to this work, they still offer a useful approach to evaluate the privacy of the presented system. The seven privacy threats and the resulting protection goals are listed in the following:

Linkability As the privacy threat to the protection goal of unlinkability explained above, it means that an adversary is able to distinguish whether two entities in the system are related with each other. Considering the case of identity management in this work, this means that an adversary would be able to discover that two digital identities are controlled by the same human user. However, it also can mean that the adversary discovers that two users are friends of each other or that two identity attributes have been created by or belong to the same user.

Identifiability If a digital identity is identifiable, that means that an adversary is able to find out which human user controls the identity. In a broader sense, identifiability can also mean that the user creating an attribute or message can be discovered. As such, it is an attack on the anonymity or pseudonymity provided by a system.

Non-repudiation This allows an adversary to prove that a user knows or has done something. As such, the relevant protection goal is plausible deniability. While not overly relevant for this thesis, this threat is often considered in communication privacy, where an adversary should be unable to prove anything about the contents of the communication towards third parties.

Detectability An adversary is able to detect whether a message, identity, or other data object exists, when it is possible to differentiate between it and random data. While there are approaches for data storage which ensure the protection goal of undetectability, communication over the Internet is difficult to hide. Related to unlinkability and pseudonymity, it is part of the goals of this thesis to protect the digital identities of a user from detection by adversaries.

Information Disclosure One of the more important points of this thesis, information disclosure means that an adversary is able to access information it is not permitted to access. Its protection goal, confidentiality, ensures that the contents of the protected data cannot be read by an adversary. While undetectability stops the adversary from even finding out about the existence of the data, confidentiality allows the adversary to notice its existence, but stops it from reading it.

Content Unawareness With some systems, the user might be unaware of the data they disclose to the system. This might be because they disclose more data than they need to, or that the system is able to discover this data due to wrong decisions of the user. Its protection goal, transparency, ensures that the user is able to find out what the system knows about them and what is done with this data. While this is an important factor when interacting with systems on the Internet, where data is kept potentially forever, it is not considered in this thesis in detail. The identities created with DecentID as well as their data are under the control of the user. However, how services on the Internet handle this data cannot be solved in this thesis.

Policy and consent Noncompliance While a system might claim to adhere to privacy policies, it might still ignore them and abuse the given data. In the case of DecentID itself, the correct handling of the data can be verified since the smart contracts forming the identity can be reviewed. What the services accessing the identities are doing with the data is out of scope for this thesis.

2.2 Trust assumptions

In this thesis, assumed trust can be considered contrary to the assumed adversaries. Different trust relationships are assumed between the participating users. In general, if one user trusts another user, it assumes that the other user will not attack them or the used system. Such an attack can mean unauthorized access to the computer of the user, which is out of scope for this thesis, but also an attempt to maliciously collect more data about the user than the user offered by themselves. This can be the case if data can be accessed by an adversary that should not be possible to access for them, but also when multiple adversaries cooperate and exchange the data each of them was granted access to.

Both approaches presented in this work, DecentID and Detasyr, are based on decentralized networks. These networks are operated by many users, which for the most part do not know each other directly. Still, it is assumed that the majority of the participating users are honest, i.e., are no adversaries, and have no intention of obstructing the operation of the system. As a consequence, it can be assumed that the results of all majority decisions within the system are correct and have been reached according to the rules of the system. Also, when a random user within the system is selected, it will more likely be an honest user than an adversary. These points are especially important for Detasyr, where random nodes are selected as trusted nodes. Based on the trust assumptions, most or all of them will execute the approach correctly, resulting in a trustworthy computation result. When two users are neighbors in the online social graph that is used by Detasyr, it is assumed that these users trust each other.

In DecentID, the user shares a digital identity with online services. Different from the trust assumptions described above, a service is not assumed to be fully trustworthy. While it is assumed that a service will not attack the user, they are considered to be an honest-but-curious adversary. As such, they will try to find out as much about the user as possible, but have no intention of harming the user, e.g., by attacking their computer. A reason for this can be that they, e.g., want to present personalized advertisement to the user. While not directly harmful for the user, the required data collection is an infringement of the users privacy. Consequently, it is assumed that the user does not completely trust the service and wants to restrict the data the service can collect. When the user interacts with multiple services, a unique partial identity is created by the user for each service. It is assumed that services can collude with each other to exchange the collected data about a user. Thus, keeping the partial identities of the user separated and unlinkable is important, since the service cannot be trusted to keep the granted data solely for itself.

2.3 Cryptography

Both approaches presented in this work make heavy use of cryptography. Hash functions are used to shorten inputs for more efficient encryption or for obfuscation, symmetric and asymmetric encryption is used for protecting data, and digital signatures are used to prove the ownership or validity of data.
This section gives a short introduction in relevant cryptography schemes, with a focus on what the algorithms can be used for. Knowledge of the mathematical functioning of the used algorithms is not required for this thesis and is therefore mostly omitted in this section. It should be noted that a few concrete algorithms are mentioned in the following as examples for their respective type of cryptographic algorithm. While they would be good candidates to use in an implementation of the approaches presented in this thesis, other secure algorithms could be used as well. Exceptions where a specific algorithm should be used or additional requirements are present are stated in the respective sections in chapter 3 and chapter 5.

2.3.1 Hash functions

A hash function is a function that reduces an input string of arbitrary length to a string of fixed length. Hashes have a number of use cases, for example they can be used to verify the integrity of a larger input efficiently or obfuscate an input. The calculation of the hash value is a quite fast operation for many hash algorithms. As such, they can be used to verify that the hash value stored at a trustworthy storage matches the hash value calculated from data in an untrusted storage.
For cryptographic hash functions, three properties are required [Alf96]. Given is a hash function H, inputs x and x', and outputs of the hash function y and y'.

(1) **Preimage resistance:** For any given output y, it is computational infeasible to find an input x so that $H(x) = y$. This means that for a known output value no matching input value can easily be reconstructed.

(2) **2nd-preimage resistance:** Given an input x, it is computational infeasible to find an input x' so that $H(x) = H(x')$.

(3) **Collision resistance:** It is computational infeasible to find inputs x and x' so that $H(x) = H(x')$. Here, both inputs can be selected arbitrarily. Consequently, it is very hard to find two input values that result in the same hash value.

Property 1 ensures that the hash function works as a one-way function. Due to that, the hash function can be used to obfuscate the input string since it is infeasible to reconstruct the input from the output.

To protect the integrity of data in publicly accessible storage, property 2 is important. It makes sure that an adversary is unable to present some manipulated data to another user for which the same trusted hash value would be calculated. If such a manipulation would be possible, the user would be unable to decide which data is the unmodified original data.
While generally important for the security of hash functions, property 3 is not explicitly required in this work. The assumed adversaries are not able to provide arbitrary data to the user, so colliding input data for the hash function cannot be introduced to attack the user.
On the blockchain Ethereum, the hash algorithm KECCAK-256 [GA11] is used. For compatibility and availability reasons, this hash algorithm is assumed to be used in this thesis as well.

2.3.2 Symmetric encryption

To protect the confidentiality of data, it can be encrypted. When the data is encrypted, it is indistinguishable from random data. Only entities knowing the secret key required to decrypt the data are able to find out the encrypted contents.
When using symmetric encryption, only one key is used for both encryption and decryption. Given a secret symmetric encryption key k, encrypting a message m works as:

$$c = enc_k(m) \tag{2.1}$$

The encrypted message c is then transmitted to the other party. Knowing the same key k and having received c, the message m can be recovered by decrypting c:

$$m = dec_k(c) \tag{2.2}$$

A consequence of using the same key for encryption and decryption is that both communicating parties require access to the same cryptographic key. While one party can generate the key, transmitting it to the other party can be difficult since no one except the expected receiver is permitted to receive the key.
A currently state-of-the-art symmetric encryption algorithm is the Advanced Encryption Standard (AES) [DR13]. Designed as Rijndael by Vincent Rijmen and Joan Daemen, it was selected by the U.S. National Institute of Standards and Technology (NIST) for protecting secret data. It is able to operate with key lengths of 128, 192, or 256 bits. As an efficient algorithm in both software and hardware, it is widely used for symmetric encryption of large amounts of data.

2.3.3 Asymmetric encryption

To avoid the need to transmit the symmetric encryption key to the receiver, asymmetric encryption algorithms were developed. In those, two different keys are used: a public key that is used for encrypting a message, and a private key that is used to decrypt the encrypted message. The private key is known only by the receiver of the message, while the public key can be known by everyone.
The first asymmetric encryption algorithm was RSA [RSA78], its basic idea is described in the following. The security of RSA is based on the difficulty of factoring the product n of two large prime numbers, which is computational infeasible for large enough values. The length of n is the key size of the algorithm and is between 2048 to 4096 bits. An asymmetric key pair based on n consists of two values (e, d), where e is public and used for encrypting the message while d is the private key to decrypt a message. To encrypt a message m, the sender calculates:

$$c \equiv m^e \pmod{n} \tag{2.3}$$

c is then transmitted to the creator of the key pair, where it can be decrypted using the private key d:

$$c^d \equiv (m^e)^d \equiv m \pmod{n} \tag{2.4}$$

While this approach protects the confidentiality of the message and avoids the problem of sharing a symmetric key, it still contains problems for a practical use: the required key size is much larger than for symmetric keys, asymmetric algorithms are quite slow compared with symmetric ones, and the sender of a message somehow has to ensure that the used public key really belongs to the intended recipient. While these problems can be solved, discussing the solutions is out of scope and not required for this document.
One way to reduce the required key length are calculations on elliptic curves. An asymmetric encryption scheme based on elliptic curves is the Elliptic Curve Integrated Encryption Scheme [Sho01]. Since elliptic curve cryptography is used by Ethereum, it is a good candidate for efficient asymmetric encryption in this thesis.

2.3.4 Asymmetric signatures

While encryption allows arbitrary senders to send confidential message to a receiver, the receiver cannot be sure who send the messages and whether they arrived unmodified. Since the public key used for encryption is publicly known, everyone is able to send an encrypted message to the owner of the public key. While the contained data

in encrypted messages cannot be read, the encrypted data can still be modified, both accidentally or maliciously. Digital signatures can be used to protect the integrity of the message, i.e., avoiding unperceived unauthorized modification, and to ensure the authenticity of the sender.
Using RSA as an signature scheme, asymmetric digital signatures work similar to encryption, but requires the sender, instead of the receiver, to posses an asymmetric key pair. Given the asymmetric key pair (e, d), a digital signature can be calculated by using the private key d:

$$h = H(m) \tag{2.5}$$

$$s = h^d \tag{2.6}$$

First, a hash h of the to-be-signed message m is calculated, as depicted in equation (2.5). Computing the signature of the hash is more efficient than signing a long message directly. The receiver of the message is able to calculate the hash of the received message by themselves and can compare the locally calculated hash value with the received hash value. Afterwards, the signature s over the hash value h is calculated the same way as if a message would be encrypted, but using the private key d instead of the public key e. This can be seen in equation (2.6), compared to the asymmetric encryption in equation (2.3).
Since the private key is used to create the signature, only the owner of the private key d is able to create a signature that can be verified with the known public key e. As such, digital signatures can also be used to prove the knowledge of a matching private key to a known public key.

$$s^e = (h^d)^e = h^{de} = (h^d)^e \equiv h \pmod{n} \tag{2.7}$$

$$h \stackrel{?}{=} h' = H(m) \tag{2.8}$$

To verify the received signature, the receiver can “decrypt” the received signature s, as displayed in equation (2.7). If the signature is valid, the result should match the locally calculated hash value h' over the received message m. If the transmitted signature s or the message m have been modified or the sender does not own a valid private key d for the public key e, h and h' will not match. In that case, the received message should be discarded and requested again from the sender.
In this work two concrete algorithms for digital signatures are used: Elliptic Curve Digital signatures, and Boneh-Lynn-Shacham signatures. In the following, these two signature schemes will be introduced.

2.3.4.1 Elliptic Curve Digital Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] is a variant of the Digital Signature Algorithm (DSA) [KSD13]. Using elliptic curves allows it to require a much shorter key length than the original DSA algorithm requires. For the same security level of 80 bits, DSA requires a key length of at least 1024 bits while ECDSA only requires a key length of 80 bits. Additionally, given the message and a signature over it, ECDSA allows the receiver to derive possible public keys for the signer from it [Bro09]. However, this also works if the signature is invalid, resulting in an invalid public key, and consequently of a seemingly correct, but actually broken verification of the signature. As such, the derivation of the public key can only be done if at least a hash of the correct public key is known to the receiver, so the validity of the public key can be ensured.
For this work, the ECDSA algorithm is important since it is used by the blockchain Ethereum to sign the transactions send to the miners of the blockchain. As such, all Ethereum clients are already able to calculate and verify these signatures.

2.3.4.2 Boneh-Lynn-Shacham signatures

Boneh-Lynn-Shacham (BLS) signatures are based on bilinear pairings to provide shorter signatures than existing approaches, while providing the same security [BLS04]. As such, they are interesting to use when many signatures have to be stored for a long time, reducing the amount of storage required. Also, using BLS signatures reduces the amount of data that has to be transmitted over the network when many signatures have to be send between many users.
An additional feature of BLS signatures is that their signatures can be aggregated [Bon+03]. Given a number of signatures σ_i and the respective messages m_i, the signatures can be combined to form a single signature $\sigma = \prod \sigma_i$, again reducing the amount of storage required. This signature σ can then be used to verify the integrity of the messages m_i when the matching public keys $pubKey_i$ and messages m_i are known.
While the messages have to be distinct for the aggregation to be secure, this does not pose a restriction [Bon+03]. The individual signers i of a shared message m can each prepend their respective public keys $pubKey_i$ in front of the message m before calculating their signatures σ_i over their modified messages $m_i = pubKey_i|m$. Since the public keys $pubKey_i$ have to be known to verify the aggregated signature σ anyway, this does not result in an additional storage requirement. It is sufficient to store the aggregated signature σ, the individual public keys $pubKey_i$, and the shared message m. To verify the signature, the distinct messages m_i can be reconstructed.

2.4 Blockchains

Simplified, a blockchain can be described as a distributed, append-only ledger. This means that the participants of the system can only append new data blocks to a blockchain, but are not able to modify already existing blocks. Since the blockchain is public and distributed between different systems, all participants can verify the correct operation of the blockchain. Modifying the blockchain, i.e., creating a new block, requires a majority decision of the participants. Together with the assumption that the majority of participants are honest, the blockchain can be considered a trustworthy data store.
The verifiability and trustworthiness of blockchains make them an interesting basis for numerous applications. The best known one is the blockchain Bitcoin (see section 2.4.1.1), where a blockchain is used to record financial transactions. Here the blockchain works as a decentralized trust anchor, making a traditional centralized bank obsolete. The blockchain Ethereum (see section 2.4.2.1) extends the included scripting language, making it possible to execute distributed computations.

2.4.1 Basics

The data structure of a blockchain consists of a chain of data blocks. This chain is known to all participating systems. Each participant is able to verify the correct construction of the blockchain, ensuring that invalid data blocks are not distributed within the system.

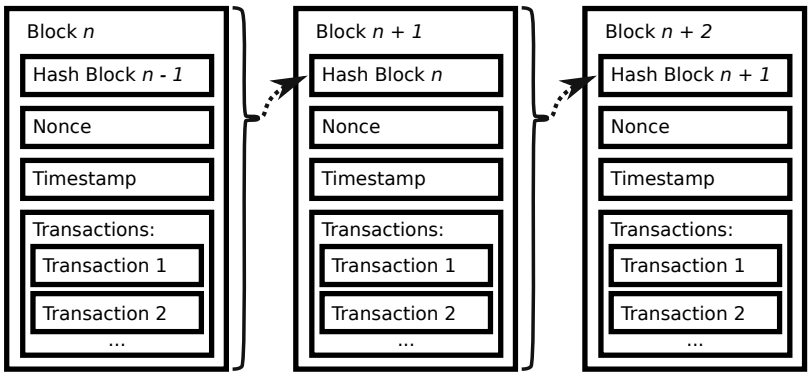

Figure 2.1: *The data structure of a blockchain (simplified).*

The structure of these data blocks is depicted in figure 2.1. To ensure the integrity and the correct linkage of the data blocks, each block contains a cryptographic hash of the previous block in the chain. The payload data of these blocks is the list of transactions.

These transactions contain the use case specific actions of the participants. For a financial blockchain, the transactions are in most cases financial transactions, i.e., orders to send money from one account within the blockchain to another one. To avoid that money is stolen, the transactions require a proof of ownership. The digital money is linked to a cryptographic public key. To prove the ownership of the money, access to the matching private key is required.
In public blockchain networks like Bitcoin, all participants are permitted to create new blocks. To avoid that many conflicting blocks are created at the same time, a *proof of work* is employed. To create a new block, a participant has to spend some amount of work, i.e., computation power. The amount of work spend is measured by calculating the hash of the newly created block and interpreting it as a number. Only if this number is smaller than a globally known threshold the new block will be accepted by the other participants. If multiple blocks fulfill this requirement, the block with the smallest hash value will be used. To be able to create multiple different hashes for a block containing the same data, the *nonce* value within the block is incremented. The advantage of this kind of proof of work is that it is quite hard to create a valid proof (since many different nonces have to be tested, requiring a lot of computational power) but it is simple to verify a finished proof (calculating a single hash about the block is computational cheap).

2.4.1.1 Bitcoin

The first widespread known blockchain was Bitcoin [Nak08], which was presented in 2008 by a person calling itself Satoshi Nakamoto. Currently, Bitcoin is the most successful blockchain and also the blockchain with by far the most valuable currency [1].
The primary purpose of Bitcoin are financial transactions, meaning that transactions can be used to send digital money between the different users of the currency. Apart from simple transactions, small transaction scripts can be added. These can be used to provide specialized requirements for spending the money, e.g., it can only be used if two users both allow it. Additionally, this scripts can be used to store small amounts of data within the blockchain.
Since it was the first blockchain, a lot of research has been done on top of it. This includes improving the bitcoin protocol, improved privacy protection, and using the blockchain for other purposes. The latter area of research often uses the integrity and trust properties as well as the storage capabilities of the blockchain to execute other protocols on top of Bitcoin.

[1] https://coinmarketcap.com/ Accessed: 11.03.2022

2.4.2 Smart Contracts

While the transaction scripts in Bitcoin allow to include some basic intelligence into the blockchain, they are still pretty limited. To improve the scripting abilities and perform more advanced computations within the blockchain, the ability to write *smart contracts* was developed. These can allow Turing complete programming within the blockchain, making it possible for all participants to verify the correct execution of the programs. Compared to traditional online services, this approach can increase the trustworthiness of the system for its users.
The smart contracts can generally execute any calculations normal computer programs can execute as well. Also, they are able to store arbitrary data within the blockchain. Interactions between the smart contracts are possible as well, both for calling functions of other smart contracts or for accessing the data stored by them. If the blockchain offers financial payments the smart contracts can also be used to transfer money, either to another smart contract or to an user. However, there are also some significant restriction compared to traditional computer programs. Since all computation and data processing has to be done within the blockchain, no external data sources can be used directly. If external data is required, trustworthy smart contracts can be used that receive their data through user input. A problem especially in the context of this work is the missing data privacy when working with smart contracts. Since all function calls and all data is, and has to be, publicly visible, the protection of users data has to be ensured before passing the data to the blockchain.
Even more so than in traditional programs, the security of the smart contract code is an issue. Everyone can analyze the deployed contract in a public blockchain and send arbitrary commands to it. Additionally, the smart contract is permanently available in the blockchain and normally it is not possible to replace it with a newer, secure, version. If a vulnerable contract is found by an adversary, not only the data stored within the smart contract can be overwritten, but also huge financial loss can occur.

2.4.2.1 Ethereum

The most popular blockchain which has been explicitly designed for the use of smart contracts is Ethereum [But+13], proposed in 2013. After Bitcoin, it is the most successful blockchain currently active. The currency used within the blockchain is called *Ether*. The mostly used scripting language within Ethereum is called *Solidity*. Solidity allows to write Turing complete smart contracts as described above.
A restriction to the programming and executing of the smart contracts is present due to their costs. Publishing a new smart contract or calling an existing one costs Ether, promoting the usage of short and efficient smart contracts.

2.4.3 Storing data in blockchains

Some blockchains have been specifically designed to handle the storage of data. An example for this is the blockchain *Namecoin*[2], which has been designed to offer an alternative to the hierarchical domain name system (DNS). Compared with other decentralized storage systems, e.g., distributed hash tables (DHTs), these blockchains can offer additional features. While the integrity of data can already be ensured in DHTs, blockchains can combine the storage of data with smart contracts and payments. For example, in the Namecoin network the registration of domain names has to be paid for. This registration expires after some time, forcing the owner of the domain to regularly confirm its interest on the domain. At the same time, the blockchain remains a distributed system which requires no single centralized trust anchor for its operation.

2.5 Online identity management

Over the last decades, online identity management has slowly evolved. While initially every online service had its own identity management, it is today often possible to use a single account for multiple services. In this section, the existing identity management approaches are discussed, as well as modern approaches which are based on decentralized systems, e.g., on blockchains.

2.5.1 Current Situation

When the need for online identities initially occurred, most online services implemented their own identity management. For the user this meant that they had to register at each service anew. Consequently, the user has to deal with numerous different logins, which is especially cumbersome if multiple Internet-capable devices are utilized by the user. Apart from the overhead of entering the user data over and over again, this also led to security problems. Regularly, another data breach of an online service is reported, where the database with the users credentials and their other data is stolen. For the ease of use, many users are using the same credentials (i.e., an email address as username and a password) over and over again. If the user database of one service is stolen, the attackers are able to access other services used by the users as well. This is especially a problem when the same password is used for securing the email account, with the email address often being used as a password recovery address at further services.

[2]https://www.namecoin.org/ Accessed: 11.03.2022

In the last years, various Single-Sign-On (SSO) providers were established. With these, a user creates only a single identity at the SSO provider. Afterwards, this identity can be used to login at other services as well. Common examples are buttons as “Login with Google” or “Login with Facebook” as alternatives to creating a local user account for one service. When the button is clicked, the user is forwarded to the respective SSO provider, can enter their credentials there, and the visited online service receives the result of a successful login attempt and can continue. Technically, this is based on the decentralized authentication protocol OpenID[3]. For the user, this simplifies their online experience since they only have to take care of a single online identity. However, the SSO provider becomes a single point of failure, a tempting target for attackers, and a potential privacy concern for the user. For once, the SSO provider stores the data of the user. Also, since the online activities of the user refer to the SSO provider for login, it is able to track the activities of the user. As such, the user has to trust its SSO provider to not abuse the collected data. If malicious, the SSO provider could even block the user from accessing services, by denying the login request of an online service the user wants to access.
A blockchain-based SSO approach is 3BI-ECC [Mal+21], where three specialized blockchains are used together to ensure the authenticity and integrity of the users identities, store the identities and their data, and maintain a directory of revoked identities. While blockchains are used, access to these blockchains, and consequently the identities, is relayed through a logically centralized server. Since the users have no direct access to their identities, the system is in practice a centralized approach under control of a single entity.
An improvement over logically centralized SSO providers is the use of federated identity providers. Instead of a single (logically) centralized provider, a number of independent identity providers are offering identity management for the user. The user is able to select any of the identity providers within the federation to manage their identity data, which is especially useful when no single identity provider is trusted by all users. An example for such a federated identity provider is the Shibboleth[4] system used between universities. In that case, the respective university operates as the users identity provider. When logging into some online service that is part of the federation, the login request is forwarded to the selected identity provider, e.g., the users university, by using a standardized protocol. At the website of their trusted identity provider the user can enter their credentials, without the forwarding online service learning them. This way users have the convenience of using Single-Sign-On but can still choose an identity provider they trust or are affiliated with.

[3] https://openid.net/ Accessed: 11.03.2022
[4] https://www.shibboleth.net/ Accessed: 11.03.2022

In [MDS19] a federated identity management based on the blockchain Ethereum is presented. Users create their identities on the blockchain themselves. Afterwards, participating authorities are able to add attributes to these identities. One goal of the approach is to reduce the required communication when authenticating. This is achieved by accessing local copies of the blockchain to verify the identity of a user. When attributes of the identity are required, the users sends these attributes directly to the service requesting them. While this increases the control the user has about their identities, it also means that the user has to directly interact with the service for all accesses to their identity.

2.5.2 Decentralized trusted identities

While the concept of federated identity providers is able to solve the issues with being a single point of failure and not being trustworthy for all participants, the privacy issue is not solved. The user has no direct control over the stored identity data while a dishonest identity provider can modify or share the stored data in any way it wants. A possible solution to this problem can be a decentralized system where users are not required to place trust into a single institution. Instead, they have to trust a (potentially verifiable) decentralized system, that is operated by multiple users or institutions. In those systems the data of the user is distributed over multiple nodes. While these nodes operate the system, they are individually unable to access or manipulate the stored user data. Only the creator of an identity is able to modify its own data since modifications by others are not accepted by the nodes of the decentralized system.
In the last years, a number of decentralized identity management systems based on blockchains have been designed. The advantage of using blockchains over other decentralized storage systems is that they act as a decentralized trust anchor. In centralized systems, some institution acts as a trust anchor and is responsible for the integrity and security of the data storage. They ensure that no unauthorized manipulation of the stored data occurs. In decentralized systems, this trust must be moved from the institutions to the used algorithms. Blockchains, with their distributed consensus algorithms, are a good candidate for this [DP18]. Consequently, many decentralized identity management systems use blockchains as a verifiable and trustworthy data storage.
Still, not all of systems using blockchains are truly decentralized. Even when they use blockchains for parts of their functionality, some of them still use a centralized server to provide parts of their service. One such system is ShoCard [SS16]. The aim of ShoCard is to provide users with digital identities, that are connected to

their offline identities. When creating a ShoCard identity, a user scans some personal identification document, e.g., their passport, and creates an asymmetric cryptographic key pair. A hash of the scan is signed and published on the blockchain of Bitcoin. The resulting transaction number is considered the ShoCard identity. The user is later on able to prove the creation of the Bitcoin transaction towards verification services. These services are then able to grant attributes to the user, which in turn can be presented towards other services. However, only the hash of the attribute is stored on the decentralized blockchain. Its data is stored centrally on ShoCards servers. While the confidentiality of the data is secured by encryption, the reliance on the servers forms a dependency towards a centralized entity. If the servers fail or their owners are no longer willing to maintain them, the generated identity is no longer of any use.
Another approach, Tawki [WGK19], is not primarily designed as an identity management system, but still allows its users decentralized identity management. It is an architecture for social communication, where the users can create their own identities and profile pages and present these profile pages to other users. Still, an important part of this is that users control their own identities. The data of their identities and profile pages is stored on user-selected backend servers, where each user can select a server they trust. To ensure the authenticity and integrity of the approach, entries in the blockchain Ethereum are used to provide a mapping of user names to the addresses of the backend servers.

2.5.3 Self-sovereign identities

Having self-sovereign identities means that each user creates and manages their own digital identities. To define this term, ten principles for were defined, which are summarized in the following [All16].

(1) **Existence** A user exists, which is digitally represented by their digital identity.

(2) **Control** Users are in control of their digital identity, including creating, updating, and deleting it. Publicly accepted algorithms ensure that no other users can manipulate their identity.

(3) **Access** Users have access to their own data, and no parts of their identity are hidden from them.

(4) **Transparency** The systems and algorithms used should be publicly known and their management and operation be accountable.

(5) **Persistence** The identity exists as long as the user wants it to. While attributes or cryptographic keys of the identity might be exchanged over time, the identity consisting of a set of attributes should stay available.

(6) **Portability** Identities should be transportable, which means that they are independent of a single service hosting them, or the current living environment of the user.

(7) **Interoperability** The identities should be as widely usable as possible, i.e., they should not be restricted to only a few services.

(8) **Consent** While sharing it is the purpose of the created identity, this should not be possible without the user explicitly permitting it to be shared.

(9) **Minimalization** Only the required amount of data should be disclosed when presenting identities. As an example, it might be sufficient to inform the service about the country the user is in, instead of giving the exact street address.

(10) **Protection** The rights of the user should be protected, also against the service using the identity.

To improve the control users have over their identities and and provide self-sovereign identities, completely decentralized identity management systems based on blockchains were designed. Still, depending on the kind of blockchain that is used, a certain centralization might still exist. One example for this is Sovrin [WR18], which is based on a permissioned blockchain. The advantages of using a permissioned blockchain are that they can add new blocks much faster, since no proof of work is required, and that no payment is required for modifying the data stored on it. Reading data from the blockchain is still possible for all users. However, it also brings some disadvantages. For once, since it is permissioned only a limited and known number of entities are able to append new blocks to it. Depending on how many entities are able to write blocks, the user is forced to trust a potentially small number of institutions, instead of the more lenient assumption that the majority of a large number of institutions is honest. Also, the institutions permitted to write to the blockchain have to be previously selected, in the case of Sovrin by a single trusted instance in form of the Sovrin Foundation. Based on this blockchain, users of Sovrin can create decentralized identities for themselves and attach attributes to them. The attributes can either be stored on the blockchain, on the smartphone of the user, or at trusted agents that are part of Sovrin. Attributes can be selectively disclosed by the user towards services.

A further step towards decentralization and self-sovereign identities is done by uPort [Lun+17]. Different from Sovrin, uPort is based on the public blockchain Ethereum and the distributed file system IPFS for attribute storage [Ben14]. Being based on these two systems, no single institution is required as trust anchor.
In uPort, the identities of users are represented by *proxy contracts* on Ethereum. These smart contracts are intentionally kept simple since their primary purpose is to provide a fixed blockchain address as identity for their creator. As such, they only provide the functionality to forward transaction send by their creator. On Ethereum, it makes no difference whether a transaction is send by a user, represented by a public key, from outside the blockchain, or send by a smart contract, represented by their blockchain address. Consequently, the called contract handles the transaction as if the user would have send it directly, but considers the proxy contract the source of the transaction. To add attributes to the identity, a single logically centralized smart contract is used for all users for uPort. For each added proxy contract, a hash is stored to lookup the attributes in the IPFS. Similar to Sovrin, users of uPort can register a number of friends within the smart contract forming their identity. If their cryptographic private key required to access the smart contract gets lost, their friends can restore the access by assigning a new key to the contract.
In [SP18] another blockchain-based approach for self-sovereign identities is presented. Different from uPort, where users can create as many identities as they want, only a single identity per user is permitted in this approach. This is intentional, since the identities are supposed to be used as a digital passport. Consequently, the digital identities created within the system are linked to government issued identities.
Two projects that provide a single decentralized identity to their users are Civic[5] and SelfKey[6]. Both aim to provide a single digital identity to use with online services. Attributes are stored on the users personal devices, and can be presented to services when required. To prove the validity of the attributes, they can be presented to certifying authorities. These authorities can be operated by whoever is able to verify and certify a certain attribute. When an authority certifies an attribute, the certification combined with a reference to the attribute is stored on the blockchain Ethereum and other services can verify it. Another project, THEKEY [THE17], follows a similar goal. It offers digital identities for its users as well, but compared to the other two projects, has a more centralized approach. Users can add attributes and allow services to access them. However, instead of granting access directly, a logically centralized instance compares the access request of the service against the permissions granted to this type of service. Only if these match the attribute can be accessed.

[5] https://www.civic.com/ Accessed: 11.03.2022
[6] https://selfkey.org/ Accessed: 11.03.2022

2.6 Online social networks and social graphs

In online social networks users can create online identities to interact with other users. To do so, they can register friendships within the social network that connect two user identities with each other. In some social networks this friendships represent trust between the connected users, though this is not valid for all online social networks. From a theoretical point of view, the friendship connections of the social network form a graph structure consisting of nodes and edges.
The *nodes* of the social graph represent identities within the system. Normally, these identities are controlled by human users, with the assumption that each human user controls one identity within the social graph. This assumption is violated by Sybil adversaries (see section 2.7), which try to create many identities controlled by one human attacker.
The bidirectional *edges* between the nodes represent trust between the identities in the system. In this work, it is assumed that the edges in the social graph represents strong trust relationships. That is, users who have a trust edge between their nodes know each other well, possibly even know and trust each other in the real world. As a consequence, it is assumed that users who create edges within the social graph trust each other not to start an attack on the system.

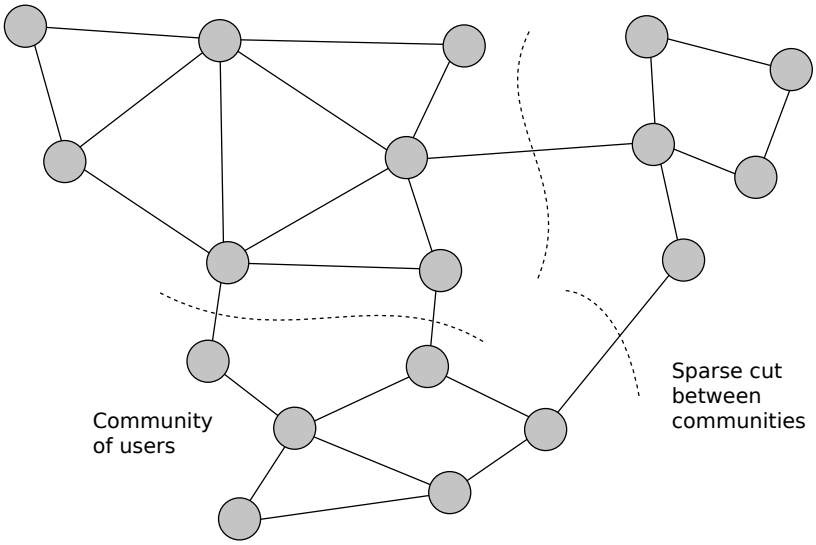

Figure 2.2: *A graph representing an online social network with multiple communities.*

It has been shown that social graphs form multiple *communities* (see figure 2.2), each representing a group of friends, e.g., sharing a common interest. Within a community numerous friendship edges are present, significant more than between the different

communities. The small number of edges connecting the communities is called a *sparse cut*, and can be discovered and used by algorithms. However, calculating the sparse cut for a given graph is computationally expensive, making it infeasible for the social graphs of current online social networks.
One algorithm operating on social graphs are *random walks*. In these, a data packet is send through the graph and on each node a next node is selected randomly. Since there are many edges connecting nodes within a community and only few edges leaving it, there is a high probability that the next node visited by the random walk is within the same community. When the number of hops done by the random walk is chosen small enough, it is relatively unlikely that the random walk ends up in another community. Due to the structure of the graph of social graphs, a length of $O(log\ n)$ is enough to reach all nodes within the graph, independent of the start node of the random walk [Yu+06].
Another approach is based on sparse cuts is flooding packets through the social graph. Since only a small number of connections to other communities exists, most of the flooded packets remain within the current community and only few packets reach other communities. This can be used by algorithms which employ flooding tickets by requiring nodes to receive and present a certain number of tickets [Tra+11]. Nodes who are unable to do so are assumed to be located in another community.
A problem for the privacy protection of users is that some graph algorithms require the structure of the social graph to be visible. It has been shown that in this case it is possible to determine that a node in an anonymized graph and a node in a second online social network belong to the same user, even without knowing anything about the users except for the structure of their friendship graph [NS09]. As a consequence, not only the identities of the users in online social graphs, but also the structure of the social graph should remain hidden.

2.7 Sybil attacks

An important part of authorization systems is the avoidance of *Sybil attacks* [Dou02]. In such an attack, a single human adversary intends to create as many identities within the system as possible, violating the “one identity per user” assumption of many systems. This can have a number of adverse effects on the functioning of a system. Depending on the intentions of the adversary, the Sybil attack can, e.g., aim to deanonymize users, overload the system, control polls or discussions within the system, or influence the public opinion. As an example, some invented information can be made to seem trustworthy to honest users by posting the same information

with multiple fake identities or writing acknowledging comments under the presented information. The honest users are unable to discern between other honest users and Sybil users. Since seemingly many users belief and confirm the information, it increases the perceived trustworthiness of it.
The risk of such attacks can be reduced by enforcing a user registration with some kind of "proof of uniqueness". As an example, one way to implement such a proof would be to force the user to present their passport to a verifying authority. In that case, an online service has to place trust into multiple institutions: For once, the government has to be trusted to only issue one passport per human, but also all verifying authorities have to be trusted into. These validations are much easier to implement for a centralized than for a decentralized system. For centralized systems, the verification only has to be done once and later on a single coherent data store can be queried about the result. In that case, the centralized data store acts as an trust anchor for the system. For decentralized systems, other approaches are required. Since no trusted centralized instance exists, all users need to be able to check the verification of all other users themselves. For once, the result of the verification has to be stored in a way to allow all users to access it. Also, the verification has to be done in a way so it can later on be reviewed for correct execution.

2.8 Sybil defense

Over time, a number of different approaches to defend against Sybil attacks have been proposed. The simplest approach is trusting some other service to take care of the problem, for example by checking passports. Another, well known, approach is that websites require their users to solve *Captchas* which are easy for humans but difficult for computers [Ahn+03]. Prominent examples are transcribing hard to read texts or selecting all pictures matching a given topic (e.g., select pictures containing cars). However, for a determined adversary it is still possible to circumvent these restrictions [RF06], for example by by hiring humans to solve the Captchas. Furthermore, both approaches are only of limited use for decentralized systems. For both a (logically) centralized entity is required, which needs either to be especially deployed or hired as an external service. In the first case, the decentralized property of the system is no longer valid while in the second case trust into a third party is needed. Captchas could be checked decentralized as well, but there is no reason why any participant should trust into other participants to perform valid verifications. Performing the verifications over and over again for each combination of users would theoretically be possible, but would practically be far too much overhead.

Even when it is not its primary purpose, the proof of work of blockchain networks can be considered a Sybil defense approach as well. Only users that are able to provide enough computation power are permitted to create new blocks for the blockchain. Creating numerous blockchain identities is of no use for a Sybil adversary, since it is unable to provide sufficient amounts of computation power for all these identities. Splitting its computation power between the Sybil identities does not increase the number of created blocks, and consequently does not increase the influence the Sybil adversary has on the blockchain. Related approaches – requiring computation power or storage capacity – have been proposed for Sybil defense. However, outside of the blockchain domain these approaches have been found lacking since they require other users to frequently verify that the questionable identities are still able to provide the required computation power or storage capacity.

2.8.1 Sybil defense based on social graphs

Newer research focused on using the trust relationships embedded within online social networks for Sybil defense. In these networks, the social graph representing the trust relationships between the users is used to estimate which identities are trustworthy and which identities may be under the control of Sybil adversaries. The basic assumption is that friends in the social network trust each other to not be a Sybil adversary. Also, it is assumed that each human only has a limited, quite small number of friends in the network.

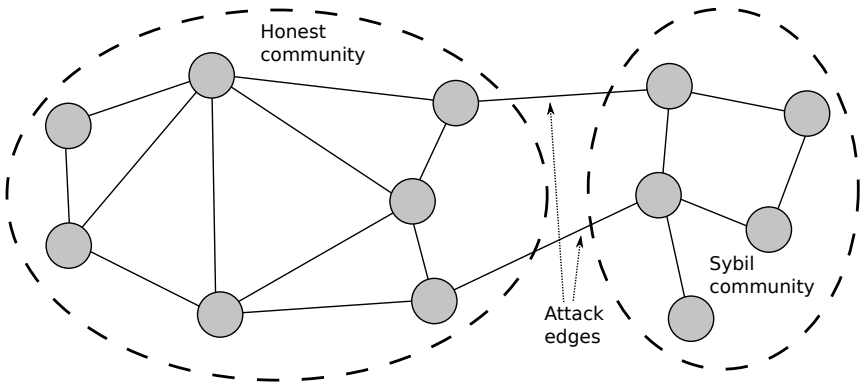

Figure 2.3: *A social graph with an honest and a Sybil community.*

Consequently, the Sybil adversary should only be able to create a small number of edges between its Sybil nodes and the nodes of honest users. When the Sybil adversary creates a lot of Sybil nodes that are only connected to other Sybil nodes, the Sybil community become discernible in the structure of the social graph. With a

global view on the graph, two communities of nodes can be separated: A community of honest nodes and a community of Sybil nodes. Within each community the nodes are closely connected by edges. However, only few edges lead from one community to the other, representing (misplaced) trust of an honest user into a Sybil identity. These edges are called *attack edges*, as depicted in figure 2.3.
Two examples for centralized approaches for Sybil defense are SybilInfer [DM09] and SybilDefender [Wei+12], with the latter one offering better scalability and performance. Both require the social graph to be visible for the user executing the Sybil defense. They also share the requirement of having at least one node in the graph that is known to be honest. The assumption is that the node is part of the honest community within the social graph. Consequently, all nodes outside of this community are considered Sybil nodes. Since these approaches know the whole social graph, they are able to not only identify a single Sybil node but try to detect the complete Sybil community.

2.8.2 Decentralized Sybil defense based on social graphs

Current decentralized Sybil defense approaches are mostly focusing on social graphs to detect and exclude Sybil identities and their communities. While other approaches exists, those have been found to be unable or too impractical to effectively restrict the creation of Sybil identities.
The first significant approach that employed the trust structure of social graphs was *SybilGuard* [Yu+06]. It uses *random routes*, an adaption of random walks, to decide whether another node in the social graph is controlled by an honest user or a Sybil adversary. As in all other approaches, it is assumed that there is only a small probability that random routes will leave the current community and enter the community of the Sybil adversary. If multiple random routes between two nodes intersect, the other node is considered to be in the same community as the testing node and is consequently considered to be honest. The successor of SybilGuard, *SybilLimit* [Yu+08], improves on the guaranties of SybilGuard with regard to the number of Sybil nodes permitted into the system. Furthermore, it experimentally confirms the assumptions regarding the propagation of random routes for multiple real-world online social networks.
An approach using random walks is SybilHedge [FFB17]. It has a high emphasis of protecting the privacy of its users, particularly by keeping the structure of the social graph hidden. Also, it limit the number of newly created Sybil nodes instead of detecting the existing nodes. To do so, a newly created node has to send random walks through the graph and collect (cryptographic) confirmations from existing, already

accepted nodes. When enough confirmations have been collected and returned to the new node, it is considered accepted as well. Sybil adversaries are excluded by directed *edge values* on all edges. If too many confirmation request are received from a neighboring node, the requests will no longer be forwarded. This way, a Sybil adversary can only get a restricted number of Sybil nodes accepted.
A different approach is used by *Gatekeeper* [Tra+11]. Instead of using random walks for accepting nodes, tickets are flooded through the graph. Nodes that receive tickets of enough different ticket sources are accepted as honest. While using a different approach, the underlying assumption remains the same: Sybil nodes are only connected by few edges and will consequently receive fewer of the flooded tickets than honest nodes do.
This assumption and its derived propagation behavior becomes a disadvantage when multiple honest communities are present. In social networks where the community structure is mostly defined by shared interests the existence of multiple communities is to be expected [KNT06]. In that case, all nodes outside of the own community will be determined to be Sybil nodes, even when they are controlled by honest users that are only in a different community. An approach that attempts to avoid this problem is *SybilShield* [Shi+13]. To avoid false positives where honest nodes in other communities are falsely marked as Sybil nodes, randomly selected nodes, called *agents*, are used. These are placed in other communities and try to verify the tested node as well. If enough of the agents accept the tested node as honest, the initial testing node accepts the tested node as honest as well.

Chapter 3

DecentID

DecentID, presented in the following, is a decentralized identity management system based on smart contracts on the blockchain Ethereum. It allows its users to manage their own self-sovereign digital identities and present them to online services, while deciding which shared identity contains which of their attributes. An early version of DecentID has been published in [FSZ18].
In section 3.1 the adversary model is defined, before in section 3.2 the design goals of DecentID based on the adversary model are enumerated and discussed. Following that, section 3.3 presents an example scenario for the use of DecentID from an end user perspective. As such, most technical details are omitted in that section and will be explained afterwards. Section 3.4 introduces the terminology used in the remainder of the chapter, before section 3.5 explains the design and functioning of the core component of DecentID, the *SharedIdentityContract*. Contrived for easier management of multiple SharedIdentityContracts, the *RootIdentityContract* is presented in section 3.6. How the cryptographic keys used within DecentID can be replaced is discussed in section 3.7. In section 3.8, two implementations of DecentID are introduced. Section 3.9 contains the evaluation of DecentID based on the previously presented design goals. Considerations for the widespread use of DecentID are discussed in section 3.10, before the results of this chapter are summarized in section 3.11.

3.1 Adversary model

For DecentID, multiple possible adversaries and related security and privacy threats are considered. These adversaries are based on the adversaries present in connection with current online services and identity management systems, as introduced in chapter 1 and in section 2.5.

Identity providers For the most part, currently existing identity providers can be considered honest-but-curious adversaries. They are able to track the actions of their users, allowing them to create user profiles which can be used for generating revenue from the user, e.g., by selling the collected user profiles. However, the identity provider controls the data of the user that is stored at its server. It could, if it wants to, manipulate and abuse the stored identity data. As such, neither the integrity, nor the privacy of the stored data is necessarily ensured. If the identity provider manipulates the stored data, the user represented by it might be misrepresented towards other online services, possibly even without the user noticing.
While being possible adversaries themselves, the identity providers can also be targets of other adversaries. Considering the often large amount of personal data stored at an identity provider, the servers storing the data become tempting targets for attacks. So even when an identity provider has no intention of abusing their users data, another adversary could try to gain access to it and use the data for its own gain.

Online services The visited online services are also expected to behave as honest-but-curious adversaries. While they will not impede the user while it is using the service, they want to learn as much about the user as possible. This includes data the user does not want the service to have. This includes, but is not limited to, the contents of stored attributes of other identities of the user, how often which service is used, or which services are used by the user at all. The latter part is related to linking multiple identities of a single user, allowing to track the behavior of the user. All of these online services might cooperate and exchange collected data: either with or without the user being aware of it. For (potentially cooperating) services it should not be possible to determine that multiple identities belong together, as long as no unique attributes, e.g., the passport ID, are used. Ensuring these goals while the users data is stored in a publicly readable data store is one of the challenges solved in this thesis. As an example, the adversaries can be visualized as being advertisement companies: they want to know as much about the user as possible to display matching advertisements on the visited services, but are not actively interested in harming the user and driving them away from the service.

Network participants The personal data of the proposed identity management system is stored in a decentralized, and as such, public, data storage network. Assuming that the majority of the participants of this network are honest, the integrity of the stored data against accidental or malicious manipulation is ensured. When using smart contracts on blockchains, additionally the authentication of the stored data can be guaranteed, i.e., the smart contracts can ensure that only authorized users are able to modify the stored data. However, all participants of the network are able to read the stored identity data. Additionally, some metadata, e.g., which identities are controlled by a single user, might be visible. As such, the privacy of the users data can be attacked and has to be protected.

3.2 Design Goals

DecentID is an approach for decentralized identity management. It is supposed to improve on the current state of the art of identity management by protecting against the adversaries described in the previous section. As such, it aims to give users full control over their digital identities, protect their privacy, and allow them to maintain multiple separate digital identities. These goals are explained in the following.

(1) **Control over the identities**
One of the motivations for DecentID is to give users the control over their digital identities. While a user seems to be in control of their identity with existing identity providers, this is only partially true. Since a centralized identity provider stores the users data on its own servers, it is able to modify or remove the users data, grant others access to it, or stop the user from accessing their own data. As such, DecentID should give users the control of their identities without any single instance being able to impede it.

Related to that, dependence on a single, centralized instance should be avoided. Having such an instance would require all users to trust it, which will be hard to achieve given a potentially global usage of an identity provider. While many people use at least one globally operating identity provider, e.g., Facebook or Google, there are still many people who do not want their data managed by large companies. One way to avoid centralized instances can be a design based on distributed ledger technology, e.g., blockchains. replacing the trust in a single provider with trust into a verifiable system. However, this leads to its own challenges. While using a distributed system, only the creator of an identity should be able to modify it, and the integrity of the created identities has to be ensured.

(2) **Privacy of identities**
Attributes in DecentID should be protected against unauthorized access, i.e., not everyone should be able to read the private data. For example, storing an email address as an attribute is useful in many cases, but could easily be abused for spam messages. Additionally, access to unique identifiers like the email address would permit linkability of multiple identities of the same user. This also includes metadata. For example, there should be no publicly readable list of attribute names, even when the attribute data itself is protected.

Furthermore, complete identities have to be protected. The identifiability of the user should be avoided, i.e., it should not be possible to find the offline identity of the creator of an identity. Related is the goal that third parties should be unable to find out that a user presents their identity to another user.

