ITeG | Interdisciplinary Research on Information System Design | 5

Olga Kieselmann

Data Revocation on the Internet

ITeG Research Center for Information System Design

kassel university press

ITeG – Interdisciplinary Research on Information System Design

Band 5 / Vol. 5

Herausgegeben von / Edited by
ITeG Wissenschaftliches Zentrum für Informationstechnik-Gestaltung
an der Universität Kassel

Universität Kassel
ITeG Wissenschaftliches Zentrum
für Informationstechnik-Gestaltung
Pfannkuchstraße 1
D-34121 Kassel

Olga Kieselmann

Data Revocation on the Internet

kassel
university
press

This work has been accepted by the Faculty of Electrical Engineering and Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Naturwissenschaften (Dr. rer. nat.).

Supervisor: Prof. Dr. Arno Wacker
Co-Supervisor: Prof. Dr. Rüdiger Grimm

Defense day: 24th October 2017

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.dnb.de.

Zugl.: Kassel, Univ., Diss. 2017
ISBN 978-3-7376-0420-8 (print)
ISBN 978-3-7376-0421-5 (e-book)
DOI: http://dx.medra.org/10.19211/KUP9783737604215
URN: http://nbn-resolving.de/urn:nbn:de:0002-404218

www.upress.uni-kassel.de

Printed in Germany

Foreword

Nowadays, especially since the invention of Web 2.0, every user is able to publish arbitrary data on the Internet. However, sometimes users change their mind and/or for any other reason want to delete their previously published data from the Internet. This data can be either some data about the user herself, which she published willingly, or data published by some third party about her. Even though there are some previous approaches to delete data on the Internet, none of them prevailed until now. The reason for this is the open and decentralized nature of the Internet itself. Previously published data resides on many and potentially different servers, maintained and controlled by different entities. Hence, in order to delete previously published data, the user would need to delete it on a foreign server, i.e., where she has absolutely no control. Some previous approaches even go beyond this and try to delete that data not only on foreign servers but also on other users' computers after they downloaded the published data. However, this is a challenging task in computer science and commonly known as the "hostile host problem". The hostile host problem states that it is generally very hard to delete data on a system, on which the user has no administrative control. Due to the lack of success from previous approaches and the fact that the hostile host problem is considered a very hard computer science problem, the term "The Internet never forgets" became very popular in the last years. Until recently, all existing approaches for deleting data on the Internet have been pure technical solutions. Due to the increased demand of many Internet users in the last years for a practical usable solution, the legislative authority reacted by creating the General Data Protection Regulation (GDPR). Most influential for a practical solution to delete data on the Internet is Article 17 of the GDPR. This article regulates that any controller within the jurisdiction of the EU must delete data on their server if a legitimate user asks for it, and the request is justified. With this regulation, for the first time, there is a common legal basis for the entire EU with respect to deleting data on the Internet. Additionally, this regulation underlines the current importance of this topic. However, there is a technical problem with Article 17. With this article, controllers must inform also third parties about the user's deletion request. In many cases, this is technically not possible. Since the legislative authority cannot demand anything that is not achievable with reasonable steps, there is a risk that controllers might use this loophole and deny the user's legitimate request for deleting her data. This is exactly the focus of Ms. Kieselmann's dissertation. In contrast to the existing approaches, she did not design another purely technical system for deleting the data on the Internet, but she designed a system to close the loophole. She therefore developed the first interdisciplinary approach, which combines computer science and legislation. Ms. Kieselmann circumvented the hostile host problem altogether by creating a system for informing all involved parties about the users' wishes. From a technical point of view, she created a distributed notification service, which informs all controllers about deleting requests. By doing so, she elegantly combines a technical solution with the legislation. Her approach closes the identified technical problem with Article 17 and makes its practical application possible.

Kassel, November 2017 *Arno Wacker*

Abstract

After publishing data on the Internet, the data publisher loses control over it. However, there are several situations where it is desirable to revoke published information. To support this, the European Commission has elaborated the General Data Protection Regulation (GDPR), which is mandatory for all countries in the European Union and must be applied from 25 May 2018. In particular, this regulation requires that providers must delete the data on user's demand. However, the data might already have been copied by third parties. Therefore, Article 17 of the GDPR includes the regulation that the provider must also inform all affected third parties about the user's request. Hence, the providers would need to track every access, which is hard to achieve. This technical infeasibility is a gap between the legislation and the current technical possibilities. To close this gap, we analyse the possibilities for a technical realisation of data deletion and propose a solution, which is based on the combination of the technical mechanisms and the obligation to follow the legal regulations of the GDPR.

Our technical solution is a distributed and decentralized Internet-wide data revocation service (DRS). Hereby, we aim to support users to remain in control over their data even after publishing them on the Internet, and to give providers a possibility to follow the legal regulations of the Article 17 of the GDPR. With the DRS, the user is able to notify automatically and simultaneously all affected providers about her revocation request of a certain data object. Thus, we implicitly provide the notification of third parties about the user's request. Hereby, we propose two approaches how to distribute the revocation notification from users to providers: (1) to *push* the revocation requests to providers or (2) providers *pull* the revocation requests by themselves.

Elaborating the DRS, protection of user's privacy and anonymity on the Internet is our central objective. To achieve this with a distributed and decentralized system, we supplement the DRS with a new privacy-aware fine-grained access control. We evaluate the DRS and its access control in terms of attacker resilience, computational and memory effort both through theoretical analysis and through simulations.

"Lady Luck favors the one who tries!"

Law of Serendipity

Acknowledgements

This dissertation marks the close of a very important chapter in my professional as well as personal life. I take this opportunity to express my deepest appreciation to all those who provided me the possibility to complete my dissertation for a doctor's degree.

First of all, I owe my deepest gratitude to my doctoral adviser, Mr. Prof. Dr. Arno Wacker, for his supervision and direction as well as for his perfect methodological and conceptual support during the entire period of my work on the present dissertation. His advices and recommendations were an