(3) **Multiple pseudonymous identities for different contexts**
On the Internet many users want to create separate identities for different contexts. For example, most people have separate digital identities for business and private life, represented by two different email addresses. As such, DecentID should support multiple identities per user, where the user can choose whether the identities represent their real or a freely chosen, pseudonymous, identity.

When maintaining multiple identities, it is important to ensure the unlinkability of that these identities. Otherwise, it would be possible to find out multiple or all identities of a user when only one of them is known. In that case, maintaining multiple identities would no longer protect the privacy of the user.

In the following sections, the design of DecentID is presented. The properties of the approach will be presented, and the taken design decision justified based on the aforementioned design goals.

3.3 Example Scenario

A use case for the identities created with DecentID would be authentication for an online service, for example a webforum. To join such a forum and communicate with other participants, a user has to create an identity for it.

Definition 3.1: User

A *user* is a person using DecentID to manage their identities. Within DecentID, a user U is represented by their public key $pubKey_{U_i}$ of their asymmetric key pair. The private key $privKey_{U_i}$ matching the public key $pubKey_{U_i}$ is kept secret and is only known to the user. Multiple public keys might be in use in parallel by the same user. $pubKey_{U_i}$ is the i-th public key of user U, e.g., the 5th public key of user B would be $pubKey_{B_5}$.

As described in definition 3.1, a user is a person using DecentID to digitally represent themselves. DecentID is based on smart contracts on the blockchain Ethereum. This is a step towards design goal 1, since the use of smart contracts ensures that only permitted users, but no single identity provider, can modify the identities. To interact with Ethereum, a user is represented by a cryptographic public key. In this scenario, a user Alice is represented by three public keys: $pubKey_{A_1}$, $pubKey_{A_2}$, and $pubKey_{A_3}$. For these keys, Alice has the matching private keys $privKey_{A_1}$, $privKey_{A_2}$, and $privKey_{A_3}$. Her ownership of these private keys can be proven towards others by using the private keys to create digital signatures over random data. Others can then verify the signatures by using the public keys she presented to them. While they are called “public keys” in a cryptographic sense as opposed to the “private keys”, a user of DecentID should not actually display all their public keys publicly at one place, e.g., listing them on their website. By using multiple public keys, a user can maintain multiple digital identities for different purposes. Those identities cannot be linked to each other, as long as it is kept secret that a single user created all these public keys.

Definition 3.2: Service

A *service* is an online entity that requires a digital identity to use its offered features. Similar to a user, a service S is identified by its public key $pubKey_S$. While a service could use multiple public keys, in this work it is assumed that only a single one is used.

Within DecentID, there is no technical difference between a user and a service as defined in definition 3.2. However, for ease of explaining the system a service is considered a special type of user which offers something to the users interacting with

it. Different from a user, the public key used by a service is commonly known by others. For example, a service S could display its public key $pubKey_S$ on its webpage so that interested users can access it. In practice, a service could also use multiple public keys the same as a user does. However, in the following it is assumed that a service only uses a single public key and has no reason to use multiple ones. In this example scenario, three services are considered which are offering three distinct webforums for their users. Within the webforums, the users can talk to other users online about the topics discussed in the specific webforum. The webforums also offer their users to present a user profile to other users of the respective webforum, where personal data can be displayed by the user.

Service S offers a webforum where its users can discuss about their hobby of skiing. The service is identified by its public key $pubKey_S$.

Service P offers a webforum where its users can discuss about their pets. The service is identified by its public key $pubKey_P$.

Service D offers a webforum where its users can discuss about their dogs. The service is identified by its public key $pubKey_D$.

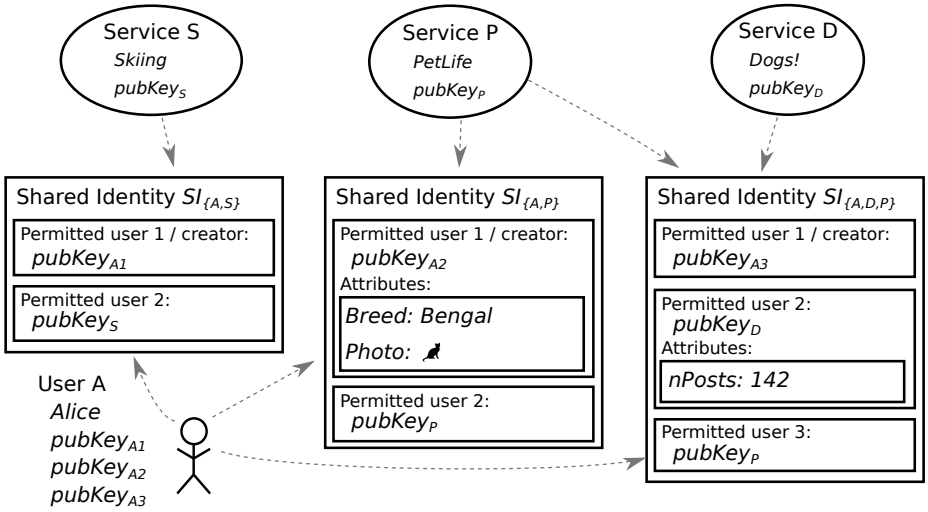

Figure 3.1: *Alice is using shared identities to interact with services. Two of the shared identities contain attributes describing them.*

In figure 3.1 the example scenario is depicted. The user Alice, identified by her public keys $pubKey_{A_1}$, $pubKey_{A_2}$, and $pubKey_{A_3}$ has created three *shared identities* $SI_{\{A,S\}}$, $SI_{\{A,P\}}$, and $SI_{\{A,D,P\}}$ to use for three webforums. The services offering the webforums are identified by their respective public keys $pubKey_S$, $pubKey_P$, and $pubKey_D$. As such, Alice has three pseudonymous identities, as described in design goal 3.

Definition 3.3: Shared Identity

A *shared identity* $SI_{\{A,S\}}$ is a digital representation of a user A and has been created by this user A. Further users and services, e.g., service S, can be permitted to access the shared identity. It can contain attributes describing the user A. These attributes could have been created by the user themselves or other permitted users. Within DecentID, a shared identity is represented by a SharedIdentityContract (see section 3.5) and the attributes that are part of it.

As described in definition 3.3, a shared identity exists between a user and a service. As seen in figure 3.1, the user Alice, identified by $pubKey_{A_1}$, has created a shared identity $SI_{\{A,S\}}$ with the service identified by $pubKey_S$. This shared identity is used to represent Alice towards the service when she uses the provided webforum to talk about skiing. Since no further data is part of this shared identity, the service only knows the shared identity $SI_{\{A,S\}}$ and the public key $pubKey_{A_1}$. It does not know that the public key belongs to Alice, does not know about her other public keys, nor does it know about her other shared identities.

The shared identity $SI_{\{A,P\}}$ that has been created by Alice to use for the webforum "PetLife" provided by service P, also contains an *attribute*.

Definition 3.4: Attributes

An *attribute* can be created by a user or a service and contains some arbitrary data. Two types of attributes are differentiated in DecentID: small attributes, that are stored on the blockchain, and large attributes, that are referenced from the blockchain but are stored in an external storage.

To describe her cat to other users, Alice added two attributes to her identity $SI_{\{A,P\}}$: a small attribute, describing the breed of her cat, and a large attribute, containing a photo of her cat. While the small attribute is stored within the shared identity on the blockchain, the large attribute is stored off-chain in an arbitrary storage to reduce the financial cost of storing it. The service P has been granted access to her shared identity $SI_{\{A,P\}}$ by Alice and is able to read the contained attributes.

To talk about dogs on the webforum provided by service D, Alice created the shared identity $SI_{\{A,D,P\}}$ while using her public key $pubKey_{A_3}$. The service D then added an attribute to the identity $SI_{\{A,D,P\}}$ created by Alice, which describes the number of posts Alice wrote within the webforum. This is depicted on the right of figure 3.1. To talk about dogs on the "PetLife" forum of service P as well, Alice added the public key $pubKey_P$ of service P to the shared identity $SI_{\{A,D,P\}}$.

Now that the public key of service P has been added, it is able to read all attributes linked to the shared identity $SI_{\{A,D,P\}}$, independent of who added the attribute to the identity. The inverse is also true: If service P adds an own attribute to the shared identity, all other users permitted to access the identity can read this attribute. However, just because a service is able to read an attribute, that does not mean that it has to use the data contained within it. For example, service P is able to read the attribute containing the number of posts written in the forum of service D. But since this is no property of the identity that is of interest for service P, it is ignored by it. In the future, service P might still decide otherwise and start using the attribute. Other users or services, which have not been added to $SI_{\{A,D,P\}}$, are not able to read the attached attributes. Due to this, the confidentiality of the attributes is ensured, which is part of design goal 2.
Since Alice used different public keys for her shared identities, each service only knows about the shared identities they have been added to as permitted users. This means, that service S only knows about the shared identity $SI_{\{A,S\}}$ and can only read the attributes contained within it (which are currently none). Service P is able to access both identities $SI_{\{A,P\}}$ and $SI_{\{A,D,P\}}$, but does not know that they have been created by the same user since different public keys have been used. The attributes contained in these identities can be read by service P. Service D is able to read the attributes contained in the shared identity $SI_{\{A,D,P\}}$, but does not know about the other two shared identities of Alice. As such, the privacy component of design goal 3 is fulfilled.

3.4 Important terms and variables

In this section, some terms are listed which are used in the rest of the chapter. These terms are shortly explained here for reference, a longer explanation is found on the listed page.

3.4.1 Definitions

Term	Description	Page
Attributes	Created by a *user* and contains some data.	39
Creator	The creator of a *shared identity* or *attribute* is the *user* that created it.	44
Off-chain attribute	An off-chain attribute is an *attribute* stored outside the blockchain in external storage.	50
On-chain attribute	An on-chain attribute is an *attribute* stored within a *SharedIdentityContract* on the blockchain.	46
Owner	The owner of an *AttributeContract* is the *user* that the AttributeContract was created for.	53
Permitted user	A permitted user is a *user* that has been permitted access to a *shared identity*.	44
Service	A service is a type of *user* that offers a platform to interact with other users.	37
Shared Identity $SI_{\{U\}}$	A digital representation of a *user* U.	39
User U	A person using DecentID to manage their identities, represented by their public keys $pubKey_{U_i}$.	37

Table 3.1: *General terms used in this chapter.*

3.4.2 Smart contracts and files

Term	Description	Page
AttributeContract AC_i	A smart contract, representing an *attribute* stored off-chain.	51
AttributeData AD_i	Stored in external storage, contains the data of an attribute.	52
AttributeLocatorFile $ALF_{\{U,S\},i}$	Stored in external storage, contains a list of blockchain addresses to *AttributeContracts*.	51
IdentityLocatorFile $ILF_{\{U\}}$	Similar to the *AttributeLocatorFile*, stored off-chain and contains references to the *SharedIdentityContracts* of a *user* U.	55
RootIdentityContract $RIC_{\{U\}}$	A smart contract, storing multiple references to the *SharedIdentityContracts* and *AttributeContracts* of a *user* U.	55
SharedIdentityContract $SIC_{\{U,S\}}$	A smart contract, the technical representation of the *shared identity* of *user* U, used for service S.	43

Table 3.2: *The smart contracts and off-chain files used in this chapter.*

3.4.3 Cryptographic keys

Term	Description	Page
$kSIC_{\{U,S\}}$	$kSIC_{\{U,S\}}$ is a symmetric *attribute* encryption key used for encrypting the attribute data stored within the *SharedIdentityContract* $SIC_{\{U,S\}}$.	44
$privKey_{U_i}$	The i-th private key $privKey_{U_i}$ of a *user* U. Forms an asymmetric key pair together with the public key $pubKey_{U_i}$.	–
$pubKey_{U_i}$	The i-th public key $pubKey_{U_i}$ of a *user* U. Forms an asymmetric key pair together with the private key $privKey_{U_i}$.	–

Table 3.3: *The cryptographic keys used in this chapter.*

3.5 Shared Identities

In this section, the creation of a shared identity for the user Alice is described step by step. The created identity is then shared with an online service. Alice is represented by her public key $pubKey_{A_1}$, the service S by its public key $pubKey_S$. Alice creates a shared identity $SI_{\{A,S\}}$ and grants the service access to it. Afterwards, Alice adds a small on-chain attribute to the shared identity, while the service adds a large off-chain attribute. The off-chain attribute can then be shown to another service by Alice.

3.5.1 Creating and sharing *SharedIdentityContract*s

Within DecentID, shared identities are represented by *SharedIdentityContract*s and additional attributes stored in or referenced by it.

Definition 3.5: SharedIdentityContract

A *SharedIdentityContract* $SIC_{\{U,S\}}$ is an instantiation of the shared identity of a user U within DecentID, that is shared with a service S. It is a smart contract on the blockchain Ethereum and can be accessed by its fixed address on the blockchain until it is deleted by its creator. The contents of a SharedIdentityContract are:

- A list with at least one permitted user, represented by their public keys. The first permitted user is the creator of the SharedIdentityContract.
- For each permitted user, a symmetric encryption key $kSIC_{\{U,S\}}$ used for the stored attributes.
- For each permitted user, an arbitrary number of attributes. All permitted users are able to read all attached attributes and add own ones.

A user is able to maintain SharedIdentityContracts towards multiple other users or services.

To begin creating her shared identity, Alice deploys a SharedIdentityContract, which is the central part of DecentID, to the blockchain. This smart contract is created at the blockchain address $SIC_{\{A,S\}}$. Initially, only Alice is able to access its data. The blockchain transaction to create this smart contract is sent from her Ethereum account identified by her public key $pubKey_{A_1}$. On construction, the smart contract automatically adds this public key as the first *permitted user* of $SIC_{\{A,S\}}$.

Definition 3.6: Permitted user

A *permitted user* is a user or service that has been permitted access to a shared identity. They are identified by their public keys which are stored within the SharedIdentityContract. Permitted users are able to read all attributes that are part of the shared identity, as well as to add own attributes to it. Only the public keys of users that have been permitted access to the shared identity are stored within it.

As permitted user, Alice is able to add and remove attributes to $SIC_{\{A,S\}}$. Since Alice is the first permitted user, she is also the *creator* of the SharedIdentityContract.

Definition 3.7: Creator

The *creator* of a shared identity or attribute is the user or service that created the identity or attribute. The public key that was used to send the blockchain transaction creating the SharedIdentityContract is automatically stored as its first permitted user and as such as its creator.

As creator, Alice is additionally able to add and remove further permitted users. Also, she is the only one able to remove the SharedIdentityContract from the blockchain if it is no longer needed at some time in the future. As such, she is the only person being in control of the SharedIdentityContract, as required by design goal 1.
Even though no attributes are contained in the new SharedIdentityContract yet, Alice has to provide an attribute encryption key $kSIC_{\{A,S\}}$ when creating the contract.

Definition 3.8: kSIC

$kSIC_{\{U,S\}}$ is a symmetric attribute encryption key used for encrypting the attribute data stored within the SharedIdentityContract $SIC_{\{U,S\}}$ as well as in linked AttributeLocatorFiles. One $kSIC_{\{U,S\}}$ is used per SharedIdentityContract. Copies of the key are encrypted with the respective $pubKey_{P_i}$ for each permitted user P and stored encrypted in the SharedIdentityContract.

$kSIC_{\{A,S\}}$ is a newly generated symmetric encryption key, used to encrypt any attributes that will be added to the identity. Ensuring the confidentiality of the attributes is an important step towards design goal 2, i.e., to protect the privacy of the users data. The same $kSIC_{\{A,S\}}$ is used for encrypting and decrypting the attributes attached by all permitted users of the identity $SIC_{\{A,S\}}$. To protect $kSIC_{\{A,S\}}$ from unauthorized access, it is encrypted with Alice's public key $pubKey_{A_1}$ before being added to the newly created SharedIdentityContract $SIC_{\{A,S\}}$.

To allow the service S to access her SharedIdentityContract $SIC_{\{A,S\}}$ and future contained attributes, Alice adds the public key of the service $pubKey_S$ as an permitted user to $SIC_{\{A,S\}}$. This public key $pubKey_S$ could for example be available on a website provided by the service. While Alice might not want that anyone knows that $pubKey_{A_1}$ is owned by her for privacy reasons, the service wants to be accessible for others and as such displays its public key. Before adding the $pubKey_S$ to the list of permitted users, the smart contract code of $SIC_{\{A,S\}}$ ensures that $pubKey_S$ is not registered as a permitted user already.
Since the future attributes will be encrypted, the service needs access to the attribute encryption key $kSIC_{\{A,S\}}$ as well. For this, Alice encrypts $kSIC_{\{A,S\}}$ with $pubKey_S$ and adds $enc_{pubKey_S}(kSIC_{\{A,S\}})$ to $SIC_{\{A,S\}}$. Now, $kSIC_{\{A,S\}}$ is stored in the SharedIdentityContract twice: once encrypted for Alice, and once for the service.

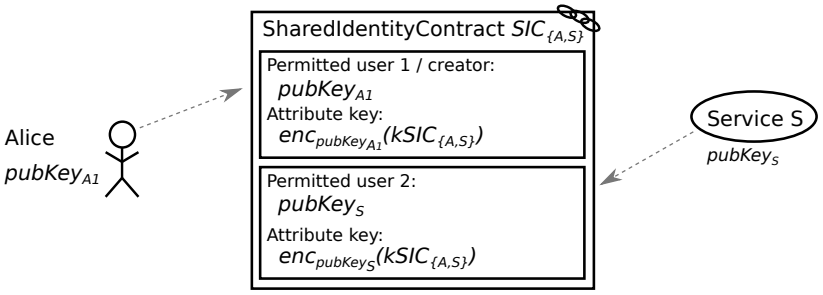

Figure 3.2: *A SharedIdentityContract with Alice as creator and a service as an additional permitted user.*

Figure 3.2 shows the state of the newly created SharedIdentityContract $SIC_{\{A,S\}}$. Alice, represented by her public key $pubKey_{A_1}$, is listed within the SharedIdentityContract as the first permitted user and creator of it. The attribute encryption key $kSIC_{\{A,S\}}$ for this contract is encrypted with $pubKey_{A_1}$ and stored in the contract as well. Since Alice has access to the private key $privKey_{A_1}$ matching $pubKey_{A_1}$, she is able to decrypt $enc_{pubKey_{A_1}}(kSIC_{\{A,S\}})$ and retrieve $kSIC_{\{A,S\}}$ to encrypt and decrypt the future attributes. Similarly, the service S which is identified by its public key $pubKey_S$ has been added as a permitted user to $SIC_{\{A,S\}}$. The service has access to its own copy of $kSIC_{\{A,S\}}$ which is encrypted with $pubKey_S$.
Now that the shared identity has been created, Alice can inform the service about the newly created SharedIdentityContract $SIC_{\{A,S\}}$. This could be done by entering the blockchain address $SIC_{\{A,S\}}$ of the SharedIdentityContract into a form on the website of the service, or automatically by a smartphone application that also automated the creation of the SharedIdentityContract. The service can then access the shared identity, and is able to recognize Alice when she wants to interact with the service.

Adding a second service to the same SharedIdentityContract $SIC_{\{A,S\}}$ would be possible as well. This would grant the second service access to the attributes appended by both Alice and the first service S, since each permitted user of a SharedIdentityContract is able to read all contained attributes. As seen in the introductory example scenario in section 3.3, using one shared identity with multiple services could be done to maintain one identity used for a certain topic on multiple online services.

3.5.2 On-Chain Attributes

Two types of attributes are supported in DecentID: on-chain and off-chain attributes. The data of on-chain attributes is stored directly in the SharedIdentityContract, whereas the data of off-chain attributes is stored in an external storage. In this section on-chain attributes are explained, the following section explains how off-chain attributes are linked to the identity.

To enable the service to send her a newsletter, Alice wants to add her email address as an on-chain attribute to $SIC_{\{A,S\}}$. While this section uses Alice as an example, it should be noted that all permitted users are able to add attributes to the SharedIdentityContract.

Definition 3.9: On-chain attribute

An on-chain attribute is a small attribute stored within a SharedIdentityContract. $Attr_{\{U,S\},i.j}$ denotes the j-th attribute of the i-th permitted user in $SIC_{\{U,S\}}$. It can be accessed with the indexing key $Index_{\{U,S\},i.j}$. Additional data about the attribute is provided by its flags.

The on-chain attributes are stored in a *mapping* within the SharedIdentityContract. A mapping in Solidity, a programming language for smart contracts in Ethereum, is a key-value data structure, where the data value is addressed with the hash value of its indexing key. As such, there is no order of attributes, the indices used in the following are only for explanation. In DecentID, the stored data value consists of the flags of the attribute, as well as the attribute data itself. For each permitted user, a separate attribute mapping exists, containing the attributes the respective permitted user added to the SharedIdentityContract. While all permitted users are able to read all attributes attached to the identity, they are only able to add, modify, or remove attributes stored in their own mapping.

The attribute mapping for one permitted user is shown in figure 3.3. In the following, the indexing key, the flags, and the data value will be explained.

```
mapping ( string => Attribute{byte flags, bytes data} )
```

Figure 3.3: *Pseudocode of a mapping of attributes.*

Within DecentID, the indexing key is an arbitrary string the user selects to address the attribute. Technically, not the string but the hash value of the string is used to index the stored data. Also, the hash value of the indexing key is not stored within the smart contract. On the one hand, this improves the privacy of the users data since it is not possible to read the used indexing keys from the stored contract. Doing so would grant an adversary information about the users attributes, even when the attributes themselves are encrypted. On the other hand, that means that it is not possible to iterate over the list of stored attributes with Solidity. The data value at a known indexing key can be retrieved, but by itself there is no possibility to iterate over all used indexing keys of a mapping. If this functionality is required, an adjacent list of indexing keys could be stored in the smart contract.
The flags stored for each attribute contain some metadata about the stored attribute data. They are stored as a bit set next to the attribute data, i.e., the single bits of the stored byte are interpreted. Specifically, the following flags are currently supported:

Set In Solidity it is not possible to determine whether a mapping entry is not set or is set to zero. Consequently, when an attribute with an empty string as data is added, some way is needed to mark the attribute as set. A possible use case why some user might want to do so could be to store a single bit of data: It is enough to know whether the attribute was set, its value does not matter. As such, this flag is set for all existing attributes. Additionally, it is also used as a parameter when calling a function to set the value of an attribute: If the flag is set in the flags of the new attribute, the attribute is stored, otherwise it is removed.

Encrypted This flag informs whether the attribute data is encrypted with the attribute encryption key $kSIC_{\{A,S\}}$ or not. Especially when storing binary data in the attribute it might be impossible to determine whether the stored data is encrypted by only checking the data itself. To ensure the confidentiality of the users data, most attributes should be stored encrypted. However, in some cases it could be desirable to explicitly not encrypt the attribute data, e.g., to store an attribute that should be read by another smart contract without access to $kSIC_{\{A,S\}}$.

External Storing on-chain attributes within the blockchain is quite financially expensive. As such, it is preferable to store large attributes, e.g., a profile picture, in some cheaper external storage. To determine whether the stored data is the attribute itself or a reference to an external, off-chain, storage, this flag is used. Off-chain attributes are explained later on in section 3.5.3.

DecentID makes no assumptions about the type of data that is stored within the identity attributes. Technically, the attribute data itself is stored as a byte array within the smart contract. This means that it can, but does not have to be, a human readable string. Consequently, arbitrary machine-readable binary data or human-readable strings can be stored in the attributes. Which data is stored in which format is determined by the use case of the attribute, e.g., it could be a human readable string or a list of binary data entries. If the attribute is supposed to be encrypted, it is encrypted with the symmetric attribute encryption key $kSIC_{\{A,S\}}$ before the attribute is added to the SharedIdentityContract.

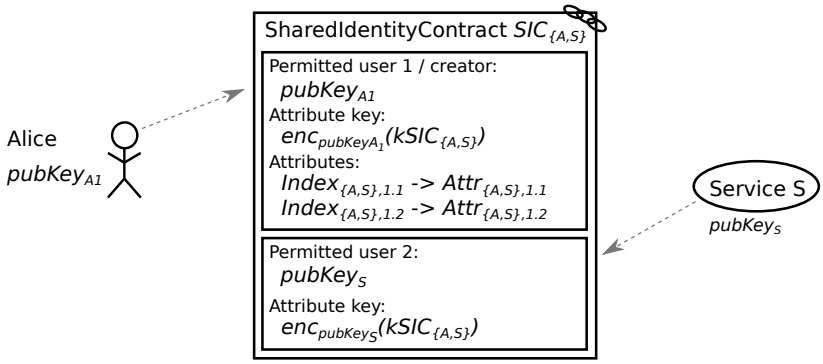

Figure 3.4: *A SharedIdentityContract with two on-chain attributes, created by Alice.*

In figure 3.4 the SharedIdentityContract $SIC_{\{A,S\}}$ is depicted. The attribute mapping of the first permitted user, Alice, contains two attributes, that can be looked up by their indexing keys $Index_{\{A,S\},1.1}$ and $Index_{\{A,S\},1.2}$: They are the first two attributes added by the first permitted user of $SIC_{\{A,S\}}$. In this example, the first attribute is the email address of Alice, which means that $Index_{\{A,S\},1.1}$ is the hash value of the string "email":

$$Index_{\{A,S\},1.1} = H(\text{"}email\text{"})$$

$Attr_{\{AS\},1.1}$ is the data stored in this mapping referenced by $Index_{\{A,S\},1.1}$. It contains Alice's email address "alice@example.com" as well as the flags of the attribute. Since Alice does not want the email address to be publicly readable, she encrypts it with the symmetric attribute encryption key $kSIC_{\{A,S\}}$ of the SharedIdentityContract: $enc_{kSIC_{\{A,S\}}}(\text{"}alice@example.com\text{"})$. As such, the flags denote the attribute as set, as encrypted, and as on-chain. Together, the flags and the encrypted email address form the stored attribute:

$$Attr_{\{A,S\},1.1} = \{Set|Encrypted;\ enc_{kSIC_{\{A,S\}}}(\text{"}alice@example.com\text{"})\}$$

Everyone with access to the blockchain is now able to retrieve $Attr_{\{A,S\},1.1}$ from the smart contract. However, only permitted users registered in $SIC_{\{A,S\}}$, e.g., the service S, are able to read the attribute encryption key $kSIC_{\{A,S\}}$, and decrypt the encrypted email address. As such, only users permitted by Alice are able to retrieve her email address from the stored attribute. The same is true for other attributes attached to $SIC_{\{A,S\}}$, for example $Attr_{\{A,S\},1.2}$.

```
1  function getAttribute (address _user, string memory _key)
2          public view returns (bytes memory) {
3      if (users[_user].key.length == 0) {
4          return "";
5      }
6      Attribute memory attr = users[_user].attributes[_key];
7      if ((attr.flags & flag_set) == 0) {
8          return "";
9      }
10     return attr.data;
11 }
```

Figure 3.5: *The source code of the SharedIdentityContract for retrieving attribute data.*

The source code of the smart contract function to read an attribute from the SharedIdentityContract is depicted in figure 3.5. The depicted function has three parts: checking whether the permitted user exists (line 3-5), checking whether the attribute is set (line 6-9), and returning the attributes data (line 10). The first **`if`**-clause in line 3 checks whether a permitted user with the given public key is stored within the mapping of permitted users. Within Solidity, public keys and blockchain addresses are both stored in the data type `address`. If the permitted user with the given public key `_user` has an attribute encryption key $kSIC_{\{A,S\}}$ stored in their variable `key`, they are registered within the smart contract. Since Solidity does not differentiate between an entry not existing in a mapping or being set to zero, part of the data value, in this case the variable `key` containing $kSIC_{\{A,S\}}$ for this permitted user, is checked whether it is empty. This should not be the case for any existing permitted user. If the variable `key` is not empty, the attribute is checked for existence in line 7. Again, a part of the data has to be checked whether it is different from zero. In this case, the flag of the attribute (`attr.flags`) is checked whether the bit of the flag "set" is zero: `(attr.flags & flag_set) == 0`. If the attribute is marked as set, its value is returned in line 10. Otherwise, if either the permitted user is unknown or the attribute is not set, an empty string is returned (lines 4 and 8, respectively).

3.5.3 Off-Chain Attributes

For small attributes, e.g., the aforementioned email address, storing them on the blockchain is financially cheap. However, for larger attributes, e.g., an email certificate, storing it within the blockchain becomes financially too expensive. As an alternative, it is possible to store attributes off-chain in external storage.

Definition 3.10: Off-chain attribute

An off-chain attribute is an attribute stored outside the blockchain. A reference to it is stored within the blockchain in an on-chain attribute.

The contents of the reference on the blockchain depends on the referred to storage. For example, when a distribute hash table (DHT) is referenced, a reference to the DHT and the hash of the referred to data has to be stored. When using a web storage, the URL would be enough to retrieve the file. However, if the integrity of the referenced data needs to be ensured, an additional hash of the data has to be stored in the smart contract. Otherwise, the referenced data could be modified by the web storage provider. When the permitted users of the shared identity are unable to verify the integrity of the attributes, the stored data cannot be trusted and should not be used.

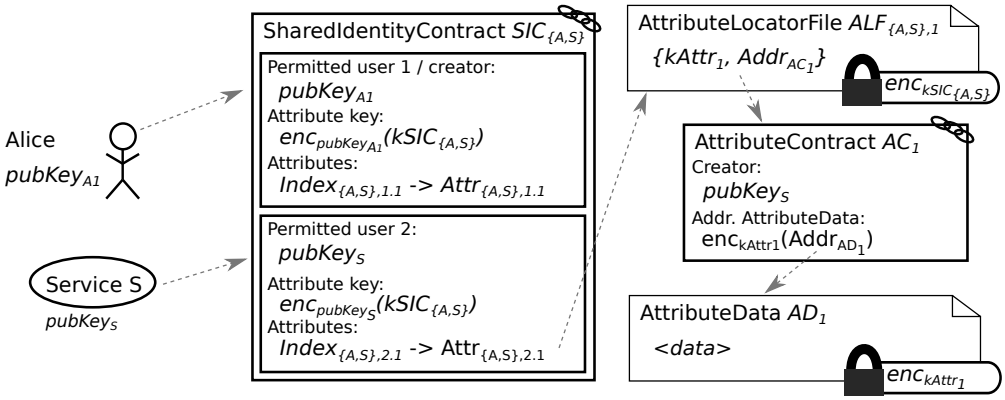

Figure 3.6: *A SharedIdentityContract with an on-chain attribute created by Alice, and an off-chain attribute created by the service.*

Figure 3.6 displays the files and smart contracts used when storing attributes outside the blockchain. $Attr_{\{A,S\},2.1}$ in the SharedIdentityContract $SIC_{\{A,S\}}$ contains a reference to an AttributeLocatorFile $ALF_{\{A,S\},1}$ stored in an external storage. Within $ALF_{\{A,S\},1}$, a list of symmetric encryption keys and blockchain addresses can be stored. In this example, only a single key and address is stored. This address refers to an AttributeContract AC_1 on the blockchain, which itself contains another reference to the AttributeData AD_1 stored in the external storage.

While the multiple references bring a certain overhead with them, they also offer some advantages to the users. Most importantly, storing data in an external storage is much cheaper than storing data on the blockchain. This way, even large attributes can be stored, which might be prohibitively expensive or even impossible on-chain. Especially when multiple attributes should be stored the presence of the AttributeLocatorFile reduces the amount of storage needed on the blockchain.
In the following, these files and smart contracts will be discussed.

Definition 3.11: AttributeLocatorFile

A file $ALF_{\{U,S\},i}$ stored in an external storage, containing a list of blockchain addresses to AttributeContracts. For each blockchain address $Addr_{AC_j}$, an additional symmetric encryption key $kAttr_j$ can be stored.

The purpose of the AttributeLocatorFile $ALF_{\{A,S\},1}$ is to reduce the number of attribute references that have to be stored on the blockchain. If many attributes should be added to a SharedIdentityContract but the list of attributes is rarely or never modified, it is more cost efficient to store the list of attribute in $ALF_{\{A,S\},1}$ instead of in $SIC_{\{A,S\}}$. Since the AttributeLocatorFile can become quite large, it is stored off-chain, e.g., in a distributed hash table. To protect $ALF_{\{A,S\},1}$ against unauthorized access, it can be encrypted with the symmetric attribute encryption key $kSIC_{\{A,S\}}$ stored in $SIC_{\{A,S\}}$. Whether this is the case can be recorded in the flags of the on-chain attribute referencing $ALF_{\{A,S\},1}$, i.e. $Attr_{\{A,S\},2.1}$ in figure 3.6. The encryption key $kAttr_1$ stored in $ALF_{\{A,S\},1}$ is used to encrypt the data in the AttributeContract AC_1, as well as Alice's email certificate itself in the AttributeData file AD_1.

Definition 3.12: AttributeContract

The *AttributeContract* AC_i is a smart contract stored on the blockchain, representing an attribute stored off-chain. It contains the public key of the user that created it, as well as the address of the AttributeData AD_i it represents.

Within the blockchain, AC_1 contains the public key of the creator of the attribute as well as the off-chain address of the attribute. Its purpose is to prove the creator of the attribute. Since the public key $privKey_S$ of the creator of the smart contract is permanently set on its deployment, no one except the owner of $privKey_S$ could have created AC_1. Additionally, AC_1 ensuring the integrity of the attribute data. The contained address $Addr_{AD_1}$ of the attribute is encrypted with $kAttr_1$.

Definition 3.13: AttributeData

A file AD_i stored in an external storage that contains the data of an attribute. It is encrypted with a unique symmetric attribute encryption key $kAttr_i$.

The attribute data file AD_1 contains the attribute itself and is stored in the external storage. As with on-chain attributes, the format of the contents of the attribute are use case specific and are not specified by DecentID.
To retrieve the data of off-chain attributes, more work is required as for retrieving on-chain attributes. The contents of $Attr_{\{A,S\},2.1}$ can be retrieved from the SharedIdentityContract $SIC_{\{A,S\}}$ and decrypted as described in section 3.5.2. However, this does not contain the attribute data but only the address of $ALF_{\{A,S\},1}$ in the off-chain storage. $ALF_{\{A,S\},1}$ can now be retrieved from the storage and its contents be decrypted with $kSIC_{\{A,S\}}$. Contained is the address of AC_1 and the symmetric attribute encryption key $kAttr_1$ used to encrypt the contents of the smart contract AC_1 as well as the attribute data in AD_1. After retrieving AC_1 from the blockchain and decrypting it, the service can verify the public key stored in AC_1. The key should be from a known user or service, in this case from service S. At last, the data can be retrieved from AD_1 in the off-chain storage where it resides at the address stored in AC_1.
As explained above, each file and contract used for off-chain attributes has a certain purpose: The AttributeLocatorFile $ALF_{\{A,S\},1}$ stores a list of attributes, the AttributeContract AC_1 proves the creator of an attribute, and the AttributeData AD_1 stores large attributes off-chain. While all three of these have been used in the above scenario, not all of them have to be used in each case.
For example, since only a single large attribute has to be stored in the presented scenario, using the AttributeLocatorFile $ALF_{\{A,S\},1}$ could be skipped and the attribute entry in $SIC_{\{A,S\}}$ could directly refer to the attribute. In that case, using the AttributeContract AC_1 would not be required either, since only the service S is able to assign an attribute reference to the list of its attributes in $SIC_{\{A,S\}}$. As another example, it might be sufficient to only use the AttributeLocatorFile $ALF_{\{A,S\},1}$ if numerous small attributes should be linked from $SIC_{\{A,S\}}$. In that case, the attribute data could be stored directly within $ALF_{\{A,S\},1}$ without using the additional AttributeContract or AttributeData. While small attributes can be stored within $SIC_{\{A,S\}}$, storing many small attributes is still financially cheaper when stored off-chain within an AttributeLocatorFile.

3.5.4 Granting Attributes

When using online identities, a frequent occurrence is that properties of a user are confirmed by another user or service. For example, email certificates are created by a certificate authority to confirm that the contained public key really belongs to a certain email address [Boe+08]. Afterwards, the user can present the certificate towards a third party, which is called verifier in the following. The verifier is then able to check the data contained in the certificate, and, if they trust the certificate authority, accept the contained public key as belonging to the email address.
When using smart contracts, parts of this confirmation process becomes simpler. Without smart contracts, the creator of such a certificate is identified by their public key, and proves the ownership of the accompanying private key by adding a signature within the certificate. With smart contracts, the public key can be automatically stored within the smart contract on deployment. Storing a signature is not required, since without the matching private key the transaction creating the smart contract could not have been send. Additionally, the program code of the smart contract ensures that the stored creator cannot be changed later on. Similarly, the program code of the smart contract can ensure that the owner of the certificate, i.e., the user for whom something is confirmed, can only be set by the creator of the smart contract and can only be set once.
To include this functionality into DecentID, the AttributeContract has been extended. This way, one user, the creator of the AttributeContract, can grant the ownership of the AttributeContract to another user, its new owner.

Definition 3.14: Owner

The *owner* of an AttributeContract is the user or service that the AttributeContract was created for. The creator of an AttributeContract is able to set the ownership of it to another user, which is identified by their public key.

For this, two public keys are now stored in the AttributeContract. The first public key is of the creator of the contract. This public key should be made generally known, e.g., by publishing it on the website of a service. By checking this key, a verifier can later on ensure that the AttributeContract has really been created by this service. The second public key is of the owner of the AttributeContract. To protect their privacy, the receiving user should create a new public key. If the same public key is used for both the AttributeContract and a SharedIdentityContract, a third party can find out that the same public key has been used, and that the AttributeContract and the SharedIdentityContract are owned by the same user. As before, the address of the AttributeData is still contained within the contract.

When creating the attribute, it has to be considered whether encryption is used for the attributes data. If the attribute is stored unencrypted, everyone can read its data. Depending on the use case of the granted attribute, this can be what is wanted by the attribute creator and owner anyway. If the attribute should be stored in encrypted form, a single-use encryption key should be generated for it. This encryption key then has to be stored in the shared identity that the verifier can access, e.g., as the encryption key for the attribute as stored in the AttributeLocatorFile. If the same encryption key is used for multiple attributes within $SIC_{\{A,S\}}$ between owner and creator of the attribute, all these other attributes become readable to the verifier as well, even when the verifier is not added as a permitted user to $SIC_{\{A,S\}}$.

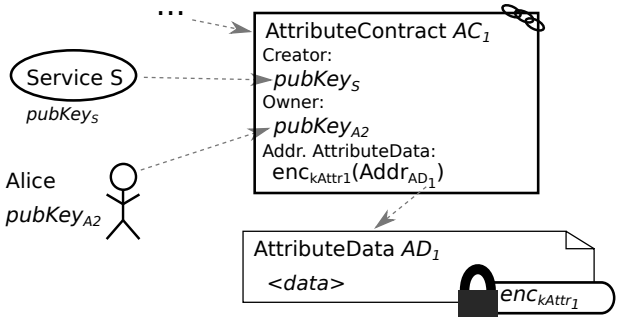

Figure 3.7: *Extended excerpt of figure 3.6: The AttributeContract AC_1 now contains a creator as well as an owner.*

In figure 3.7, the AttributeContract AC_1 presented in the previous section has been extended by the public key of an owner. Service S created the AttributeContract, so its public key $pubKey_S$ is automatically stored as the creator of AC_1 on deployment of the contract. Since this public key is generally known, other users can verify that AC_1 has been created by the service S. The user Alice created a new public key, $pubKey_{A_2}$, to be registered as owner of AC_1. This way, the AttributeContract can also be referenced from other SharedIdentityContracts Alice uses, without disclosing the address of $SIC_{\{A,S\}}$ or Alice's public key used for $SIC_{\{A,S\}}$, which would allow a third party to find out that both AC_1 and $SIC_{\{A,S\}}$ are owned by Alice. Due to the code of AC_1, only its creator is able to set the owner of it. Consequently, a verifier can be sure that Alice, identified by $pubKey_{A_2}$, has been granted this attribute by the service S, identified by its known $pubKey_S$.
As before, AD_1 is linked from AC_1. Continuing the example of an email certificate, AD_1 would contain Alice's email address, the encryption key used to write encrypted emails to it. Other information, i.e., the identity of the certificate authority, its public key, and a signature over the certificate data, is no longer required and has been made obsolete by the source code of the AttributeContract.

3.6 RootIdentityContract

For easier management of the owned identities, the *RootIdentityContract* $RIC_{\{A\}}$ has been designed.

Definition 3.15: RootIdentityContract

The *RootIdentityContract* $RIC_{\{U\}}$ is a smart contract stored on the blockchain, storing references to the SharedIdentityContracts and AttributeContracts of a user U.

The RootIdentityContract simplifies the use of DecentID, especially when multiple end devices are used in parallel, e.g., a smartphone and a desktop computer at home. Since all SharedIdentityContracts can be linked from the RootIdentityContract, all of them are known on all devices without updating the list of owned identities by some external means.
Since it should only be accessed by its creator, the address of the users RootIdentity-Contract is kept private and not shared with any other users or services. Similar to the SharedIdentityContract and its use of AttributeLocatorFiles, the RootIdentityContract contains two references to lists of blockchain addresses, with the lists themselves being stored off-chain. One of them is an AttributeLocatorFile, storing references to AttributeContracts that are currently not referenced by any SharedIdentityContracts. The other referenced list is an IdentityLocatorFile.

Definition 3.16: IdentityLocatorFile

Similar to the AttributeLocatorFile, an *IdentityLocatorFile* $ILF_{\{U\}}$ is stored off-chain and contains references to the SharedIdentityContracts of a user U.

It contains the blockchain addresses of all the SharedIdentityContracts of the user. Additionally, $RIC_{\{A\}}$ contains an symmetric encryption key $kRIC_{\{A\}}$, to encrypt the addresses as well as the contents of $ILF_{\{A\}}$ and $ALF_{\{A\}}$.
As seen in figure 3.8, the user Alice created a RootIdentityContract $RIC_{\{A\}}$. To do so, a new public key $pubKey_{A_3}$ was used, that is not used in any other context. This way, third parties are unable to link $RIC_{\{A\}}$ to any other blockchain transaction or smart contract, nor to Alice herself. Within $RIC_{\{A\}}$, a symmetric encryption key $kRIC_{\{A\}}$ is stored, that is encrypted with the public key $pubKey_{A_3}$. As with the symmetric encryption keys stored in the SharedIdentityContracts, this allows only permitted users to access the data stored within the contract. In the case of a RootIdentityContract, only its creator is permitted to access its contents, further permitted users cannot be added.

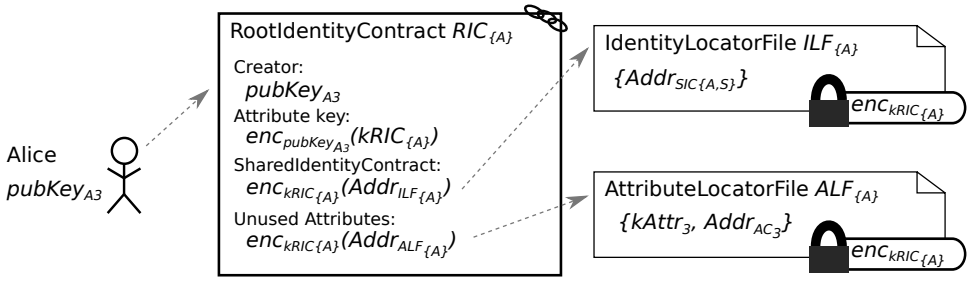

Figure 3.8: *A RootIdentityContract linking the SharedIdentityContracts and unused AttributeContracts of the user Alice.*

The first reference stored within $RIC_{\{A\}}$ contains the address of the IdentityLocatorFile $ILF_{\{A\}}$. Only Alice is able to decrypt the contents of this file, and retrieve the blockchain addresses of the SharedIdentityContracts listed within it. Also, the address of an AttributeLocatorFile $ALF_{\{A\}}$ is stored in $RIC_{\{A\}}$. Within it, the symmetric encryption keys for and the addresses of a number of AttributeContracts are stored. In this example, only a single AttributeContract AC_3 is referenced. Since AC_3 is currently not used by any SharedIdentityContract, its address and encryption key are stored in the $ALF_{\{A\}}$ referenced from $RIC_{\{A\}}$, to avoid losing access to it.

3.7 Replacing Cryptographic Keys

In this section, the different cryptographic keys used within DecentID are discussed. For each used key, it is considered what has to be done to replace it with a newly generated key. This can be required for various reasons, most importantly when a key has been compromised by an adversary.

3.7.1 Keys of permitted users

Within a SharedIdentityContract $SIC_{\{U,S\}}$ two asymmetric public keys are stored: $pubKey_U$ for the first permitted user and creator U of $SIC_{\{U,S\}}$, and $pubKey_S$ for the service S that has been permitted access to the shared identity (see figure 3.4). Additionally, for each permitted user a copy of $kSIC_{\{U,S\}}$ is stored within $SIC_{\{U,S\}}$, that is encrypted with the public key of the respective permitted user. Instead of storing complete public keys within $SIC_{\{U,S\}}$, it could be considered to only store the cryptographic hashes of the public keys. This would reduce the amount of data stored on the blockchain. However, the complete public key is required for encrypting $kSIC_{\{U,S\}}$, and can also be used to encrypt messages send to the respective permitted

user. If the public keys are not stored within the smart contract, both the identity creator and the permitted users would need to store these public keys locally on all of their used devices.
To replace a public key of a permitted user, the respective user has to create a new asymmetric key pair, and store the new public key within $SIC_{\{U,S\}}$, thereby replacing the old public key. Additionally, the copy of $kSIC_{\{U,S\}}$ encrypted with the old public key of this user has to be retrieved from the blockchain, decrypted with the old private key, encrypted with the new public key, and stored in $SIC_{\{U,S\}}$ again.

3.7.2 Attribute encryption keys

To encrypt the potentially large attributes within a shared identity $SI_{\{U,S\}}$, symmetric encryption is used. In general, symmetric encryption is more efficient than asymmetric encryption and has shorter keys. The latter point is important for DecentID, since the key has to be distributed to all permitted users of a shared identity. Two symmetric keys are used to encrypt attributes: $kSIC_{\{U,S\}}$, to encrypt the attributes stored within the SharedIdentityContract $SIC_{\{U,S\}}$ and the AttributeLocatorFiles $ALF_{\{U,S\},i}$ referenced from it, and $kAttr_j$, to encrypt a single off-chain attribute (see figure 3.6). To replace $kSIC_{\{U,S\}}$, the cooperation of all registered permitted users is required. The permitted user U starting the replacement is able to generate a new symmetric attribute encryption key $kSIC'_{\{U,S\}}$, encrypt it with the public keys of all permitted users, and store the encrypted keys in the smart contract $SIC_{\{U,S\}}$. However, all the attributes and the linked AttributeLocatorFiles have to be re-encrypted with the new encryption key. For their own attributes, user U is able to do so themselves. The attributes can be retrieved from the blockchain, decrypted with the old $kSIC_{\{U,S\}}$, encrypted with the new $kSIC'_{\{U,S\}}$, and stored within $SIC_{\{U,S\}}$ again. It works similar for a linked AttributeLocatorFile $ALF_{\{U,S\},i}$: the $ALF_{\{U,S\},i}$ is retrieved from the off-chain storage, decrypted, re-encrypted, and stored in the off-chain storage again. Independent of whether the address of $ALF_{\{U,S\},i}$ changes through the re-encryption, the reference to it in $SIC_{\{U,S\}}$ has to be re-encrypted with the new $kSIC'_{\{U,S\}}$ as well. This is done the same way as described above for on-chain attributes.
When $kAttr_j$ is replaced with a new symmetric attribute encryption key $kAttr'_{\{j\}}$, the AttributeData AD_j stored off-chain has to be re-encrypted, as well as the reference to it stored within the AttributeContract AC_j stored on-chain. Afterwards, the new key $kAttr'_{\{j\}}$ can be stored in the AttributeLocatorFile $ALF_{\{U,S\},i}$, where AC_j is referenced from. Depending on the off-chain storage used, it can be that the address of $ALF_{\{U,S\},i}$ changes when its data is modified, for example when a distributed hash table is used. If the address changes, the new address of the modified $ALF_{\{U,S\},i}$ has to be

stored in $SIC_{\{U,S\}}$ in place of the previous reference to $ALF_{\{U,S\},i}$. Otherwise, the SharedIdentityContract would still reference the old AttributeLocatorFile, containing the now no longer valid encryption key $kAttr_j$.

3.7.3 Attribute ownership

Two public keys are stored within AttributeContracts AC_i (see figure 3.7). These denote the creator as well as the current owner of the linked AttributeData AD_i, and are not necessarily the same public keys as in the SharedIdentityContract referencing AC_i. Different from the public keys stored within a SharedIdentityContract, they are not used for encryption. As such, replacing them is only a matter of storing new public keys within the smart contract. Both the public keys of the attribute creator as well as the attribute owner can only be replaced by the user denoted by the creators public key. The purpose of the AttributeContract is that the creator C of the attribute can grant an attribute to an arbitrary owner O. Due to this, the assumption of third parties is that the owner presenting an AttributeContract AC_i has been granted this attribute by the creator of AC_i. If the owner of the attribute would be able to replace their public key by themselves, this assumption no longer holds. The owner O of AC_i would be able to pass on the attribute to an arbitrary other user O', allowing them to pretend towards third parties that the attribute has been granted to them. However, the creator C might know nothing about the new owner O', despite the new owner being able to wrongly prove that they do. By using the smart contract code to enforce that only the creator C is able to change the public key of the owner, this can be avoided.

3.7.4 Keys in the RootIdentityContract

Within the RootIdentityContract $RIC_{\{U\}}$, two cryptographic keys are used: an asymmetric key pair representing the creator of $RIC_{\{U\}}$, and a symmetric encryption key $kRIC_{\{U\}}$ to protect the data stored within $RIC_{\{U\}}$ (see figure 3.8).
The public key $pubKey_U$ of the asymmetric key pair is stored within the smart contract $RIC_{\{U\}}$ and is used to encrypt the stored symmetric encryption key $kRIC_{\{U\}}$. Since the user is the only one having access to the RootIdentityContract, both entries can be replaced without interacting with other users. After a new key pair has been created, the new $pubKey_{U'}$ can be stored in $RIC_{\{U\}}$. The stored $kRIC_{\{U\}}$ can be decrypted with the old private key $privKey_U$, encrypted again with the new public key $pubKey_{U'}$, and the old entry in $RIC_{\{U\}}$ can be overwritten.

The symmetric encryption key $kRIC_{\{U\}}$ is used to encrypt the references to and the contents of the linked IdentityLocatorFile $ILF_{\{U\}}$ and the AttributeLocatorFile $ALF_{\{U\}}$. Replacing it is more complicated than replacing the stored public key $pubKey_U$, but works similar to replacing $kSIC$ within a SharedIdentityContract. A new symmetric key $kRIC_{\{U'\}}$ has to be generated by the user, encrypted with $pubKey_U$, and stored in the smart contract $RIC_{\{U\}}$. Both locator files $ILF_{\{U\}}$ and $ALF_{\{U\}}$ have to be retrieved from the off-chain storage, decrypted, and encrypted again with the new encryption key $kRIC_{\{U'\}}$. After encrypting them, they can be stored in the off-chain storage again. Their storage addresses are then encrypted with $kRIC_{\{U'\}}$ as well, and stored in $RIC_{\{U\}}$. If the off-chain storage supports removing data from it, the old $ILF_{\{U\}}$ and $ALF_{\{U\}}$ can subsequently be removed.
Replacing one of the attribute encryption keys $kAttr_i$ within the linked $ALF_{\{U\}}$ can be done the same way as replacing $kAttr_j$ within an AttributeLocatorFile of a SharedIdentityContract.

3.8 Implementation

Two implementation of DecentID have been developed with different goals. The intention when developing the first implementation was to demonstrate the use and functionality of DecentID. The second implementation has been a prototypical implementation of a smartphone application. For once, it showed that DecentID is efficient enough to use it on a smartphone. Furthermore, the implementation can be used as a basis for an application used by the general public.
In the following, these two implementations will be presented. For ease of implementation and demonstration, both implementations operate on local Ethereum instances. Compared to using the official Ethereum network, a local blockchain instances offers some advantages. For example, it can operate without the delays necessary for the functioning of a public blockchain and as such allows faster operations, especially important when developing or demonstrating DecentID. Also, no fees have to be paid to write data to the local blockchain. However, only a change in the configuration of the implementations is required to use the official Ethereum network instead.

3.8.1 Demonstrator

The demonstrator for DecentID consists of four web applications. It demonstrates how an identity is created, accessed by online services, and how attributes are granted and displayed. The four web applications can be run on a single as well as on multiple

computers. Independent of whether the web applications are run on one or multiple computers, they do not interact with each other directly. Instead, all data exchange is only done through the SharedIdentityContracts stored in the local blockchain. The web applications are listed here and explained in detail in the following:

- The first web application allows to create a DecentID identity.
- The "university website" allows the user to receive an attribute for their DecentID identity, confirming the student status.
- At an "online shop", the student attribute can be presented for a discount.
- At the fourth web application, the state of all identities stored on the local blockchain is visualized.

The initial version of this demonstrator has been implemented as a prototype as part of the master thesis of Ingo Sobik [Sob17], while most of the demonstrator was designed and implemented in cooperation with the student assistant Philipp Matheis. It is based on the JavaScript framework Meteor[1] to display the current state at the users webbrowser and communicate with the server. There, the library web3x[2] is used to interact with a locally running Ethereum client. For storing the attributes, the distributed hash table Swarm[3] is employed.

3.8.1.1 Architecture

The software architecture of the demonstrator follows the usual approach for using the JavaScript framework Meteor. As depicted in figure 3.9, the web applications of the demonstrator each consist of a client and a server part, with the server part using server side services for blockchain access and data storage.
For Meteor projects, the database MongoDB is used as a default for storing the users data. However, for the demonstrator this is not needed since all data is stored on the blockchain Ethereum or in the distributed hash table Swarm. Swarm can be accessed from the JavaScript code directly since it offers a HTTP interface. To access Ethereum, the library web3x is employed. It offers descriptions of the used smart contracts in JSON format and allows to send transactions and call smart contract functions on the blockchain.

[1] https://www.meteor.com/ Accessed: 11.03.2022
[2] https://github.com/xf00f/web3x Accessed: 11.03.2022
[3] https://www.ethswarm.org/ Accessed: 11.03.2022

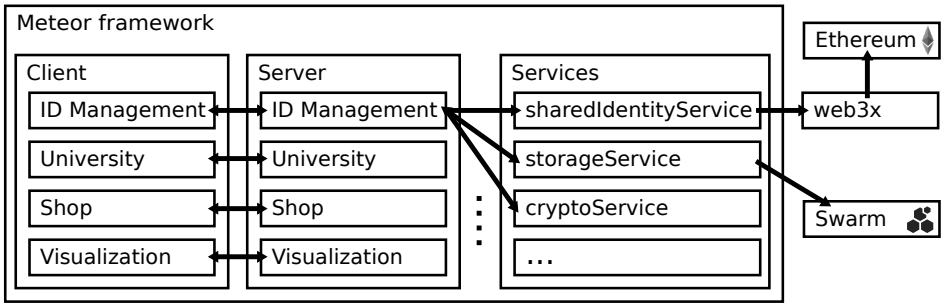

Figure 3.9: *An overview over the software architecture of the demonstrator.*

To allow web applications within the demonstrator to interact with smart contracts or the distributed hash table, a number of Meteor services were implemented. For each smart contract class, one such service exist. They work similar to software libraries and hide the complexity of the interaction behind JavaScript functions. Besides interacting with existing smart contracts, they also allow to deploy new smart contracts to the blockchain. Additionally, a Meteor service providing cryptographic functionalities has been implemented.
For each web application, which are described in the following sections, a client and a server part has been implemented. The client part is executed in the web browser of a user and displays the current state of the relevant data. For example, in the identity management this means listing which identities have been created and with which attributes. If the user wants to modify an identity, the respective request is send to the server part of the web application. This part is run on the web server and can access the provided Meteor services and through them the blockchain as well as the distributed hash table. With support of the Meteor framework, the server part is afterwards able to notify the client of the modified data, to allow a reactive web design without the client having to wait for long page load times.

3.8.1.2 Identity management

The first web application provided by the demonstrator is an identity management interface, allowing the user to create their own shared identities. These identities can be augmented with arbitrary attributes, which are stored on the locally running distributed hash table Swarm.
One of the screens of the identity management interface is displayed in figure 3.10. The user has already created two SharedIdentityContracts, one with the university and one with the online shop. Shown is the shared identity with the university. The user has not added any attributes to the identity themselves, but the university has

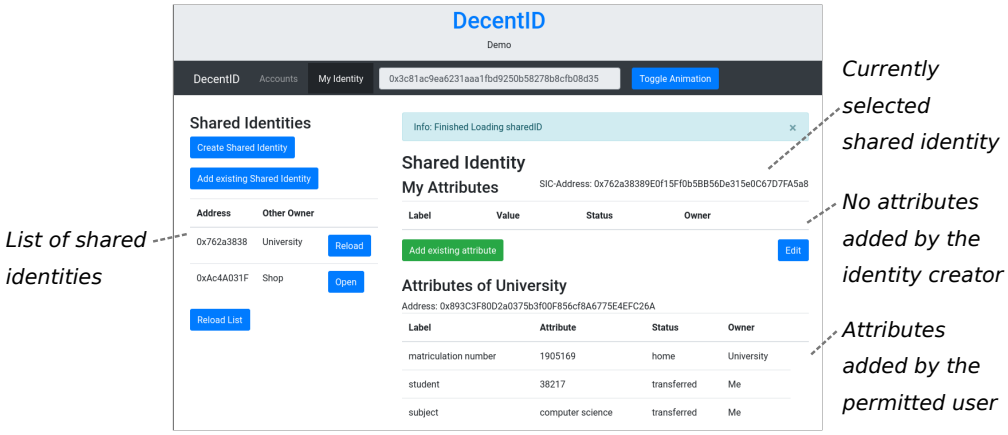

Figure 3.10: *The web interface to create and modify identities.*

added three attributes. Two of these attributes ("student" and "subject") have been granted to the user, which can be seen since their owner has been set to the user. As such, the user is able to add these existing attributes to their other identities, for example to the identity with the online shop.

3.8.1.3 University

The created identities can then be used with online services. As one such service, the web portal of a university has been implemented.

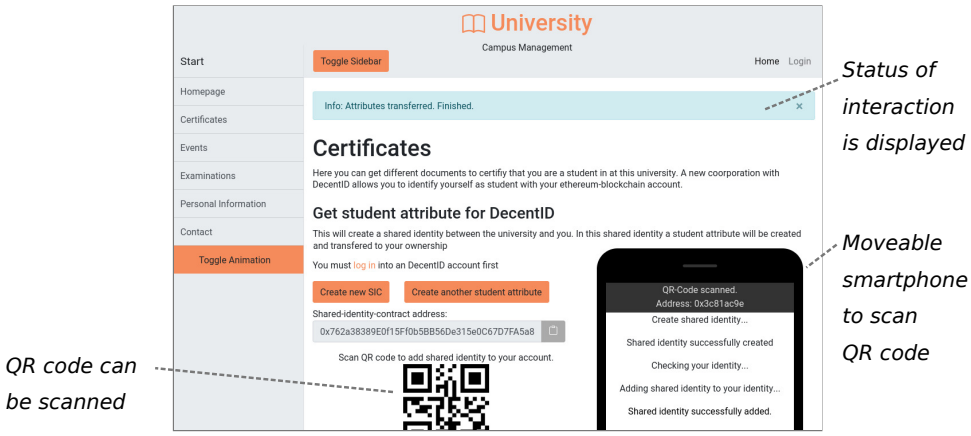

Figure 3.11: *The web interface to receive an attribute from the "university".*

Depicted in figure 3.11, it allows students to automatically create the shared identity displayed in figure 3.10. To do so, the user of the demonstrator has to "login" at the

university website, and scan the provided QR code[4] with a virtual smartphone that is part of the website. When doing so, a new SharedIdentityContract between the user and the university is automatically created. This identity contains a number of attributes created by the university, for example the matriculation number of the student and their course of studies, as displayed above.
While simplified for the demonstrator, a real implementation could follow the same sequence. The smartphone application would create a SharedIdentityContract for the user and add the public key of the university, which is contained in the QR code, as a permitted user to it. Then, the application can inform the university webpage about the new shared identity by sending its blockchain address to the URL also contained in the QR code. The university is then able to attach the relevant attributes to the new identity of the user.

3.8.1.4 Online shop

As a second web application an “online shop” has been implemented. For once, this online shop allows its visitors to login by providing a DecentID identity with the online shop as a permitted user. The creation process of this identity and also the following login processes can be handled as described for the university by scanning a QR code.

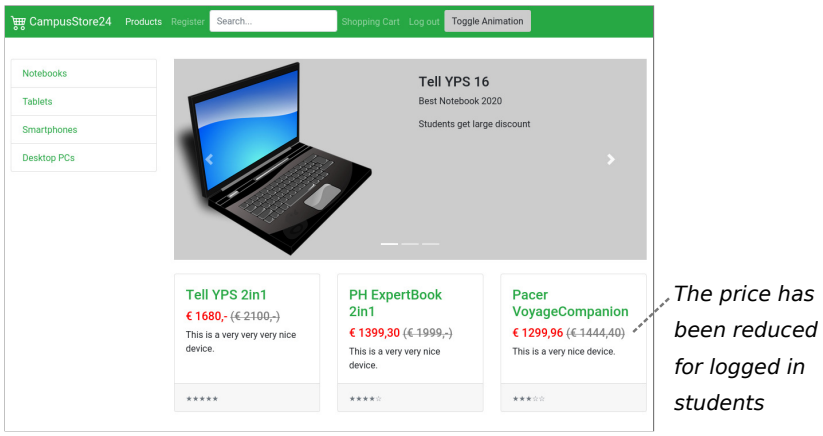

Figure 3.12: *The web interface of the “online shop” with reduced prices for students.*

After creating the identity, the user can login at the online shop but will only see the normal prices for the wares. However, the user can add the “student” attribute granted by the university to the SharedIdentityContract between the user and the

[4]https://www.qrcode.com/ Accessed: 11.03.2022

online shop. This discloses towards the shop that the user is a student of the university, but does not disclose the real identity of the user. Afterwards, the online shop can verify that the user is a student, and can offer a student discount for its wares, as is depicted in figure 3.12.
When the “student” attribute is displayed towards the online shop, the online shop learns that the user is a student at a certain university. While the real identity of the user is kept secret, the online shop still learns something about the user. Whether the loss of privacy, e.g., disclosure of the student status, is worth the advantage gained, e.g., a price reduction in the online shop, has to be decided by the user.

3.8.1.5 Visualization

While the other web applications are demonstrating how DecentID could be used in practice, this application displays the data on the local blockchain, visualizing the created identities. It cannot be used on the official blockchain of Ethereum, since it requires access to the private keys of the users which created the identities.

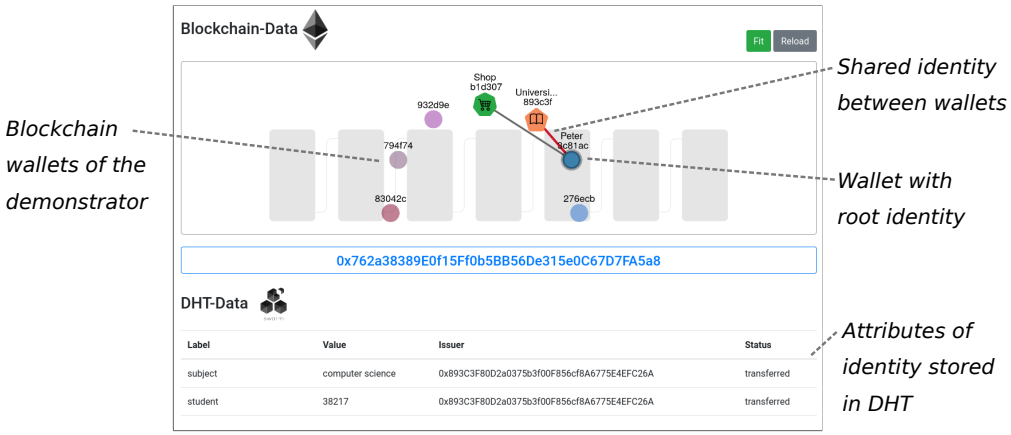

Figure 3.13: *A visualization of the identities currently stored on the local blockchain.*

The display as depicted in figure 3.13 consists of two parts: In the upper half the state of the local Ethereum blockchain is presented, while in the lower half the attributes of the selected SharedIdentityContract are shown, which are stored in the distributed hash table Swarm. In the upper half, the blockchain wallets created for the demonstrator are presented, placed in a circle and labeled with there shortened public keys. Only the wallet owned by the user “Peter” has been used and named yet. With it, two SharedIdentityContracts, respectively with the university and the online shop, have been created. This is depicted by the lines connecting the wallet of the user with the wallets of the two web services. The SharedIdentityContract with the

university is currently selected, with its blockchain address being displayed in the middle of the screen. The attributes linked to it are stored in the distributed hash table Swarm, and are listed in the lower half of the screen.

3.8.2 Smartphone application

The second implementation of DecentID is a mobile smartphone application. It allows to manipulate the owned identities of the user, similar to the identity management interface of the demonstrator.
The application has been implemented under supervision in the bachelor thesis of Evgeni Cholakov [Cho20]. It is written in Kotlin[5], the programming language currently recommended by Google for Android applications. To interact with Ethereum the web3j[6] library is employed. For attribute storage in the distributed hash table Swarm, the library Retrofit[7] is used.

3.8.2.1 Architecture

Different to traditional software, the applications running on smartphones have to be strictly partitioned in a user interface and the model. Due to constrained memory and processing power on smartphones, the user interface parts might be destroyed by the operating system at any time. To avoid data loss in these events, all data has to be stored within the model.

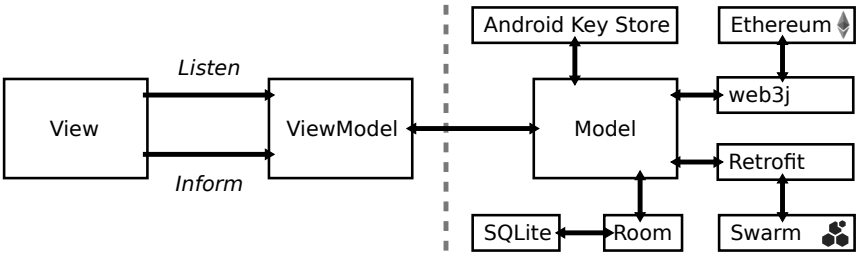

Figure 3.14: *An overview over the software architecture of the smartphone application. The components on the left side might be destroyed by the operating system at any time.*

As displayed in figure 3.14, the user interface consists of the View and the ViewModel. The View contains the presentation logic of the application and displays the data to the user. Additionally, it handles the user input and forms requests for the ViewModel.

[5]https://kotlinlang.org/ Accessed: 11.03.2022
[6]https://docs.web3j.io/ Accessed: 11.03.2022
[7]https://square.github.io/retrofit/ Accessed: 11.03.2022

The ViewModel interacts with both the View and the Model. It maintains the state of the View, informs the View of data changes in the Model, and forwards change requests from the View to the Model.

The Model manages the application data. In the DecentID application, not much data is stored on the smartphone itself. To store the cryptographic keys needed for DecentID, the Android key store is used. This way, the keys are kept secret even from other applications running on the smartphone. For other data, e.g., the blockchain address of the RootIdentityContract, an SQLite database is used. To access it, the library Room is used.

To interact with the blockchain, the web3j library is used. Besides communicating with the blockchain, it also offers local classes to represent the smart contracts on the blockchain. The distributed hash table Swarm is accessed over its HTTP interface, by using the library Retrofit for HTTP communication.

3.8.2.2 Usage

In the following the usage of the application will be explained, based on the four screens shown in figure 3.15.

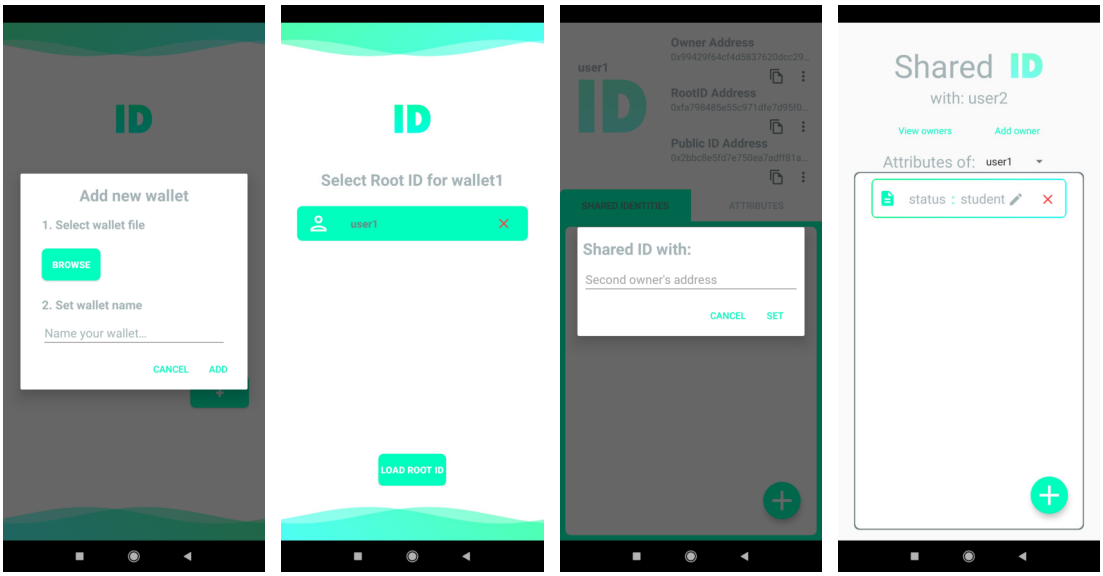

Figure 3.15: *Four screens of the smartphone application for DecentID: creating an Ethereum wallet, selecting the RootIdentityContract, creating a SharedIdentityContract, viewing an existing SharedIdentityContract containing an attribute.*

When starting the application, an Ethereum wallet has to be selected or created. To use DecentID, this wallet must contain funds to pay for the blockchain transactions. Sending funds into the wallet is not supported by the DecentID application, since this would require interaction with financial institutions to transfer money. The first screen displayed in figure 3.15 shows the creation of a new blockchain wallet by selecting a file to store its data in and entering a password for its encryption. The private keys used for DecentID are created and stored in secure storage provided by the operating system on the smartphone.
After loading the wallet, a new RootIdentityContract can be created for the selected blockchain wallet, or the existing one can be selected. Selecting an existing RootIdentityContract is shown in the second screen.
Referred to from the RootIdentityContract, a number of SharedIdentityContracts can be maintained by the user. When creating a new identity, as seen in the third screen, the blockchain address of a user to share it with has to be entered.
The last screen in figure 3.15 shows an existing SharedIdentityContract. It is currently shared with one other user, and the user that created the identity added one attribute to it. The attributes added by the other user can be displayed as well.
While the first implementation, the demonstrator, is running on normal computer hardware, the hardware of a smartphone is much more restricted. Still, the application has shown that DecentID is usable even on less powerful devices. The operation with the longest execution time, that is executed on the smartphone itself, is the loading of the blockchain wallet, which took around 6 seconds. All further operations, e.g., creating identities and attributes, are no longer restricted by the computational capabilities of the smartphone, but are instead delayed by the network and the blockchain speeds. As such, these delays cannot be avoided, independent of the implementation or hardware used.

3.9 Evaluation

Since DecentID is based on smart contracts on a public blockchain, a “traditional” evaluation of quantitative features like response time, throughput, or memory consumption is not useful. These features are directly inherited from the used blockchain Ethereum and are not or only marginally influenced by DecentID. For example, on average each 13 seconds[8] a new block is published on Ethereum. The speed of changing an attribute in DecentID is directly linked to publishing new blocks, independently of how efficient the smart contracts themselves are written.

[8] https://ethereum.org/en/developers/docs/blocks/#block-time Accessed: 11.03.2022

Instead, qualitative features of DecentID are discussed in the following. Based on the design goals presented in section 3.2, various aspects are evaluated and their contribution towards the goals are described. Afterwards, practical considerations for using DecentID are considered.
Another factor for the security and privacy of DecentID are the applications and the end systems that are used to interact with DecentID. If the used application or the end system contains security vulnerabilities that can be abused by an adversary, the security of DecentID itself is irrelevant. However, evaluating the security of all these systems, e.g., applications, smartphones, or computers, is out of scope for this thesis.

3.9.1 Control over the identities

Users should be in complete control over their digital identities. In how far this is the case within DecentID will be discussed in the following. This includes whether anyone else can access or modify their identities, as well as whether trust into a single instance is required. Additionally, it will be discussed whether permitted users can trust the authenticity and integrity of the linked attributes. Lastly, the removal of attributes, permitted users, or whole identities is discussed.

3.9.1.1 Properties of smart contracts

When developing and executing smart contracts, many aspects are similar to traditional software. Still, some aspects regarding their security have to be considered, which are discussed in the following.

Unmodifiable code Different from traditional software systems, the openness of public blockchains allows all participants to review the code of smart contracts and verify its execution. While this simplifies finding weaknesses for adversaries, it also means that the code can be easily reviewed. This is especially important since smart contracts on the blockchain cannot be overwritten by newer versions of them. If a bug in an already deployed contract c is found, a new contract c' has to be deployed at another blockchain address. Since this disrupts existing uses of the old contract c, it should be avoided as much as possible. Consequently, smart contracts should be extensively tested and analyzed before deploying them to the blockchain.

Verified execution When a function f of a smart contract is executed on the blockchain, the miners of the blockchain execute the function and store the results r of the execution in the next published block. While only the block created by a single miner will become part of the blockchain, all other miners will verify the correct construction

of this block. This includes that the function f must have been executed as expected and all miners reach the same result r. If their execution deviates from the expected execution based on the known smart contract code, the input parameters, and the current state of the blockchain, the other miners will not accept the published block. Consequently, if an execution result of a smart contract ends up on the blockchain, the majority of miners agrees that the smart contract was executed as it should be. As such, users of smart contracts, and consequently DecentID, can be sure that the contract has been executed as it is written.

State-dependent execution The result r of calling a function f in a smart contract depends on the function parameters p as well as the state s stored in the smart contract: $f(p,s) \rightarrow r$. However, the blockchain is a distributed system and the state of the smart contract might change before the function call is executed: $f(p,s') \rightarrow r'$. In that case the result of the function might be different than the user calling the function expected. This can also happen if the same user calls two functions nearly at the same time. Even if the function call $f(p,s)$ is done before the function call $f'(p',s)$, the order of execution is not fixed until one of the results is stored on the blockchain. As such, it can happen that $f'(p',s)$ is executed first, modifying the state of the smart contract and leading to for the user unexpected results when f is executed on this changed state.
When modifying a SharedIdentityContract, this can happen as well. However, only permitted users are able to call modifying functions of the smart contract. Additionally, each permitted user U_i can only modify the attributes they added themselves. As such, their function calls operate on different states s_i, even when they are all operating on the same SharedIdentityContract: $f_{U_1}(p,s_1)$ does not conflict with $f_{U_2}(p',s_2)$. Still, the function calls of a user U_i can conflict with earlier function calls of themselves if both function calls modify the same state, e.g., the same attribute.

The above explained properties of smart contracts together with the program code of DecentID ensure that only the permitted users have control over the respective SharedIdentityContracts. Due to the decentralized architecture of blockchains, this can be achieved without relying on centralized instances or trust anchors.

3.9.1.2 Authenticity and integrity

In DecentID, the privacy of the stored data is ensured by encrypting it, while its authenticity and integrity is ensured by checking the authenticity and authority of the user that wants to modify an attribute. The access rights for the functions in the SharedIdentityContract can be differentiated between three user groups: unknown

users, that are not listed as permitted users within the shared identity, permitted users of the identity, and the creator of the identity. Unknown users, which are not registered in the SharedIdentityContract, are not permitted to call any modifying functions and are unable to modify the state of the contract.

Checking authorization Some functions of the SharedIdentityContract can only be called by permitted users registered in the smart contract. An example for this is the function `setAttribute()` to write attributes. To check the access rights, it is checked whether the public key $pubKey_C$ of the caller C, representing their identity, is stored in the SharedIdentityContract. Only when $pubKey_C$ is known as a permitted user is C permitted to execute the protected function `setAttribute()`. Since the instruction to call the function is digitally signed with the private key $privKey_C$ belonging to the public key $pubKey_C$, the miners of the blockchain can verify that the caller C has access to the private key $privKey_C$ and, consequently, really is the permitted user stored in the SharedIdentityContract. If an unpermitted user tries to call this function, they are unable to create the required signature and cannot prove that they own a permitted public key. Consequently, their access to the function is denied.

Separation of permitted users Within the SharedIdentityContract, multiple lists $lAttr_{U_i}$ of attributes, separate for each permitted user, are stored. By using the public key $pubKey_C$ of the caller C as index into the list of permitted users, only the attributes in this users attribute list $lAttr_C$ can be written. Both measures, checking whether the caller is a permitted user and accessing the right entry in the list of permitted users, are enforced by the program code of the smart contract. The miners ensure that this code is executed as written, consequently ensuring that the linked attributes can only be modified by their respective user.

Integrity of data Since an append-only blockchain is used the attributes in published blocks cannot be modified, ensuring the integrity of the data in the smart contract. As such, other permitted users can be sure that the attributes have been added by the respective users and that no-one else was able to modify them.

Permissions of the creator In a SharedIdentityContract the creator of the identity is listed as the first permitted user. As a permitted user, they are able to modify their own attributes, but are unable to modify the attributes added by other permitted users. Additionally some functions, most importantly to add and remove permitted users, can only be executed by the creator. As explained before, this is ensured by checking the public key of the user calling the function. Assuming that the user creating the SharedIdentityContract keeps their private key secret, they are the only person able to grant and revoke access to the identity and its attributes.

3.9.1.3 Removals

In addition to appending and modifying attributes and the list of permitted users, it is also possible to remove attributes and permitted users, or even the complete shared identity. After some general remarks, these three cases are discussed.

Removing from the blockchain In general, removing data from blockchains, and consequently DecentID, is problematic. While it is no problem, e.g., to remove an attribute from the current state s_i of a SharedIdentityContract, all older states $s_{i-1}, s_{i-2}, \ldots$ of the smart contract can be recovered from the blockchain. Compared to other digital storage systems, this is a pitfall the user is most likely unaware about. To mitigate this problem, some approaches should be taken if DecentID is widely deployed. For once, the identity management application used to access DecentID should warn its user about this issue. However, for most use cases of DecentID storing some attributes in the shared identity cannot be avoided. Instead of storing attributes on the blockchain itself, they could be stored in an off-chain storage which permits removal of data. While this can improve the privacy of the attributes, other smart contracts can no longer retrieve these attributes, which might be required depending on the use case of the identity.

Removing attributes When removing attributes, they are removed from the current state of the identity but, as explained above, older states can still be reconstructed. Still, when permitted users read the attributes of the identity the removed attribute is no longer part of the identity. As such, the permitted users know that the data in the removed attribute should no longer be considered part of the shared identity.

Adding additional permitted users It can happen that a user removes their attribute $Attr_i$ and later on adds a new permitted user U_j to the SharedIdentityContract $SIC_{\{A\}}$. This new user U_j should not be able to access the removed attribute $Attr_i$, but would still be able to do so by reconstructing the older state of the smart contract. If the symmetric attribute encryption key $kSIC_{\{A\}}$ has not been changed in the meantime, the newly added permitted user U_j could use the attribute encryption key, which they now have access to, to decrypt the attribute $Attr_i$ that was linked to the identity at an earlier time. To change $kSIC_{\{A\}}$, all linked attributes have to be re-encrypted with the new attribute encryption key. After re-encrypting the attributes, they have to be written to the blockchain or the off-chain storage again. Additionally, since each permitted user is only able to modify the attributes they linked to the identity themselves, this requires cooperation with all other permitted users. Alternatively, a new SharedIdentityContract $SIC_{\{B\}}$, with a new $kSIC_{\{B\}}$, could be created, instead of continue using the existing identity $SIC_{\{A\}}$.

Removing permitted users Removing a permitted user from the shared identity leads to a similar effect. The removed user no longer has access to the encrypted $kSIC_{\{A\}}$ stored within the SharedIdentityContract $SIC_{\{A\}}$. However, if they kept a copy of $kSIC_{\{A\}}$ locally or are accessing an old status of $SIC_{\{A\}}$ on the blockchain, they are still able to decrypt the attributes linked to the SharedIdentityContract. For the attributes that were already linked at the time of removal this is most likely not problematic: The (then still permitted) user could have already read them at that time. When new attributes are added to the identity, however, the removed user might still be able to decrypt them. To avoid this, $kSIC_{\{A\}}$ has to be replaced and all attributes have to be re-encrypted, or a new SharedIdentityContract has to be created.

Removing a shared identity In some cases, the creator of a SharedIdentityContract might want to remove their shared identity completely. While a functionality for this is offered, the persistent history of the blockchain means that the smart contract could be reconstructed. As such, the removal of a SharedIdentityContract only shows the intent of the user that the identity should no longer be used, but does not stop previously permitted users from accessing the linked attributes.

In summary, the creator of a SharedIdentityContract is able to remove data from it, but it is mostly a declaration of intent. However, this is similar as with traditional, centralized identity providers: When a user removes data from their identity, they have to trust the identity provider to really remove the data and not keep any copies. Regarding the removal of attributes or permitted users and the following addition of new users respectively new attributes, this does not occur with traditional identity providers. Still, a malicious identity provider could forwards these data without the users consent. At least, in DecentID this potential privacy leak is visible, and can be avoided when a DecentID application informs the user of the privacy risk and suggests the above mentioned mitigation approaches, i.e., re-encrypting attributes and creating separate shared identities.

3.9.1.4 Comparison with the state of the art

In table 3.4 the most relevant related works are compared to DecentID. Some of them, e.g., uPort, have been selected for comparison since their design is similar to the design of DecentID. Other approaches, e.g., Sovrin, use a different technical approach but want to reach similar goals as DecentID with regard to the identity management and the privacy protection. However, all compared approaches claim to provide a decentralized, blockchain-based, system. The compared approaches, as well as further, less similar, approaches, are described in section 2.5.3.

Approach	Full Control	Decentralized	Integrity	Authenticity	Removable
DecentID	√	√	√	√	√
uPort	√	(√)	√	(√)	(√)
Sovrin	(√)	(√)	√	√	√
ShoCard	X	X	√	√	(√)
[MDS19]	X	(√)	√	√	√
[SP18]	√	√	√	√	√
Tawki	√	√	√	√	√
3BI-ECC	X	X	√	(√)	X

Table 3.4: *Comparison with related work regarding the control of the user, depicting whether the approach fulfills the requirement completely √, mostly (√), or does not X fulfill it.*

For self-sovereign identities it is important that the user is in full control over their identity and its data. However, for ShoCard and [MDS19] this is not sufficiently supported, since the user requires other, hierarchically appointed, entities to receive attributes for their identities. For a user this means that it is not possible to receive attributes from any service that might want to grant an attribute to them. Additionally, this restricts the use of the identities to the contexts the maintainers of the approaches permit. Within Sovrin the distributed ledger is maintained by selected entities, while in 3BI-ECC a centralized middleware software is used to forward requests to the blockchain. In both cases the access of users to their own identities might be restricted. In the other compared approaches, including DecentID, the user is able to control their identity without another entity being able to restrict them from doing so.

Both ShoCard and Sovrin contain centralized control by their respective organizations, ShoCard for storing attributes, Sovrin for selecting participants for its permissioned blockchain. While uPort does not depend on centralized off-chain entities, a centralized registry smart contract is used for linking attributes. This contract could become a central point of failure, either through attack from malicious users or when its creator decides to remove it. Furthermore, the existence of multiple of these registry contracts could result in multiple, separated instances of uPort which are unable to accept their mutual identities. For 3BI-ECC the middleware software is an undesirable point of centralization. Compared to these approaches, DecentID does not contain any centralized instances and all identities and interacting users and services can operate independently.

All compared approaches ensure the integrity of the stored attributes. The integrity is ensured by storing the attributes or, when off-chain storage is used, their hashes on the blockchain. Since the data on the blockchain can only be modified by authorized users, other users accessing the data can be sure that the data is still the same as when the user stored it on the blockchain.
Regarding the authenticity, most approaches support proving who created an attribute. While most approaches simply store the identity of the attribute creator, DecentID improves over this by storing the identity creator and additionally assigning the current owner of the attribute. Exceptions are uPort and 3BI-ECC. There, only the creator of the identity can attach attributes to it. As such, the creator of an attribute has to use additional measures, e.g., an attached digital signature, to prove that they created the attribute. The attribute data combined with the signature can then be added as an attribute by the creator of the identity.
If an identity or some of its attributes are no longer needed, a user might want to remove it. This functionality is supported and under the users control in nearly all compared systems. For uPort, it is unclear whether a user can remove their identity entry from the registry contract. However, removing the linked attribute data is possible, which means that the users data can be removed, even when the previous existence of the identity can still be discovered. With ShoCard, the identity is stored on the centralized server used by the system. As such, the user can request that their identity is removed, but has no direct control whether this really happens. Removing attributes or identities is not considered in the description of 3BI-ECC.

3.9.2 Privacy of identities

When evaluating the privacy protection supported by DecentID, multiple aspects are considered in the following. For once, the confidentiality of the attributes on the blockchain has to be ensured. Also, a shared identity should not permit others to find out which person created it. Additionally, the risk of services passing on the private identity data is discussed.

3.9.2.1 Confidentiality of attributes

DecentID is based on smart contracts on Ethereum, which is a public blockchain. As such, all data on the blockchain and all transactions send to it are visible to all interested parties. Additionally, external storage might be linked to the shared identities. The data stored in this storage should be accessible by permitted users of shared identities. Consequently, this data has to be public as well, but might,

depending on the used storage, at least be hidden by a hard to guess address[9]. In the following, the existence of a SharedIdentityContract $SIC_{\{U\}}$ with the symmetric encryption key $kSIC_{\{U\}}$ is assumed. $kSIC_{\{U\}}$ is stored encrypted in the smart contract and can be decrypted with the private keys $privKey_{U_1}$ and $privKey_{U_2}$, which are owned by the two permitted users U_1 and U_2 of $SIC_{\{U\}}$.

Encryption of attributes Since DecentID manages personal data of its users, the protection of their privacy is an important design consideration. In general, all attributes should be encrypted before they are stored, independently of whether they are stored on-chain or off-chain. On-chain attributes are encrypted with $kSIC_{\{U\}}$, and can be decrypted by one of the permitted users U_1 or U_2. Other, potentially malicious, users have no access to $kSIC_{\{U\}}$, since they have no access to $privKey_{U_1}$ or $privKey_{U_2}$. Consequently, they are unable to decrypt the attribute. For off-chain attributes, $kSIC_{\{U\}}$ is used to encrypt the reference to the AttributeLocatorFile $ALF_{\{U\},i}$, as well as $ALF_{\{U\},i}$ itself. As for on-chain attributes, an adversary has no access to $kSIC_{\{U\}}$ and thus is unable to find nor decrypt $ALF_{\{U\},i}$. Whether an attribute is encrypted is decided by the user adding the attribute, with the decision being stored in the flags of the attributes.

Encryption of indexing keys For the indexing keys of the attributes there is no flag to describe their encryption status. Instead, it is up to the application using DecentID to decide whether an indexing key $Index_i$ is encrypted. Adding a flag for it would bring no advantage: The flags can only be accessed after the attribute has been looked up using the indexing keys. Consequently, a permitted user would have to look up the attribute with both the unencrypted $Index_i$ and the encrypted indexing key $enc_{kSIC_{\{U\}}}(Index_i)$ to retrieve the flags and find out whether encryption was used. As described in section 3.5.2, the indexing keys are not stored in the SharedIdentity-Contract. While this stops an adversary from receiving the list of used indexing keys from the current state of the smart contract, they are still able to fetch the blockchain transactions that added the attributes, retrieving the indexing keys from them. So while the confidentiality of the attribute data can be ensured, the metadata, which attributes exist, can be discovered by a determined adversary. If this is problematic, off-chain attributes can be used. There, the AttributeLocatorFile $ALF_{\{U\},i}$ stores the indexing keys $Index_i$, encrypted by $kSIC_{\{U\}}$. As such, they are not transmitted over the blockchain, keeping them confidential.

[9]For example, this approach is used by Google Docs: Shared files can be made available without authentication, but their address is randomly generated and quite long, making it unlikely that someone manages to discover it.

Unencrypted attributes For some attributes it might not be practical to store them in encrypted form. When the value of the attribute should be automatically evaluated by another smart contract, e.g., to verify an authorization token stored as an attribute, the attribute cannot be encrypted. If the verifying smart contract is able to decrypt the attribute, then it would leak the secret key material for decryption on the blockchain. Afterwards, everyone else is able to decrypt the attribute as well. This is the case independently of whether symmetric encryption with $kSIC_{\{U\}}$, a new symmetric encryption key, or asymmetric encryption with a new key pair is used. In all cases the key required for decryption would be visible on the blockchain. As such, the encryption of the attribute does not improve its confidentiality, if a smart contract needs to access its unencrypted data.
While the attribute data has to be readable, the information contained within it can be limited to exactly what is required for its purpose. For example, depending on the use case it can be sufficient to prove that a certain user is an employee of a certain company. Disclosing their name or any other information about the employee is not required, and can be omitted from the attribute. Alternatively, verifying the attribute outside the blockchain could be done without disclosing the secret key material. However, this does not help in the scenario where a smart contract wants to verify an attribute, since smart contracts are unable to access data outside the blockchain.

3.9.2.2 Identifying users

The idea behind using pseudonymous identities on the Internet is that online services are unable to determine the real-world, offline identity of a user. To reach this goal, the pseudonymous identity must not contain information that allows to identify a single human.

Attribute combinations Unfortunately, determining which information are critical for identification is hard to decide. In some cases, the combination of non-identifying data can result in identifying a single human. Assuming the existence of three attributes: attribute A, the email address of the user, attribute B, the home address of the user in a large building, and attribute C, the users work address. If an adversary discovers attribute A, the user can most likely be identified since most email addresses are only used by a single person. However, neither attribute B nor C are sufficient to identify the user by themselves, if only one of them is given. On the other hand, if both attributes B and C are known, it is quite probable that only a single person remains where both attributes are correct. This identification can happen with every identity system, including DecentID. A smart user interface can try to warn the user about this privacy risk and can even try to find identifying attributes, e.g., by trying to

recognize email addresses and attribute combinations. However, this most likely will not work in every case due to the many different connections that can exist between individual attributes.

Cryptographic keys Specific to blockchain based systems is the asymmetric key pair, $pubKey_{U_i}$ and $privKey_{U_i}$, representing a blockchain user U. Continued use of one key pair i can allow adversaries to track the actions of the user. Depending on what the user U is doing, this might lead to their identification. Consequently, each key pair should only be used for a single purpose, e.g., one shared identity, or a limited time. Since new key pairs can easily be created locally without interacting with other users or systems, this does not pose a problem.

Transferring funds Another privacy risk is caused by the financial properties of blockchains. To send transactions through the blockchain, either to transmit money or to call a function of a smart contract, Ether is required. This Ether has to be obtained previously, often by paying government-issued money at cryptocurrency exchanges. If the money is paid from a regular bank account, the Ether transaction can be traced back to the real-world identity of a user. This is a general problem when trying to interact with blockchains anonymously. Different approaches have been proposed to obfuscate the origin of funds on blockchains [Liu+18; Xia+21]. Depending on the use case for a users DecentID identity, a link to a regular bank account might not be a problem. Services on the Internet, which the user interacts with, can only see that the Ether was send to the users blockchain account from the blockchain account of the cryptocurrency exchange. However, the services cannot find out which bank account was used to pay for the Ether at the exchange, nor the identity of the bank account owner. Neither the cryptocurrency exchange nor the bank are able to uncover all steps of the link either. So to link a DecentID identity to a real-world person, multiple entities, the bank, the exchange, and the service, have to cooperate. This is unlikely to happen, since the entities have different interests in the user and are additionally often bound by legal contracts, forbidding them to pass on the private data of the user.

Interacting users In some scenarios, it might be interesting for an adversary to find out whether two users U_A and U_B interact with each other, i.e., have a SharedIdentity-Contract $SIC_{\{A,B\}}$ where both of them are permitted users. Whether the adversary is able to do so, depends on multiple factors.

- In the most privacy preserving case, both users created new asymmetric key pairs for the SharedIdentityContract $SIC_{\{A,B\}}$, and these new key pairs cannot be linked back to real-world persons. In that case, the adversary can only observe

that two users are interacting, but cannot say anything about the identities of the users. Still, care has to be taken to not disclose the identities, e.g., by storing a street address as an unencrypted attribute.

- If the user U_A is a publicly known service, its public key $pubKey_{U_A}$ is known to the adversary as well. Assuming that the other user U_B uses a new key pair, the adversary only knows that someone interacts with the service, but not who it is.
- The same is true if the identity of one user is known for some other reason, e.g., due to earlier uses of the key pair.
- Only when the identities of the owners of both key pairs are known to the adversary, the adversary can be sure that the two users U_A and U_B interact. For example, the key pairs might be known to the adversary due to previous uses of the respective key pairs.

As such, an adversary cannot detect the interaction of two users U_A and U_B, as long as atleast one of them uses an asymmetric key pair unknown to the adversary.

3.9.2.3 Passing on data

A problem that cannot be solved technically is that permitted users with access to a shared identity $SI_{\{X\}}$ might make local copies of the linked attributes $Attr_{\{X\},i}$. Even more concerning, the retrieved data could be passed on to other users or unknown third parties. When two online services A and B are cooperating, sharing the private data of their users might allow them to identify a user U by using a unique attribute, e.g., their e-mail address, and establish that two shared identities $SI_{\{X\}}$ and $SI_{\{Y\}}$ on their respective services A and B are controlled by the same human user U.
While undesirable, this same problem exists for all other data stores as well, be it on the blockchain, with centralized online providers, or even with offline services. Technically, this possibility cannot be inhibited when someone is able to access private data. As a partial solution, legal contracts and terms of business might forbid passing on the data, but the user has to trust the used services to observe their contracts.

3.9.2.4 Comparison with the state of the art

When incorporating blockchains in the design of an identity management system, privacy is an important challenge to consider. While data stored within a smart contract can be removed from the current state of the smart contract, all its older states can be restored. As such, data written to the blockchain once will stay accessible permanently and have to be protected. For large attribute data and to avoid this permanent

Approach	Encrypted attributes	Encrypted metadata	Smart contract interactions	Selective disclosure	Interactions hidden
DecentID	√	(√)	√	√	√
uPort	√	X	X	√	√
Sovrin	√	(√)	X	√	√
ShoCard	√	√	X	√	(√)
[MDS19]	√	(√)	X	√	√
[SP18]	√	X	X	√	√
Tawki	√	√	X	√	√
3BI-ECC	X	X	X	√	(√)

Table 3.5: *Comparison with related work regarding their privacy protection.*

record, using other storage is supported by all compared systems. However, whether the data is really removed when requested remains an open question. Additionally, ShoCard and Sovrin use centralized storage providers, which have to be trusted to operate as expected. Independent of the used storage, nearly all systems compared in table 3.5 are using encryption to protect the data of their users. An exception is 3BI-ECC, where encryption is not considered in the design.

Whether metadata is encrypted differs between the approaches. Metadata here means for once the indexing keys used to access a specific attribute, but also other administration data, e.g., references from the identity to the attributes. In ShoCard and Tawki, the attributes are stored off-chain together with their encrypted metadata. As long as every identity links to own off-chain data, no metadata is leaked. In uPort the attributes and their metadata are stored off-chain as well. While the attribute itself is encrypted, the metadata containing additional information about the attribute is stored unencrypted, which can give an adversary clues about the type of the stored data. For other approaches, the metadata is partially encrypted. In DecentID the metadata is encrypted when the attributes are stored off-chain. If the attributes are stored on-chain, the blockchain transactions contain the indexing keys for the on-chain attributes.

While the visible indexing keys in DecentID are undesirable from a privacy point of view, they are the result of a design decision. With DecentID, a user can explicitly decide to store an attribute unencrypted on the blockchain. To protect their confidentiality, the attributes should normally be stored encrypted. However, when storing an attribute unencrypted, other smart contracts are able to interact with the

shared identity to retrieve this attribute and use its data, e.g., for access control. This feature is not supported by the other approaches in this comparison. Mostly, the other approaches store their data completely off-chain, making it impossible to retrieve it within a smart contract. The attributes that are stored on-chain are always encrypted, stopping smart contracts from reading them.
In all compared approaches it is possible to selectively disclose attributes to other user. This is done by granting other users access to an attribute or a whole identity, mostly by sharing an attribute encryption key,
Except with ShoCard and 3BI-ECC, disclosing identities to other users can not be observed by third parties. That two users are accessing one identity might be known, e.g., by observing blockchain transactions. However, the access to single attributes within a shared identity cannot be observed. With ShoCard and 3BI-ECC, the interactions between the users require access to the centralized server respectively the middleware software which manages the attributes for the users. As such, this central instance knows which attributes are accessed by whom.

3.9.3 Multiple pseudonymous identities

To improve their privacy, users might want to maintain multiple identities at the same time. For example, a user U might want to have separate identities $SI_{\{U,A\}}$ and $SI_{\{U,B\}}$ for different online services A and B. By doing so, they improve their privacy since each service receives less information about the user, as each service is only able to retrieve the attributes that are part of the respective shared identity. Even when the services A and B are cooperating, they should be unable to discover that the two identities belong to the same user U. By linking the identities $SI_{\{U,A\}}$ and $SI_{\{U,B\}}$, they would be able to assemble a single identity $SI_{\{U,A\cup B\}}$ containing all data about the user, which the user wants to avoid. Consequently, DecentID should prevent links between the two identities or their attributes. Some measures have been taken in the design of DecentID, while other measures should be included in an application to manage the identities, e.g., warning the user that AttributeContracts should only be reused if needed. Further measures have to be taken by the users themselves, for example determining which pieces of information might allow an adversary to link identities. In the following, potential vulnerabilities are discussed.
A special case of having multiple shared identities $SI_{\{U,A\}}$ and $SI_{\{U,B\}}$ for multiple services A and B, is having multiple shared identities $SI_{\{U,A\},1}$ and $SI_{\{U,A\},2}$ for a single service A. One use case for this is to maintain different identities for different discussion topics within one online forum. When analyzing the privacy protection provided by maintaining multiple identities this does not differ to interacting with

multiple services which cooperate with each other. In both cases the user wants to avoid that the single service or the multiple services can link their identities. However, when a service cannot find out that two identities belong to the same user, Sybil attacks on the service become possible. In these attacks, a single user creates many DecentID identities for a single service to abuse its functionality. This threat and a countermeasure to it is discussed below.

3.9.3.1 Explicit linkability

Explicit linkability exists if the user U adds an identifying attribute $Attr_{\{U\},i}$ to two shared identities $SI_{\{U,A\}}$ and $SI_{\{U,B\}}$. For example, such an identifying attribute might be an email address. In that case, the two services A and B can both read the address from the identity, and, if they are cooperating, find out that they both have a user with this email address. Most email addresses are only used by a single person, so two shared identities containing the same email address most likely belong to the same person. Consequently, the services A and B can assume that both shared identities have been created by the same user U. This allows them to create a more extensive profile for the user, which can be abused by the services, e.g., to track the users actions or display personalized advertising.
For the user, this is undesirable: The use case for creating separate identities is to avoid such tracking, but now the user has to maintain two identities without the desired gain of privacy. This explicit linkability cannot be prevented by technical means only, and a solution is out of scope for this work. An application for DecentIDs identity management could warn the user when the same attribute data is assigned to attributes of two identities. However, deciding whether the attribute is identifying for the user, e.g., their email address, or not, e.g., their hair color, is not always possible for an application. Additionally, some attributes are only identifiable in certain contexts. For example, the hair color might be an identifying attribute when only a few users are considered, but will no longer be identifiable when a larger user group is regarded. Consequently, an application can only warn a user but the user has to decide whether they want to use that attribute or not.

3.9.3.2 Implicit linkability

Another vulnerability is due to implicit linkability. When an AttributeContract or an off-chain storage is used to store an attribute A for the shared identity $SI_{\{U\}}$, it has a unique address $addr_A$ on the blockchain or in the external storage, respectively. This address $addr_A$ can then be used by two services to discern that two shared identities, $SI_{\{U\},1}$ and $SI_{\{U\},2}$, which link to the same attribute address, are controlled by the

same user U. As long as the data within the attribute does not identify the user, i.e., no explicit linkability exists, the user can copy the attribute A and the copy A' can be referenced to by the second shared identity. Copying the attribute can either be done by the user manually, or it can be automated by an identity management application for DecentID. After copying the attribute, the attribute addresses, $addr_A$ and $addr_{A'}$, are different and no longer allow the services to link the identities. This increases the overhead for the user since multiple attributes have to be kept updated and synchronized, but this can be simplified by an identity management application.

3.9.3.3 Trade-offs

To protect the privacy of a user, maintaining multiple SharedIdentityContracts between the user and a single service can be an advantage. Especially for larger services, e.g., a general discussion forum with many sub-forums, this can be preferable to maintain separate identities to discuss different topics. However, at the same time this introduces two vulnerabilities for the service.

Sybil attacks One vulnerability is that malicious users can execute Sybil attacks. In those, a single malicious user U creates numerous identities $SI_{\{U,S\},1}$, $SI_{\{U,S\},2}$, $SI_{\{U,S\},3}$, ... for a service S, without the service being able to recognize that they are from the same user. Those identities are then used to abuse features of the service, e.g., by participating many times in a poll. Sybil defense systems offer a solution for this problem by being able to recognize Sybil identities. One such system, Detasyr, is presented in chapter 5. By linking the "proof of authorization" p_A created by Detasyr, i.e., a proof that a user is no Sybil adversary, with a shared identity, each user is only able to create a single identity per online service. Consequently, Sybil attacks are no longer possible. The tracking protection of DecentID between multiple services is still working: While shared identities with the same service can be recognized by the service, even cooperating services cannot find out that a single user has shared identities with both of them.
Whether Sybil attacks are a problem for the service or whether the ability of the user to maintain multiple identities is more important, depends on the service in question. For example, a service providing a general discussion forum might not require a proof of authorization of its users. Its users are then able to create multiple identities $SI_{\{U,S\},1}$, $SI_{\{U,S\},2}$, ... for this forum, and the service is unable to discover that they are controlled by the same human. The threat due to the multiple identities can be considered low: There is not much to gain for an adversary by creating numerous identities, but for honest users it can be preferable to maintain separate identities for separate discussion topics. For a political discussion forum this threat can be

considered higher: A Sybil attacker could try to use its many identities to manipulate the public opinion of a party or politician. When a proof of authorization p_A is required, only one identity can be maintained for the service, which restricts the abilities of the adversary.

Blocking users Another vulnerability due to having multiple identities is that misbehaving users cannot be blocked from accessing a service. A single shared identity $SI_{\{U\},1}$ can easily be blocked, e.g., the service can attach an attribute describing the shared identity as blocked. However, even when the service is blocking one identity of a user U, the user can simply create a new identity $SI_{\{U\},2}$ and continue using the service. While some collected attribute data might be lost this way, it does not stop a malicious user from using the service. When a service requires that a shared identity refers to a Detasyr proof p_A, the service is able to block the user from continued use of the service. Since all shared identities of the user for one service use the same proof p_A, a second shared identity of the user can be recognized and blocked. To do so, the service has to keep a local list of blocked Detasyr proofs. When a user informs the service about their new shared identity, the service can check in the local list for the proof contained in the new identity, and block it directly if the user is already known as misbehaving.

3.9.3.4 Comparison with the state of the art

Most of the decentralized identity management systems compared in table 3.6 support maintaining multiple pseudonymous identities. However, while maintaining multiple identities was a design goal for DecentID, the other systems only support additional identities by creating a completely new identity, independent of the already existing identity of the user. Support for creating or maintaining these multiple identities is not part of their designs. For the systems that do not support having multiple identities, a trusted identity document, e.g., a government ID, has to be presented on identity creation. As such, maintaining pseudonymous identities is not possible. For these systems, this is acceptable since their intended purpose is a digital representation of the real identity of the user.
When a user maintains multiple identities, these identities should not be linkable to each other. Otherwise, an adversary could detect that they are maintained by the same user. If the user created the multiple identities to avoid tracking, their goal of privacy protection would not be fulfilled. For most systems which support multiple identities, this requirement is fulfilled. With uPort or Sovrin the identities might be linked. These systems support to recover an identity when the used private key is lost. To do so, the identity creator can register a group of friends that confirm that the new

Approach	Pseudonymous identities	Unlinkable identities	Transferring attributes	Sybil protection	Blocking users
DecentID	√	√	√	√	√
uPort	√	(√)	X	X	X
Sovrin	√	(√)	X	X	X
ShoCard	X	X	√	√	√
[MDS19]	√	√	√	X	X
[SP18]	X	X	√	√	√
Tawki	X	X	X	X	X
3BI-ECC	√	√	X	X	X

Table 3.6: *Comparison with related work regarding multiple identities.*

private key is from the same user. When having multiple identities, different groups of friends or different identities of the same friends should be used. Otherwise, the identities of the user can be linked since all of them the refer to the same friends. For ShoCard, [SP18], and Tawki, no unlinkable identities are possible since a trusted identity document is required for identity creation.

When using multiple identities, it can be of interest to the user to receive a certification of some kind from one service, and present it to another service. In DecentID, this is possible since services can grant attributes to users that have a shared identity with them. ShoCard, [MDS19], and [SP18] support a similar functionality, by allowing their users to integrate attributes created by participating services into their identities. For the remaining approaches, no such functionality is integrated by design.

DecentID avoids Sybil attacks while maintaining privacy protection through the integration of the Sybil defense system Detasyr. While doing so, the real identity of the user is kept secret. Contrary to that, ShoCard and [SP18] require a proof of identity, e.g., presenting a passport towards a trusted authority, on creation of a digital identity. This protects against Sybil attacks, but also links the digital identities to the real identity of their users, a linkage that has been avoided in DecentID. The other approaches do not offer any Sybil protection.

The protection against Sybil attacks and the ability to block a user from accessing a certain service are related: If the unlimited creation of identities is possible, the user can simply create a new identity to access a service again. Consequently, only the approaches which protect against Sybil attacks are able to permanently block a user from accessing a service.

3.10 Applicability

The previous section evaluated DecentID analytically based on its design. In this section, practical considerations are evaluated that influence whether and how an implementation of DecentID would be used in everyday life.

3.10.1 End user interaction

From a technical viewpoint, the blockchain-based DecentID is more complicated than a traditional centralized identity provider. For the user however, the used technology does not really matter. To manage their identities, users need some management application. Considering the amount of cryptographic keys and addresses used by DecentID, manual management of the shared identities is not feasible. Given the prevalent availability of smartphones, a smartphone application is a good candidate for this. This application can hide the complexity of DecentID from the user, offering a simple interface as the user is already used to from other identity providers.

3.10.1.1 Privacy protection

Compared to centralized identity providers, the improved privacy protection of DecentID can increase the difficulty of maintaining the shared identities. While the technical complexity can be hidden within an application, the user has to decide by themselves which attributes are added to which shared identities. To protect their privacy, the user has to make a conscious decision which attributes to add. When using multiple centralized identity providers for different services, the user has to face the same decisions. Still, using DecentID can make this easier since all identities can be managed in one application, instead of interacting with multiple websites.

3.10.1.2 Interaction with online services

While developing the demonstrator described in section 3.8, scanning QR codes with ones smartphone turned out to be a feasible approach to display ones identity to a service. The scanned QR code can contain a web address the smartphone can automatically access to provide the service with the blockchain address of the shared identity. If no shared identity for this service is present yet, the smartphone application is able to create a shared identity automatically and add the service as an owner to the identity. When the service receives the address of the users shared identity, it can verify that it is registered as a permitted user within it and that the attributes stored in the identity grant access to the service. Afterwards, the website displaying the QR

code can be automatically refreshed for the user to access the service. Compared to traditional website logins with username and password, the effort for the user is comparable, since the complexity of the identity management can be automated by the smartphone application.

3.10.2 Overheads

Compared to traditional systems, where a service provider stores the users data on its own server, DecentID generates a certain overhead for its operation, especially due to the use of a blockchain.

3.10.2.1 Writing to the blockchain

When modifying attributes, either creating or changing them, the state of the blockchain has to be updated. Since Ethereum only publishes new blocks around each 13 seconds, a significant delay occurs compared to a local database run by the service provider. Additionally, the transaction containing the change might not end up in the block published next but in a later one, increasing the delay. Even when the transaction has been added to a block, a user should wait for another few blocks to be added since the newest blocks might still be removed again. While the service might just assume that the attribute will be assigned and might pretend that it exists to improve the user experience, the user would not be able to see the attribute in their identity management application immediately.

3.10.2.2 Reading from the blockchain

When reading attributes from the blockchain, no delay due to block creation occurs. However, both the user and the service provider either need to request the data from blockchain participants, or keep a local copy of the blockchain. Which approach is preferable is a trade-off: When requesting the data, a trusted blockchain participant is required and a small delay occurs. If a local copy of the blockchain is maintained, a lot of irrelevant data is stored and the local state has to be constantly updated. While both approaches are possible for service providers, users will most likely prefer the first approach using a trusted blockchain participant due to storage restrictions on their smartphones. A point that should be noted is that with DecentID both the service provider and the user always have access to the most recent user data. In traditional systems using a local database at the service provider, the user has no direct access to their own data and can only access it through the service provider.

3.10.2.3 Cryptography

Another overhead is increased computation effort due to the used cryptography, when compared against a centralized data store under the control of a service provider. All data stored in the blockchain has to be encrypted to protect the privacy of the user. Additionally, sending transactions towards the blockchain requires the creation of digital signatures. Given that today's smartphones have enough processing power to easily do so, this should not pose any obstruction to the deployment of DecentID. Significant computation effort is required to create new blocks for the blockchain. However, to use DecentID creating new blocks is not required from the users, since other blockchain participants are already doing so independently of DecentID.

3.10.3 Cost considerations

A significant difference to traditional systems is the direct financial cost incurred by using a blockchain. In Ethereum, all calculations done by the miners have to be paid for. While ordinary Ether transactions or function executions are relatively cheap, writing data to a variable in a smart contract or, especially, deploying a new smart contract is expensive. Unfortunately, the dollar price for Ether has drastically increased in the last years and is significantly fluctuating.
In July 2021, deploying a SharedIdentityContract would cost around $80 - $300, depending on how fast the smart contract should be deployed to the blockchain. Obviously, this is much too expensive for most use cases. Compared to the current state, where creating an online identity only costs a few moments of time, it is unlikely that most users are willing to pay that much money. In comparison, adding a small attribute to an already existing shared identity cost only about one percent of the gas, which, while still expensive, is a more feasible cost. Using only a single shared identity with many attributes for multiple services would save money, but would abandon the privacy protection provided by using separate identities.
Using another blockchain instead of Ethereum to avoid or at least reduce the monetary costs is not advisable. Technically, DecentID is not bound to Ethereum and could operate on another blockchain supporting smart contracts as well. However, an important factor for the security of blockchains is the number of users who participate in mining blocks for the blockchain. If more than half of the mining power of a blockchain is controlled by one, potentially malicious, user, the blockchain can no longer be trusted. This user would be able to publish arbitrary blocks, invalidating the assumed security guarantees of the blockchain. A consequence of this is that it is a security advantage if many users participate in mining the blockchain. Using

Ethereum, the second largest blockchain by market value[10], as a basis for DecentID means that it is very unlikely that a malicious adversary is able to control most of the miners and bypass the security features integrated into the smart contracts of DecentID. Using a smaller blockchain, e.g., because a special purpose blockchain allows faster or cheaper execution, would forfeit this advantage. While some blockchains seem to be promising regarding their capabilities, they are currently supported by significantly less miners than Ethereum. If a blockchain is only controlled by a small number of entities, it is no more secure than a traditional centralized system.

3.11 Summary

DecentID is a self-sovereign identity management system based on the blockchain Ethereum. It gives users the control over their own identities, protects the privacy of its users, and allows to maintain multiple pseudonymous identities to access different online services. Decentralized identities can be created and arbitrary attributes added to them. For large attributes external storage can be used, to avoid the costs of writing to the blockchain. The added attributes can then be shared with online services, granting the service access to the linked attributes and allowing them attach attributes of their own. When doing so, the service is only able to read the attributes attached to the identity shared with it, but does not receive any information about other identities or attributes of the user.
The security and privacy features of DecentID were evaluated, and its advantages and limits were discussed. A comparison with the state of the art was conducted and DecentIDs unique characteristics were identified. Improving over existing work, the creation of multiple identities per user has been considered in the design and evaluation of DecentID, while related work only considers one identity per user. Attributes can be granted by other users or services, and can be presented to third parties. This is especially useful when interacting with other smart contracts, a feature of DecentID that is not supported by related work. However, this flexibility and increased privacy protection comes with increased costs for operations on the blockchain. Furthermore, aspects for the applicability of DecentID in practice were considered. Using a prototype smartphone application and a computer-based demonstrator, two implementations were presented.

[10] https://coinmarketcap.com/ Accessed: 11.03.2022

Chapter 4

Use Cases for DecentID

With DecentID, identities can be managed by their creator without a dependence on a centralized service. This allows users to maintain self-sovereign identities completely under their own control. However, when no service supports using these identities, DecentID is of no use. As exemplary use cases for DecentID two example scenarios are presented and evaluated in this chapter, where DecentID is coupled with existing systems. The first example is using the identities created with DecentID for access control in Palinodia, while as a second example DecentID is extended with the voting system Open Vote Network for improved decentralization of online services, by reducing the dependency on a single user. Both Palinodia and the voting system are implemented by using smart contracts, allowing to interact with DecentID without the help of off-chain programs.
These two examples have been chosen since their coupling with DecentID, as presented in chapter 3, is achieved by different approaches. For coupling with Palinodia, presented in section 4.1, DecentID does not have to be modified. The smart contracts of DecentID already support the required interface, namely the ability to retrieve on-chain attributes from other smart contracts. For the integration of the voting system, presented in section 4.2, DecentID had to be extended. The voting system is supposed to allow users to assign attributes to the SharedIdentityContracts of other users. Normally, this requires the private key of a permitted user of the target SharedIdentityContract. With the integration of the voting system, authorized users are able to execute a poll in the name of a permitted user. To enable the voting system to modify the attribute after a successful poll without knowing the private key, additional functions had to be added to the SharedIdentityContract of DecentID.

4.1 Coupling with Palinodia

By itself, an identity management system as DecentID is of limited use. Only when other services require identities and are able to interact with the identity management system it becomes useful. One possibility for such a service is an off-chain service that uses the identities. As such, the service is able to access the data stored in the smart contracts on the blockchain as well as the data stored in off-chain attributes. To do so, use case specific software is required for the service. In other cases, DecentID can be used by services running as smart contracts on the blockchain. How the smart contracts of DecentID can be coupled with other smart contracts, and what approaches are feasible to do so, is discussed in the following. As an example contract to discuss the possible coupling approaches Palinodia is used. However, the evaluated coupling approaches and the reached conclusions apply to other services implemented as smart contracts as well.

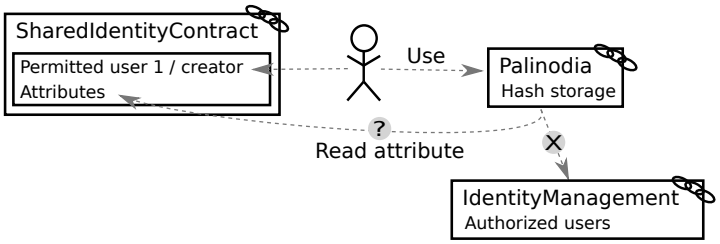

Figure 4.1: *Replacing the identity management of Palinodia with DecentID.*

Palinodia, which will be introduced in the following, requires identities to decide which users are authorized to update the stored hashes. Instead of using a specialized smart contract to store the identities, approaches to couple with the generalized identities of DecentID have been evaluated. This is depicted in figure 4.1. How these two systems can be coupled, that is, which smart contracts have to be added or modified, is evaluated.

After introducing Palinodia, the coupling approaches that will be evaluated are introduced. These approaches are evaluated based on multiple criteria. Afterwards, the finding will be summarized, both generally for coupling of smart contracts and for identity management systems, especially for DecentID.

An earlier state has also been presented in [Fri+21]. The design and implementation of the coupling approaches as well as the costs measurements were mostly done by myself, while the other parts of the evaluation and the discussion have been collaborative work.

4.1.1 Palinodia

Palinodia [Ste+19], designed by Stengele, et al., is a blockchain-based system to verify the integrity of downloaded software binaries. It allows to verify the binary integrity of downloaded software automatically, by calculating the hash of the local software binary and comparing it with the known hash stored in the blockchain. If a software version should no longer be used, e.g., because security vulnerabilities have been found, the respective binary hash for this version can be marked on the blockchain as no longer valid. When the user tries to run the software the next time, they are automatically warned that the used version should not be used.
Similar systems already exist based on centralized servers. However, if an adversary can take over the server providing the software hashes, they can report arbitrary hashes and even manipulate the user to update to an insecure version. Using blockchain as a storage for the hashes avoids this problem since its security guarantees protect against unauthorized manipulation, and avoids a single trust anchor.

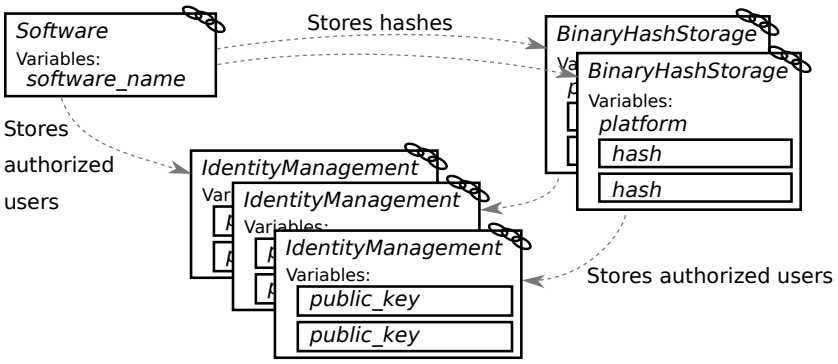

Figure 4.2: *The smart contracts of Palinodia and their interactions.*

Palinodia by itself consists of three smart contracts, which are described in the following. Additionally, their interaction is displayed in figure 4.2.

Software contract The software contract represents one specific software with all its versions on the blockchain. Also, it links to a number of BinaryHashStorage contracts that store hashes for specific platforms. Since not everyone should be able to add or remove links to BinaryHashStorage contracts, an IdentityManagement contract is used to store the public keys of permitted software developers.

BinaryHashStorage contract BinaryHashStorage contracts, one for each supported platform, store the hashes for the software binaries that were published. Such a platform consists of an operating system and the respective hardware the operating

system is running on. If a software version has been revoked, the respective hash stored in the contract is marked as revoked, resulting in future software verifications to fail. Only the software maintainers that are listed in a linked IdentityManagement contract are permitted to add or revoke hashes from the contract.

IdentityManagement contract Multiple of these contracts are used for a single software to form a simple role-based access control system. One contract lists the authorized software developers for a specific Software contract, while the other IdentityManagement contracts list the software maintainers for specific platforms. Whether they list developers or maintainers is not stored in these contracts, but is implicitly defined depending on which contract links the respective IdentityManagement contract. Technically, these contracts store a list of public keys of permitted users. Adding or removing public keys from this list is only permitted to the users already listed.

While being sufficient for the requirements of Palinodia, the abilities of the IdentityManagement contract are pretty limited. The IdentityManagement contract stores identities simply as the public keys of the authorized users. This is enough to check whether a function call on one of the other contracts of Palinodia should be executed. However, this approach is lacking most features which are expected from digital identities, restricting their use in other contexts. In the following, it is discussed how this contract can be replaced by using the shared identities of DecentID, and which aspects have to be considered when linking two blockchain-based systems.

4.1.2 Coupling approaches

As traditional software development has shown, a standardized interface is the best approach to couple two systems to work together. However, with blockchain technology being relatively new, no such interface exists yet for identity management. While some approaches exist [Bra03; Fab02; Pel03], they are all still considered drafts and no widely supported interface has emerged yet.
Missing a standardized interface, several coupling approaches were evaluated. As it turned out, the most restricting factor is the question whether one or both of the contracts are already deployed. If one of the systems that should be coupled is already deployed to the blockchain, its smart contracts can no longer be modified. Consequently, the other system has to be adapted. Alternatively, a proxy contract can be used that translates the function calls of the calling smart contract to the supported interface of the called system. Which approach is possible depending on

		Palinodia	
		In development	Deployed
DecentID	In development	Proxy Contract Adapting Palinoda Adapting DecentID	Proxy Contract Adapting DecentID
	Deployed	Proxy Contract Adapting Palinoda	Proxy Contract

Figure 4.3: *The possible coupling approaches based on the deployment status of the smart contracts.*

the deployment status of Palinodia and DecentID is displayed in figure 4.3. In the following, the possible approaches are presented.

No coupling The simplest approach is to use the integrated identity management of Palinodia. Since it is specially designed for its purpose, it is simple to use and quite efficient. However, the identities stored within it are not very useful in other contexts.

```
1 function checkIdentity(address _addr) public view returns(bool) {
2     return (arr_idents[map_idents[_addr].array_index] == _addr);
3 }
```

Figure 4.4: *A function in the IdentityManagement contract of Palinodia which checks the authorization of users.*

In figure 4.4 the function used for authorization checks in the IdentityManagement contract of Palinodia is displayed. As a function argument, it receives the public key[1] of the user that wants to call a restricted function in Palinodia. Within the smart contract, a mapping `map_idents` allows to retrieve an array index, and possibly further data, for all registered public keys. This array index is then used to look up the respective array entry in the array `arr_idents`, which should be the public key of the authorized user. Due to the way Solidity works, a non existing mapping entry would return a zero initialized array index, resulting in a lookup of the first element in the array `arr_idents`. Consequently, the retrieved array element has to be compared to the function parameter to ensure that the correct element has been retrieved, and that the public key is registered as an authorized user within the IdentityManagement

[1] In Solidity, the programming language for smart contract on the blockchain Ethereum, public keys and blockchain addresses are both represented by the `address` data type.

contract. If this check succeeds, `true` is returned and the restricted function in Palinodia can be executed. When replacing the IdentityManagement contract with either a proxy contract or DecentID, the functionality of this `checkIdentity()` function has to be provided in some other way.

Using a proxy contract When both systems, in this example Palinodia and DecentID, already have been deployed, no further changes to their function interfaces can be done. Consequently, a special purpose proxy contract has to be used. It offers the interface expected by Palinodia for its IdentityManagement contract, namely the `checkIdentity()` function, and adapts the calls to the interface DecentID supports. If required, the proxy contract can execute multiple function calls to DecentID or even modify the data passed to or received from DecentID to match the data format required by Palinodia. With further proxy contracts, even more identity management systems could be coupled with Palinodia without the need to adapt the function interfaces. As such, it is a very universal solution for most coupling scenarios, but due to the additional function calls and data modifications it is not as efficient as matching interfaces.

```
1 function checkIdentity(address _addr) public view returns(bool) {
2     SharedIdentityContract sic = map_idents[_addr];
3     if (sic == SharedIdentityContract(address(0))) {
4         return false;
5     }
6     bytes memory attr = sic.getAttribute(admin,
7         addressToHexString(msg.sender));
8     return attr.length == 1 && attr[0] == role;
9 }
```

Figure 4.5: *The `checkIdentity()` function in the proxy contract, previously provided by the IdentityManagement contract.*

The adaption between the interfaces required by Palinodia and provided by DecentID is displayed in figure 4.5. As seen in line 1, the function declaration is the same as in the IdentityManagement contract of Palinodia. This is required since Palinodia should not be modified, and expects this function declaration from its identity management. Within the function, the SharedIdentityContract linked to the given public key is retrieved from the mapping `map_idents`. Previously, this mapping has been established in the proxy contract by the user identified by this public key. While registering the mapping, the cryptographic hash of the smart contract code at the registered address is checked to match the code of a known SharedIdentityContract.

Also, the proxy contract ensured that the user really is the creator of the registered SharedIdentityContract. Further authorization to add the mapping, e.g., checking the authorization for Palinodia, is not required. Checking the authorization only once on registration to the proxy contract is insufficient, since the authorization might be revoked at some time in the future. Instead, it has to be checked on each access of Palinodia. Since the mapping has to be stored, the proxy contract maintains some state for each identity that should be used. A stateless proxy contract would be preferable, since it is more flexible and financially cheaper to use. However, this is not possible due to the smart contract code of Palinodia: Only the public key of the user calling one of Palinodias functions is passed on to the identity management, without the possibility to pass on the address of a SharedIdentityContract with it.

After the address of the SharedIdentityContract has been retrieved from the mapping, the returned address is compared whether it is valid. If the address is equal to a default constructed address, no existing entry in the mapping has been found and the public key `_addr` is not registered in the proxy. In that case, the access to Palinodia is denied by returning `false` from this function. From the SharedIdentityContract at the found address an attribute is retrieved. This attribute was granted by the permitted user `admin`, which has been set in the proxy contract earlier. This user has granted an attribute with the address of the Software or BinaryHashStorage contract, that is calling the `checkIdentity()` function, as indexing key. This way, separate permissions can be granted for the different software projects and their binaries. In line 8 the actual authorization check is done and the result of it returned. The retrieved attribute is expected to have a length of one byte, containing the role this user has for the smart contract calling this function. If the length differs, the attribute is either not set or set to an unexpected value. In both cases the access to Palinodia should be denied. As with the `admin` address, the `role` that should be checked has been stored in the proxy contract on its deployment. For this exemplary implementation of a proxy contract, two roles have been defined: a “developer” role for access to a Software contract, and a “maintainer” role for access to a BinaryHashStorage contract. Depending on the specific requirements, additional roles could be defined for other use cases.

Adapting Palinodia When the smart contracts of Palinodia have not been deployed to the blockchain yet, they can be modified to support the generic function interface of DecentID. Normally, Palinodia expects a certain function, i.e., `checkIdentity()`, from its identity management system that takes a public key as parameter and returns whether the user represented by it is permitted to access Palinodia. Whether the user is a software developer or maintainer is not passed towards the identity management system, but is inferred from the context of the call. As such, a generic identity of

DecentID cannot be called with this function directly. Instead, Palinodia has to be modified to retrieve an attribute from a given shared identity and check whether the attribute allows its owner to perform the requested action.

```
1  function checkIdentity(SharedIdentityContract sic, address sender)
2          internal view returns (bool) {
3      if (getContractHash(address(sic_root)) !=
4                  getContractHash(address(sic))
5              || sic.getCreator() != sender) {
6          return false;
7      }
8      bytes memory attr = sic.getAttribute(sic_root.ownerAddrs(0),
9          addressToHexString(address(this)));
10     return attr.length == 1 && attr[0] == byte(0x02);
11 }
```

Figure 4.6: *Added code in Palinodia for checking authorization in the attributes of DecentID.*

The code that has to be added to both the Software and the BinaryHashStorage contracts can be seen in figure 4.6. From its functionality, it is similar to what is done in the proxy contract. The checks in the lines 3 – 6 are the same as executed when registering a new SharedIdentityContract within the proxy contract. As done in the proxy contract, the hash of the provided contract is compared to the hash of a known SharedIdentityContract to ensure that no modified smart contract is provided by an adversary. Furthermore, the creator as stored in the SharedIdentityContract is checked to be the user trying to access Palinodia. In the proxy contracts these checks only have to be done once when registering the SharedIdentityContract. Here, the checks have to be executed each time since the address of the SharedIdentityContract is not stored within Palinodia. This is avoided since the addresses would have to be stored for all authorized users, increasing the size and the cost of the smart contract.

Afterwards, nearly the same attribute check as used in the proxy contract is done in lines 8 – 10. While using the same functional logic, the values used to access and check the attribute has been replaced to refer to the contract containing the function. Instead of a stored administrator public key, the administrator of the current Palinodia contract is used in line 8. In the following line 9, `msg.sender`, which refers to the entity calling the function, has been replaced by `this`, which refers to the contract containing the function. Finally, instead of a previously stored role the value `0x02` is used, which denotes a “maintainer” role.

Adapting DecentID Instead of adapting Palinodia as described above, DecentID could be adapted to support the expected interface required for checking the authorization of users.

```
1 function checkIdentity(address _addr) public view returns(bool) {
2     if (getCreator() != _addr) {
3         return false;
4     }
5     bytes memory attr = getAttribute(palinodia_admin,
6         addressToHexString(msg.sender));
7     return attr.length == 1 && attr[0] == palinodia_role;
8 }
```

Figure 4.7: *The `checkIdentity()` function as it could be added to the SharedIdentityContract.*

Technically, this would require code as displayed in figure 4.7. The function would receive the public key of the user that wants to access Palinodia as the function parameter `_addr`. Afterwards, the function ensures that the user identified by the given public key is the creator of this shared identity. If it is, the attribute is checked as done in the other implementations as well.

However, for multiple reasons this approach was not evaluated in the following. For once, including this function into DecentID would mean that the generic identities of DecentID would contain a function and variables specific to a single use case, i.e., Palinodia. From a design perspective, this is very undesirable. If further use cases ought to be supported, even more single use functionality would have to be included in the SharedIdentityContract.

Furthermore, this approach introduces a security vulnerability: If an arbitrary smart contract could be provided by a user to call the `checkIdentity()` function on it, the user can simply provide a smart contract where the function always returns `true`. This would mean that they are able to access any contract of Palinodia, since the authorization check is bypassed. This is also the reason why the hash of the smart contract code needs to be verified in the previous coupling approaches.

Finally, the smart contract of Palinodia expect one fixed smart contract address where the identity management resides at. When a different SharedIdentityContract is supposed to be used for each authorization, an Palinodia administrator would be needed to manually change the called address each time. Besides being impracticable, changing the address of the identity management is currently not supported by

Palinodia. Consequently, Palinodia would have to be adapted to add this functionality, which should have been avoided by adapting DecentID. Also, adapting Palinodia is impossible if Palinodia has already been deployed.

Based on the first three approaches, an evaluation given multiple criteria is performed in the following.

4.1.3 Evaluating the approaches

In this section, multiple criteria related to the coupling of smart contracts are discussed. As an example, these criteria are applied to the approaches described above. To do so, a proxy contract as well as a modified version of Palinodia were implemented. These approaches are compared to the native identity management system included into Palinodia, which requires no coupling with DecentID.

4.1.3.1 Security dependency

Blockchains ensure that the smart contract code is executed as it is written, but at the same time they do not allow later modification of the code. As such, it is even more important than for traditional software to ensure that the deployed code does not contains bugs or security vulnerabilities. While a single smart contract can be verified, it becomes much more complicated when multiple smart contracts are coupled. For once, the called contracts might contain security vulnerabilities themselves, endangering the correct execution of the calling contract. Also, calling other contracts add additional complexity to handle the call, which might lead to security vulnerabilities by itself.
An additional problems is that the address of the called contract has to be known. If all functionality is contained within one smart contract, all functions are simply called on the current contract without additional addresses being required. When a smart contract should call another smart contract, by whichever coupling approach, the addresses of the contract that should be called has to be given to the calling contract at some time. This might either happen rarely or only once, e.g., on deployment, or be done on each function call. The security becomes even more difficult to evaluate when the called contract calls a further contract. However, this case is not explicitly regarded in this section, since the same considerations apply even with multiple consecutive call steps.
In the original version of Palinodia the IdentityManagement contract I is called by the other contracts, e.g., Software or BinaryHashStorage contracts, to check the authorization of a user. The address of I is provided when deploying the calling

contract C to the blockchain, and then stored as an attribute within the contract C. This has the advantage that the creator of C can verify the code of the smart contract residing at the address of I, i.e., they can check the functionality and security of the code of I. When a function of I is called later on, it is already known beforehand what will happen.
When adapting Palinodia to call SharedIdentityContracts directly, another approach is used. Instead of providing the address at construction of the calling contract C, the address of the called SharedIdentityContract SIC_i is provided when a function is called that requires authorization. When accessing C, each authorized user i provides their own SIC_i. Consequently, a different blockchain address is required each time. Storing all possible addresses of authorized SharedIdentityContract beforehand is for once expensive, but would also defeat the purpose of providing a SharedIdentityContract in the first place: If all users are already known and registered in C, there is no reason to check the authorization in SIC_i. Instead, the smart contract SIC_i at the provided address has to be verified by C to ensure its validity. As described in section 4.1.2, this is achieved by verifying the hash of the smart contract code, followed by verifying the ownership and attributes stored in the provided SharedIdentityContract. If the code of SIC_i, respectively its hash, would not be checked by C, an adversary could provide a modified contract SIC_i' which always returns that the adversary is permitted to access Palinodia, even when the relevant attribute has not been granted to them. A similar approach can also be used when coupling other smart contract, independently of the example used in this section. However, a drawback of verifying the hash of the smart contract code is that only those smart contracts can be used, where the hash has been previously stored in the calling contract. Using an updated or a completely different smart contract is not possible, even when this contract would still provide the same functionality.
When using a proxy contract to couple Palinodia and DecentID, both variants of passing the address are used. To call the proxy contract P from within one of Palinodias contracts C, a fixed blockchain address is provided at deployment of C. This has to be done, since the unmodified smart contracts of Palinodia expect a fixed address for their identity management and do not support working with dynamic addresses. When retrieving the attribute from the user-provided SharedIdentityContract SIC_i, the address of SIC_i has been previously stored in the proxy contract P. Technically, this differs from the direct coupling by adapting Palinodias contracts. From a security standpoint however, it results in the same challenges and required considerations. Since the address of SIC_i is provided and registered in P by the user, the address of SIC_i has to be considered an untrusted address. Consequently, the code residing at this address has to be verified before calling a function on this contract.

4.1.3.2 Costs

In Ethereum, calling smart contract function and deploying new smart contracts has to be paid for in Ether, with the costs depending of the amount of code executed or written. This cost, measured in *gas*, is paid to the miners, motivating them to include the execution of the call in the next block. As such, function code should be written as efficient as possible, to avoid high costs for the calling user. To evaluate the costs, a simple scenario has been chosen: One instance of Palinodia, consisting of a Software contract and a BinaryHashStorage contract, is deployed, together with the identity management contracts of the different approaches. The latter are either the IdentityManagement contracts of Palinodia, a SharedIdentityContract of DecentID, or a SharedIdentityContract together with a proxy contract. The created identity is then authorized as a developer for the Software contract. While deploying contracts is much more expensive than executing contracts, it is also a relatively rare occurrence. Much more frequent are code executions and small data modifications on the blockchain, e.g., authorizing a new user or updating a stored binary hash.

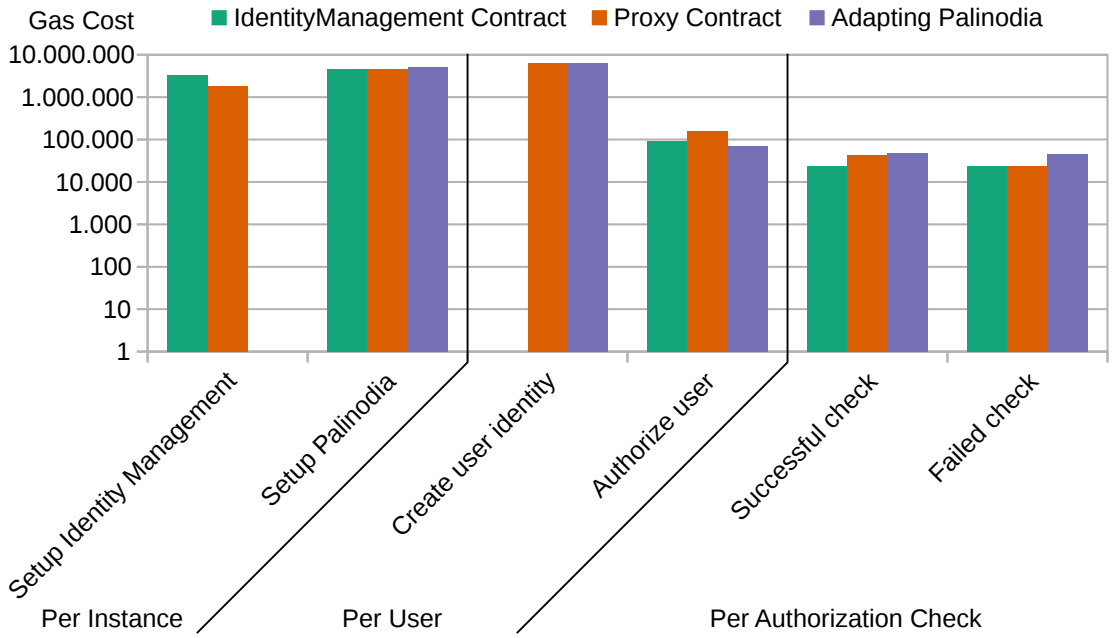

Figure 4.8: *Gas costs when deploying and using Palinodia and DecentID plotted on a logarithmic scale.*

The results of the cost measurements are displayed in figure 4.8. Depending on how often these costs occur, they are separated between once per deployed instance, once per authorized user, and once per authorization check of a user.

Per Instance
The costs per instance occur only once per deployed Palinodia instance, independently of how many users are authorized for it or how much the system is used.

Setup Identity Management In the original design of Palinodia, two IdentityManagement contracts are created per deployment of Palinodia. If a proxy is used, the costs are based on deploying the two proxy contracts, which are replacing the IdentityManagement contracts of Palinodia. The IdentityManagement and proxy contracts are always created, even when no users will be authorized. When more than one BinaryHashStorage contract is deployed, an additional IdentityManagement or proxy contract has to be deployed as well. Since shared identities are created per user, no setup cost for the identity management occurs when using an adapted version of Palinodia, which directly interacts with DecentID.

Setup Palinodia When using the IdentityManagement contracts or the proxy contracts, the cost to deploy the other contracts of Palinodia remains the same. In both cases, Palinodia does not need to be adapted. The cost to deploy the Palinodia contracts is slightly higher when coupling directly with DecentID. In that case, the adapting code towards the function interface of DecentID is included within Palinodias contracts, instead of not being required or being part of the proxy contract.

Per User
The costs per user describes the costs when authorizing a single user to access Palinodia. It is separated between creating a new identity to do so, and the authorization itself.

Create user identity When the IdentityManagement contract is used, no costs occur for creating user identities. Since the IdentityManagement contract only stores the public keys of authorized users, there are no identities which need to be created or stored on the blockchain. When DecentID is used, the relatively high costs of deploying shared identities is visible. If the created shared identities are only used to access Palinodia, this cost might be too high to be acceptable for many users. However, when a shared identity already exists for some other purpose and can be used for Palinodia as well, this cost does not apply, significantly reducing the costs for using DecentID with Palinodia.

Authorize user When using the IdentityManagement contract, the cost to authorize a user is the cost to add a new public key to the list of authorized users stored within the contract. When using DecentID, either directly or through a proxy, a new attribute is added to the SharedIdentityContract describing the role of the user. Adding the

attribute is cheaper than adding the public key, since the attribute is shorter and requires less storage on the blockchain. If a proxy is used, the cost is slightly higher than when using DecentID directly, since state within the proxy contract has to be established.

Per Authorization Check
When a user wants to modify the state of Palinodia, e.g., add a new binary hash, their authorization is verified.

Successful check At a successful authorization check the public key is found in the IdentityManagement contract, respectively a matching attribute found in the shared identity. Since additional checks are required to verify the SharedIdentityContract, the costs for the authorization check is higher. If the proxy contract is used, some of these additional checks were already done when registering the SharedIdentityContract within the proxy contract, reducing the costs compared to a direct coupling when checking the authorization later on.

Failed check A failed authorization check is executed when a new user should receive access permissions, since already authorized users should not be authorized a second time. With the IdentityManagement contract, the cost is the same as for a successful check, since in both cases a check for the existence of the public key is executed. The cost when using the proxy is in the evaluation scenario lower than a successful check, since the authorization check already fails when the missing state of the proxy is discovered, instead of needing to retrieve the attribute from the shared identity. Using an adapted version of Palinodia and accessing the SharedIdentityContract directly is slightly cheaper than a successful check as well, since some of the required checks, e.g., processing the value of the attribute, are not executed.

The previous cost evaluation has regarded the creation of a minimal Palinodia deployment. Contrary to that, figure 4.9 considers the costs of larger, more realistic, deployments. The depicted costs for only one user are the same as the sum of the costs displayed in figure 4.8. Based on that, the costs for systems with more authorized users have been calculated. These calculations assume that one instance each of the Software and BinaryHashStorage contracts have been deployed to the blockchain. If the IdentityManagement contract or the proxy are used, two of those, one per Palinodia contract, have been deployed.
The first three bar groups assume that for each user that should be authorized a new identity is created and authorized. When the IdentityManagement contract is used, the costs are only increasing slowly, since for each newly authorized identity only

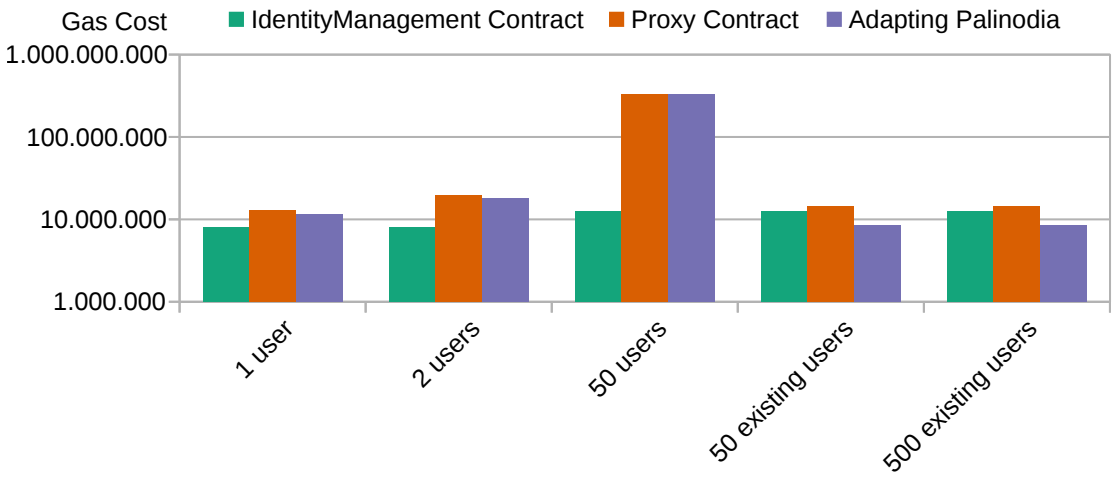

Figure 4.9: *Costs with different amounts of authorized users on a logarithmic scale.*

a single public keys has to be written to the blockchain. When DecentID is used, either directly or through the proxy contract, the costs are increasing much faster. Deploying SharedIdentityContracts is much more expensive than writing a single public key, resulting in the higher costs.

The last two bar groups assume that the users of Palinodia already have SharedIdentityContracts and want to add the authorization to access Palinodia to their existing shared identities. For the IdentityManagement contract this does not change anything and the costs are increasing as in the previous bars. The IdentityManagement contract only lists authorized users, so it is not possible to reuse existing identities for it. However, when DecentID is used the costs are significantly reduced. If a proxy contract is employed, the costs are higher than with the IdentityManagement contract, since the state stored within the proxy is larger, and as such more expensive to store, than the public key stored in the IdentityManagement contract. If Palinodia is coupled with DecentID directly, the cost is lower than with the IdentityManagement contract. As already seen in figure 4.8, the cost for authorizations is lower when creating an attribute in DecentID, than adding a public key to the IdentityManagement contract. When more identities are authorized, this saving becomes more apparent.

This evaluation has shown that the costs of using the general purpose identity management system DecentID is higher than using the special purpose identity management included in Palinodia. This is most likely true for every generic identity management system that is coupled instead of using a special purpose approach. The costs are especially high when users have to or want to create a new SharedIdentityContract for their Palinodia authorization. However, when they already own a shared identity and are reusing it for Palinodia, the costs are even cheaper compared to the included

IdentityManagement contract. In conclusion it can be said that DecentID gives the user of Palinodia more flexibility to use their identity, but results in higher financial costs if this flexibility is not used.

4.1.3.3 Implementation effort

In section 4.1.2 the relevant code segments for checking the authorization in the different coupling approaches were shown. As is apparent, less smart contract code is required when the IdentityManagement contract of Palinodia is used (displayed in figure 4.4). Compared to that, the other coupling approaches require similar amounts of code, since they are all providing the same functionality: Adapting the function interface provided by DecentID to the requirements of Palinodia.
When adapting Palinodia, an adapter function has to be implemented that reads the attributes from the SharedIdentityContracts and checks whether the returned values match the requirement for the desired modifying action in Palinodia. Additionally, this function has to check whether the given blockchain address even is a SharedIdentityContract and whether the identity has been created by the calling user. Both the attribute check and the verification of the provided smart contract are not required when using the IdentityManagement contract.
If a proxy contract is used, the same checks have to be done. Additionally, the proxy contract has to contain code to fulfill the expected function interface of Palinodia. Since Palinodia only passes the public key of the function caller to the identity management, which is represented by the proxy contract, the proxy has to contains a mapping of public keys to addresses of shared identities. Consequently, a user has to register their shared identity in the proxy before using Palinodia. Furthermore, using a proxy contract requires the implementation and deployment of a complete smart contract. However, the additional implementation effort for this smart contract is negligible, since it only consists of the adapting function.
Based on this, it appears that coupling with an existing identity management system is more effort than using a special purpose identity management, especially when comparing the shown code segments required to check the authorization. However, while a special purpose identity management offers a simpler function interface, using one requires the implementation of a complete smart contract, with functionalities to create, authorize, and delete identities. Additionally, the security of this system has to be considered and evaluated, to avoid that adversaries authorize themselves. So while coupling with an existing identity management system might seem to be more effort, it can actually reduce the required amount of work, both in the design and in the implementation of the system.

4.1.3.4 Interoperability

To improve the interoperability between smart contracts, the function interface of a contract should be designed as generic as possible. Given a publicly known interface, two contracts can be coupled even when their developers did not consider this specific pairing of contracts. Doing so can avoid the costs for deploying additional, single purpose, contracts, as well as increasing the reusability and flexibility. Preferably, changes to the design of smart contracts should try to improve the coupling options for their users and not replace one fixed coupling with another.

In the original design of Palinodia, the function interface of the IdentityManagement contract was tailored to the requirement of receiving a public key and checking whether it has been authorized. As such, the Software and BinaryHashStorage contracts expected such a function interface, which simply takes and checks a provided public key. While this is sufficient for the use case of Palinodia, it restricts the abilities of the identity management system and hinders possible other uses of the stored identities, e.g., using the received authorization for other smart contracts.

DecentID, on the other hand, provides a generic function interface to read and write the attributes of the SharedIdentityContracts. While this permits other smart contracts to interact with the shared identities, it also means that the calling smart contracts have to interpret the returned attribute data. In the case of Palinodia, that means that the retrieved attribute data has to be checked whether it represents a valid authorization to interact with the system. However, after adapting Palinodia any other identity management system, that supports the same function interface as DecentID, could be coupled with Palinodia as well.

Considering the costs of deploying additional contracts like the proxy contract, the interfaces should be as generic as possible for the use case of the called system. This way, the called system can be exchanged, while at the same time multiple systems can use its functionality. If a proxy contract is implemented, it should try to avoid keeping state for the desired coupling. Without state in the proxy contract, multiple other smart contract can call the proxy contract, allowing to use one proxy contract to couple between multiple systems. In that case, the proxy contract only adapts the mismatching function interfaces of the systems that are coupled, without being tied to any of them.

In general, standardized interfaces, offering use case independent functions, between smart contracts are preferable to improve their interoperability. If this is not possible, e.g., because one or both of the smart contracts that should interact are already deployed to the blockchain, a proxy contract can help to adapt the interfaces and permit interacting with other smart contracts.

4.1.4 Findings

Similar to what is common practice in software development outside of the blockchain, a modular design for smart contracts is preferable, since it can reduce the costs of implementation and deployment of smart contracts. However, how the coupling interface between modular smart contracts is designed is not quite clear yet. In the future, conforming to a standardized function interfaces will be preferable, to permit coupling with as many other smart contracts as possible. Unfortunately, no such interface for identities management has emerged yet. For now, a clear recommendation which of the evaluated coupling approaches is best cannot be given and depends heavily on the specific use case of the to be coupled systems.
The evaluation has shown that the different coupling approaches each have their own advantages and disadvantages. If one of the contracts that should be coupled has not been deployed to the blockchain yet, it can be preferable to adapt its interface to the already deployed contract. However, this should not be done if this results in adding use case specific functionality in an otherwise generalized smart contract. In that case, or if both contracts have already been deployed, a proxy contract can help to bridge the gap. While always applicable and allowing additional adaptions between the smart contracts, they bring their own disadvantages of higher implementation effort, more costs, and potentially additional security vulnerabilities.
For DecentID, this evaluation has shown that it can be used as an identity management system for other blockchain-based systems. Its generic interface can be used to fulfill purposes that have not been an explicit design goal, without increasing the required adaption overhead for the calling contract too much. A restriction that should be noted again is that smart contracts are unable to access data stored outside the blockchain. Both Palinodias IdentityManagement contract and the SharedIdentityContract store their data on the blockchain, while some other identity management systems, e.g., ShoCard and uPort, store their data off-chain. As such, identity attributes stored off-chain are not available to other smart contracts on the blockchain. While off-chain storage is cheaper, it limits the versatility of the identities. However, since computation on the blockchain incurs financial costs, there probably is no demand for accessing and processing large amounts of data on-chain anyway. Accessing the small attributes stored directly within the shared identities on-chain should most likely be enough when using DecentID from within other smart contracts.

4.2 Voting integration

Given a SharedIdentityContract $SIC_{\{A,S\}}$, only the user A and the service S are able to add, modify, or delete attributes from $SIC_{\{A,S\}}$. For many use cases, this is sufficient: The user A can add attributes describing themselves, while the service S reads these attributes and might add own attributes. However, given a group of users having shared identities with a service, it can be useful if some of the users are able to add attributes to the other users in the name of the service.
In the following, such a scenario is described. Afterwards, the decentralized voting system Open Vote Network is introduced, together with its Ethereum implementation and the adaptions required for its integration into DecentID. The modified and newly added smart contracts are presented, before the technical execution of a poll is detailed. In the following, the security and privacy of the approach is analyzed.

4.2.1 Example Scenario

In this example scenario, each user of a group of users, Alice, Bob, Carol, and Thomas, has their own SharedIdentityContract with the service S. The users do not have access to the shared identities of the other users. This can be seen in figure 4.10.

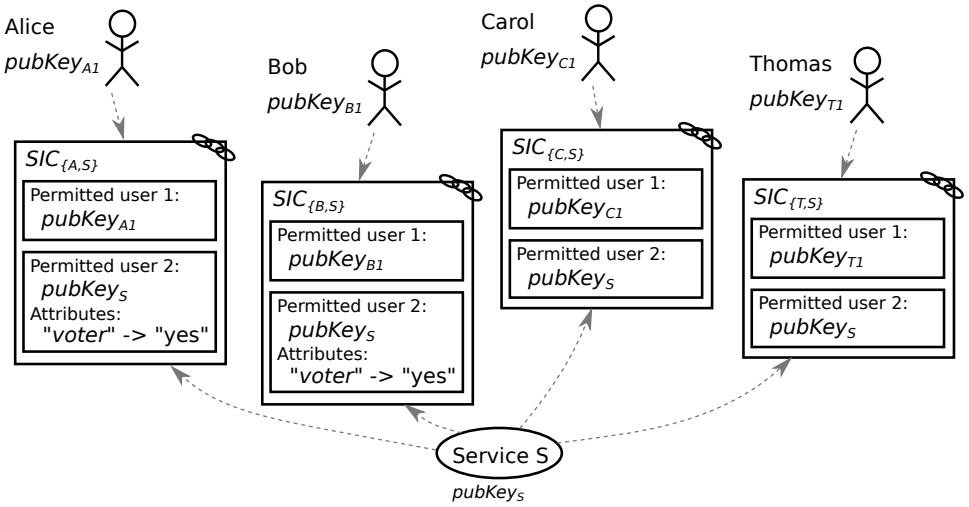

Figure 4.10: *Example scenario for voting. The attribute keys and other attributes are omitted in this figure.*

As explained in chapter 3, only the permitted users of a SharedIdentityContract are able to attach attributes to it. For example, only Thomas and the service S are able to attach attributes to $SIC_{\{T,S\}}$. The same holds true for the other SharedIdentity-Contracts in the example. If the service S wants to add an attribute to $SIC_{\{T,S\}}$, the

private key $privKey_S$ of the asymmetric key pair used by the service is required. This means, that other users are not able to attach attributes to $SIC_{\{T,S\}}$, especially not in the name of the service. In most scenarios this is not needed anyway. One example would be when the service offers an online shop: Only Thomas and the service need to access $SIC_{\{T,S\}}$, other users should not be able to do so, and there is no reason for them to do so. In other scenarios, e.g., when the service offers an online forum, this might be different. Here, the service might want to authorize certain users of the forum to add attributes to other users of the forum.

Initially, most online forums are managed by a single human, here represented by the service S. Consequently, only S has access to the private key $privKey_S$, meaning that this single person becomes a single point of failure. If this manager is not available for some reason, management tasks of the forum regarding a SharedIdentityContract cannot be executed. These management tasks could, for example, be to grant an advanced forum rank to a user of the forum. While other people might be able to agree on the decision to grant the rank, they are unable to assign the respective attribute to the user. Assigning the attribute would require access to $privKey_S$, which should not be disclosed to other users.

As an alternative to disclosing $privKey_S$, a voting system has been integrated into DecentID. By executing a poll and agreeing on the action that should be done, authorized users are able to add, modify, or remove attributes in the name of the service S to identities that have been shared with it.

Definition 4.1: Voter

A *voter* V is a user having a SharedIdentityContract $SIC_{\{V,S\}}$ with a service S, that has been authorized by the service S to participate in polls and add attributes to other users of the service S by voting.

In the depicted example scenario in figure 4.10, the service S has added an attribute "voter" to the SharedIdentityContracts $SIC_{\{A,S\}}$ and $SIC_{\{B,S\}}$ created by Alice and Bob, authorizing them as voters. Carol and Thomas have not received this attribute, prohibiting them from participating in polls for this service.

After executing the poll and agreeing on the desired change, the voters are able to add a new attribute to the SharedIdentityContract $SIC_{\{T,S\}}$ between Thomas and the service. This new attribute is added to $SIC_{\{T,S\}}$ as if the service S had added it itself using its private key $privKey_S$.

Due to the involved smart contracts, the addition or modification of the attribute can be executed without requiring the private key $privKey_S$ of the service. Still, the attribute is modified in the name of the service S. As explained in section 3.5.2, this

means that manipulating attributes attached to $SIC_{\{T,S\}}$ by Thomas is not possible. Only Thomas is able to modify the attributes he added to $SIC_{\{T,S\}}$, the integration of the voting system does not change this. Furthermore, the voters are only able to write, overwrite, or remove attributes: Reading encrypted attributes that are already attached to $SIC_{\{T,S\}}$ is not possible.
On the blockchain Ethereum, functions in smart contracts are able to execute code and call other functions and smart contracts. However, they cannot to be called automatically, e.g., periodically each ten minutes. For this reason, the smart contract used for polls require a voting administrator.

Definition 4.2: Voting administrator

The *voting administrator* is the voter that started the poll, and that calls the required smart contract functions to advance it.

It should be noted that voting in this context is not meant to be and is not sufficient for legal votes, e.g., for political offices. Since the offline identity of a DecentID user is not known, it cannot be checked when voting. Consequently, the voting should only be used for informal opinion polls, where a wrong result, e.g., due to a non-terminating poll or malicious manipulation, is not critical. Also, the bootstrapping problem is not completely solved within DecentID: The first two voter authorizations have to be assigned by the administrator of the forum. How the administrator of the forum selects these first two voters is out of scope for this work. Afterwards, further voting permissions can be granted or withdrawn through a poll by the existing voters.

4.2.2 Open Vote Network

Designing and evaluating a secure and privacy-preserving digital voting system is by itself a formidable task, which is no goal of this work. Instead, an existing decentralized voting system, *Open Vote Network* [Zie10], was integrated into DecentID. It has been selected since it fulfills the requirement for the desired voting system: no dependency on centralized trust anchors, protection of the voters privacy, and verifiability of the results, besides being frequently referred to by related approaches. Additionally, there already exists an implementation based on smart contracts for Ethereum, simplifying the integration into DecentID.
Since this work focused on the integration of the system into DecentID and not on the design of a decentralized voting system as such, only a high-level overview over the functioning of Open Vote Network is given. For an extensive description of the approach and a security analysis, see the paper by Hao, et al. [Zie10]. The approach

presented in the following only supports votes of “yes” or “no”. However, it would be possible to extend the voting approach to support more choices if required.
In general, Open Vote Network operates in two rounds: Publishing of the voting public keys and publishing of the votes. Before executing the two rounds, a finite cyclic group G with its generator g is selected.

(1) In the first round, the voting public keys are created and published. To do so, each voter i selects a random value x_i and calculates their voting public key as g^{x_i}. This voting public key is send to the other voters, together with a zero-knowledge proof demonstrating that they know the value of x_i. When every voter has done so, the received zero-knowledge proofs are verified and a reconstructed key g^{y_i} based on all g^{x_i} is calculated by each voter.

(2) In the second round, each voter publishes $g^{x_i y_i} g^{v_i}$, with v_i being their own vote in $\{0, 1\}$. Again, a zero-knowledge proof is published together with this value, this time to prove that v_i is in $\{0, 1\}$. After all votes have been published, the zero-knowledge proofs are verified.

Due to the inclusion of the secret x_i values within their published vote, other voters or observers are unable to deduce the vote of a single voter. However, by calculating the product of the published votes, the final tally can be calculated: $\Pi_i g^{x_i y_i} g^{v_i} = g^{\Sigma_i v_i}$. As shown in [Zie10], $\Pi_i g^{x_i y_i}$ evaluates to 1. Since only a small number of voters is present, the discrete logarithm of $g^{\Sigma_i v_i}$ can be calculated, revealing $\Sigma_i v_i$ which is the number of “yes” votes. By comparing this number with the known total number of voters, it can be determined whether the vote passed or not.

4.2.2.1 Ethereum implementation

In [MSH17], an Ethereum implementation of the Open Vote Network based on smart contracts is presented. The smart contracts have been published under the MIT license and are available online[2]. Additionally, a webinterface to commence a vote has been implemented, which is not regarded in the following.
While the theoretical design of the approach only consists of two rounds, five stages are required for its implementation in Ethereum. These stages are *SETUP*, *SIGNUP*, *COMMIT*, *VOTE*, and *TALLY*, which are described in the following. In the previous section all voters have been regarded as equal. However, for the practical implementation as smart contracts, one of the voters acts as a voting administrator to create the smart contracts of the poll and advance the stages, as defined above.

[2]https://github.com/stonecoldpat/anonymousvoting Accessed: 11.03.2022

While not necessary for the execution of the poll, the implementation demands a deposit of Ether to participate in it. This deposit has to be paid by voters when they register for the poll, and is returned when they submit their vote. This way, misbehaving voters, i.e., voters that do not contribute a vote despite having registered, are punished by losing their deposit.
Another property of the implementation is that a number of durations can be set for the stages of the poll. The set minimum durations ensure that there is enough time for the voters to interact with the voting contract, even when their blockchain transactions need some time to get processed. The maximum durations make sure that the deposits can be returned to the voters, when either one of the voters or even the voting administrator stops participating in the poll.

SETUP In the first stage, the voting administrator creates the voting contract and sets the parameters for the poll. This includes setting the voting question, the size of the deposit, whether the optional COMMIT stage should be used, and the durations of the stages. Also, a list with the public keys of the authorized voters is stored.

SIGNUP In this stage, all voters which have previously been authorized by the voting administrator can now register for the poll, but do not have to do so. To register, they send their voting public key g^{x_i}, a zero-knowledge proof for x_i, and their deposit to the voting contract on the blockchain. When enough time passed, the voting administrator can advance the stage. When this happens, the smart contract calculates the keys g^{y_i}.

COMMIT This stage is optional to avoid an advantage for the last voter. Without this stage, the last voter is able to calculate the final tally even before voting themselves, since all previous votes are already stored on the blockchain and their own vote is known to them. When the COMMIT stage is used, all voters publish a hash of their vote on the blockchain. In the next stage, if the hash of their vote does not match this committed hash, their vote is declined. Since the number of voters is known, the stage can advance automatically once the last commit has been received.

VOTE All voters send their votes $g^{x_i y_i} g^{v_i}$ to the voting contract together with a zero-knowledge proof for its validity. If the vote is accepted, i.e., it matches the commitment and the zero-knowledge proof is correct, their deposit is refunded. Once the last vote is cast, the voting administrator can notify the voting contract to compute the final tally of the poll.

TALLY The voting contract calculates the tally $g^{\Sigma_i v_i}$. Afterwards, the discrete logarithm of it is calculated by testing all possible values, up to the known number of voters, until $\Sigma_i v_i$, the number of “yes” votes, is known.

4.2.2.2 Adaptions

To integrate the Ethereum implementation of Open Vote Network into DecentID, a number of adaptions are required, both to the voting implementation and to the SharedIdentityContract of DecentID. These changes are listed in this section, before describing the relevant smart contracts and the execution of a poll in detail in the following sections.

Separation of contracts The Ethereum implementation of Open Vote Network uses a single smart contract to store all data about the poll and the code for the required calculations. From a financial viewpoint this can be optimized. Instead of deploying the smart contract code to the blockchain again and again for each poll, this smart contract has been split into two parts: one permanent smart contract containing the static program code and a lightweight smart contract containing the state variables for a currently running poll.

Authorizing voters In the voting implementation, the voting administrator has to add the public keys of the authorized voters to the voting contract. When integrated into DecentID, this is no longer required. Instead, the smart contracts can automatically check whether a user is authorized to vote in the currently running poll, by accessing their SharedIdentityContract.

Security verifications Beside verifying the authorization of the voters, the use of the correct smart contracts has to be verified as well. Since multiple contracts interact with each other, they have to ensure that the correct smart contracts are the callers respectively parameters of functions. Otherwise, an adversary could use a modified voting contract which, e.g., always returns that the poll has passed, independently of the real votes submitted.

Removing the COMMIT stage The COMMIT stage was only added in the Ethereum implementation of Open Vote Network, and is no required part of the algorithm. Its purpose is to avoid the vulnerability that the last voter can predict the result of the poll even before they are submitting their own vote. If the other voters ended up in a tie, the last voter would be able to reconsider their vote. Since the threat due to this vulnerability is quite low and the COMMIT stage increases the financial costs of voting, it was omitted when integrating voting in DecentID.

Adapting the SharedIdentityContract Normally, the SharedIdentityContract only accepts attribute modification by permitted users. For voting, an additional method has been added that permits to modify attributes when the result of a passed poll is given as a parameter.

4.2.3 Smart Contracts

Voting in DecentID is executed with the help of three smart contracts on the blockchain: a modified version of the SharedIdentityContract, the VotingContract, and the VotingDataContract. These smart contracts and their interactions are displayed in figure 4.11 and will be described afterwards.

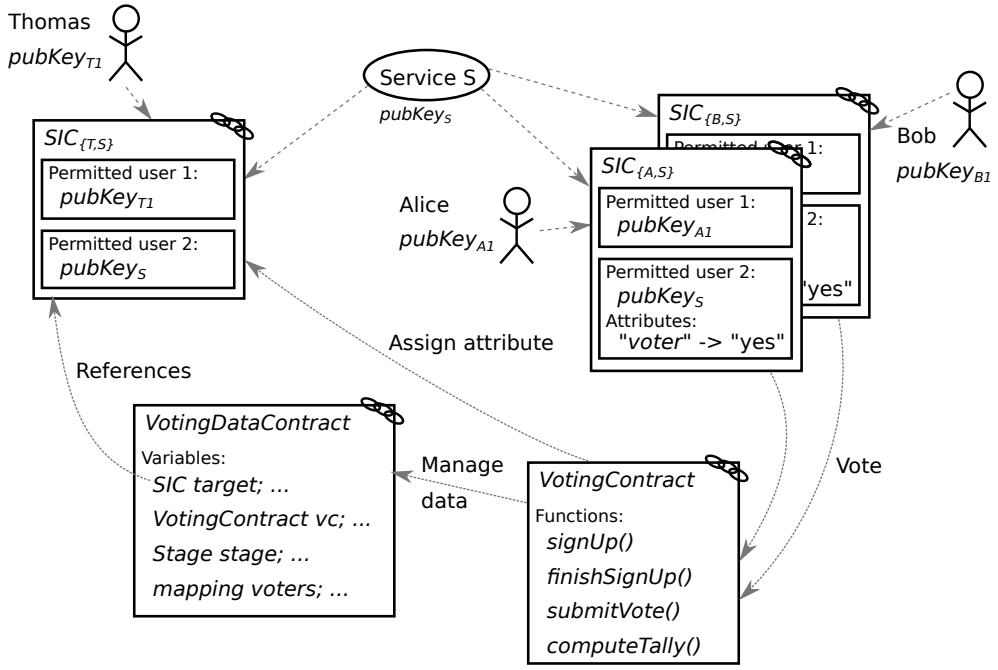

Figure 4.11: *The smart contracts used when voting. The additional parts of DecentID, e.g., for attribute management, have been omitted since they are not relevant for voting.*

4.2.3.1 SharedIdentityContract

To support voting, the SharedIdentityContract, introduced in section 3.5, had to be adapted. Two new functions are required: a function to start a poll, and a function to modify the attributes of the SharedIdentityContract as decided in a passed poll. Their function declarations are displayed in figure 4.12. In its previously described state, adding attributes to a SharedIdentityContract is only possible when knowing the private key of a permitted user. To circumvent this requirement, the following voting functions were added. The already existing functions of the smart contract as well as its existing functionality is not changed by integrating the voting system, only the new voting functionality is added.

```
1 function startVoting(SharedIdentityContract sic, address _pkService,
2         string memory _key, byte _flags, bytes memory _data)
3     public returns (VotingDataContract);
4
5 function updateAttribute(VotingDataContract vdc) public;
```

Figure 4.12: *The function declarations for voting in the SharedIdentityContract.*

startVoting() This function starts the voting process by creating and initializing the VotingDataContract before returning its address. Before doing so, the authorization of the function caller is checked whether the caller is authorized to start a poll. This includes checking that the passed service is a known permitted user of this SharedIdentityContract, that the SharedIdentityContract of the caller uses the same contract code as the called contract, that the caller is the creator of the passed SharedIdentityContract, and that the passed SharedIdentityContract is an authorized voter for the given service. If these checks pass, a new VotingDataContract is deployed to the blockchain. It is initialized with the parameters given to the `startVoting()` function: the address of the service, as well as the indexing key, the flags, and the data for the new attribute. Additionally, the new VotingDataContract stores the address of the to-be-modified SharedIdentityContract and of the VotingContract that is permitted to modify its data. The to-be-modified SharedIdentityContract is the contract where the `startVoting()` function is called on, while the address of the VotingContract is stored as a variable in this SharedIdentityContract. The address of the new VotingDataContract is returned to the caller of the function, and can be later on given to other interested voters so they can participate in the poll.

updateAttribute() When the poll is finished and has passed, this function is called to modify the attributes of the SharedIdentityContract as requested. To protect against unauthorized modifications, a number of checks are executed. First, it is checked whether the given VotingDataContract uses the known smart contract code and has been created to modify this SharedIdentityContract. Since the code of the VotingDataContract is known, the SharedIdentityContract can be sure that the poll has been correctly executed. Afterwards, the state of the poll is checked: it has to be finished, and the proposed change has to be accepted by the majority of the voters. If all checks succeed, the attribute is added, modified, or deleted as described in the VotingDataContract. It should be noted that only attribute modifications in the name of the service can be done. Especially this means, that no attributes added by other permitted users, e.g., the creator, of the SharedIdentityContract can be modified.

4.2.3.2 VotingContract

The VotingContract contains the program code required to execute polls. It only contains the code but no state variables, which allows to deploy it to the blockchain only once and use the same smart contract for all executed polls. The states of the currently running polls are stored in VotingDataContracts, which are explained in the next subsection. The function declarations of the VotingContract are displayed in figure 4.13, and represent the stages of Open Vote Network. The code of the functions is partially inherited from the Ethereum implementation of Open Vote Network [MSH17], with adaptions to work together with the VotingDataContract and the SharedIdentityContract. These adaptions include the authorization checks for the voters, the checks of the smart contracts that are interacted with, and the code modifications required to store the state of the poll in the VotingDataContract instead of in local variables.

```
1  function signUp(VotingDataContract vdc, SharedIdentityContract sic,
2          uint[2] memory xG, uint[3] memory vG, uint r)
3      public payable returns (bool);
4
5  function finishSignUp(VotingDataContract vdc) public;
6
7  function submitVote(VotingDataContract vdc, uint[4] memory params,
8          uint[2] memory y, uint[2] memory a1, uint[2] memory b1,
9          uint[2] memory a2, uint[2] memory b2)
10     public returns (bool);
11
12 function computeTally(VotingDataContract vdc) public;
```

Figure 4.13: *The function declarations of the VotingContract.*

signUp() After a poll has been started by creating a VotingDataContract, users can sign up to participate in the poll. First, the function ensures that the given VotingDataContract is supposed to be used with this VotingContract, and that it is in the correct voting stage. Afterwards, the time is checked whether signing up is still allowed. To participate in the poll, the user has to place a deposit that is returned when voting in the respective stage. Whether a large enough Ether deposit was send with the sign up request is checked next. The last check ensures that the user is an authorized voter for this vote, i.e., the user uses a SharedIdentityContract with known program code and the user has been authorized as a voter. After these security checks,

the voting public key `xG` of the new voter is stored in the VotingDataContract. To ensure that the voter really knows the private key for the voting public key `xG`, a zero-knowledge proof consisting of `vG` and `r` is submitted to the function and automatically checked before storing the voting public key.

finishSignUp() After the time for the SIGNUP stage has expired, `finishSignUp()` can be called by the voting administrator to proceed to the next stage. Before the stage is advanced, the function checks that the poll is currently in the SIGNUP stage, that at least two voters signed up, and that the time for the SIGNUP stage has passed. When the checks are successful, the reconstructed keys g^{y_i} are calculated and stored in the VotingDataContract. Finally, the stage is advanced and the voting itself can be started.

submitVote() In this stage, all signed up voters that have not voted yet can submit their votes. Similar to the checks of the other functions, this function checks whether the poll is currently in the VOTE stage and whether its time has not expired yet. Before the vote `y` is stored in the VotingDataContract, the zero-knowledge proof consisting of the other function parameters is checked to ensure that the vote is either 1 or 0. After voting successfully, the voters receive their deposit back.

computeTally() After all voters have submitted their vote or the time allotted for the VOTE stage is over, the voting administrator can end the poll by computing the voting result. The result is calculated as described in section 4.2.2.1, and stored in the VotingDataContract. If the poll passed, the function `updateAttribute()` is called on the SharedIdentityContract of the user that should receive the new attribute. Afterwards, the poll is over and the voting administrator can delete the VotingDataContract to reclaim some of the Ether spend in its deployment.

4.2.3.3 VotingDataContract

The purpose of the VotingDataContract is to store all data required for a currently running poll. The variables it contains are displayed in figure 4.14. While all of them are declared as **`public`**, this only means that other smart contracts can read the variables directly. To write them, a number of trivial functions are present which take the new values and store them in the respective contract variables. However, they additionally verify that the caller of the function is the VotingContract at a known address, which already checked the requirements to set the variable. This way, as much code as possible has been moved to the static VotingContract, reducing the cost of deploying the VotingDataContract.

Most of the stored variables are required by the implementation in [MSH17]. However, the authorization checks for modifying the variable values had to be added, since they should only be set by the VotingContract. Additionally, storing the to-be-assigned attribute in the VotingDataContract is not required for the original voting system.

```
1  SharedIdentityContract public target;
2  address public pkService;
3  string public key;
4  byte public flags;
5  bytes public data;
6
7  VotingContract public votingContract;
8  address public votingAdministrator;
9
10 Stage public stage;
11 uint public timeNextStage;
12 uint public totalregistered;
13 uint public totalvoted;
14 uint[2] public finaltally;
15
16 struct Voter {
17     address addr;
18     uint[2] registeredkey;
19     uint[2] reconstructedkey;
20     uint[2] vote;
21 }
22 mapping (address => uint) public addressid;
23 mapping (uint => Voter) public voters;
24 mapping (address => bool) public registered;
25 mapping (address => bool) public votecast;
26 mapping (address => uint) public refunds;
```

Figure 4.14: *The data stored in the VotingDataContract.*

Attribute data The first block of variables, in lines 1-5, contains the data that should be stored in the attribute of the linked SharedIdentityContract `target`. This data should be added as an attribute of the service identified by `pkService`.

Access control The variables in line 7 and 8 are used to prevent unauthorized modification of the stored data. All calls to the functions of the contract have to be send by the VotingContract. When the poll is done, the listed voting administrator is able to remove the deployed VotingDataContract from the blockchain.

Voting progress In lines 10-14 the variables describing the progress of the poll are stored. The variable `stage` describes the current stage of the poll: one of SIGNUP, VOTE, or FINISHED. `timeNextStage` is the time when the next stage should start and interactions for the current stage are no longer possible. Whether this temporal requirement is fulfilled is checked by the VotingContract. `totalregistered` and `totalvoted` are counters used to determine the progress of the current stage, and, e.g., decide whether the VOTE stage can be ended early because all voters have already voted. This data could be calculated by checking the status of the registered voters. However, this would be more expensive than storing the counter in the contract. Additionally, the final result of the poll will be stored in `finaltally` after it has been computed by the VotingContract.

Voter status The lines 16-26 store the current status of the registered voters. Each voter is identified by their public key, represented by the datatype `address` in the smart contract. For each voter it is stored whether they registered, voted, how much deposit they placed, which voting public key they registered, and what their respective calculated reconstructed key and vote are.

4.2.4 Poll Execution

In this section, an exemplary execution of a poll is presented in detail. As introduced in the example scenario in section 4.2.1, two moderators of an online service S, named Alice A and Bob B, want to add an attribute to the user Thomas T. They all have SharedIdentityContracts with the service, i.e., $SIC_{\{A,S\}}$, $SIC_{\{B,S\}}$, and $SIC_{\{T,S\}}$. The two SharedIdentityContracts of Alice and Bob contain an attribute marking them as voters for the service S. Since the real-world identities of the users are not known within DecentID, the integrated voting system should only be used for informal opinion polls. As such, only a small number of users is expected to participate in each poll, and a different subgroup of the authorized voters might participate each time.

4.2.4.1 Preparing the poll

To start a poll, one of the authorized voters of the service S, Alice A in this example, calls the function `startVoting()` on the SharedIdentityContract $SIC_{\{T,S\}}$ of the target user Thomas T, where the attribute should be modified at. When doing so, Alice has to provide the address of her SharedIdentityContract $SIC_{\{A,S\}}$, the address of the service S, and indexing key, flags, and data for the attribute. Depending on the given flags and on whether the indexing key already exists at $SIC_{\{T,S\}}$, the poll is either

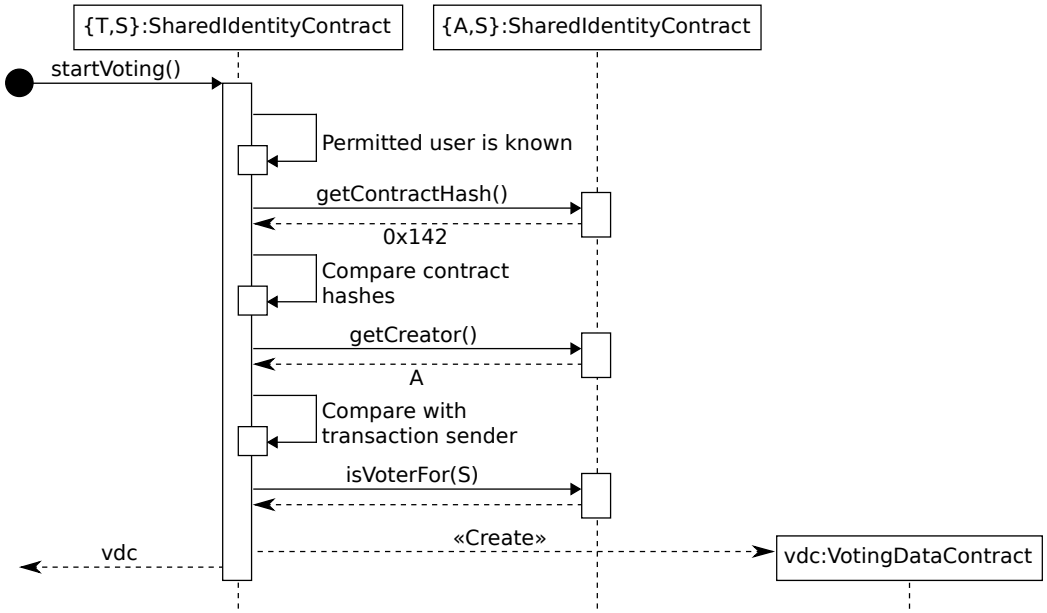

Figure 4.15: *Sequence of starting a poll for a SharedIdentityContract $SIC_{\{T,S\}}$.*

about adding, modification, of removal of the attribute. Since Alice started the poll, she becomes the voting administrator for this poll and has to advance the stages of it. The purpose of the function `startVoting()` is to create the required VotingDataContract vdc that is used to store the state of the poll. Before the smart contract creates the vdc, a number of prerequisites are checked. This is depicted in figure 4.15.

- First, it is checked whether the service S is a permitted user in $SIC_{\{T,S\}}$. If it is not, S has no right to modify the attributes attached to the shared identity, and consequently it should not be possible to do so by voting.

- As a precaution, the binary hash of the code of the users $SIC_{\{T,S\}}$ is compared to the hash of the code of $SIC_{\{A,S\}}$. If these hashes do not match, one of the participants is using a modified SharedIdentityContract. This might either happen due to different versions of DecentID being used by the participants, or due to a malicious attempt to bypass the authorization checks. For example, if Alice would be using a maliciously modified SharedIdentityContract, this contract could always return that Alice is a valid voter for any service. Since fully automated code reviews on the blockchain are not possible, it is instead enforced that the same smart contract code, as represented by its hash value, is used by all participants of the poll.

- For the given shared identity $SIC_{\{A,S\}}$, it is checked whether the caller of the function `startVoting()` is the creator of it, i.e., Alice *A*. If this is not the case, a malicious user is possibly trying to abuse the voting permission in $SIC_{\{A,S\}}$ even though $SIC_{\{A,S\}}$ is not their own SharedIdentityContract. Afterwards, it is checked whether $SIC_{\{A,S\}}$ contains the attribute representing voting permissions for service S.

When these check pass, the VotingDataContract vdc is created and its address returned by the function. When creating vdc, the constructor stores the public keys $pubKey_A$ and $pubKey_S$ of Alice and the service as well as the blockchain addresses of $SIC_{\{T,S\}}$ and the static VotingContract used for later security checks. Also, the attribute to be modified is stored. The vdc is now in the *SIGNUP* stage and allows registration of voters that want to participate in the poll. If Alice wants to participate in the poll, she has to register as a voter as well.

4.2.4.2 Registering

In the *SIGNUP* stage, authorized voters are able to register for a started poll, as depicted in figure 4.16. The authorized voters are able to do so by calling the function `signUp()` of the VotingContract by providing the address of the VotingDataContract vdc, that stores all data about this poll, as a parameter. On registration, a voter V provides their public key of an asymmetric key pair used for the voting process, a zero-knowledge proof that the matching private key is known, the address of their SharedIdentityContract $SIC_{\{V,S\}}$, and a small deposit in Ether. In the example scenario, Alice *A* and Bob *B* register as voters for the poll with their respective SharedIdentityContracts $SIC_{\{A,S\}}$ and $SIC_{\{B,S\}}$. Before the voting public key of the new voter is stored, a number of checks are executed to ensure the correct and secure execution of the poll.

- To ensure that the VotingContract vc is permitted to modify the state stored within the VotingDataContract vdc, it is checked that the address of the VotingContract stored within vdc matches the address of the VotingContract vc being executed.
- The stage of the voting process as stored in vdc has to be at SIGNUP.
- The time for the stage is not over yet. If the time is reached, the registering for the vote is no longer permitted and the voting administrator should advance the stage.

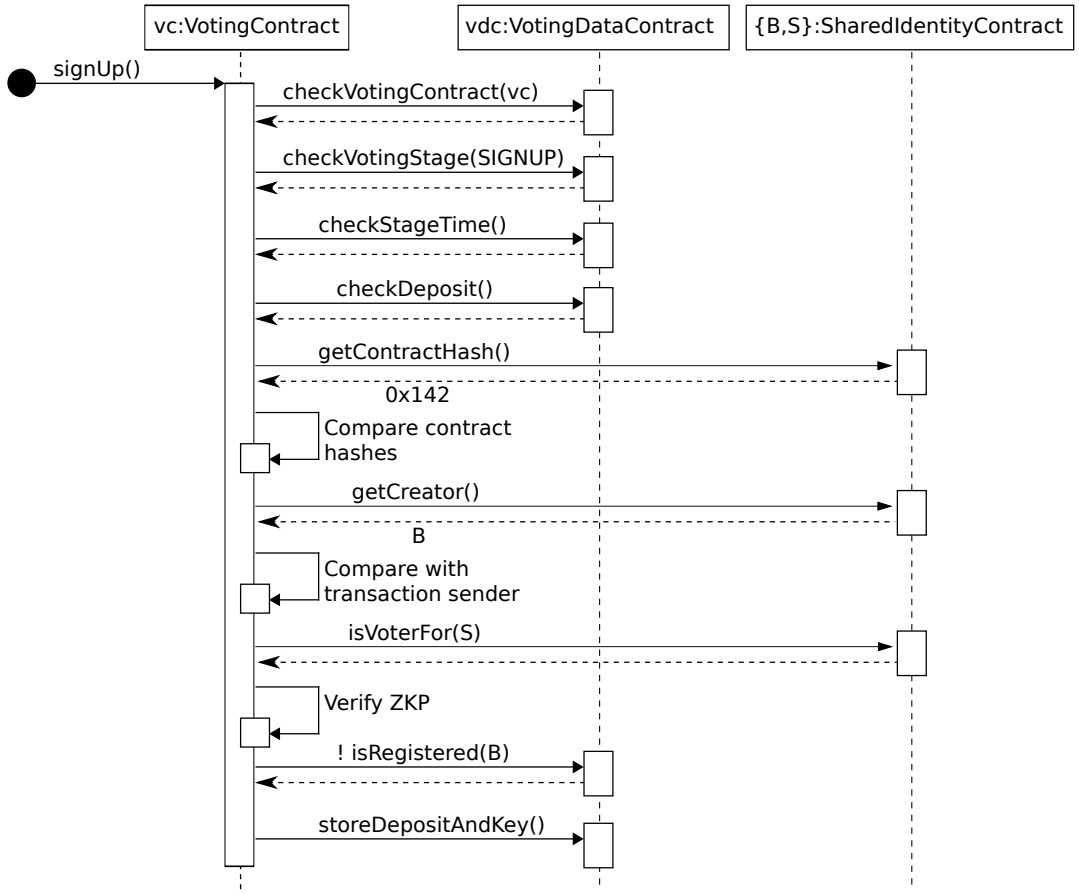

Figure 4.16: *Bob B is calling the `signUp()` function at the VotingContract vc to register for a poll. The poll is identified by the provided VotingDataContract vdc. To register, Bob provides his $SIC_{\{B,S\}}$ and the required voting public key.*

- To stop malicious voters from blocking the poll, a deposit of Ether has to be send with the `signUp()` function call on the VotingContract. If the deposit does not fulfill the expected amount, the function execution is aborted. The deposit ensures that the voter has a motivation to continue with the poll and does not simply abandons it and keeps the other participants waiting.

- When registering Bob B as a voter for the poll, the same checks are executed as before when the voting administrator Alice A created the VotingDataContract vdc: the hashes of the program codes of the SharedIdentityContracts $SIC_{\{T,S\}}$ and $SIC_{\{B,S\}}$ have to be the same, B has to be the creator of the provided SharedIdentityContract $SIC_{\{B,S\}}$, and B has to be an authorized voter for the service S, as stored in vdc.

- To ensure the correct further execution of the poll, a non-interactive zero-knowledge proof has to succeed. This ensures that the voter has the matching private key to the submitted voting public key.
- The voter B must not already be registered as a voter for this poll, i.e., Bobs blockchain public key $pubKey_B$ must not already be stored in vdc.

If the checks are passed, the deposit amount as well as the provided voting public key are stored in the VotingDataContract vdc. Later on, the voter is able to use their voting public key to encrypt their vote.

4.2.4.3 Starting the voting

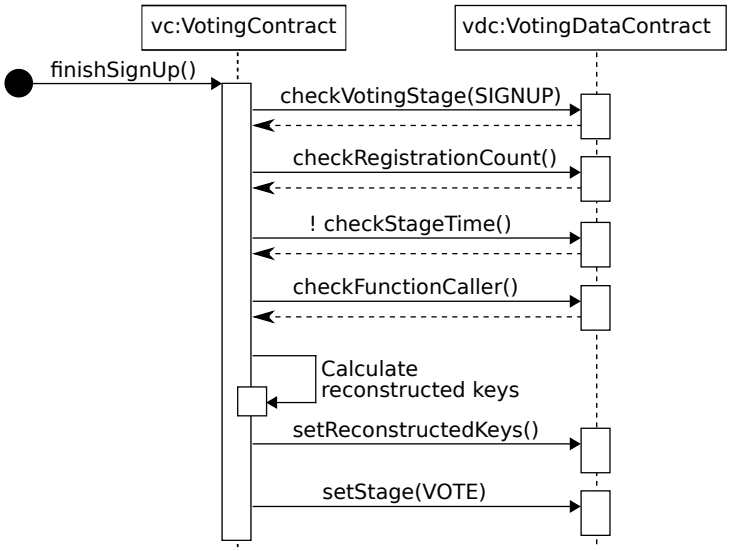

Figure 4.17: *The voting administrator Alice A advances the stage of the poll.*

When the time of the SIGNUP stage is up and at least two voters have registered, the voting administrator is able to advance the stage. Different from the later stages, waiting for the whole time of the stage is important to give the potential voters the possibility to register. To advance the stage, the VotingContract vc offers the function `finishSignUp()`, as depicted in figure 4.17. Only the address of the VotingDataContract vdc is expected as a parameter. Based on the registered voting public keys, which are stored in vdc, the VotingContract vc is able to calculate a number of reconstructed keys used when encrypting the votes. These keys are stored in the VotingDataContract vdc as well. They are calculated in a way that when combining all voting public keys, reconstructed keys, and votes, only the votes remain.

So while the votes are encrypted individually, the final result can be decrypted after all registered participants casted their votes. Consequently, the poll cannot complete and has to be restarted if some of the registered participants do not vote.

4.2.4.4 Voting

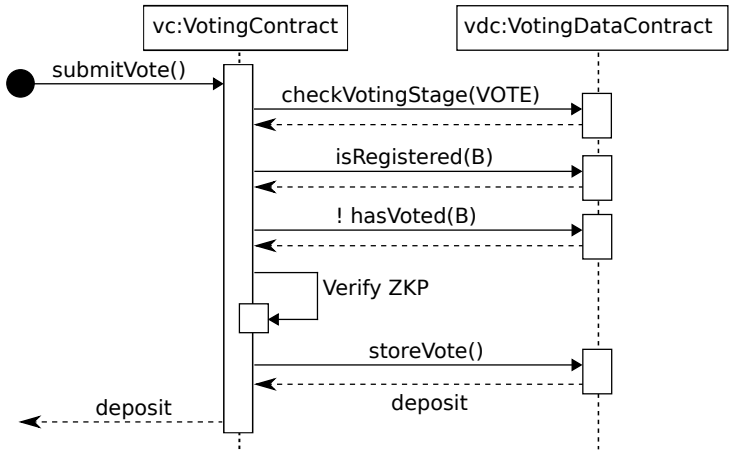

Figure 4.18: *Bob B votes for the poll. The same sequence is repeated for all voters.*

In the VOTE stage, the registered voters submit their encrypted vote as well as a zero-knowledge proof to the VotingContract, by passing them as parameters to the function `submitVote()`. Its execution is shown in figure 4.18. As for all other voting operations, the address of the VotingDataContract vdc has to be provided.

For the voter Bob B his public key $pubKey_B$ is used to check that he registered as a voter before, and that he has not voted in this poll yet. Additional checks, e.g., whether Bob is an authorized voter for the service, are not required again. If a user on Ethereum is able to send a blockchain transaction signed with from $privKey_B$, they are considered the owner of $pubKey_B$. They could only have been added as a voter in the smart contract vdc, after they have signed up at the VotingContract. As such, it must be the same user that previously registered with their checked SharedIdentityContract $SIC_{\{B,S\}}$, making another check unnecessary.

Afterwards, the one-out-of-two zero-knowledge proof is verified. It proves that the encrypted vote either has the value 0 or 1, without disclosing which of the two it is. If this check is passed, the submitted vote is stored in vdc. Since this voter has now cast their vote, their deposit is returned to them.

4.2.4.5 Ending the poll

When all participants voted, the voting administrator can call the respective VotingContract function `computeTally()` to calculate the result of the poll. Its successful execution is depicted in figure 4.19.

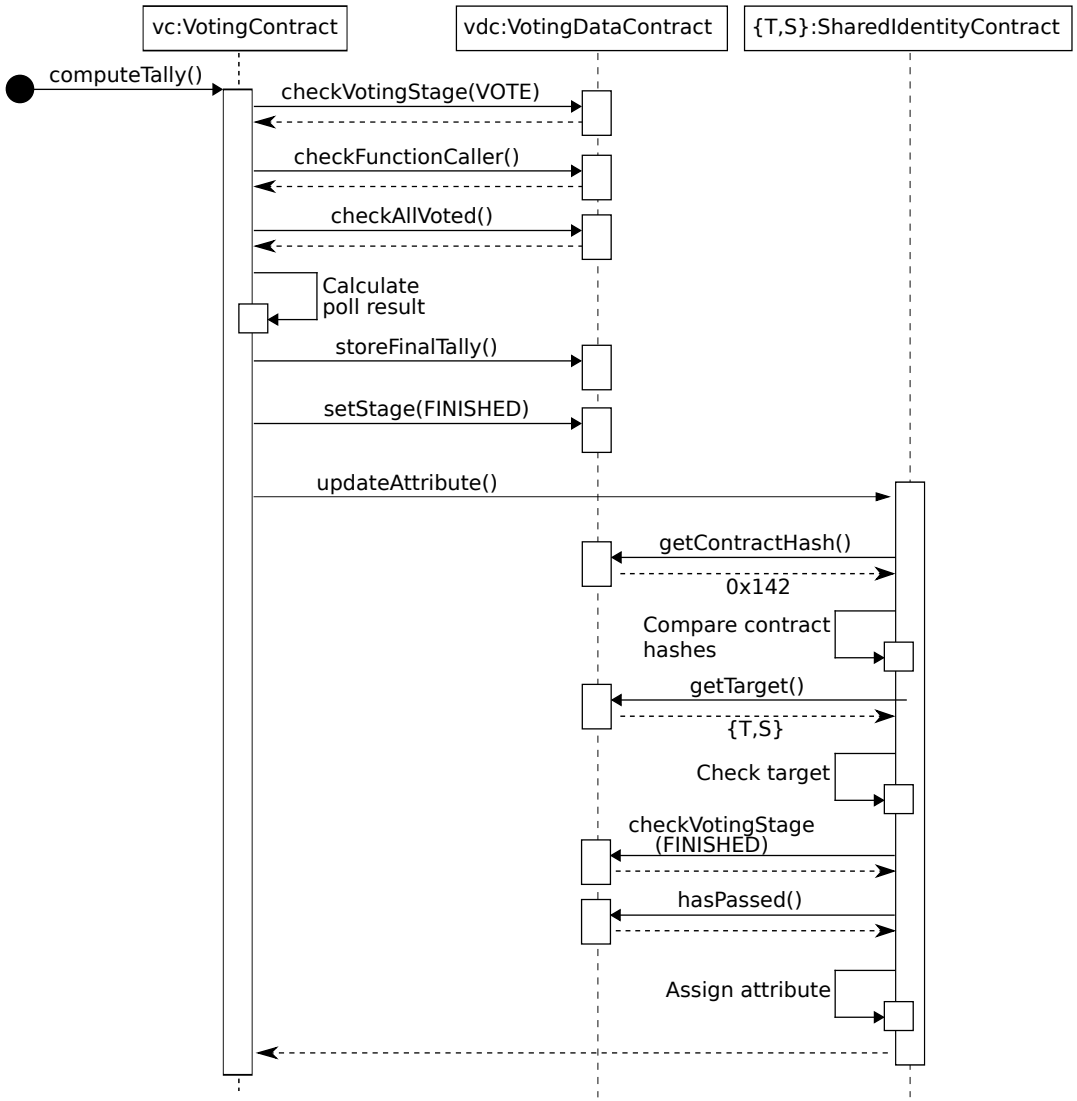

Figure 4.19: *The voting administrator Alice A ends the poll, assigning the attribute.*

The calculation of the result is done the same way as in [MSH17]. As a first step, the mathematical product of all the submitted votes is calculated. Due to the way the encryption keys for voting have been constructed, they cancel each other out while doing so. What remains is $g^{\Sigma_i v_i}$, the base g of the cyclic group used for encryption

raised to the power of the sum of the votes v_i. Since the number of participants is known and quite small, the logarithm can be calculated by brute forcing it. Summing up g should result in $g^{\Sigma_i v_i}$ after only a small number of repetitions j less than or equal to the number of voters. When j is found, it is equal to the number of "yes" votes in the poll. Consequently, if j is greater than half the number of participating voters, the poll passed and the attribute will be assigned. After calculating the result, the stage of the poll is set to FINISHED and the final tally is stored in vdc.
Normally, a SharedIdentityContract only accepts attribute modifications from permitted users registered within it. For voting, the SharedIdentityContract contains an additional function `updateAttribute()` that accepts the address of a vdc as a parameter. Before the attribute is assigned, the state of the passed vdc is verified. These checks include whether the hash of the vdc matches a known hash of a VotingDataContract, whether the target SharedIdentityContract of the vdc is the called SharedIdentityContract, whether the poll is finished, and whether the poll passed. Afterwards, depending on the attribute flags stored in the VotingDataContract vdc, the new attribute stored in vdc is either written to the SharedIdentityContract $SIC_{\{T,S\}}$ as an attribute of the permitted user S, or removed based on its indexing key. When this is done, vdc is no longer needed and can be removed by the voting administrator.

4.2.5 Evaluation

In this section, the integration of Open Vote Network into DecentID is evaluated. This includes the modifications on DecentID to support the voting, as well as an analysis regarding the voters privacy. For an evaluation of the voting process itself, see the original publication of Open Vote Network [Zie10].

4.2.5.1 Security

While the integration of Open Vote Network into DecentID offers new abilities for the users of DecentID, it also offer new attack approaches for malicious users. One such approach is that an adversary might try to forge the state of a VotingDataContract vdc to use it to manipulate the attributes in a SharedIdentityContract $SIC_{\{T,S\}}$ by passing the address of vdc to the function `updateAttribute()` in $SIC_{\{T,S\}}$. If successful, the adversary would be able to add, modify or remove attributes of any permitted user in any SharedIdentityContract. To protect against this, the function `updateAttribute()` checks that the received smart contract vdc uses the same program code as a known benign VotingDataContract. If they use the same program code, the adversary would have been unable to modify the code of the smart contract to their advantage. To

avoid manipulation of the data stored within a VotingDataContract, only the linked VotingContract is permitted to call its functions. The VotingContract itself verifies that its callers are authorized to call its functions as well, as described in the previous sections. So as long as the guarantees of the blockchain, namely that the smart contract code is executed as written, hold, the added voting functionality should not offer new attack approaches.

4.2.5.2 Voting for off-chain attributes

In the previous paragraphs, voting has only been described for modifying on-chain attributes. Due to the restrictions of the blockchain Ethereum, smart contracts can only access and modify data stored on the blockchain itself, meaning that only on-chain attributes can be assigned by voting. However, the voting administrator could store arbitrary attribute data in an off-chain storage by themselves. Afterwards, not the attribute data itself would be assigned to $SIC_{\{T,S\}}$ within the poll, but instead a reference towards the off-chain attribute. The other voters would still be able to retrieve the off-chain attribute, allowing them to verify what they are voting for.
For off-chain attributes it can be useful to contain a digital signature by their creator, i.e., service S, to ensure their integrity and authenticity. When assigning off-chain attributes by voting, this is not possible, though, since this would require the private key of the service. However, the hash value of the attribute data can be stored in the assigned attribute within the SharedIdentityContract, ensuring the integrity of the off-chain data. Since the attribute in the SharedIdentityContract $SIC_{\{T,S\}}$ is assigned as an attribute of the service S, the authenticity of the data is known to the user T as well: Only the service S or authorized voters are able to assign this attribute, which means that the referred off-chain attribute has been endorsed by them.

4.2.5.3 Voting for encrypted attributes

To keep the attribute data confidential, it should be encrypted before the poll is started. This way, the already encrypted data is included in the VotingDataContract, which means that the unencrypted data is never seen on the blockchain. Technically, it does not matter for the execution of the poll whether the to-be-assigned attribute data is encrypted or not. The used smart contracts require no access to the unencrypted attribute data, since the contents of the attribute are not relevant for the execution of the poll. However, the voters need to know the contents of the attribute they are voting about. Voting to add an unknown attribute would mean that the voting administrator could add arbitrary attributes, making the joint decision to add the attribute meaningless. Still, by encrypting the attribute data it could be avoided to

display the data publicly by sharing the unencrypted data only between the authorized voters by off-chain communication. If, e.g., asymmetric encryption is used to encrypt the attribute, the voters are able to locally encrypt the shared unencrypted attribute with the known public key, and verify that the result matches the encrypted attribute as stored in the VotingDataContract.
A challenge when the attribute is to be encrypted is the question which encryption key to use. Normally, attributes are encrypted with the symmetric encryption key $kSIC_{\{T,S\}}$ which is stored in $SIC_{\{T,S\}}$. However, this key is only available to permitted users and not to the voters. An alternative would be to encrypt the new attribute with the public key of either the creator T of the SharedIdentityContract $SIC_{\{T,S\}}$ or of the service S that is a permitted user in $SIC_{\{T,S\}}$. A drawback of this alternative is that only the respective permitted user can decrypt the attribute. Instead, a new symmetric encryption key k could be generated and used to encrypt the attribute. Then, copies of k could be respectively encrypted with the public keys of all permitted users and the encrypted copies of k could be stored together with the attribute. This approach would be similar to the way $kSIC_{\{T,S\}}$ is stored and encrypted in $SIC_{\{T,S\}}$.

4.2.5.4 Privacy analysis

Regarding the privacy of the voting approach and its integration, multiple facets have to be considered. These are discussed in the following.

Privacy of the vote The use of Open Vote Network ensures that the vote of each user is kept secret and cannot be discerned from the data written to the blockchain [Zie10]. However, since the total number of approving votes is calculated by Open Vote Network, it is possible to calculate the vote of a certain user if the votes of all other voters are known to an adversary. This $n-1$ attack, with n being the number of voters, is a frequent problem in privacy preserving approaches, and the attack cannot be avoided. When only two voting options are available, e.g., for voting in DecentID, the attack can be improved. If, e.g., the adversary knows the votes of c of n voters and knows that they have all voted against the proposal, but there were $n-c$ votes for the proposal, the adversary knows that the uncompromised voters voted for the proposal. Nevertheless, the risk due to this attack is assumed to be low. To determine the votes of the individual voters, the adversary has to either compromise the endsystems of the voters, or participate in the poll with many own voters. Both variants are unlikely given the effort required to do so. Additionally, depending of the context of the poll, the vote of a specific user might not be worth knowing for the adversary. As long as the polls executed within DecentID are only informal opinion polls, the effort required from an adversary might be larger than their gains.

Privacy of the voting permission While the vote itself is encrypted, the voting permission can be seen on the blockchain. Furthermore, since the voting permission has to be checked automatically by a smart contract, the permission is currently stored as an unencrypted attribute. As such, it is possible for third parties to find out whether a shared identity is a voter for a certain service. Encrypting the voting permission is by itself not sufficient to protect the privacy of the voter status. For once, since the attribute has to be verified by a public smart contract, all data to check the voter status has to be public. Consequently, everyone else is able to verify the voter status as well. Also, when setting the voter permission a publicly visible blockchain transaction has to be used. Since the indexing key for the voting attribute has to be known and fixed, writing data to this attribute will reveal the voter status. This metadata leak could be fixed by assigning this attribute with a value of "no-voter" to all SharedIdentityContracts shared with the service. However, as long as the other leaks exist this does not improve the privacy. At last, participating in a poll as voter will reveal the status as voter permanently on the blockchain. As long as the participation in a poll is publicly visible, protecting the state of the attribute does not improve the privacy of the user.

Privacy of the participation Besides being out of scope for this work, since it is a restriction brought by the use of Open Vote Network, hiding the participation in the poll is a difficult problem. An relatively easy solution would be to abandon the idea to execute the poll on the blockchain and execute it in private off-chain. This would protect the participation of users, but brings up the question whether the result of the vote can be trusted, and how this trust can be proven towards the smart contracts on the blockchain to set the desired attribute. An on-chain approach could possibly use "dummy" voters, which participate in the poll without their votes being counted. However, if the VotingContract can determine whether a vote should be ignored, then an adversary can do so as well. Using equal numbers of "yes" and "no" dummy voters would avoid that problem, since their votes would not change the result of the poll. If observers of the poll do not know how many dummy voters were added the result would look distorted, though. Seemingly, many voters participated and the result of the poll would still nearly be a draw. This could be interpreted as the voters being split about the poll, even though all real voters might have voted the same. How the numerous dummy voters are selected and in which way they secretly receive the information that and how they are supposed to vote, is an open question that is out of scope for this work. Additionally, it would not be possible to verify whether they really voted as they were supposed to. Since this can open up new ways to attack the voting system, a solution for this will probably not be trivial.

4.3 Conclusion

In this chapter two use cases for DecentID have been discussed: using DecentID as an identity management system for other smart contracts, and extending DecentID by integrating a voting system.
Using the identities for other blockchain-based systems, as well as different aspects for coupling the systems, were analyzed. To maintain the generality of DecentID, providing self-sovereign identities independently of a specific use case, coupling approaches without modifying the smart contracts of DecentID were evaluated. As long as other smart contracts are able to use the interface provided by DecentID, either directly or by using a proxy contract, they can access the shared identities and retrieve the attributes contained within them. While all stored attributes can be retrieved, some of them might be encrypted. Since smart contracts have no access to the private keys required for decrypting the attributes, only unencrypted attributes can be used automatically.
To reduce the dependency on single service administrators, a voting system was integrated into DecentID. It allows authorized users of a service to conduct a poll, which can be used to modify attributes of other users of the service. To support this functionality, the code of the SharedIdentityContract had to be modified. The integration of the voting system requires that attributes can be automatically written to an identity by another smart contract, i.e., the VotingContract. Without the private key of a permitted user this was not possible before modifying the SharedIdentityContract. This modification was implemented in a way that does not limit the already existing capabilities of DecentID.
In conclusion, reading unencrypted attributes of SharedIdentityContracts from other smart contracts is possible without modifying DecentID, since reading the attributes requires no private keys. This was demonstrated by coupling DecentID with Palinodia. Adding support for modifying attributes, however, requires a modification of DecentID to circumvent the requirement for private keys. Integrating the voting system into DecentID demonstrated this.
If DecentID would be designed anew, the adherence to a publicly known and used interface for smart contract identity management would be recommendable. This would permit easier interactions with other smart contracts which require identities for their functioning. Unfortunately, such an interface has not been established yet. Whether such an interface would permit to write attributes to identities remains to be seen. Some kind of access control is required to ensure the authenticity and integrity of the identities. Providing such access control in a generalized fashion independently of a single use case, e.g., integrating voting, will be a challenge.

Chapter 5

Sybil defense

DecentID, as presented in chapter 3, allows its users to create pseudonymous identities. These identities can then be used to access online services. To protect the users privacy, multiple identities of the same user cannot be linked to each other. While this prevents the services from tracking the user, they are also unable to link multiple identities of one user within one service, which this permits *Sybil attacks*.
In these, a single adversary tries to create as many identities within a system as possible. Depending on the attacked system this allows a number of attacks, e.g., manipulating polls or the public opinion. Sybil defense sums up the various approaches to recognize and protect against Sybil attacks. To avoid Sybil attacks when DecentID is used, a Sybil defense system should be integrated into it.
For this only decentralized Sybil defense systems that hinder the creation of Sybil identities are relevant. Centralized systems would introduce a single trust anchor, something that should not exist in DecentID. A Sybil detection system could be used as well, but restricting the Sybil adversary already at registration time is preferable. To detect the Sybil identities within the user group of a system a significant number of existing Sybil identities is required, which should be avoided in the first place.
In this chapter, the Sybil defense system *Detasyr* is presented. After defining the capabilities of Sybil adversaries in section 5.1, the approach of Detasyr is presented in section 5.2. Its design is explained based on its core components, listed in section 5.3, and their operation, as explained in section 5.4. To further improve the Sybil defense capabilities of Detasyr, two possible extensions are discussed in section 5.5, before evaluating the extensions in section 5.6. Finally, the integration of the created proofs of authorization into DecentID is presented in section 5.7.

5.1 Adversary model

The presented Sybil defense approach is based on the trust relationships embedded into online social graphs (see section 2.6). These graphs consist of nodes, representing the users, and edges, connecting the nodes and representing trust between the connected users. It is assumed that the majority of the participants in the network are honest, with a smaller number of nodes controlled by a *Sybil adversary*.

Definition 5.1: Sybil adversary

A *Sybil adversary* wants to create as many Sybil identities as possible, to manipulate or take over the attacked system. The same as for the identities of honest users, each Sybil identity is represented by a node in the social graph.

The adversary tries to create the Sybil identities without other participants of the system realizing that the nodes are controlled by a Sybil adversary, which would lead to them blocking the nodes of the adversary. The following abilities of the Sybil adversary are similarly used in other work [Yu+06; DM09; Wei+12; Vis+10].

Any number of Sybil nodes is possible The Sybil adversary in most systems is able to create as many Sybil nodes as it requires. Since normal users are unable to see the structure of the social graph, the number of and connectivity between the Sybil nodes is unknown to them.

Limited number of attack edges *Attack edges* are the edges between a Sybil node and a node controlled by an honest human user. Since the effectiveness of the system depends on the relationship edges being trustworthy, the users in the social graph should not accept anyone as neighbor but only accept other users they trust not to launch a Sybil attack. Preferably, the neighboring users know and trust each other in the real world as well. It is not entirely sure whether this trust really exists in online social networks. Research has shown that on widespread online social networks the acceptance rate of friendship requests of a fake account is over 20 percent [Bil+09]. For an adversary creating friendship requests automatically this allows to create large numbers of friendships. However, due to the purpose of these networks their users are not highly concerned about their privacy. Another, more privacy oriented, social network could possibly avoid this problem when the users are sufficiently aware of the implications of their friendships. In these social networks, the Sybil adversary would be unable to have an unlimited numbers of honest friends, i.e., attack edges, within the social graph.

Existence of communities As in the real world, online social graphs tend to exhibit a graph structure containing multiple communities [KNT06]. Users have many friendships, i.e., edges between nodes, within a community where the whole community shares interests. Between these communities, fewer edges exist. Since Sybil adversaries are able to create as many nodes as they want, but are only able to create a restricted number of attack edges, a separate Sybil community is present in the social graph. This varying degree of connectivity between the communities, especially towards the Sybil community, can be detected and used by Sybil defense algorithms.

Control over own Sybil community Since the adversary can modify the edges in its community of Sybil nodes arbitrarily, the paths of random walks or the distribution of flooded packets within the Sybil community can be manipulated by the Sybil adversary. Since the structure of the social graph is hidden for the honest users, they are unable to determine whether the Sybil adversary is manipulating the data exchange within its community or is correctly following the protocol. Consequently, the adversary is able to deviate from the protocol in any way. However, sending invalid data to the honest users will lead to the honest users dropping the received data and possibly blocking further communication with the deviating Sybil identity.

Restricted abilities Similar with other work, we do not assume the adversary to have an omnipotent view of the social graph. It is also unable to listen to or manipulate all network traffic within the network. Especially the adversary is not able to manipulate a significant number of honest nodes, since that would allow the adversary total control of the system. A small number of controlled honest nodes can be tolerated, since they do not offer any advantage over having more Sybil nodes.

5.2 Approach

In this chapter the Sybil defense system *Detasyr* is presented, which restricts possibilities for Sybil attacks, works decentralized, and protects the privacy of participating users. An earlier version of Detasyr has been published in [FMZ19].
To protect against Sybil adversaries, the graph structure of an online social graph is utilized. Through this graph small data packets, called *authorization tickets*, are flooded. To become authorized as honest, i.e., not being a Sybil adversary, users have to collect a sufficient number of these tickets. After being authorized as honest, users can prove their successful authorization towards third parties, for example by referencing this proof within an identity attribute of DecentID. This feature is missing in most existing decentralized Sybil defense systems (e.g., [Wei+12] or [Yu+06]).

Due to the structure of the social graph, the influence of Sybil adversaries can be restricted. Since users are assumed to only create connections with users they trust, the number of connections towards the Sybil adversary is restricted. Consequently, the number of authorization tickets that can be collected by Sybil adversaries is restricted as well, and the Sybil adversary can only authorize a bounded number of Sybil identities within the system. The unrestricted creation of Sybil identities and the malicious take over of the system is prevented.
Since honest users only want to authorize their own, single, identity, they are mostly unhindered by this restrictions. Due to Detasyr operating decentralized, the disadvantages of single trust anchors are avoided. Also, the privacy of the users relationships is protected, averting, e.g., linking their digital identities to their real-world ones.

5.3 Core components

The design of Detasyr is based on some core components, namely the ticket sources, a blockchain, authorization tickets, and rounds. These components are explained in this section. Ticket sources execute the management tasks of Detasyr, which includes maintaining the blockchain which stores the authorized nodes. To become authorized, the nodes are required to collect a certain amount of authorization tickets emitted by the ticket sources. In each recurring round, new ticket sources are active, a new block is appended to the blockchain, and new authorization tickets are distributed.

5.3.1 Ticket Sources

Each round a group of *ticket sources* is selected out of the authorized nodes. The ticket sources for the next round are randomly selected by the ticket sources of the current round, which is described in detail in section 5.4.3.2.

Definition 5.2: Ticket Source

Ticket sources (TS) are a group of $s \geq 3$ authorized nodes, that are randomly selected for a single round. They are responsible for the management tasks of Detasyr. Each ticket source is identified by a cryptographic public key.

For the current round, in which the ticket sources are active, they are responsible for the management tasks in Detasyr. This includes the creation of authorization tickets, the verification of authorization requests, the selection of the ticket sources for the next round, and the creation of a new block for the blockchain. These tasks are explained in detail in section 5.4.

Every authorized node has an asymmetric key pair that is used to identify the node. In the case of the ticket sources, it also is used to digitally sign the authorization requests of other nodes as well as the new block for the blockchain.
Selecting new ticket sources each round serves multiple purposes. For once, due to the new placement of the ticket sources, authorization tickets are propagated through different parts of the social graph, allowing nodes to become authorized that might have been not well enough connected in the previous round. Furthermore, it leads to a distribution of effort required to maintain the system. Operating a ticket source is more effort for a node than simple being part of the social graph, so by regularly changing the ticket sources the effort for each single node is reduced. It also increases the security of the system. If a malicious node is elected as a ticket source, it could try to hinder the operation of the system. However, this malicious node is only a ticket source for one round. Additionally, the changing ticket sources pose a moving target for an adversary. If an adversary wants to take over the system, they would need to compromise most of the ticket sources. If the ticket sources are regularly changing, this has to be achieved within a restricted time period.
To increase the efficiency of the protocol, one of the ticket sources is selected as a *prime ticket source*. If the prime ticket source goes offline or acts malicious, the next ticket source in the list of active ticket sources, based on the newest block of the blockchain, takes over its duties. For all management tasks that are executed by the group of ticket sources, a number of messages have to be exchanged. Always sending these messages to all other ticket sources would result in many unnecessary messages, increasing the overhead of the protocol and slowing its execution. Instead, all messages are only send to the prime ticket source. There, the messages are aggregated as required and then send to the other ticket sources which still need to handle the message. For example, the current ticket sources have to exchange the results of their random walks done to select new ticket sources (see section 5.4.3.2). To reduce the number of transmitted messages, all ticket sources send their results to the prime ticket source, which then forwards the collected results to the other ticket sources. While the prime ticket source is trusted with doing these organizational tasks, it has no elevated rights compared to the other ticket sources. Since all actions have to be confirmed and digitally signed by a group of ticket sources, even a malicious prime ticket source cannot decide anything by itself.

5.3.2 Blockchain

One design goal of Detasyr is to enable users of the system to prove to others that they have been authorized. This makes it possible to use the received authorization in other contexts, for example within DecentID. To allow them to do so, the public keys of authorized users are stored in a publicly readable blockchain, together with some other data required for operating Detasyr. Each round a new block, shown in figure 5.1, is appended to the blockchain by the ticket sources.

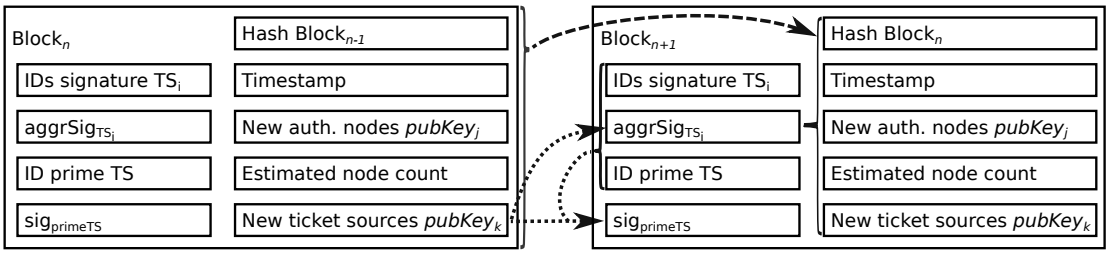

Figure 5.1: *Structure of the data blocks of the Detasyr blockchain.*

The blocks are replicated between the nodes participating in Detasyr, increasing the availability of the stored data. The contained data elements can be divided into three categories: node authorization, management data, and linking the blocks. These categories and their associated block elements are discussed in the following.

Node authorization
The primary purpose of the blockchain is to store a list of public keys $pubKey_j$ of authorized nodes, to allow the nodes to prove their authorization towards third parties. Each node that successfully finished its authorization has its public key recorded in the next published block of the blockchain, enabling the node to prove its authorization toward others.
Individual signatures by the ticket sources over each authorized public key are not required, since the whole block is digitally signed. This allows to reduce the storage space required for the blockchain, which becomes increasingly important if the length of the blockchain, and with that the number of authorized nodes, grows over time.

Management data
While not directly important to a user of Detasyr, some data has to be stored in the blockchain to ensure a continued operation of the system. These block elements are described in the following.

New ticket sources As part of the current round, new ticket sources for the next round are selected. These new ticket sources are randomly selected out of the nodes that have been authorized in earlier rounds. The public keys $pubKey_k$ of the new ticket sources are stored as part of the block published at the end of the current round. This way, the nodes participating in the next round can verify that the active ticket sources have been selected by the ticket sources of the previous round. When multiple blocks are considered, these public keys increase the security since they form a forward directed link between the blocks, complementing the hashes linking back to the previous blocks. Since the public keys $pubKey_k$ are listed in a fixed ordering, it becomes possible to reference ticket sources by their ID within the block.

Timestamp A timestamp contained in the block documents the time when the next round begins. After this time, the previous ticket sources return to being normal authorized nodes, while the newly selected ticket sources take over their duties. Also, authorized nodes can use this timestamp to prove that they have become authorized at a certain time.

Estimated count of active nodes The number of currently active nodes is used to determine how many authorization tickets have to be flooded through the graph. While the number of authorized nodes is known and can be calculated based on the authorizations stored in the blockchain, some of these nodes might temporarily or permanently be offline when a new block is created. As such, the number of active authorized nodes is most likely smaller than the number of authorized nodes. Since the number of active nodes is required at the beginning of a round, their number is determined in the previous round as described in section 5.4.3.1 and stored in the new block of the blockchain.

Linking the blocks

To ensure the security and integrity of the blockchain, the blocks have to be cryptographically linked to each other, and the integrity of the blocks data has to be ensured. The link towards the next block, by listing the future ticket sources, has been described above. To link the previous block, a cryptographic hash is used. To ensure the integrity and authenticity of the block, the current ticket sources digitally sign its contents.

Hash of the previous block The blockchain of Detasyr, similar to the blockchains of existing cryptocurrencies, consists of a cryptographically secured chain of publicly known data blocks. As in the blockchains of existing systems like Bitcoin, each block contains a cryptographic hash of the previous block [Nak08]. This creates a linked

list of blocks, stopping adversaries from replacing older blocks within the blockchain. If any of the older blocks would be modified, all newer blocks would need to be replaced by the adversary as well, replacing the whole blockchain.

Signature over the block Different from existing public blockchains, no *proof of work* is used in Detasyr. Instead, the active ticket sources are the only ones able to create a new block. This is ensured by an aggregated signature over the previously listed contents of the block, created by the active ticket sources. Based on the public keys $pubKey_k$ in the previous block and the signature, all nodes can verify that the block has been created by the selected ticket sources, ensuring the integrity and authenticity of the current block.
To avoid a stalling system, only a configurable percentage $b \in]0.5, 1.0[$ of all s ticket sources are required to participate in the aggregated signature. As such, up to $(1-b)*s$ offline or malicious ticket sources can be tolerated. For the verification of the aggregated signature $aggrSig_{TS}$ the public keys of the participating ticket sources are required. To select the respective public keys $pubKey_k$ in the previous block, their IDs within the block are required. Consequently, the list of IDs is stored in the block as well.

Signature of the prime ticket source When the new block is complete and the aggregated signature has been created by the current group of ticket sources, the prime ticket source adds the aggregated signature, the list of IDs of participating ticket sources, and its own ID to the block. These new data elements of the block are signed by the prime ticket source, and the new signature added to the block as well. This additional signature protects the signed data elements against, potentially malicious, manipulation. If the list of IDs or the aggregated signature would be modified, the verification of the integrity and authenticity of the block would fail.
If required, each of the currently active ticket sources could declare itself the prime ticket source for block creation. This is possible since the ID of the public key of the prime ticket source is not digitally signed by the other ticket sources. However, without the signatures created by the other ticket sources a new block cannot be created, meaning that the other ticket sources have to cooperate with the prime ticket source. This allows to replace the prime ticket source if the initial prime ticket source is not able to finish the creation of the block, e.g., because it went offline.

5.3.3 Authorization Tickets

To differentiate between Sybil and honest nodes, *authorization tickets* are used.

Definition 5.3: Authorization Ticket

Authorization tickets are signed data packets flooded through the social graph by the ticket sources. If a node receives enough authorization tickets, it can become authorized.

New nodes, that have not been authorized yet, need to collect a sufficient number of authorization tickets to become authorized. They are issued by ticket sources and flooded through the social graph, following the trust relationships between users. The basic idea is that only nodes well connected within the social graph receive enough tickets to become authorized, while Sybil adversaries are unable to receive sufficient tickets to create a large amount of Sybil nodes [Tra+11].

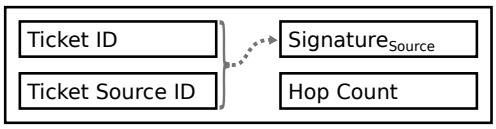

Figure 5.2: *Structure of authorization tickets.*

The structure of an authorization ticket is depicted in figure 5.2. The contained ticket ID is unique regarding the creating ticket source. It is used to differentiate between the tickets created by a specific ticket source, in order to avoid that the same ticket is used for multiple times. The sending ticket source is referred to by its ID within the list of ticket sources in the blockchain. The signature digitally signs the IDs of the ticket and of the ticket source, protecting the integrity of the ticket and proving that it has been created by the referenced ticket source. Based on the ID of the ticket source, receiving nodes are able to retrieve its public key from the blockchain and verify the signature. Adding this public key to the authorization ticket would be possible, but the receiving user would still have to access the blockchain to ensure that it belongs to a valid ticket source. The hop count ensures that flooded tickets do not circulate though the graph indefinitely.
Authorization tickets only stay valid for one round and can no longer be verified when new ticket sources become active. When a new round begins, the public keys of the new ticket sources are present in the most recent block of the blockchain. Now, the public key referenced within the authorization ticket by the ID of the ticket source refers to the public key of the new ticket source. Consequently, the signature in the ticket can no longer be verified.

5.3.4 Round-based operation

Since each authorization ticket can only used once, periodic flooding of new tickets is required. To do so, the operation of Detasyr is divided into repeating *rounds*.

Definition 5.4: Round

In each *round*, new authorization tickets are created and flooded through the social graph, allowing further nodes to become authorized. In parallel, some management tasks are executed, e.g. selecting new ticket sources and counting the number of active nodes.

In the following it is assumed that the system has already been running for some time, i.e., some authorized users already exist and ticket sources have been selected. Additionally, some new nodes want to become authorized.
The round starts after a new block has been added to the blockchain. The new ticket sources listed in the new block start the round by sending tickets through the graph, allowing the authorization of new nodes. In parallel to sending the tickets, two preparing steps for the next round are executed: counting the number of currently active nodes, to determine the number of authorization tickets that have to be created in the next round (see section 5.4.3.1), and selecting the new ticket sources for the next round (see section 5.4.3.2).

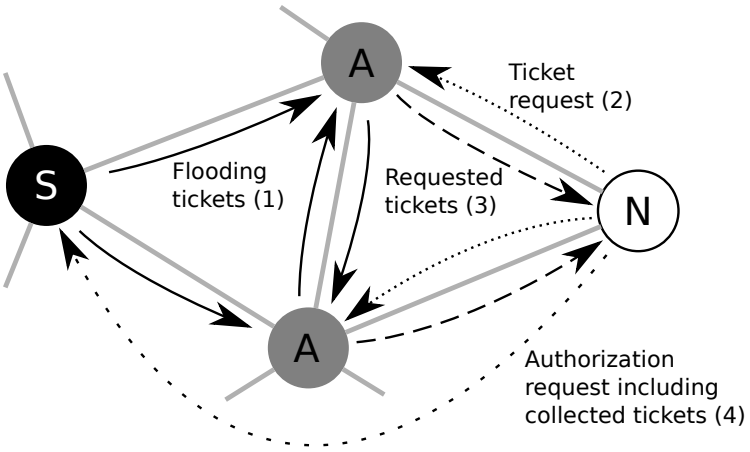

Figure 5.3: *Example of the data for authorization being passed by neighbors through the social graph between ticket sources (S), authorized nodes (A), and new, unauthorized, nodes (N).*

The high-level procedure of a node to become authorized is depicted in figure 5.3 and described in the following. A detailed explanation can be found in section 5.4.4.

(1) Flooding tickets All ticket sources flood a number of tickets through the social graph to the authorized nodes. The number of flooded tickets is based on the number of currently active authorized nodes. Each authorized node keeps one ticket from each ticket source for themselves and forwards the remaining tickets to its other authorized neighbors.

(2) Requesting tickets If a new node wants to become authorized, it sends ticket requests to its authorized neighbors. When a node is well connected within the social graph, tickets from more neighbors can be requested.

(3) Receiving tickets The received ticket requests are answered with the tickets the neighbors collected. Due to the assumption that Sybil nodes have only a limited number of neighbors, they will only receive a small number of authorization tickets.

(4) Authorization request When the new node has received tickets from enough ticket sources, it creates an authorization request. This request is send back to an active ticket source. The ticket sources verify the validity of the received authorization request and, if the request is valid, add the public key of the new node to the next block of the blockchain.

When the authorization requests received by the ticket sources have used most of the created tickets or no authorization requests have been received for some time, the round ends. The assumption is that most authorization tickets have either been used or are stored at nodes without unauthorized neighbors. In the latter case, the stored tickets are of no use and no further authorizations are possible in this round. This is detailed in section 5.4.3.3.
To end the round, the current ticket sources prepare and publish the new block of the blockchain. When publishing the block, the old ticket sources become normal authorized nodes again. At the same time, the new ticket sources start a new round by flooding new authorization tickets through the graph. The new block also contains the public keys of the nodes that managed to send valid authorization requests in this round, making them authorized nodes. An explicit synchronization between the nodes is not required for changing the round, and diverging local clocks at the different nodes can be tolerated. As soon as the nodes receive the newest block of the blockchain, they are able to verify the contained signatures to ensure the correct linkage of the blockchain. Afterwards, they are able to verify the contained signatures of messages send and signed by the new ticket sources. At which time this happens is not important for the functioning of the system. Remaining authorization tickets of the previous round can be removed.

5.4 Design

In this section the design of Detasyr is described in detail, based upon the core components presented in the previous section. First, the required basics and assumption are explained, before the details of the ticket creation and forwarding are presented. Then, a number of management tasks that need to be executed by the ticket sources are discussed. Finally, how new nodes use the received authorization tickets to become authorized is explained.

5.4.1 Basics and assumptions

Based on the assumption that users only accept trustworthy friends as neighbors in the social graph, the Sybil adversary only has a restricted number of neighbors and is unable to convince unlimited numbers of users to add its Sybil identities as friends. The trust relationships between users in the social graph are represented by network connections between the nodes operated by the users, e.g., as software on their smartphones. Each node only knows about its relationships to its directly neighboring nodes, no global view of the social graph is available. This protects the privacy of the users since it has been shown that it is possible to infer the real-world identities of users from the structure of a social graph, even when the individual identities are anonymized [NS09].
Each node within Detasyr owns an asymmetric digital key pair for the Boneh–Lynn–Shacham (BLS) signature scheme [BLS04]. For a short introduction into BLS signatures, see section 2.3.4.2. To identify a node when sending messages through the network, the public key is used. As usual for asymmetric cryptography, the private key is used to create digital signatures used to prove the authenticity of send data. Compared to other asymmetric signature schemes, BLS signatures offer the advantage that they are quite small and can be aggregated. Out of the signatures contained in the flooded authorization tickets, a single signature can be calculated. With this signature, each of the single tickets can be verified by using the public key of the respective ticket source that created the ticket. The advantage of the aggregation is that less data has to be transmitted and stored since only one signature is used instead of one signature per ticket. Instead of BLS signatures, another signature scheme could be used as well. While BLS signatures offer these advantages, Detasyr does not depend on this features.
For the following description of the design, it can be assumed that the users of Detasyr are always online, even after they have been authorized. This is no restriction of the presented approach: nodes going offline do not hinder the execution of the

protocol. Measures to ensure the continued operation are explained where required. Regarding the structure of the social graph, offline nodes can be treated as if they do not exist. For the neighbors of the offline nodes this means that they have less connections to other nodes. In some cases, this can result in nodes that temporarily are no longer able to become authorized or even are disconnected from the social graph. While unfortunate for the affected nodes, it is unproblematic for the other nodes or the general execution of the system. When running only Detasyr itself as an own application, the assumption that all nodes are always online is probably unrealistic, since the participation induces network traffic and requires computation efforts. Due to this, the participation within the system should be included in a popular service, e.g., integrating Detasyr into the app of some web service that wants to avoid Sybil attacks.

5.4.2 Ticket creation

The most important factor to restrict the number of Sybil nodes that can join the system each round is the creation, flooding, and collection of authorization tickets. Only a restricted number of tickets are flooded each round, so that only well connected nodes receive tickets, i.e., nodes that are in the same community as the emitting ticket source. Due to the common assumption that Sybil nodes only have a limited number of honest friends in the social graph, they receive a restricted number of tickets and are less likely to be able to authorize additional Sybil nodes [Yu+08; DM09].

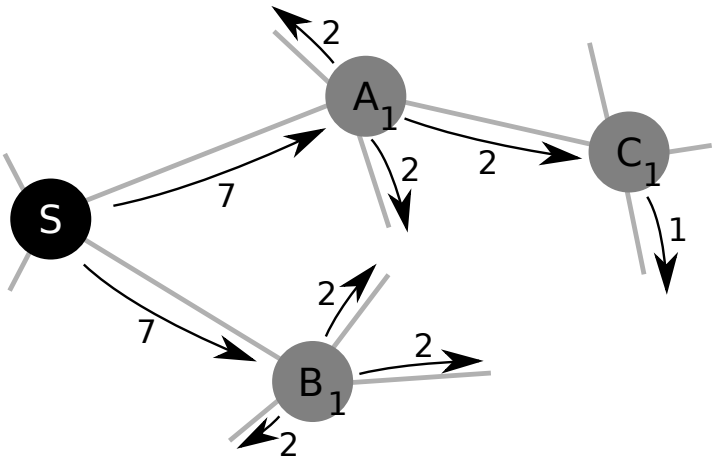

Figure 5.4: *Created by the ticket source (S), the labeled amounts of authorization tickets are flooded through the social graph between the nodes (A-C). Node C is unable to forwards tickets to all its neighbors.*

Tickets are created by the currently active ticket sources. The number of tickets created per source is based on the estimated number of active nodes in the previous round, as stored in the newest block of the blockchain, multiplied by a fixed factor $t \in]0, 1[$. Each ticket source creates its tickets independently from the other ticket sources and floods them to its neighboring authorized nodes. If a node receives a ticket from a ticket source, where it has not received a ticket from yet, it keeps one of the tickets for itself. After reducing the hop count of the tickets, the other tickets are evenly distributed to all authorized neighbors, except the one the tickets have been received from. This process is depicted in figure 5.4.
Due to BLS signatures being used for the tickets, the signatures of multiple tickets can be aggregated, reducing the amount of data that has to be transmitted. A disadvantage of this aggregation is that it becomes impossible to use only a single of the aggregated tickets for an authorization request, since the aggregated signatures cannot be separated again. Since for each authorization request only a single ticket from each ticket source is required, this would lead to tickets that cannot be used. Consequently, the signatures of multiple tickets of one ticket source should not be aggregated, while aggregating tickets from different ticket sources is preferable to reduce the amount of data that needs to be stored and transmitted.

5.4.3 Round management

In each round of Detasyr a new block with the newly authorized nodes is published, new ticket sources are selected and new tickets are flooded. This subsection deals with the management of these rounds, i.e., the different tasks ticket sources have to do to run the system. These and the other presented procedures are running in parallel, i.e., counting active nodes, selecting new ticket sources, distributing tickets, and handling authorization requests can all be done in parallel.

5.4.3.1 Counting active nodes

The number of currently active nodes is used to set thresholds for other parts of the system. As an example, the number of active nodes is used to determine the number of authorization tickets flooded through the social graph by each ticket source. When using a fixed number of tickets, either too many tickets would be created, resulting in too many Sybil nodes joining the system, or not enough ticket, resulting in nodes not receiving tickets and being unable to become authorized. In centralized systems, the number of active nodes is known to the single centralized server. Due to the decentralized design of Detasyr, no such server exists. While the number of nodes

already authorized in the system is known from the authorizations in the blockchain, not all authorized nodes will be available at all times. Instead, the current number of active nodes needs to be estimated in each round.
For this, each ticket source selects a group of authorized nodes out of the blockchain randomly. The amount of selected nodes is logarithmic in the known total number of authorized nodes. A list containing the public keys of the selected nodes is signed by the ticket source and flooded through the graph. When a node receives the list it first checks the signature created by the sending public and dismisses the list if the signature is invalid. Also, based on local state stored for this round, the node checks whether the list has been received before and dismisses the list if it is not receiving it for the first time. When these initial checks are done, the node checks whether its public key is part of the received list. If it is, it assembles an answer and sends it back to the ticket source as a confirmation that it is active. This answer contains its public key, the public key of the ticket source that send the list, and a signature over the two public keys. Afterwards, the list is flooded to the neighbors of the node, independently of whether the node was part of the list.
At the end of the round the ticket sources exchange their received percentages of responses. Based on the measurements, an average percentage of responses is calculated. This average is assumed to represent the percentage of active authorized nodes compared to the total number of authorized nodes. Multiplied with the total number of authorized nodes, it is written to the blockchain and used as the estimated number of active nodes, e.g., for calculations of ticket amounts, in the following round of the system.

5.4.3.2 Selecting new ticket sources

For each round, new ticket sources are selected. To do so, a random walk is started by each currently active ticket source. In random walks each node that receives the random walk forwards its to another, randomly selected, neighbor. As previous work has shown, random walks are successful in traversing social graphs and selecting nodes randomly, while having a reduced probability of entering another, possibly Sybil, community [Yu+06; DM09].

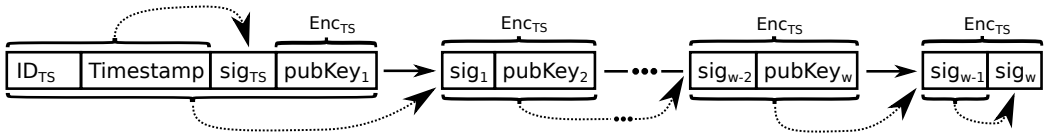

Figure 5.5: *Structure of the data blocks of a random walk for ticket source selection.*

The data forwarded in the random walk consists of a chain of data blocks. These data blocks are connected by a signature sig_i over its predecessor block i, as well as the public key $pubKey_{i+1}$ of the following node $i+1$ of the walk (see figure 5.5). The complete block is encrypted with the public key of the ticket source that started the random walk before being passed on to the next node. Visited nodes only know about the respective previous and next node of the random walk, since they are unable to decrypt the public keys contained in earlier blocks. No additional information is learned by this, since they have this information anyway since they communicate with these nodes directly. If the block would not be encrypted, all nodes receiving the random walk would learn information about the structure of the social graph. Since they can reconstruct the path used by the random walk, they can infer which nodes are neighbors of each other. This should be avoided, since over time it would allow nodes to map the whole graph. The signatures in the data blocks are used to avoid manipulation of the random walk. A malicious node is unable to modify or remove elements of the chain of data blocks without the ticket source noticing the breach of the signature chain. If such a breach is noticed, the ticket source discards the manipulated random walk and starts a new one.
The first block and the last block of the chain are different from the later ones. The first block is not encrypted and contains the number ID_{TS} of the public key of the sending ticket source TS within the current block of the blockchain, a timestamp, a signature sig_{TS} over public key and timestamp created by the sending ticket source, and the public key $pubKey_1$ of the next node to visit. While being send unencrypted by the ticket source, the public key of the next node will be encrypted by that node before forwarding the blocks. The same as later blocks, the second block contains a signature sig_1 over the first block as well as the public key $pubKey_2$ of the second receiver and will be encrypted with the public key of the ticket source TS. The special contents of the first block allows receivers of the random walk to check whether it has been started by a valid ticket source. Starting with the second block, the correct linkage of the chain is ensured by the public keys and the signatures.
After $w = log(n)$ hops, with n being the estimated number of active nodes, the random walk ends and is sent back to the originating ticket source. Due to the fast mixing property of social graphs, this length w should be enough to end the random walk at any connected node [WS98]. The last block w of the chain differs from the other blocks as well since it does not contain a pointer to a further next block. Instead, it contains a signature sig_w over the signature sig_{w-1} of the previous block. Since only the last visited node is able to create the signature sig_w, this ensures that the node referenced by the public key $pubKey_w$ in the previous block $w-1$ is a valid node and is currently online.

After the random walks are finished, the ticket sources exchange lists of the collected public keys. To avoid that the social graph is recreated based on the walked paths, the lists of public keys are sorted by the respective ticket source. For the resulting sorted list, a cryptographic commitment [KL14] is created and shared with the other ticket sources. The commitment is used to avoid a potential attack, in which a malicious ticket source would be able to forge its own list to match the ones received from other ticket sources. Candidates for the new ticket sources for the next round are selected out of the nodes that are part of the lists provided by multiple ticket sources. As such, a malicious ticket source could create an own list that contains all the Sybil nodes that also appear in the lists of honest ticket sources, which is prevented by sharing the commitments. After the random walks of the active ticket sources are finished and their commitments have been received by the other ticket sources, all commitments are opened and verified. This means that the lists themselves are exchanged and all ticket sources check the commitments whether they really belong to the respective lists. If a commitment cannot be successfully verified, the respective ticket source is considered malicious and excluded from the following process. The same applies to ticket sources that are no longer available, e.g., due to being offline. Similar to other decisions that have to be concluded in Detasyr, only a configurable percentage in $]0.5, 1.0]$ of all ticket sources needs to be involved to select new ones by executing the random walks and exchanging the lists of public keys. This avoids that misbehaving or offline ticket sources stop the process from progressing.
Based on the exchanged lists the potential ticket sources are determined. To do so, the intersection of the lists is calculated, i.e., the public keys are found that appear in more than one of the lists. The nodes in the intersection are candidates to become new ticket sources. For each pair of lists, at most two public keys are used from the intersection. This is done to avoid an attack when two Sybil nodes already are ticket sources. Without this restriction, these two malicious ticket sources could both publish the same list, which would result in up to $log(n)$ ticket source candidates appointed by the Sybil adversary. When providing so many candidates, the Sybil adversary would most likely be able to gain control over the majority of ticket sources. If less candidate nodes are found as there are current ticket sources, additional random walks are executed. When there are enough candidates, the new ticket sources are selected. First, all public keys of the candidates are summed up by bitwise XOR. The candidates whose public keys have the smallest hamming distance to this sum are selected as new ticket sources. In case two candidates have the same hamming distance, the one that has been authorized earlier, based on the block number in the blockchain, is selected. This approach brings some deterministic randomness into the selection process to avoid possible attacks due to specifically

chosen public keys. Otherwise, a Sybil adversary might be able to specifically select the public keys of its nodes with the goal of becoming ticket sources. However, since the Sybil adversary cannot influence the public keys contributed to the process by the honest ticket sources, the calculated XOR value, and with that the optimal public key to become a ticket source, cannot be predicted. In each round the same number of ticket sources is selected, i.e., as many new ticket sources are selected as there are ticket sources in the current (and consequently every) round.
For publishing the newly selected ticket sources, an aggregated signature over their public keys is calculated by the current ticket sources. Together the list of public keys and the signature are published in a new block on the blockchain. When doing so, the public keys of the new ticket sources are listed in the same order as they were selected. The first ticket source in the list becomes the new prime ticket source.
If less than a configurable percentage in $]0.5, 1.0]$ of the ticket sources sign the list of public keys, the selection of new ticket sources is considered invalid. For example, this could happen if an adversary provides a list of public keys for new ticket sources to sign that has not been created according to protocol. In case no new group of ticket sources is selected in time, the ticket sources of previous rounds are starting own random walks and are joining the selection process. There is no explicit notification for the authorized nodes when new ticket sources have been selected. Instead, they are informed about the new round, and with that the new ticket sources, by receiving the newest block of the blockchain when it is flooded through the social graph.

5.4.3.3 End of round

Authorization tickets are only created and flooded by the ticket sources once per round. To create further tickets, new rounds have to be started from time to time. To decide when a new round should be started, multiple conditions are monitored by the ticket sources. If any of the conditions is reached, the current round ends by publishing a block for the blockchain. Afterwards, a new round is started with new ticket sources, repeating the operation of the previous rounds.

Enough nodes authorized In each round a new block for the blockchain is created by the ticket sources. This block contains, besides other data, a list of public keys of the newly authorized nodes. If a certain number of nodes, depending on the number of emitted authorization tickets, has been added to this list, the round ends. Only a limited number of tickets are available for authorization in each round and successful authorizations use some of them. Consequently, it becomes increasingly unlikely that further new nodes are able to collect enough tickets to become authorized, so a new round is started.

Not enough tickets left The number of unused authorization tickets is monitored by the ticket sources. While related to the number of newly authorized nodes, the number of unused authorization tickets can shrink faster then strictly necessary to fulfill all received authorization requests. This is possible since authorization requests can contain more tickets than are required for a successful authorization, e.g., since nodes have aggregated authorization tickets. As the ticket sources know how many tickets they created, they are able to calculate the number of still available tickets based on the tickets used in the authorization requests they receive. The number of tickets set for this condition should be higher than the number of tickets required to fulfill the previous condition regarding the authorized nodes.

No authorization requests The last condition checks if no authorization requests have been received by the ticket sources for some time. In this case, two possibilities exist. Either, no further nodes want to become authorized. Alternatively, the unused authorization tickets still present at some nodes do not reach the new nodes that need them. In that case, starting a new round and flooding tickets from the positions of the new ticket sources within the social graph leads to a different distribution of tickets that allows further authorizations.

If the fulfillment of one of these conditions is noticed by a ticket source, it notifies the other ticket sources about it. To increase the efficiency of the protocol, this message is not broadcasted between the ticket sources directly. Instead, the prime ticket source is informed about the reached condition.
The prime ticket source will then start to assemble a new block for the blockchain. As described in section 5.3.2, among other entries this block contains a list of the public keys of the newly selected ticket sources, and the list with the public keys of newly authorized nodes. Also, the other entries in the new block are filled with the exception of the ID of the prime ticket source and its signature. This block is then send to all other ticket sources which will verify and digitally sign the created block. The prime ticket source can then aggregate the created signatures, sign the block itself and broadcast the finished block to the other ticket sources and through the social graph, thus ending the current round and starting a new one.

5.4.4 Gaining authorizations

For its authorization to succeed, a node has to possess a number of authorization tickets, each from a different ticket source. The number of required tickets is $c * s$ with a configurable constant $c \in]0.0, 1.0]$ and s being the number of ticket sources. The parameter c influences the security as well as the usability of Detasyr. With a

low value for c, new nodes can join the system easily but Sybil nodes are able to join easily as well. On contrary, a too high value of c results in a system that can hardly be used. Tickets from only some ticket sources are required so that ticket sources which become unconnected within a round, or are only reachable over many other nodes, do not block the authorization of new nodes.

```
while not check_if_authorized():
    sources_with_tickets = 0
    tickets = []
    while sources_with_tickets < c * s:
        neighbors.send_ticket_requests()
        tickets.add(neighbors.receive_tickets())
        sources_with_tickets = tickets.count_unique_sources()
    tickets.aggregate()
    tickets.send_to_ticket_source()
    wait_for_new_block()
```

Figure 5.6: *Pseudocode for the process of gaining authorization.*

As explained in section 5.4.2, only already authorized nodes receive the tickets send by the ticket sources. To collect the required tickets, a new node can send *ticket requests* to all its neighbors. The neighbors respond with the separated or aggregated authorization tickets they have collected. When the new node has acquired tickets from enough different ticket sources, an authorization request can be assembled and send to any of the ticket sources. This process is depicted in figure 5.6.
To identify the sending node, an *authorization request* contains the public key of the new node. As proof that they are connected to other already authorized nodes, the aggregated signatures of the received tickets, the indices of the tickets, and the indices, within the current block of the blockchain, of the ticket sources that contributed the respective ticket, are added. To protect the integrity and to prove the ownership of the matching private key, the data of the ticket request is signed by the new node. The data and the signature are encrypted with the public key of the addressed ticket source to stop malicious nodes from stealing the authorization tickets contained within the ticket request. Otherwise, a malicious node could intercept the ticket request, replace the public key of the new node with its own, and sign the ticket request itself. Finally, the data packet, as depicted in figure 5.7, is send to the ticket source the authorization request has been encrypted for.
After decrypting the packet, the ticket source receiving the authorization request first checks the contained signature of the new node and the number of tickets by different

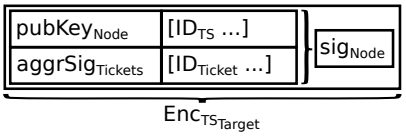

Figure 5.7: *The structure of an authorization request as send to the ticket sources.*

sources. If the signature is invalid or not enough different tickets are provided, the authorization request is invalid and will be ignored. To avoid duplicate usage of tickets, tickets are only accepted if they have not been used in a previous authorization request. This is checked by each ticket source for the tickets it generated, based on a list of previously used tickets stored locally. This list only needs to be stored as long as the node is a ticket source, i.e., for one round. If the authorization request is accepted by this ticket source, it creates a signature over the public key of the new node, representing the acceptance of the request. The signature as well as the authorization request are then send to the other ticket sources. Each ticket source will then repeat the checks, create an own signature, and send the ticket request together with its signature to the prime ticket source for inclusion in the next block of the blockchain. Only when a high enough percentage $r \in]0.5, 1]$ of ticket sources are accepting the request the new node will be authorized. As for the number of tickets that have to be collected, the number of ticket sources that have to participate in accepting the request can be less than the total number of ticket sources. This increases the robustness of the process in case some of the ticket sources are no longer available. Especially important if an adversary is present, this means that a misbehaving ticket source is unable to block the authorization of a node, i.e., if it intentionally does not sign the authorization request. Still, more than half of the ticket sources are required to sign the request to ensure the acceptance of a majority of ticket sources and to avoid that a single, or a few, malicious ticket sources are able to authorize nodes without the honest ticket sources confirming the authorization.

5.5 Further restrictions for Sybil identities

For Sybil adversaries, the design presented so far restricts them from authorizing an unlimited number of nodes per round. The same as honest nodes, they have to collect authorization tickets from their neighbors to send authorization requests to the ticket sources. However, different from honest nodes, a Sybil adversary does not stop authorizing nodes after the first node has been authorized. Instead, they try to authorize enough nodes to gain control over the system. While the number of

authorized nodes per round is limited by the number of received authorization tickets, a similar number of nodes can be authorized by the Sybil adversary every round. In this section two extensions are presented that aim to provide an absolute upper bound to the number of Sybil nodes: a maximum age of authorizations and edge values. Both extensions restrict the absolute number of Sybil nodes, independently of the number of rounds passed, leading to a smaller relative number of Sybil nodes as the size of the social graph increases.

5.5.1 Aging Authorizations

In the design presented so far, successful authorizations are stored in the blockchain and stay valid indefinitely. This is changed by the extension presented in this subsection. The underlying idea is, that only authorizations in the ν newest blocks are considered valid. If regarding, e.g., only the newest $\nu = 30$ blocks, a node becomes unauthorized again 30 blocks after it successfully authorized itself. Afterwards, it has to reauthorize itself again.
This extension results in an upper bound for the number of Sybil nodes that can become authorized. While the basic design already ensures that only a limited number of Sybil nodes is authorized each round, it allows an unlimited number of Sybil nodes over time. With this extension authorizations become invalid after some rounds, so the Sybil adversary has to spend the received tickets to reauthorize its nodes. Consequently, no further nodes can be authorized, leading to a bounded number of Sybil nodes. Even when this upper bound can be rather large in absolute numbers (depending on the number of nodes the Sybil adversary has as neighbors and the number of valid blocks ν), it is an absolute upper bound. As a result, the relative number of Sybil nodes decreases as the number of authorized honest nodes increases. Another point is that the security of the blockchain is improved. Since only the last few blocks are important for the verification of authorizations, an adversary has only limited time available to forge the signatures of the ticket sources for these blocks or compromise their nodes. In the basic design an adversary has unlimited time to compromise the signature of a block, and, e.g., add numerous Sybil nodes as authorized nodes within the block.
However, this extension also leads to a higher overhead for honest nodes. All nodes, both controlled by a Sybil adversary and and by honest users, have to reauthorize themselves periodically. In the basic design presented so far, an “idle-state” establishes after some time for a finite social graph, as can be seen in the evaluation in section 5.6.1.2. In this state, the required work for honest nodes is reduced, since no further (honest) authorizations happen as all nodes already are authorized.

Another advantage is that with this extension it becomes possible to withdraw trust from other users, e.g., because a user now assumes that their neighbor is a Sybil identity. In the basic design, an authorized node stays authorized permanently, even when its neighbors no longer trust it. Even with this extension, removing the edge to a neighboring node does not immediately revokes its authorization. However, when the node tries to reauthorize itself after v rounds, this can fail due to it not having enough authorized neighboring nodes anymore.

5.5.2 Edge Values

Applying unidirectional edge values to the edges between nodes is another possible extension to limit the absolute number of Sybil nodes. When using edge value, each node stores one of these values for each of its neighboring nodes, representing the trust into the relationship with the respective neighboring node. The idea is that receiving authorization requests from a node reduces this trust. When the edge value for one neighbor reaches zero, no more authorization requests are accepted from this neighbor by the receiving node, restricting the number of requests that are possible over each edge. While only slightly hindering for honest nodes, this extension stops Sybil nodes from sending unlimited numbers of authorization requests.

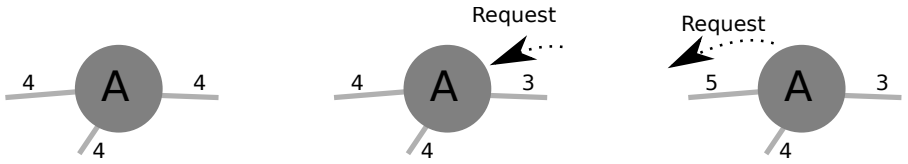

Figure 5.8: *Local edge values on the three edges of a node with the initial state on the left. When an authorization request is received the edge value of the respective edge is reduced (middle). When the request is accepted by a neighbor, the edge value of the respective edge is increased (right).*

As depicted in figure 5.8, the edge values are considered when a node receives an authorization request for forwarding. First, it checks its local edge value on the edge to the sending node, i.e., the edge where the request have been received over. If the edge value is greater than zero, the edge value in the direction of the sending node is reduced by one and the request is forwarded in the direction of the ticket source. Otherwise, the request is declined and send back. If the neighboring node in the direction of the ticket source accepts the request, the edge value in its direction is increased by one. Combined this means that the sum of edge values stored by a node keeps the same. While the individual edge values on its edges are modified,

the trust towards the sending neighbor is moved to another edge in the direction of the ticket source. For the edges towards the node initially sending the authorization request, the edge values are only reduced. This node does not receive the request from another node, so no other node increases its trust in it. This results in a limited number of requests that can be send by each node, since the neighboring nodes will stop accepting authorization requests. For Sybil adversaries this results in a restriction of their abilities, which prohibits them from creating as many Sybil identities as they want. The sum of the edge values over all edges towards a node, i.e., as seen by its neighbors, is only increased when a node becomes a ticket source and handles authorization requests itself. In that case no authorization requests are send to neighboring nodes, so their edge values are not reduced. However, the ticket source will receive a lot of authorization requests from its neighbors, leading to its edge values reaching zero and blocking further authorizations. To avoid this problem, the current ticket sources, and also their direct neighbors, should not modify their edge values when handling and forwarding authorization requests.
While the introduction of edge values is technically a restriction for all nodes, honest nodes should only be slightly restricted by it in practice. They only send a single authorization request and should have many different paths to send their request over. Since the ticket sources are placed differently each round, it is likely the authorization request in the next round succeeds, even when it failed in the current round due to the edge values. However, Sybil adversaries are assumed to send unlimited numbers of authorization requests to create further Sybil nodes. Sending these requests will reduce the edge values of all of their honest neighbors for all their attack edges to zero, stopping the Sybil adversary from sending any further authorization requests. Since they are unable to authorize further nodes, this restricting them to $InitialEdgeValue * \#Neighbors$ Sybil nodes. This bound can only be exceeded if a Sybil node becomes randomly selected as a ticket source which becomes increasingly unlikely with a growing graph. If this happens, the Sybil adversary can accept its own ticket requests without having to send out authorization requests, avoiding the restriction by the edge values. Still, the other ticket sources have to confirm the authorization of the new Sybil nodes. This means that the Sybil node is not relieved from collecting enough tickets for the authorization, which was already restricted by the basic design of Detasyr. Consequently, unlimited creation of Sybil nodes is still not possible if the Sybil adversary controls a ticket source.
The initial edge value e is a parameter of this extension. It determines the balance between speed of authorization for honest users and the maximum number of Sybil nodes, with a high value for e favoring both of them. As in the other extension, the maximum number of Sybil nodes is an absolute number.

5.6 Evaluation

In this section the evaluation of Detasyr is presented. After introducing the simulation environment used for the conducted measurements, the results of simulating the extensions presented in section 5.5 are shown and discussed. Afterwards, the consequences due to Sybil controlled ticket sources as well as the overhead introduced by participating in the Detasyr network are analyzed. Finally, a comparison with related work is conducted.

5.6.1 Increase of Sybil strength

In this subsection the evaluation results for the two extensions presented in section 5.5 are shown and discussed. This includes the authorization rates for honest nodes as well as Sybil nodes. Simulation results for the basic design, i.e., without either of the extensions, are shown for comparison as part of the other simulation results.

5.6.1.1 Evaluation environment

An event-based simulator has been implemented for the evaluation. It creates a synthetic social graph and simulates the state of the nodes as well as the message passing and processing. Two social graphs were evaluated, based on the models by Barabási-Albert [BA99] and Watts-Strogatz [WS98]. With the Barabási-Albert model a scale free graphs is generated that is assumed to be close to current online social networks. In those, some users have huge amounts of friends, while most users have much less or only a few friends. As an alternative, the Watts-Strogatz model generates a graph with a strong community structure. This means that multiple communities exists which are extensively connected within themselves but have only few connections to other communities. Such a graph would most likely be the result if users would only accept trustworthy real-world friends as their neighbors in the social graph, which is the preferable scenario for executing Detasyr.
Due to the execution time of the simulations, graphs with 10,000 nodes are used for the presented evaluation plots. Initially, only $n = 20$ connected nodes are already authorized. Of these nodes, $s = 10$ are ticket sources. Comparative simulation runs with larger graphs and an excerpt of a real social graph have shown that the qualitative results of the simulations, e.g., the authorization rates for both honest and Sybil nodes, are the same. For each parameter combination, 40 repetitions have been run. The Sybil adversary starts the creation of nodes in round 7. Starting the Sybil adversary already in the first round would be possible, but would allow the

adversary to authorize more nodes than honest users are currently authorized which results in the Sybil adversary controlling the system. Considering a real-world use of Detasyr this delayed start of the Sybil adversary is probably realistic. Initially, only real-world friends have nodes in the initial group forming the social graph. Since this small group of people knows how important trust is for the operation of the system, they will most likely be careful about selecting their friends in the social graph. Additionally, even when a Sybil adversary is present immediately, it would not be beneficial to start the attack, by creating many Sybil nodes, early on. As long as only few users are part of the system such an attack would be noticed and the system would be restarted, thwarting the attack of the Sybil adversary.
The evaluation of the number of nodes the Sybil adversary controls focuses on the behavior of the system when no further honest nodes in the simulated graph become authorized. As long as honest nodes are becoming authorized, the relative number of Sybil nodes is naturally restricted even when the absolute number of Sybil nodes increases. Only when no further honest nodes are becoming authorized the real increase of the relative number of Sybil nodes can be evaluated.

5.6.1.2 Aging authorizations

In the first extension, described in section 5.5.1, authorizations only stay valid for v rounds. In this subsection, the simulation results using this extension are displayed and discussed.

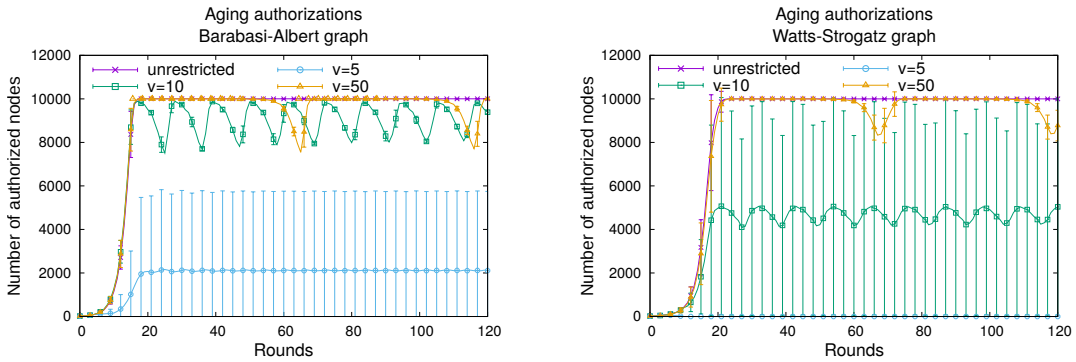

Figure 5.9: *Simulation results of the aging extension for different numbers v of valid rounds. Depicted are the numbers of authorized honest nodes.*

In figure 5.9 the simulation results regarding the successful authorization of honest nodes are displayed. Depicted are simulation runs on the two social graphs used for the evaluation for different values of v, which is the number of rounds authorizations stay valid. The simulations were running for 120 rounds.

The "unrestricted" curve is based on simulation runs without using the aging extension and is displayed for comparison. As can be seen, the curve starts to grow only slowly at first. Since only 20 nodes are authorized initially, only relatively few nodes are neighbors of an authorized node and can become authorized themselves in the current round. However, the number of authorized nodes increases exponentially, since each newly authorized node allows all its neighbors to become authorized as well. At about round 10 the initially slow start is over and the number of authorized nodes significantly increases with each following round. It should be noted that in a real-world implementation of Detasyr the number of authorized nodes would most likely increase much slower. While many nodes could be authorized each round, users would probably join the system much slower than authorizations are possible. On the other hand, this means that every new joining user would likely become authorized in its first round in the system. At round 17 it can be seen that the rate of authorization starts decreasing again. Around this time, nearly all nodes of the fixed size generated social graph already have been authorized. The remaining nodes that have not been authorized yet are either connected to too few nodes and are not receiving enough tickets to become authorized immediately, or are part of a group of unauthorized nodes which become authorized one after the other. After a few rounds, all 10,000 nodes of the social graph have been authorized. While the structures of the simulated social graphs differ, both "unrestricted" curves look nearly the same, even though the community structure of the Watts-Strogatz graph leads to a slightly slower authorization of honest nodes.

When using the extension with a maximum age of authorizations, periodic dips in the graphs of both plots are visible due to the limited validity of the authorizations. Every ν rounds the authorizations of the nodes become invalid and have to be renewed. The form of the resulting dip in authorizations depends on the speed the nodes have been authorized with earlier. Further evaluations have shown that the dips are shallower but longer when the nodes are getting authorized at a slower pace.

For low values of ν, not all nodes manage to become authorized. Furthermore, a significant variance between the executed simulation runs is visible. When inspecting the underlying data that has been plotted, this misleading variance can be explained. For a value of $\nu = 5$ in the graph based on the Barabási-Albert model, most simulations result in one of two frequent outcomes. In the better outcome, around 8000 nodes manage to become authorized, which is still less than the total number of 10,000 nodes. In the other frequent outcome, no nodes are authorized at the end of the simulation. To become authorized, a node requires at least one authorized neighbor and at least 6 (indirectly) connected ticket sources. If nodes are forced to reauthorize too early, this can lead to a fragmented structure for the social graph. In that case,

each graph fragment contains some ticket sources, but no fragment contains enough ticket sources to continue authorizing further nodes. At most v rounds later, all nodes are unauthorized again. A value $v = 10$ with the graph based on the Watts-Strogatz model leads to a similar result. If a large enough value for v is used, the authorized subgraph of the generated social graph reaches a large enough size with a high enough connectivity between its nodes. When some nodes become unauthorized again, no significant fragmentation of the social graph occurs.
Additionally, a difference between the generated social graphs becomes visible. In the graph generated with the Barabási-Albert model, a validity duration of $v = 10$ is sufficient to allow all 10,000 nodes of the social graph to become authorized. With the graph based on the Watts-Strogatz model, a value $v = 10$ is not sufficient to authorize all nodes yet. This difference can be explained due to the structure of the graphs. With the Barabási-Albert model, some nodes have a huge number of friends. As such, many neighboring nodes can be authorized at once, leading to a faster authorization of nodes and consequently to a faster reauthorization as well. Additionally, these well connected nodes introduce shorter paths through the social graph, which is beneficial for flooding authorization tickets and allows more nodes to receiving tickets. With the Watts-Strogatz model a stronger community structure is present, with less edges between the different communities. Due to being only connected by relatively few edges, the distribution of authorization tickets is delayed, leading to a slower authorization of nodes. This results in reauthorizations being required before all nodes were able to become authorized, repeating each v rounds.

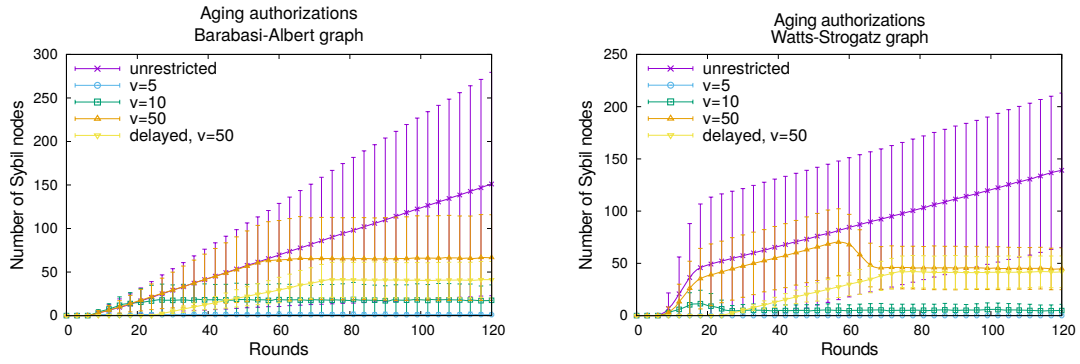

Figure 5.10: *Simulation results of the aging extension for different numbers v of valid rounds. Depicted are the numbers of authorized Sybil nodes.*

The plots in figure 5.10 display the number of Sybil nodes that have been authorized, based on the same simulation runs as the previous plots. Again, the “unrestricted” curves are showing the increase of Sybil nodes when no extension is used. As has

been explained before, the Sybil adversary is able to only register a restricted number of nodes per round, but can repeat this in every round. Consequently, the number of Sybil nodes keeps increasing, albeit with a limited rate.

Compared to the previous plots in figure 5.9, where the authorization of honest nodes was shown, no periodic structure is visible in the graphs. This can be explained with the much lower amount of nodes compared with the previous plots. While previously up to 10,000 nodes were authorized and became unauthorized again, here only around 65 nodes have been authorized by the Sybil, which additionally happened at a slower rate. Consequently, the periodic structure is not visible in the graph and only visible as a small deviation in the numeric representation.

What is clearly visible in both graphs of figure 5.10 is that a higher value of ν results in more authorized Sybil nodes. Still, for all depicted values of ν the Sybil adversary is restricted in its ability to authorize nodes. At round 20, when all 10,000 honest nodes have been authorized, the Sybil adversary only managed to authorize 16 nodes. Since the Sybil adversary started its attack after seven rounds, this means that the Sybil adversary was only able to authorize around one node per round, even with the basic design of Detasyr without the extensions. At the end of the evaluation period of 120 round the Sybil adversary had on average 150 authorized nodes in the unrestricted simulation runs, which constitutes for 1.5 percent of the social graph. When aging authorizations where used with a maximum age of $\nu = 50$ rounds, only 65 Sybil nodes where authorized, resulting in a Sybil presence of less than one percent. Both results show the effectiveness of Detasyr. It should be noted that with aging authorizations the absolute number of Sybil nodes no longer increases. If the size of the honest social graph increases over time, which is to be expected in a real deployment, the percentage of Sybil nodes decreases.

Two phases can be discerned in the increase of the number of authorized Sybil nodes. Initially, the number of Sybil nodes increases with the same rate as in the unrestricted simulation runs. However, after some rounds $(\nu + 7)$ the increase of Sybil nodes stops. At this point, the first nodes of the Sybil adversary become unauthorized again. As are honest nodes, the Sybil nodes become unauthorized with the same rate as they have become authorized ν rounds before. Since the Sybil adversary is unable to increase the amount of authorization tickets it receives, it is only able to reauthorize its nodes with the same rate as nodes become unauthorized. As such, the number of Sybil nodes remains the same and is no longer increasing.

Compare to the authorization of honest nodes, a higher variance between the simulation runs is present. In the honest part of the social graph there are always neighboring unauthorized nodes which can be added to the graph. With Sybil nodes this is not the case. Instead, the nodes form an own community within the social

graph which is only connected to few honest nodes. Due to this, the Sybil adversary is more dependent on its placement within the social graph, leading to varying simulation results depending on this placement.

In the plot based on the Watts-Strogatz graph, the curve shows a more complicated pattern than in the graph generated by the Barabási-Albert model. As an example, the curve for $\nu = 50$ is discussed, with the other curves behaving similar. Initially, in the rounds 7 – 16, the increase of Sybil nodes is quite high. This is also visible in the unrestricted curve. In this phase, the Sybil adversary profits from the small size of the social graph, which leads to the ticket sources being close to the Sybil nodes. Consequently, the Sybil adversary receives many authorization tickets and is able to authorize more nodes in these rounds. When the Sybil adversary starts its attack at a later time, as in the "delayed" curve which starts the attack in round 30, this steep increase of Sybil nodes is not present.

As depicted in figure 5.9, the number of honest nodes significantly increases around round 16. This leads to higher distances between the ticket sources and the Sybil adversary. Additionally, the Watts-Strogatz graph has a community structure. Since most flooded authorization tickets stay within the community of the sending ticket source, the Sybil adversary receives even less tickets. In the rounds 17 – 57, the Sybil adversary is able to authorize similar amounts of nodes as in the unrestricted simulations, and also as in the Barabási-Albert graph. In the following rounds 58 – 67 the number of Sybil nodes declines. The Sybil adversary does not receive enough tickets to keep all of its nodes authorized, since it no longer receives as many tickets as it did in the earlier rounds 7 – 16. For the rest of the simulation, the two curves using a validation duration $\nu = 50$ are at similar numbers of Sybil nodes. While the Sybil adversary that started its attack in round 7 was able to temporarily authorize more nodes, this early advantage is no longer relevant after round 57. Afterwards, the number of Sybil nodes is only influenced by the number of authorization tickets the Sybil adversary receives each round, which is the same independently of the round it started the authorization of Sybil nodes.

Based on these results, it can be seen that a larger value for the validity duration ν is required for a graph structure which contains more communities, as well as for an increasing size of the social graph. An "ideal" value for ν cannot be derived analytically. For once, it could be beneficial to increase the size of ν together with the size of the nodes in the social graph. Starting with a large, but fixed, value for ν would not benefit the honest nodes, while it would permit the Sybil adversary to authorize more of its nodes. A fixed small value for ν would restrict the Sybil adversary, but would result in a maximum number of authorized nodes, both for honest users and the Sybil adversary. Additionally, the structure of the social graph

has been shown to have a significant influence on the required value for v. Since this structure changes depending on the social graph of the online service Detasyr is run on, a "good" value for v has to be found experimentally by running Detasyr on the social graph in question.

5.6.1.3 Edge values

As introduced in section 5.5.2, this extension limits the amount of Sybil nodes by only accepting a restricted number of authorization requests from each neighbor. If a neighboring node sends too many authorization requests, further requests are no longer forwarded.

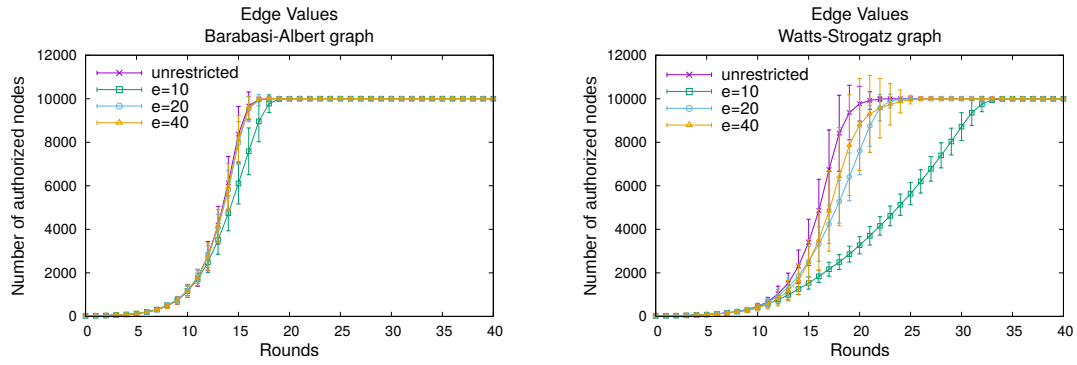

Figure 5.11: *Evaluation results of the extension with edge values for different initial edge values e. Depicted are the numbers of authorized honest nodes.*

The simulation results for the number of authorized honest nodes are shown in figure 5.11. Displayed are the first 40 rounds of the simulation. Similar to the previously discussed extension using a maximum age of authorizations, the graph based on the Watts-Strogatz model leads to a slower authorization of honest users. Since multiple communities exist in this graph model, authorizations are restricted by the fewer edges between the communities. The edge values on these edges become zero, prohibiting further authorization requests from crossing the boundary between the communities. Missing alternative paths over other edges, the community is unable to authorize further nodes in this round. Only when ticket sources are present in this community in a later round authorizations become possible again. This effect is especially prevalent with low initial edge values, since this leads to a faster depletion of the trust in neighboring nodes. This can be seen in both graphs, were an initial edge value of $e = 10$ leads to a significant slower authorization than with higher edge values. However, independently from the initial edge value all nodes become authorized eventually.

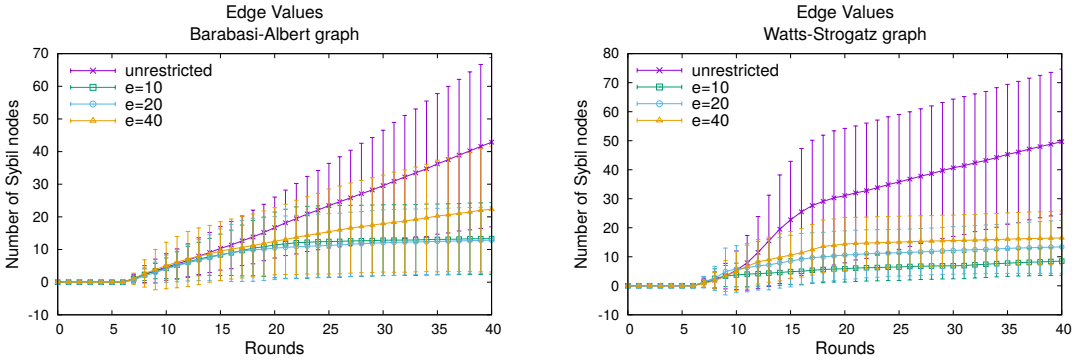

Figure 5.12: *Evaluation results of the extension with edge values for different initial edge values e. Depicted are the numbers of authorized Sybil nodes.*

While this is a restriction for honest users, it is an even more significant restriction for Sybil adversaries, as shown in figure 5.12. As already seen in the previous section, the number of Sybil nodes keeps increasing when no restriction is applied. With edge values however, the maximum number of authorized Sybil nodes is restricted to an absolute upper bound. Initially the number of Sybil nodes increases, e.g., as seen in rounds 7 – 20 in the graph generated by the Barabási-Albert model. In these rounds, the edge values towards the Sybil community have not been depleted to zero yet, allowing the Sybil adversary to send authorization requests through the honest community to the ticket sources. Later on, the edge values for $e = 10$ and $e = 20$ are depleted and no further authorization request can be send. For an initial edge value of $e = 40$ this state is not reached in the 40 rounds that are displayed in the plot. Due to the higher initial edge value, the Sybil adversary is able to send more authorization request before being blocked. Still, when considering the authorization of honest users, a higher initial edge value, e.g., $e = 40$, might be preferable. With an increasing size of the social graph the relative number of Sybil nodes decreases, even when the Sybil adversary is able to authorize a higher absolute number of nodes initially. At the same time, the higher initial edge value improves the authorization speed for honest users, which is an important consideration for the general acceptance of the system by its users.

Compared to the simulation results of the aging authorizations displayed in figure 5.10, even less nodes can be authorized by the Sybil adversary. Though, as already seen for the different values of the validity duration v, the initial edge value e influences how many Sybil nodes become authorized. Both these parameters v and e have to be chosen by the operator of Detasyr to achieve an acceptable compromise between the ease of authorization for honest users, and the restriction of the Sybil adversary.

5.6.2 Sybil controlled ticket sources

New ticket sources are selected randomly out of the authorized nodes which are currently active, by executing random walks through the social graph and calculating the intersection of these walks. This is explained in detail in section 5.4.3.2. As previous work has shown, random walks have an increased probability to stay within one community of the social graph [Yu+06; DM09], reducing the chance that a Sybil node is proposed as a ticket source. Still, it can happen that Sybil nodes become selected as ticket sources.
If only a single Sybil node or a few Sybil nodes are selected as ticket sources, the Sybil adversary is unable to significantly disrupt the functioning of Detasyr. Most operations executed by the ticket sources are based on a majority decision, e.g., authorizing new nodes, selecting the next ticket sources, or adding a new block to the blockchain. To disrupt these operations, the Sybil adversary can attempt three attacks: providing invalid data, withholding acceptance to a correct decision, or intercepting authorization requests.

Invalid data When providing invalid data, e.g., to authorize further Sybil nodes without enough collected authorization tickets, the honest ticket sources will recognized that the received data is invalid. In that case, they will not confirm it, thwarting the attack. For some procedures, e.g., selecting new ticket sources, providing invalid data even results in the malicious ticket source being excluded from the rest of the procedure, limiting the influence of the Sybil adversary.

Withholding acceptance To block the honest ticket sources from operating, the Sybil controlled ticket source could ignore requests send to it, e.g., to sign the new block for the blockchain. However, only a configurable percentage in $]0.5, 1.0]$ of all ticket sources have to confirm these requests. If the Sybil adversary refuses its acceptance, this is not worse than a ticket source which goes offline. As long as a sufficient number of honest ticket sources remain active, a small number of malicious ticket sources ignoring a request can be tolerated.

Intercepting authorization requests An attack that is possible for the Sybil adversary is to intercept the authorization requests of honest nodes. For once, this stops an honest node from being authorized. The advantage for the Sybil adversary is that the correct operation of the system is at least slightly disrupted, and that the size of the social graph does not increase, keeping the percentage of Sybil controlled nodes from decreasing. For the requesting node, this is only a temporary inconvenience. With a high probability a second authorization request in the next round will not be received by a Sybil controlled ticket source, resulting in a successful authorization.

However, the Sybil controlled ticket source is able to decrypt the encrypted authorization request, something that normal nodes forwarding authorization requests are not possible to do. This allows the Sybil adversary to steal the authorization tickets contained within the request, and create an own authorization request for one of its own Sybil nodes.
How successful this attack is depends on the specific situation, most significantly on the number of nodes which try to become authorized in this round. Additionally, the authorization requests can be send by the nodes to arbitrary selected ticket sources. On average, a single Sybil controlled ticket source receives $\frac{1}{s}$ of the authorization requests, with s being the number of ticket sources. Finally, when aging authorizations are used, this gain of authorized Sybil nodes is only temporary. While the Sybil adversary is able to authorize more of its nodes in this round, it is unable to reauthorize this nodes v rounds later when the nodes become unauthorized again.

To gain complete control over the system, the Sybil adversary can try to get as many ticket sources under its control as possible. If the adversary controls more as the configurable percentage in $]0.5, 1.0]$ of ticket sources required to confirm decisions, it is able to appoint new ticket sources at will and control all authorizations within the social graph. Since the behavior of possibly remaining honest ticket sources is no longer relevant when this state is reached, the system can be considered broken and under the control of the Sybil adversary. Consequently, the required percentage for decisions should be selected as high as possible, while still allowing for offline or single malicious ticket sources.
To control further ticket sources, the Sybil adversary has to manipulate the selection of new ticket sources in its favor. Candidates for new ticket sources are selected by intersecting random walks, with at most two candidates taken from each intersecting pair of random walks. Assuming that s_a ticket sources are controlled by the Sybil adversary, up to $s_a * (s_a - 1)$ candidates can be presented by the Sybil adversary to the honest ticket sources. For $s_a \geq 3$ the adversary would be able to introduce more candidates as it currently controls ticket sources, potentially increasing its number of ticket sources in the next round.
However, this only allows the Sybil adversary to propose more ticket source candidates as it currently controls. As described in section 5.4.3.2, the selection of ticket sources out of the candidates is based on calculating the XOR sum of the candidates public keys. The Sybil adversary is unable to influence or predict this sum, since it does not know the public keys of the other candidates until after it committed to its own candidates. Consequently, the Sybil adversary cannot enforce that its candidates are selected as the next ticket sources, resulting in a low probability that it can even maintain its current number of Sybil controlled ticket sources.

5.6.3 Overhead estimation

The effort required to participate in Detasyr is hard to estimate, since it depends on a number of factors, most importantly the current role of a node within the system. This role changes every round, with most nodes being normal authorized nodes.

Ticket sources From the roles nodes can have in Detasyr, operating as a ticket sources requires the most computation and communication from a node. Ticket sources have to create and send authorization tickets, receive and process authorization responses, and maintain the state of Detasyr and its blockchain. Another factor is the size of the social graph, with a larger graph resulting in more authorization tickets that need to be flooded and potentially more authorization requests that are received. However, nodes are normally only a ticket source for a single round. Furthermore, the number of ticket sources is low: For the simulated evaluation only 10 ticket sources were used for a graph of 10,000 honest nodes. So while the overhead of operating a ticket source is the highest overhead that can occur within Detasyr, it is a relatively rare occurrence for a single node.

Authorized nodes The tasks of authorized nodes are forwarding tickets and random walks. How much effort this requires depends on the placement of the node relative to the placement of the ticket sources. If placed near one or multiple ticket sources, the node receives many authorization tickets it has to forward to other nodes. On the other hand, if placed at the border of the social graph far from ticket sources, the node receives only few tickets. Additionally, when having an neighboring unauthorized node, the authorized node has to answer received ticket requests and potentially forward an authorization request. If aging authorizations are used, additional effort is inflicted to ticket sources and authorized nodes by the periodic reauthorizations.

Unauthorized nodes Within Detasyr, both authorization tickets and random walks are not forwarded to unauthorized nodes. As such, their overhead is basically non existent: All effort that is required is to become authorized itself.

Based on measurements in the simulations, authorized nodes had to process on average 3 messages per minute in a graph of 10,000 nodes. However, these messages are not equally distributed, neither about the nodes nor about the time. For a single node, the required effort can significantly change depending on the distance to the next ticket source. Also, within one round more effort is required at the beginning of the round when the authorization tickets are flooded, while later on only sporadic messages have to be processed. In conclusion, running a Detasyr node should on average not result in excessive overhead compared to other network traffic on the users device, but can occasionally cause an burst of activity.

5.6.4 Comparison with related work

In the last years multiple approaches for Sybil defense based on social graphs have been proposed. All of them are based on the assumption that there is a sparse cut consisting of only few edges between the community of honest nodes and of Sybil nodes. How the sparse cut is employed by the approaches differs, though. Also, the approaches differ in their requirements and their abilities. An overview is given in table 5.1 and discussed in the following.

Approach	Privacy / Decentralized	Dynamic graph	Multi-community	Group auth.	Proof of auth.
Detasyr	√	√	√	√	√
SybilGuard	√	√	X	X	X
SybilLimit	√	√	X	X	X
SybilShield	√	√	√	X	X
SybilInfer	X	X	X	√	X
SybilDefender	X	X	X	√	X
SybilHedge	√	√	X	X	√
Gatekeeper	√	X	√	√	√

Table 5.1: *Comparison with related Sybil defense approaches.*

For integration into DecentID, it is important that the Sybil defense approach operates decentralized and protects the privacy of its users. Otherwise, the privacy protection of DecentID could be circumvented by looking up the identities used in the Sybil defense approach. Most approach considered in this comparison operate decentralized and do not require a complete view of the social graph. An exception to this are SybilInfer and SybilDefender. In these approaches, a complete view of the social graph is required. While detrimental for the privacy of the users, it allows the approaches to authorize complete communities of nodes as honest or Sybil at once.
When the size of a dynamic social graph increases, it becomes more and more impractical to authorize all nodes in the graph at once, due to the increasing required effort to do so. As such, it is preferable if a Sybil defense approach is able to authorize further nodes when required, without evaluating the authorization of all nodes again. In the case of Detasyr this becomes possible by operating in recurring rounds, with new nodes becoming authorized in each round. In the remaining approaches which support this ability, individual nodes are authorized one by one, i.e., each node

verifies all other nodes it wants to interact with by themselves. While this reduces the efficiency of the approaches, it allows to authorize newly joined nodes.

As analyses of real-world social graphs have shown, there is no single honest community in the social graph but instead multiple communities with varying sizes [KNT06]. If this structure is not considered, as is especially the case for older Sybil defense approaches, these other communities are detected as Sybil communities, despite consisting of honest users. Detasyr, SybilShield, and Gatekeeper are able to handle multiple honest communities by using additional nodes in the other communities to help with authorization, instead of only executing the Sybil detection from the position of a single node.

To improve the efficiency of authorization, it is beneficial to authorize or decline entire groups of nodes at once, instead of authorizing each node by itself. Based on the assumption that the nodes controlled by the Sybil adversary form an own community in the social graph, this allows to exclude all Sybil nodes at once. In SybilInfer and SybilDefender this is done by finding a single Sybil node and afterwards detecting its surrounding community. Detasyr and Gatekeeper are using another approach, where tickets are flooded through the complete social graph, allowing to authorize the whole graph, including the communities in it, at once.

When using a Sybil defense approach coupled with another system, e.g., DecentID, it is important for the users to be able to prove their successful authorization towards third parties. Most Sybil defense systems do not support such proofs. Instead, each user has to authorize each other user if an authorization is required. In Detasyr, the authorization is stored on a public blockchain which can be used to prove the authorization towards others. With SybilHedge and Gatekeeper, the authorization is not explicitly stored somewhere. Instead, the result of the successful authorization, respectively a successful random walk or the collected tickets, can be stored by the individual nodes and presented to a third party if required.

As apparent from this comparison, the available approaches to Sybil defense require different conditions and offer varying advantages. Depending on the use case, different approaches might be appropriate. For integration with DecentID, none of the existing approaches was fully eligible, most of them due to the missing ability to prove a successful authorization. While the ability to authorize whole groups of nodes at once improves the efficiency, it is no hard requirement for the integration into DecentID. On the other hand, modern online social networks consist of constantly changing social graphs with multiple, interest-based, communities, making support for dynamic social graphs an important feature for Sybil defense as well. With Detasyr, a previously missing Sybil defense system matching the requirements of DecentID has been designed.

5.7 Integration into DecentID

DecentID enables its users to create pseudonymous identities to use them for different online services. Since users can create these identities by themselves, DecentID is susceptible for Sybil attacks. This allows a Sybil adversary to create numerous DecentID identities as well as using them for online services. Since the services cannot recognize that these identities are controlled by the same user, Sybil attacks on these services become possible.
To avoid this problem, Detasyr can be used to restrict the number of identities a Sybil adversary can use with the same service. The simplest approach would be to enforce the successful authorization within Detasyr. The service can require a cryptographic proof, e.g., a digital signature, from the user that it has access to the private key of a public key registered within the blockchain of Detasyr. This would allow the service to recognize that the different identities are in fact all controlled by the same user. However, this is undesirable, since it also allows tracking of the user through different services. With privacy protection being an important design goal of DecentID and Detasyr, a different approach has to be used. Instead, a cryptographic link between the service and the proof of authorization created by Detasyr is created.

5.7.1 Assumptions

In the following, a number of assumptions for the correct functioning and privacy of the integration approach are listed.

5.7.1.1 Existing identities

It is assumed that the user already created a Detasyr identity and their public key $pubKey_{user}$ is registered as entry ID_{pk} in a block ID_{block} of the blockchain of Detasyr. The user also has created a shared identity with an online service, where the service has presented itself with an identifier $ID_{service}$. For example, $ID_{service}$ could be the public key of the service, as discussed later on in section 5.7.4. To avoid Sybil attacks, the service now requires proof of a successful Detasyr authorization before allowing access to its offered features.

5.7.1.2 Trustworthy ticket sources

It is assumed that the randomly selected ticket sources of Detasyr are trustworthy. This means, that they will not try to track which services a user is using. While they do receive $ID_{service}$ and could identify a single visited service, the frequent selection

of new ticket sources means that the user cannot be tracked by a single malicious ticket source. Also, it is assumed that no ticket source maliciously cooperates with a service. If doing so, they would be able to discover the permanent Detasyr identities of the users, allowing the service to track its users.
No assumptions about the trustworthiness of the service are made. Today, many advertisement services try to track visitors of website showing their advertisement. This way, they are able to create user profiles and present user targeted advertisement. While the unlinkable identities of DecentID prevent this, such services would be interested to find out the permanent identifiers of their users within Detasyr.

5.7.1.3 Access to data and systems

The user trying to create a cryptographic link needs to know the position of their public key $pubKey_{user}$ within the Detasyr blockchain and needs to have access to the matching private key. Access to the blockchain itself is not required. However, they need to be able to communicate with at least one currently active ticket source.
The Detasyr ticket sources require access to at least the block of the Detasyr blockchain, where the users public key is stored. Given that the ticket sources are supposed to have access to the whole Detasyr blockchain anyway, this should be the case.
The service needs access to the Detasyr blockchain to verify the link presented by the user. Specifically, it needs access to the data block where the public keys of the ticket sources that created the link are stored. Communication between the service and the ticket sources is not required.

5.7.2 Requirements

Based on the desired functionality and privacy features, a number of requirements emerge. While stronger privacy guarantees would be desirable, these are not possible due to the conflict between privacy and the required verifiability of the created link.

Services should recognize multiple shared identities of one user To avoid Sybil attacks, it should only be possible for each user to create a limited number of shared identities for a single service. For most services, a single identity is sufficient to use all features offered by the service. As such, only a single link should be created for each pair of users and services.

The service should be able to verify the created link By accessing the blockchain of Detasyr, services should be able to verify the validity of a link presented to them by one of their users. This should be possible without interacting with the user.

The link should be independent from the shared identities When the user creates multiple shared identities for one service, the same link should be created each time. Consequently, the created link should be independent of the shared identity used for the service. If the user presents the created link in multiple shared identities with the service, the service is permitted to find out that it is the same link.

Shared identities on multiple services should be unlinkable A single user is permitted to create shared identities for multiple services. To prevent tracking, even cooperating services should be unable to find out whether these identities all belong to the same user.

Users permanent identities should be kept private A service should not be able to find out which Detasyr identity, i.e., which proof of authorization, was used to create the cryptographic link. Since the Detasyr identity is a permanent identifier for the user, it should be kept secret.

5.7.3 Approach

Creating a link consists of three parts: First, the data that should be proved to the service is assembled and send to the ticket sources. The ticket sources then verify the request and sign it if it is valid. Afterwards, the user combines the responses of the ticket sources to form the cryptographic link that will be presented to the service. This link is stored in an attribute of the used shared identity of DecentID, no modification of the code of the smart contracts is required. The service can later on use the data contained in the link to verify that it was created by valid ticket sources and that they certify the authorization of the user within Detasyr.

5.7.3.1 Sending data to ticket sources

Before the ticket sources can create their signatures to prove the previous Detasyr authorization of the user, the data that should be signed has to be provided by the user to the ticket sources.

$$ID_{pk}, ID_{block} \tag{5.1}$$

Together, the values in equation (5.1) describe the position of the users public key $pubKey_{user}$ within the Detasyr blockchain. While ID_{block} is the number of the block on the blockchain, ID_{pk} describes the number of $pubKey_{user}$ within the list of public keys contained in the block.

$$ID_{service} \tag{5.2}$$

The service provides some identifying value $ID_{service}$ to the user when a link is required. This value should be unique and only used by this service, so the service can be sure that the link was created specifically for accessing it. At the same time, the value should be the same for all users of the service so two DecentID shared identities using the same Detasyr identity can be recognized by their same link. From the viewpoint of the ticket sources, $ID_{service}$ appears to be random data, i.e., no specific structure or cryptographic properties are required.

$$S = Sig_{user}(ID_{pk}|ID_{block}|ID_{service}) \tag{5.3}$$

The cryptographic signature S in equation (5.3) is created by the user using the asymmetric key pair for their Detasyr identity. As such, the public key $pubKey_{user}$ found in the blockchain location at the position of ID_{block}, ID_{pk} can be used to verify the signature. It proves towards the ticket sources that the user sending the request is actually the owner of $pubKey_{user}$, since they are able to access the matching private key and create a valid digital signature.

$$Req = \{ID_{pk}, ID_{block}, ID_{service}, S\} \tag{5.4}$$

Together, these values are send as a request Req to a currently active ticket source.

5.7.3.2 Handling at ticket sources

At the ticket source, the request is first verified. Based on ID_{block} and ID_{pk}, the public key $pubKey_{user}$ of the requesting user is recovered from the blockchain. With the public key, the signature S is checked. If the signature is valid the link request is forwarded to the other ticket sources.

$$H(Sig_{user}(ID_{pk}|ID_{block}|ID_{service})) \tag{5.5}$$

Each ticket source calculates a cryptographic hash of S. Hashing the user created signature provides additional privacy for the user. If the signature itself would be passed on to the service later on, the service would be able to iterate over all possible (ID_{block}, ID_{pk}) pairs and try to verify the signature with the referenced public keys taken from the Detasyr blockchain. If a matching public key is found, the service would have managed to find the Detasyr identity of the user.

Hashing the signature prevents this attack, since recovering the signature from the hash is not possible. Calculating the signature itself and hashing it is not possible for the service, since the service is missing the private key of the user.

$$M = H(Sig_{user}(ID_{pk}|ID_{block}|ID_{service}))|ID_{service} \tag{5.6}$$

$$\sigma_i = Sig_i(M) \tag{5.7}$$

After verifying the request, each ticket source separately calculates a Boneh–Lynn–Shacham signature σ_i over the message $M = H(S)|ID_{service}$ and sends it back to the requesting user. $ID_{service}$ is appended to $H(S)$ so the service can later on verify that the link was created specifically for itself and not for some other service. To create the signature, the ticket sources use their asymmetric key pair matching the public key stored for the respective ticket source in the newest block of the Detasyr blockchain.

5.7.3.3 Assembling the link

After receiving the different signatures σ_i created by at least half of the ticket sources, the user can assemble the cryptographic link to present to the service.

$$\sigma = \prod \sigma_i \tag{5.8}$$

To do so, the user aggregates the received signatures σ_i to a single signature σ.

$$P = \{H(S), \sigma, [i_1, \ldots, i_n], ID_{blockTS}\} \tag{5.9}$$

Afterwards, the user assembles a link data structure consisting of the hash of the users signature $H(S)$, the aggregated signature σ of the ticket sources, the indices $i_1, \ldots, i_n$ of the public keys of the contributing ticket sources within the newest block of the Detasyr blockchain, and the number $ID_{blockTS}$ of this block.
It should be noted that the created link does not contain any reference towards the shared identity of the user. This has to be avoided, since otherwise a different link would be created for each shared identity, again permitting Sybil attacks. At the same time, this allows users to use the created link in any of their shared identities without being restricted by a single one. Since they are all recognized as being controlled by the same user, by referring back to the same Detasyr identity, no Sybil attack is possible by doing so. Also, the user would be able to pass on their link to another user to permit them to use a service. However, this would exclude the user that originally created the link from accessing the service.

5.7.3.4 Verifying the link

After receiving the link data structure, e.g., by reading it from an attribute of the shared identity of the user, the service can verify it. To verify the aggregated signature of the ticket sources, the message that was signed and the public keys of the signing ticket sources are required.
To reassemble the signed message, the service concatenates the value $H(S)$, as contained within the attribute, with its own identifier $ID_{service}$. By appending $ID_{service}$ itself, and not receiving the complete message M from the user, the service can ensure that the link was created especially for itself. The required public keys of the ticket sources can be read from the block denoted by $ID_{blockTS}$ in the Detasyr blockchain. This block contains the public keys of all ticket sources that were active at the time of the creation of the link. The public keys required to verify the signature are indexed by $[i_1, \ldots, i_n]$ as given in the link data structure.
After all required information is collected, the aggregated signature can be verified as described in [BLS04]. If the verification succeeds, the service knows that the link was created specifically for this service for a user authorized within Detasyr. As a last step, the service can compare the received hash $H(S)$ to hash values already known from other shared identities[1]. If another shared identity already uses the same hash value, it means that the two shared identities are using the same Detasyr identity and are most likely controlled by the same user. Whether this is a reason to decline the new user access to the service or accept another identity of the user, depends on the service. For example, it could be permitted by the service to maintain, e.g., five different identities per user.

5.7.4 Identification of the service

As explained above, $ID_{service}$ is a value provided by the service to the user as an identifier. In its simplest case, it is an identifier unique for a service but the same for all users of it, e.g., its public key $pubKey_{service}$. Consequently, all links created by a single user for this service will contain the same user-signed value within the message M, even when the signatures created by the ticket sources differ. As intended, this allows the service to recognize multiple identities of a single user. However, this also allows the ticket sources to find out which service is used by the user, since they are able to compare the $ID_{service}$ contained in the signed data with the different $ID_{service_i}$ disclosed by known services.

[1] In practice, it might be more efficient to first check whether the hash value is already known, and verify it only when it is not known yet. Which order is preferable depends on specifics of the service, e.g., on the number of stored hash values or access speed to the Detasyr blockchain.

Unfortunately, this cannot be avoided using the presented approach. The $ID_{service}$ used within the hashed signature of the user in equation (5.5) has to remain the same for all links created by a user for a certain service. The service must not receive the data that has been signed (since that would disclose the Detasyr identity of the user) and consequently cannot check that the correct $ID_{service}$ is part of the signed data. If different $ID_{service}$ would be used, the hashed signature would be different and the service would be unable to recognize Sybil identities.
Since different values for $ID_{service}$ cannot be used to permit multiple shared identities per user at a single service, the service has to maintain a list of permitted shared identities locally. Instead of only maintaining a list of all known links to check for duplicates, the shared identities used for each link are stored as well. This way, the service can permit, e.g., up to five shared identities per link, depending on how many identities per user are acceptable for the service. If too many shared identities already presented a single link, access by the new shared identity can be declined.

5.8 Conclusion

Detasyr is a decentralized, privacy preserving system to protect against Sybil attacks. By leveraging the trust relationships of the social graph between its users, attempts to create numerous Sybil identities are thwarted.
By flooding messages through the digital representation of the social graph, Sybil adversaries with only a limited number of real-world friends are restricted from authorizing unlimited amounts of identities. Additionally, two extensions for restricting the number of Sybil identities to an absolute upper bound are presented, based on aging authorizations and edge values, respectively. The simulative evaluation has shown that this goal was reached without excessive restrictions or computational overhead for honest users. While both extensions are able to restrict the Sybil adversary, they offer different advantages. Compared to the extension using edge values, the aging authorizations can permit faster authorization for honest users and allows to withdraw authorizations. The use of edge values on the other hand requires less computation and communication effort from the participating nodes and reaches a lower amount of Sybil identities.
Since DecentID provides pseudonymous identities to its users, it is easy for an adversary to create numerous Sybil identities within it. To restrict this abuse, an approach is presented to use the authorizations gained within Detasyr as a requirement for access to online services. While doing so, the privacy offered by DecentID is not compromised, i.e., identities of a single user at multiple services cannot be linked.

Chapter 6

Conclusion

In this thesis, two novel systems were designed and evaluated: DecentID, an identity management system, and Detasyr, a Sybil defense system. Together, they provide trustworthy, self-sovereign identities, while ensuring the privacy of their users and preventing Sybil adversaries from utilizing the identities. Additionally, two use cases for DecentID were analyzed.

DecentID has been designed to grant users the control over their digital identities. Especially when interacting with online services on the Internet, digital identities are becoming more and more prevalent in everyday use. To simplify the identity management for their users, many online services support Single-Sign-On (SSO) providers, where the user creates a single identity through which they interact with multiple services. While convenient for the user, this creates a high dependency on the provider and increases the risk to the users privacy. The SSO provider can observe all activities of the user, learning which services are accessed and also how often which service is used. This can be used to create extensive user profiles, e.g., allowing targeted advertisement, and threatening the users privacy in general. Additionally, users depend on the availability and correct functioning of their identity providers: An SSO provider is, accidentally or maliciously, able to block access to the used services or report wrong user data to them.

To avoid these drawbacks, the decentralized identity management system DecentID was introduced in this thesis. It provides the functionality of SSO providers, without the need to depend on a presumably trustworthy identity provider. Furthermore, it enables users to create self-sovereign identities to access online services. Compared to existing centralized SSO providers, the user remains in control about which identity

is presented to which service. As part of that, access to the identities data is restricted to authorized users and services, protecting the privacy of the users data, without a centralized identity provider being able to pass on their personal data or track their actions when interacting with online services.
How online services can interact with DecentID was analyzed based on two use cases. In the first use case, demonstrating how other blockchain-based systems can use DecentID as an identity provider, attribute data was read from the smart contracts of DecentID without requiring external support. In the second use case, a voting system was integrated into DecentID to evaluate how data can be written to the identities. This integration enables other services to improve their decentralization by supporting majority decisions instead of depending on a single user.
The second presented system, *Detasyr*, is an approach to restrict Sybil attacks. In those, a Sybil adversary creates numerous identities to subvert the correct functioning of a service. To prevent effective Sybil attacks, the number of pseudonymous identities that a single human can create has to be restricted. However, existing Sybil defense approaches exhibit shortcomings when verifying the identity of many nodes at once and when proving the successful verification towards third parties. To fill that gap, Detasyr has been presented in this thesis. It offers three key benefits: it allows to verify many user identities at once, restricts the creation of numerous Sybil identities by a single user, and enables users to prove their successful verification. In particular, proving the verification towards third parties allows the integration of Detasyr into DecentID. By cryptographically linking the created proof, users can prove that their identity is no Sybil identity, while remaining pseudonymous.

6.1 Results

In the following, the most important contributions of DecentID, its use case analysis, and those of Detasyr are listed.

DecentID Based on the blockchain Ethereum, DecentID allows users to manage their own self-sovereign identities, which means that users are in full control of their created identities. The blockchain acts as a decentralized trust anchor, avoiding the dependence on any centralized instance. To reduce the financial costs of storing data on the blockchain, identity attributes can be stored both on-chain and off-chain. The security and integrity of the off-chain data is still ensured by cryptographic data, e.g., encryption keys and hash values, stored on the blockchain. For all attribute data, encryption can be used to ensure confidentiality. Only the identity creator controls the access to their identity: by adding permitted users to the identity, access

to the contained attributes can be granted to online services. If desired, the user can maintain multiple identities. Attributes granted by a service to one identity can be added to other identities as well, allowing the user to prove to third parties that they received a certification from the service. The security and privacy of DecentID has been evaluated and compared to the state of the art.

Use cases for DecentID To evaluate DecentIDs applicability and usability, two use cases were implemented and evaluated. The first use case analyzed how DecentID can be coupled with other smart contract-based systems, by using DecentID as an identity management for another system. Multiple approaches for coupling the two contracts were designed and evaluated, generalizing findings where possible. To further reduce the dependency of online services on centralized entities, an existing voting system was integrated into DecentID. It enables services to distribute their administration privileges by granting some of their users the right to assign attributes in the name of the service without disclosing the cryptographic private key of the service. Additionally, based on this integration, it was analyzed how attributes can be written to DecentIDs identities without requiring private keys, while still maintaining the security of the integrated access control.

Detasyr To avoid that a Sybil adversary creates large numbers of identities within DecentID and use these to attack an online service, Detasyr has been designed. It operates decentralized on an online social graph, protecting the privacy of its users by only directly communicating with the neighbors in the social graph and keeping the topology of the graph secret. A group of special nodes, called ticket sources, are periodically selected, replacing a centralized trust anchor. The ticket sources flood tickets through the social graph, allowing other nodes to generate a proof of authorization, i.e., a proof that they are no Sybil adversaries. The periodic operation of Detasyr allows to avoid a computational expensive reauthorization of all users when additional users want to be authorized. The generated proof of authorization can be shown to third parties which are able to verify the correctness of the proof. This also makes it possible to refer to this proof from within DecentID, allowing its users to prove that they are no Sybil adversaries, without compromising their privacy. The simulation-based evaluation has shown that the Sybil adversaries are effectively restricted to a small constant number of Sybil nodes. At the same time, honest users can get authorized with only a negligible delay.

6.2 Perspectives for Future Work

In their current state, both DecentID and Detasyr could be used for their respective purposes. Regarding the practical usability of DecentID, an as of now unsolved challenge is the currently high cost of creating identities and adding attributes. Since beginning the design of DecentID, the costs of calculations and storage on the blockchain Ethereum significantly increased. At present, creating an identity would most likely be unacceptable expensive for most users. Since the blockchain Ethereum itself is still under development, it remains to be seen if future changes lead to reduced costs. While there may be a small optimization potential for the smart contracts used for identities, it is unlikely that these changes will reduce the costs to an acceptable level for widespread deployment of DecentID. Alternatively, DecentID could be adapted to operate on a sidechain, i.e., a cheaper blockchain inheriting its security from Ethereum.

When coupling DecentID to other smart contracts, adherence to a standardized function interface for identity management would be preferable. Since such an interface has not been defined yet, it remains a challenge for future work.

The evaluation of DecentIDs security and privacy is mostly based on theoretical analyses. While there already exist first approaches for formal verification of Ethereum smart contracts, not all vulnerabilities can be detected and false positives can still occur. When it becomes possible, a formal verification of the designed smart contracts would be desirable, to provide a solid basis of trust for DecentID.

For Detasyr, there has to be an incentive to remain online even after a user has become authorized. This is required to allow further users to become authorized, and might be solved by coupling Detasyr with another application that the users are using anyway.

Another challenge for Detasyr is the willingness of users to accept most friendship requests in current online social networks. This breaks the assumptions of Sybil defense systems, since a Sybil adversary would be able to find further friends and add further Sybil identities. To solve this problem, users have to be educated about the importance of the relationships represented within Detasyr and that they should not just accept everyone to be their friend in the online social network.

Appendices

Appendix A

Important terms

A.1 Definitions

Attributes Page 39, definition 3.4
An attribute can be created by a *user* and contains some arbitrary data.

Authorization Ticket Page 139, definition 5.3
Flooded data packets that have to be collected to become authorized.

Creator Page 44, definition 3.7
The creator of a *shared identity* or *attribute* is the *user* or *service* that created the identity or attribute.

Off-chain attribute Page 50, definition 3.10
An off-chain attribute is an *attribute* stored outside the blockchain in external storage.

On-chain attribute Page 46, definition 3.9
An on-chain attribute is an *attribute* stored within a *SharedIdentityContract*.

Owner Page 53, definition 3.14
The owner of an *AttributeContract* is the *user* or *service* that the AttributeContract was created for.

Permitted user Page 44, definition 3.6
A permitted user is a *user* or *service* that has been permitted access to a *shared identity*.

Round Page 140, definition 5.4
Detasyr operates in rounds. In each round, new users become authorized.

Service Page 37, definition 3.2
A service is a type of *user* that offers a platform to interact with other users.

Shared Identity Page 39, definition 3.3
A shared identity $SI_{\{U\}}$ is a digital representation of a *user* U.

Sybil adversary Page 132, definition 5.1
An adversary that tries to create as many identities as possible to subvert a system.

Ticket Source Page 134, definition 5.2
Ticket sources are selected nodes that manage the operation of Detasyr for one *round*.

User Page 37, definition 3.1
A user U is a person using DecentID to manage their identities, represented by their public key $pubKey_{U_i}$.

Voter Page 108, definition 4.1
A *user* that is authorized to participate in polls in the name of a *service*.

Voting administrator Page 109, definition 4.2
A *voter* that additionally starts and manages the execution of a poll.

A.2 Smart contracts and files

AttributeContract Page 51, definition 3.12
The AttributeContract AC_i is a smart contract stored on the blockchain, representing an *attribute* stored off-chain.

AttributeData Page 52, definition 3.13
A file AD_i stored in an external storage that contains the data of an attribute.

AttributeLocatorFile Page 51, definition 3.11
A file $ALF_{\{U,A\},1}$ stored in an external storage, containing a list of blockchain addresses to *AttributeContracts*.

IdentityLocatorFile Page 55, definition 3.16
Similar to the *AttributeLocatorFile*, an IdentityLocatorFile $ILF_{\{U\}}$ is stored off-chain and contains references to the *SharedIdentityContracts* of a *user* U.

RootIdentityContract Page 55, definition 3.15
A RootIdentityContract $RIC_{\{U\}}$ is a smart contract stored on the blockchain, storing references to the *SharedIdentityContracts* and *AttributeContracts* of a *user* U.

SharedIdentityContract Page 43, definition 3.5
A SharedIdentityContract $SIC_{\{U\}}$ is the technical representation of the *shared identity* of a *user* U within DecentID.

VotingContract Page 115, section 4.2.3.2
The static VotingContract contains the program code required to execute polls in DecentID.

VotingDataContract Page 116, section 4.2.3.3
A new VotingDataContract is created for each poll, storing the state of the poll.

A.3 Cryptographic keys

$kSIC_{\{U\}}$ Page 44, definition 3.8
$kSIC_{\{U\}}$ is a symmetric *attribute* encryption key used for encrypting the attribute data stored within the *SharedIdentityContract* $SIC_{\{U\}}$ as well as linked AttributeLocatorFiles.

$privKey_{U_i}$
The i-th private key $privKey_{U_i}$ of a *user* U. Forms an asymmetric key pair together with the public key $pubKey_{U_i}$.

$pubKey_{U_i}$
The i-th public key $pubKey_{U_i}$ of a *user* U. Forms an asymmetric key pair together with the private key $privKey_{U_i}$.

Appendix B

Smart Contracts of DecentID

B.1 Mortal.sol

The smart contract *Mortal* is a common contract on the blockchain Ethereum, designed to be used as a parent contract for others. When created, it stores the public key of the user creating it in the variable `creator`. Later on, it allows this user to delete the smart contract from the blockchain by calling the function `kill()`. This returns some of the Ether spend on deploying the contract back to its creator. Also it supports the modifier `onlyByCreator`, allowing to provide access control to functions that should only be called by the creator of the smart contract.

```
1  pragma solidity ^0.5.0;
2
3  contract Mortal {
4  address payable creator;
5
6  constructor() public {
7      creator = msg.sender;
8  }
9
10 modifier onlyByCreator {
11     require(msg.sender == creator);
12     _; // This line is replaced by the
13        // code where onlyByCreator is applied to
14 }
15
16 function kill() public onlyByCreator {
17     selfdestruct(creator);
18 }
19 }
```

B.2 RootIdentityContract.sol

As described in section 3.6, the *RootIdentityContract* stores references to the SharedIdentityContracts of the user as well as to currently unused attributes.

```
pragma solidity ^0.5.0;

import "./Mortal.sol";

contract RootIdentityContract is Mortal {

string public identityCLFref;
string public attributeCLFref;
string public clfKey;

constructor(string memory _clfKey) public {
    clfKey = _clfKey;
}

function updateIdentityCLF(string memory _storageReference)
        public onlyByCreator {
    identityCLFref = _storageReference;
}

function updateAttributeCLF(string memory _storageReference)
        public onlyByCreator {
    attributeCLFref = _storageReference;
}

function updateCLFkey(string memory _clfKey) public onlyByCreator {
    clfKey = _clfKey;
}
}
```

B.3 SharedIdentityContract.sol

The *SharedIdentityContract*, as described in section 3.5, is the central part of the digital identities within DecentID. It stores identity attributes and references to off-chain attributes for each user in the `attributes` mapping within the `PermittedUser` structure.

```
pragma solidity ^0.5.0;

import "./Mortal.sol";
import "./VotingDataContract.sol";

contract SharedIdentityContract is Mortal {

struct Attribute {
    byte flags;
    bytes data;
}

// flag_set is used to check whether indexing key is used
byte public constant flag_set = 0x01;
byte public constant flag_external = 0x02;
byte public constant flag_encrypted = 0x04;

struct PermittedUser {
    bytes key;
    mapping (string => Attribute) attributes;
}

mapping (address => PermittedUser) users;
// The list of addresses is used by user interfaces to display
// which permitted users are registered.
// Entry 0 is the creator of the identity
address payable[] public userAddrs;

VotingContract public votingContract;

// The hash of the VotingDataContract can be  set before deployment
bytes32 private vdcHash = 0x3A...;
```

```
34 function getContractHash(address _addr) public view
35         returns (bytes32) {
36     bytes32 codehash;
37     assembly {
38         codehash := extcodehash(_addr)
39     }
40     return codehash;
41 }
42
43 constructor (VotingContract _votingContract, bytes memory _key)
44         public {
45     require(_key.length != 0);
46     users[msg.sender] = PermittedUser(_key);
47     userAddrs.push(msg.sender);
48     votingContract = _votingContract;
49 }
50
51 function addPermittedUser (address payable _user, bytes memory _key)
52         public onlyByCreator {
53     require(_key.length != 0);
54     require(users[_user].key.length == 0);
55     users[_user] = PermittedUser(_key);
56     userAddrs.push(_user);
57 }
58
59 function removePermittedUser (address _user) public onlyByCreator {
60     assert(users[_user].key.length != 0);
61     // First entry is skipped: Creator cannot be removed
62     for (uint i = 1; i < userAddrs.length; i++) {
63         if (userAddrs[i] == _user) {
64             userAddrs[i] = userAddrs[userAddrs.length - 1];
65             delete userAddrs[userAddrs.length - 1];
66             userAddrs.length--;
67             delete users[_user];
68             assert(users[_user].key.length == 0);
69             return;
70         }
71     }
72 }
```

```
74  function setAttribute(string memory key, byte flags,
75          bytes memory data) public {
76      assert(users[_user].key.length != 0);
77      users[msg.sender].attributes[key] =
78          Attribute(flags | flag_set, data);
79  }
80
81  function deleteAttribute(string memory key) public {
82      require (users[msg.sender].key.length != 0);
83      delete users[msg.sender].attributes[key];
84  }
85
86  function getAttribute (address _user, string memory _key)
87          public view returns (bytes memory) {
88      if (users[_user].key.length == 0) {
89          return "";
90      }
91      Attribute memory attr = users[_user].attributes[_key];
92      if ((attr.flags & flag_set) == 0) {
93          return "";
94      }
95      return attr.data;
96  }
97
98  function getAttributeFlags (address _user, string memory _key)
99          public view returns (byte) {
100     if (users[_user].key.length == 0) {
101         return 0x00;
102     }
103     Attribute memory attr = users[_user].attributes[_key];
104     return attr.flags;
105 }
106
107 function updateKey (address _user, bytes memory _newKey)
108         public onlyByCreator {
109     users[_user].key = _newKey;
110 }
```

```
112 function startVoting(SharedIdentityContract sic,
113         address _pkService, string memory _key, byte _flags,
114         bytes memory _data) public returns (VotingDataContract) {
115     require (users[_pkService].key.length != 0);
116     require (getContractHash(address(this)) ==
117         getContractHash(address(sic)));
118     require (sic.userAddrs(0) == msg.sender);
119     require (sic.isVoterFor(_pkService));
120     VotingDataContract vdc
121         = new VotingDataContract(msg.sender, votingContract,
122             _pkService, _key, _flags, _data);
123     assert(vdc.target() == this);
124     return vdc;
125 }
126
127 function updateAttribute(VotingDataContract vdc) public {
128     require (getContractHash(address(vdc)) == vdcHash);
129     require (vdc.target() == this);
130     require (vdc.state() == VotingDataContract.State.FINISHED);
131     require (vdc.finaltally() > vdc.totalvoted() / 2);
132     if ((vdc.flags() & flag_set) != 0) {
133         users[vdc.pkService()].attributes[vdc.key()] =
134             Attribute(vdc.flags(), vdc.data());
135     } else {
136         delete users[vdc.pkService()].attributes[vdc.key()];
137     }
138 }
139
140 function isVoterFor (address _user) public view returns (bool) {
141     if (users[_user].key.length == 0) {
142         return false;
143     }
144     return (users[_user].attributes["voter"].flags & flag_set)
145         != 0;
146 }
147 }
```

The functions `startVoting()`, `updateAttribute()`, and `isVoterFor()` have been added to support the integration of Open Vote Network.

B.4 AttributeContract.sol

The *AttributeContract*, introduced in section 3.5.3 and section 3.5.4, references to attribute data stored outside the blockchain with its `storageReference`. Additionally, it allows to set an owner of the attribute that is different than the attributes creator. The code of the smart contract ensures that only the creator is able to set the owner.

```
1  pragma solidity ^0.5.0;
2
3  import "./Mortal.sol";
4
5  contract AttributeContract is Mortal {
6
7  address public owner;
8  string public storageReference;
9
10 constructor(address _owner, string memory _storageReference) public {
11     owner = _owner;
12     storageReference = _storageReference;
13 }
14
15 function updateReference(string memory _storageReference) public
       onlyByCreator {
16     storageReference = _storageReference;
17 }
18
19 function transferOwnership(address _newOwner) public onlyByCreator {
20     owner = _newOwner;
21 }
22 }
```

B.5 VotingDataContract.sol

As described in section 4.2.3.3, the *VotingDataContract* stores the data of a currently running poll. Its functions ensure that they are called by the registered VotingContract, and then store the function parameters in the matching state variables.

```
pragma solidity ^0.5.0;

import "./Mortal.sol";

contract VotingDataContract is Mortal {

// Verification of callers
SharedIdentityContract public target;
address public pkService;

// Attribute to create
string public key;
byte public flags;
bytes public data;

VotingContract public votingContract;
address public voteAdmin;

enum Stage { SIGNUP, VOTE, FINISHED }
Stage public stage;
uint constant stageDuration = 3 * 60;
// Timestamp when the next stage starts in seconds
uint public timeNextStage;
// Total number of participants that have submited a voting key
uint public totalregistered;
uint public totalvoted;
uint[2] public finaltally; // Final tally

struct Voter {
    address addr;
    uint[2] registeredkey;
    uint[2] reconstructedkey;
    uint[2] vote;
}
```

```
36 // Address to index in voters
37 mapping (address => uint) public addressid;
38 // Registered voters, indices in addressid
39 mapping (uint => Voter) public voters;
40 // Address registered?
41 mapping (address => bool) public registered;
42 // Address voted?
43 mapping (address => bool) public votecast;
44 // Have we received their deposit?
45 mapping (address => uint) public refunds;
46
47
48 constructor(address _admin, VotingContract _vc,
49         address _pkService, string memory _key, byte _flags,
50         bytes memory _data) public {
51     votingContract = _vc;
52     target = SharedIdentityContract(msg.sender);
53     pkService = _pkService;
54     key = _key;
55     flags = _flags;
56     data = _data;
57     voteAdmin = _admin;
58     stage = Stage.SIGNUP;
59     timeNextStage = block.timestamp + stageDuration;
60 }
61
62 function signUp(address _sender, uint[2] memory xG)
63         public payable {
64     require (msg.sender == address(votingContract));
65     refunds[msg.sender] += msg.value;
66     refunds[_sender] += msg.value;
67     uint[2] memory empty;
68     addressid[_sender] = totalregistered;
69     voters[totalregistered] = Voter({addr: _sender,
70         registeredkey: xG, reconstructedkey: empty, vote: empty});
71     registered[_sender] = true;
72     totalregistered += 1;
73 }
```

```
75 function setReconstructedKey(address sender, uint i, uint a,
76         uint b) public {
77     require (msg.sender == address(votingContract));
78     require (sender == voteAdmin);
79     voters[i].reconstructedkey[0] = a;
80     voters[i].reconstructedkey[1] = b;
81 }
82
83 function setStage(address sender, Stage s) public {
84     require (msg.sender == address(votingContract));
85     require (sender == voteAdmin);
86     stage = s;
87     timeNextStage = block.timestamp + stageDuration;
88 }
89
90 function submitVote(address payable sender, uint[2] memory y)
91         public {
92     require (msg.sender == address(votingContract));
93     require (registered[sender]);
94     require (!votecast[sender]);
95
96     uint i = addressid[sender];
97     voters[i].vote[0] = y[0];
98     voters[i].vote[1] = y[1];
99     votecast[sender] = true;
100    totalvoted += 1;
101
102    // Voter has voted, send back the deposit
103    uint refund = refunds[sender];
104    refunds[sender] = 0;
105    // If sending the Ether failed, store the deposit
106    if (!sender.send(refund)) {
107        refunds[sender] = refund;
108    }
109 }
```

```
111 function setFinalTally(address payable sender, uint a, uint b)
112         public {
113     require (msg.sender == address(votingContract));
114     require (sender == voteAdmin);
115     finaltally[0] = a;
116     finaltally[1] = b;
117 }
118 
119 function getRegisteredKey(uint voterId) public view
120         returns (uint[2] memory) {
121     return voters[voterId].registeredkey;
122 }
123 
124 function getReconstructedKey(uint voterId) public view
125         returns (uint[2] memory) {
126     return voters[voterId].reconstructedkey;
127 }
128 
129 function getVoterAddr(uint voterId) public view returns (address) {
130     return voters[voterId].addr;
131 }
132 
133 function getVoterVote(uint voterId) public view
134         returns (uint[2] memory) {
135     return voters[voterId].vote;
136 }
137 }
```

B.6 VotingContract.sol

The static *VotingContract*, described in section 4.2.3.2, stores the program code required to execute a poll. All data specific for a individual poll is stored in a VotingDataContract. The implementation of the VotingContract is based on the implementation of OpenVoteNetwork which was written by McCorry et al. [MSH17].

```
pragma solidity ^0.5.0;

import "./SharedIdentityContract.sol";
import "./VotingDataContract.sol";
// Import mathematical libraries ECCMath and Secp256k1
// offering elliptic curve cryptography
import "./ECCMath.sol"

// Implementation of OpenVoteNetwork by Patrick McCorry and Jon Johnson
// written in 2017 at https://github.com/stonecoldpat/anonymousvoting
// Adapted to work with SharedIdentityContracts and VotingDataContracts

contract VotingContract {

function signUp(VotingDataContract vdc, SharedIdentityContract sic,
        uint[2] memory xG, uint[3] memory vG, uint r)
        public payable returns (bool) {
    require (vdc.votingContract() == this);
    require (vdc.stage() == VotingDataContract.Stage.SIGNUP);
    require (block.timestamp < vdc.timeNextStage());
    require (msg.value >= 10);
    require (sic.ownerAddrs(0) == msg.sender);
    require (sic.isVoterfor (vdc.pkService()));

    if (Secp256k1.verifyZKP(msg.sender, xG,r,vG)
            && !vdc.registered(msg.sender)) {
        vdc.signUp.value(msg.value)(msg.sender, xG);
        return true;
    }
    return false;
}
```

```
32
33 function finishSignUp(VotingDataContract vdc) public {
34
35     require (vdc.totalregistered() >= 2);
36     require (vdc.stage() == VotingDataContract.Stage.SIGNUP);
37     require (block.timestamp >= vdc.timeNextStage());
38     require (msg.sender == vdc.voteAdmin());
39
40     uint[2] memory temp;
41     uint[3] memory yG;
42     uint[3] memory beforei;
43     uint[3] memory afteri;
44
45     // Step 1 is to compute the index 1 reconstructed key
46     {
47         uint[2] memory key = vdc.getRegisteredKey(1);
48         afteri[0] = key[0];
49         afteri[1] = key[1];
50         afteri[2] = 1;
51     }
52
53     for (uint i=2; i<vdc.totalregistered(); i++) {
54         Secp256k1._addMixedM(afteri, vdc.getRegisteredKey(i));
55     }
56
57     ECCMath.toZ1(afteri,Secp256k1.pp());
58     vdc.setReconstructedKey(msg.sender, 0, afteri[0],
59         Secp256k1.pp() - afteri[1]);
60
61     // Step 2 is to add to beforei, and subtract from afteri.
62     for (uint i=1; i<vdc.totalregistered(); i++) {
63
64         if (i==1) {
65             uint[2] memory key = vdc.getRegisteredKey(0);
66             beforei[0] = key[0];
67             beforei[1] = key[1];
68             beforei[2] = 1;
69         } else {
70             Secp256k1._addMixedM(beforei, vdc.getRegisteredKey(i-1));
71         }
```

```
72
73          // If we have reached the end, just store beforei
74          // Otherwise, we need to compute a key.
75          // Counting from 0 to n-1
76          if (i==(vdc.totalregistered()-1)) {
77              ECCMath.toZ1(beforei,Secp256k1.pp());
78              vdc.setReconstructedKey(msg.sender, i,
79                  beforei[0], beforei[1]);
80          } else {
81                  // Subtract 'i' from afteri
82                  uint[2] memory key = vdc.getRegisteredKey(i);
83                  temp[0] = key[0];
84                  temp[1] = Secp256k1.pp() - key[1];
85                  Secp256k1._addMixedM(afteri,temp);
86                  ECCMath.toZ1(afteri,Secp256k1.pp());
87                  temp[0] = afteri[0];
88                  temp[1] = Secp256k1.pp() - afteri[1];
89                  yG = Secp256k1._addMixed(beforei, temp);
90                  ECCMath.toZ1(yG,Secp256k1.pp());
91
92                  vdc.setReconstructedKey(msg.sender, i, yG[0], yG[1]);
93          }
94      }
95      vdc.setStage(msg.sender, VotingDataContract.Stage.VOTE);
96  }
97
98  function submitVote(VotingDataContract vdc, uint[4] memory params,
99          uint[2] memory y, uint[2] memory a1, uint[2] memory b1,
100         uint[2] memory a2, uint[2] memory b2) public returns (bool) {
101
102     require (vdc.stage() == VotingDataContract.Stage.VOTE);
103     require (block.timestamp < vdc.timeNextStage());
104
105     if (vdc.registered(msg.sender) && !vdc.votecast(msg.sender)) {
106         uint i = vdc.addressid(msg.sender);
```

```
107
108         // Verify the ZKP for the vote being cast
109         if (Secp256k1.verify1outof2ZKP(msg.sender,
110                 vdc.getReconstructedKey(i), vdc.getRegisteredKey(i),
111                 params, y, a1, b1, a2, b2)) {
112             vdc.submitVote(msg.sender, y);
113             return true;
114         }
115     }
116     return false;
117 }
118
119 function computeTally(VotingDataContract vdc) public {
120
121     require (vdc.stage() == VotingDataContract.Stage.VOTE);
122     require (block.timestamp >= vdc.timeNextStage()
123         || vdc.totalregistered() == vdc.totalvoted());
124     require (msg.sender == vdc.voteAdmin());
125     require (vdc.totalregistered() == vdc.totalvoted());
126
127     uint[3] memory temp;
128     uint[2] memory vote;
129     uint refund;
130
131     // Sum all votes
132     for (uint i=0; i<vdc.totalregistered(); i++) {
133             // Confirm all votes have been cast...
134             if (!vdc.votecast(vdc.getVoterAddr(i))) {
135                     revert();
136             }
137             vote = vdc.getVoterVote(i);
138
139             if (i==0) {
140                 temp[0] = vote[0];
141                 temp[1] = vote[1];
142                 temp[2] = 1;
143             } else {
144                     Secp256k1._addMixedM(temp, vote);
145             }
146     }
```

```
147     // Now temp contains G^sum(vi), that is G^(sum of yes votes)
148     vdc.setStage(msg.sender, VotingDataContract.Stage.FINISHED);
149 
150     // Each vote is represented by a G.
151     if (temp[0] == 0) {
152         vdc.setFinalTally(msg.sender, 0, vdc.totalregistered());
153         return;
154     } else {
155         // There must be a vote. Add 'G' until we find the result.
156         // temp is G^(sum of yes votes), so brute force the logarithm
157         ECCMath.toZ1(temp,Secp256k1.pp());
158         uint[3] memory tempG;
159         tempG[0] = Secp256k1.G()[0];
160         tempG[1] = Secp256k1.G()[1];
161         tempG[2] = 1;
162 
163         // totalregistered() is the maximal number of tries,
164         // there can't be more "yes" votes than total votes
165         for (uint i=1; i<=vdc.totalregistered(); i++) {
166 
167             // We hit a matching value.
168             if (temp[0] == tempG[0]) {
169                     vdc.setFinalTally(msg.sender, i,
170                         vdc.totalregistered());
171                     if (i > vdc.totalregistered() / 2) {
172                         SharedIdentityContract sic =
173                             SharedIdentityContract(vdc.target());
174                         sic.updateAttribute(vdc);
175                     }
176                     return;
177             }
178             Secp256k1._addMixedM(tempG, Secp256k1.G());
179             ECCMath.toZ1(tempG,Secp256k1.pp());
180         }
181         // Error: Should never get here
182         vdc.setFinalTally(msg.sender, 0, 0);
183     }
184 }
185 }
```

Bibliography

[Ahn+03] L. von Ahn et al. “CAPTCHA: Using Hard AI Problems for Security.” In: *Advances in Cryptology — EUROCRYPT 2003*. Ed. by E. Biham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 294–311. ISBN: 978-3-540-39200-2.

[Alf96] S. V. Alfred Menezes Paul van Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996. URL: https://cacr.uwaterloo.ca/hac/about/chap9.pdf.

[All16] C. Allen. *The Path to Self-Sovereign Identity*. Accessed: 2022-01-12. 2016. URL: https://www.coindesk.com/markets/2016/04/27/the-path-to-self-sovereign-identity/.

[BA99] A.-L. Barabási and R. Albert. “Emergence of Scaling in Random Networks.” In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: http://science.sciencemag.org/content/286/5439/509.full.pdf. URL: http://science.sciencemag.org/content/286/5439/509.

[Ben14] J. Benet. “IPFS - Content Addressed, Versioned, P2P File System.” In: *CoRR* abs/1407.3561 (2014). URL: http://arxiv.org/abs/1407.3561.

[Bil+09] L. Bilge et al. “All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks.” In: *Proceedings of the 18th International Conference on World Wide Web*. Madrid, Spain: Association for Computing Machinery, 2009. DOI: 10.1145/1526709.1526784.

[BLS04] D. Boneh, B. Lynn, and H. Shacham. “Short signatures from the Weil pairing.” In: *Journal of cryptology* 17.4 (2004), pp. 297–319.

[Boe+08] S. Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. DOI: 10.17487/RFC5280. URL: https://rfc-editor.org/rfc/rfc5280.txt.

[Bon+03] D. Boneh et al. “Aggregate and verifiably encrypted signatures from bilinear maps.” In: *Proceedings of Eurocrypt* 2656 (2003), pp. 416–32.

[Bra03] P. Braendgaard. *ERC-1812: Ethereum Verifiable Claims*. 2019-03-03. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1812.md (visited on 03/19/2021).

[Bro09] D. R. L. Brownt. “SEC 1: Elliptic Curve Cryptography.” Version 2.0. In: (May 2009). URL: https://www.secg.org/sec1-v2.pdf (visited on 08/28/2021).

[But+13] V. Buterin et al. “Ethereum white paper: a next generation smart contract and decentralized application platform.” In: (2013). Accessed: 2021-12-23. URL: https://ethereum.org/en/whitepaper/.

[Cho20] E. Cholakov. “InDependency: An App UI for DecentID.” Bachelor Thesis. Karlsruhe Institute of Technology, 2020.

[Dan+14] G. Danezis et al. *Privacy and Data Protection by Design - from Policy to Engineering*. Dec. 2014. ISBN: 978-92-9204-108-3. DOI: 10.2824/38623.

[Den+11] M. Deng et al. “A privacy threat analysis framework: Supporting the elicitation and fulfillment of privacy requirements.” In: *Requir. Eng.* 16 (Mar. 2011), pp. 3–32. DOI: 10.1007/s00766-010-0115-7.

[DM09] G. Danezis and P. Mittal. “SybilInfer: Detecting Sybil Nodes using Social Networks.” In: *NDSS*. San Diego, CA. 2009.

[Dou02] J. R. Douceur. “The Sybil Attack.” In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS ’01. London, UK: Springer-Verlag, 2002, pp. 251–260. ISBN: 3-540-44179-4.

[DP18] P. Dunphy and F. A. Petitcolas. “A First Look at Identity Management Schemes on the Blockchain.” In: *IEEE Security Privacy* 16.4 (2018), pp. 20–29. DOI: 10.1109/MSP.2018.3111247.

[DR13] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[Fab02] T. Y. Fabian Vogelsteller. *ERC-725: Smart Contract Based Account*. 2017-10-02. URL: https://github.com/ethereum/EIPs/issues/725 (visited on 03/19/2021).

[FFB17] S. Friebe, M. Florian, and I. Baumgart. "Decentralized and sybil-resistant pseudonym registration using social graphs." In: *14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 12th - 14th December 2016*. IEEE. 2017, pp. 121–128. ISBN: 978-1-5090-4379-8. DOI: 10.1109/PST.2016.7906946.

[FMZ19] S. Friebe, P. Martinat, and M. Zitterbart. "Detasyr: Decentralized Ticket-based Authorization with Sybil Resistance." In: *IEEE 44th Conference on Local Computer Networks (LCN), Osnabrueck, Germany, 14-17 Oct. 2019*. 44th IEEE Conference on Local Computer Networks. LCN 2019. 2019, pp. 60–68. ISBN: 978-1-7281-1029-5. DOI: 10.1109/LCN44214.2019.8990773.

[Fri+21] S. Friebe et al. "Coupling Smart Contracts: A Comparative Case Study." In: *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 27-30 Sept. 2021*. 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services. IEEE. 2021, pp. 137–144. ISBN: 978-1-6654-3925-1. DOI: 10.1109/BRAINS52497.2021.9569830.

[FSZ18] S. Friebe, I. Sobik, and M. Zitterbart. "DecentID: Decentralized and Privacy-Preserving Identity Storage System Using Smart Contracts." In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), New York, NY, August 1-3, 2018*. 2018, pp. 37–42. ISBN: 978-1-5386-4388-4. DOI: 10.1109/TrustCom/BigDataSE.2018.00016.

[GA11] M. P. G. Bertoni J. Daemen and G. V. Assche. *The Keccak reference, SHA-3 competition (round 3)*. 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf (visited on 08/24/2021).

[HC48] J. P. Humphrey and R. Cassin. *Universal Declaration of Human Rights*. Dec. 1948. URL: https://www.un.org/en/about-us/universal-declaration-of-human-rights.

[JMV01] D. Johnson, A. Menezes, and S. Vanstone. "The elliptic curve digital signature algorithm (ECDSA)." In: *Int. J. Inf. Sec.* 1 (Aug. 2001), pp. 36–63. DOI: 10.1007/s102070100002.

[KL14] J. Katz and Y. Lindell. *Introduction to modern cryptography*. 2014.

[KNT06] R. Kumar, J. Novak, and A. Tomkins. "Structure and Evolution of Online Social Networks." In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 611–617. ISBN: 1595933395. DOI: 10.1145/1150402.1150476.

[KSD13] C. F. Kerry, A. Secretary, and C. R. Director. *FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS)*. 2013.

[Liu+18] Y. Liu et al. "Unlinkable Coin Mixing Scheme for Transaction Privacy Enhancement of Bitcoin." In: *IEEE Access* 6 (2018), pp. 23261–23270. DOI: 10.1109/ACCESS.2018.2827163.

[Lun+17] D. C. Lundkvist et al. *[Whitepaper] uPort: A Platform for Self-Sovereign Identity*. 2017.

[Mal+21] D. Maldonado-Ruiz et al. "An Innovative and Decentralized Identity Framework Based on Blockchain Technology." In: *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2021, pp. 1–8. DOI: 10.1109/NTMS49979.2021.9432656.

[MDS19] P. Mell, J. Dray, and J. Shook. "Smart contract federated identity management without third party authentication services." In: *arXiv preprint arXiv:1906.11057* (2019).

[MSH17] P. McCorry, S. F. Shahandashti, and F. Hao. "A smart contract for boardroom voting with maximum voter privacy." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 357–375.

[Nak08] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[NS09] A. Narayanan and V. Shmatikov. "De-anonymizing Social Networks." In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009. DOI: 10.1109/SP.2009.22.

[Pel03] J. T. Pelle Braendgaard. *ERC-1056: Lightweight Identity*. 2018-05-03. URL: https://github.com/ethereum/EIPs/issues/1056 (visited on 03/19/2021).

[PH10] A. Pfitzmann and M. Hansen. *A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management*. Version 0.34. Aug. 2010. URL: https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf.

[RF06] A. Ramachandran and N. Feamster. "Understanding the Network-level Behavior of Spammers." In: *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '06. Pisa, Italy: ACM, 2006, pp. 291–302. ISBN: 1-59593-308-5. DOI: 10.1145/1159913.1159947.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342.

[Shi+13] L. Shi et al. "SybilShield: An agent-aided social network-based Sybil defense among multiple communities." In: *2013 Proceedings IEEE INFOCOM*. Apr. 2013, pp. 1034–1042. DOI: 10.1109/INFCOM.2013.6566893.

[Sho01] V. Shoup. "A proposal for an ISO standard for public key encryption." Version 2.1. In: (Dec. 20, 2001). URL: https://www.shoup.net/papers/iso-2_1.pdf (visited on 08/28/2021).

[Sob17] I. Sobik. "DecentID: Distributed and Privacy-preserving Identity Management using Blockchain-based Smart Contracts." Master Thesis. Karlsruhe Institute of Technology, 2017.

[SP18] Q. Stokkink and J. Pouwelse. "Deployment of a blockchain-based self-sovereign identity." In: *2018 IEEE international conference on Internet of Things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*. IEEE. 2018, pp. 1336–1342.

[SS16] SITA and ShoCard. *White Paper: Travel Identity of the Future*. Accessed: 2022-01-12. May 2016. URL: https://blockchainlab.com/pdf/2016-05-00-idm-ShoCard-travel-identity-of-the-future.pdf.

[Ste+19] O. Stengele et al. "Access Control for Binary Integrity Protection Using Ethereum." In: *Proc. of the 24th ACM SACMAT*. Toronto ON, Canada: ACM, 2019, pp. 3–12. ISBN: 9781450367530. DOI: 10.1145/3322431.3325108.

[THE17] THEKEY. *A Decentralized Ecosystem of an Identity Verification Tool Using National Big-data and Blockchain [Whitepaper]*. 2017.

[Tra+11] N. Tran et al. “Optimal Sybil-resilient node admission control.” In: *The 30th IEEE International Conference on Computer Communications (INFOCOM 2011)*. Apr. 2011.

[Vis+10] B. Viswanath et al. “An Analysis of Social Network-based Sybil Defenses.” In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 363–374. ISSN: 0146-4833. DOI: 10.1145/1851275.1851226.

[Wei+12] W. Wei et al. “SybilDefender: Defend against sybil attacks in large social networks.” In: *2012 Proceedings IEEE INFOCOM*. Mar. 2012, pp. 1951–1959. DOI: 10.1109/INFCOM.2012.6195572.

[WGK19] M. Westerkamp, S. Göndör, and A. Küpper. “Tawki:Towards self-sovereign social communication.” In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE. 2019, pp. 29–38.

[WR18] P. Windley and D. Reed. “Sovrin: A protocol and token for self-sovereign identity and decentralized trust.” Version 1.0. In: *Whitepaper, The Sovrin Foundation* (2018).

[WS98] D. J. Watts and S. H. Strogatz. “Collective dynamics of ’small-world’ networks.” In: *nature* 393.6684 (1998), p. 440.

[Xia+21] R. Xiao et al. “A Mixing Scheme Using a Decentralized Signature Protocol for Privacy Protection in Bitcoin Blockchain.” In: *IEEE Transactions on Dependable and Secure Computing* 18.4 (2021), pp. 1793–1803. DOI: 10.1109/TDSC.2019.2938953.

[Yu+06] H. Yu et al. “SybilGuard: Defending Against Sybil Attacks via Social Networks.” In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. New York, NY, USA: ACM, Aug. 2006, pp. 267–278. DOI: 10.1145/1151659.1159945.

[Yu+08] H. Yu et al. “SybilLimit: A Near-Optimal Social Network Defense Against Sybil Attacks.” In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–17. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.13.

[Zie10] P. Zieliński. “Anonymous voting by two-round public discussion.” English. In: *IET Information Security* 4 (2 June 2010), 62–67(5). ISSN: 1751-8709. URL: https://digital-library.theiet.org/content/journals/10.1049/iet-ifs.2008.0127.

List of Publications

Conference articles

1. Sebastian Friebe, Martin Florian, and Ingmar Baumgart. Decentralized and sybil-resistant pseudonym registration using social graphs. In *14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 12th - 14th December 2016*, pages 121–128. Institute of Electrical and Electronics Engineers (IEEE), 2017

2. Sebastian Friebe, Ingo Sobik, and Martina Zitterbart. Decentid: Decentralized and privacy-preserving identity storage system using smart contracts. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), New York, NY, August 1-3, 2018*, pages 37–42. Institute of Electrical and Electronics Engineers (IEEE), 2018

3. Sebastian Friebe, Paul Martinat, and Martina Zitterbart. Detasyr: Decentralized ticket-based authorization with sybil resistance. In *IEEE 44th Conference on Local Computer Networks (LCN), Osnabrueck, Germany, 14-17 Oct. 2019*, pages 60–68. Institute of Electrical and Electronics Engineers (IEEE), 2019

4. Jonas Schiffl, Matthias Grundmann, Marc Leinweber, Oliver Stengele, Sebastian Friebe, and Bernhard Beckert. Towards correct smart contracts: A case study on formal verification of access control. In *SACMAT '21: Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*, pages 125–130. Association for Computing Machinery (ACM), 2021

5. Sebastian Friebe, Oliver Stengele, Hannes Hartenstein, and Martina Zitterbart. Coupling smart contracts: A comparative case study. In *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 27-30 Sept. 2021*, pages 137–144. Institute of Electrical and Electronics Engineers (IEEE), 2021

Demonstrators

1. Sebastian Friebe and Martin Florian. Desyps : A decentralized, sybil-resistant, pseudonymous online discussion platform. In *Proceedings of the International Conference on Networked Systems, NetSys 2017, Gttingen, Germany, 13th - 16th March 2017*, pages 1–2. Institute of Electrical and Electronics Engineers (IEEE), 2017

2. Sebastian Friebe and Martin Florian. Dps-discuss: Demonstrating decentralized, pseudonymous, sybil-resistant communication. In *SIGCOMM Posters and Demos'17. Proceedings of the 2017 SIGCOMM Posters and Demos, Los Angeles, CA, August 22-24, 2017*, pages 74–75. Association for Computing Machinery (ACM), 2017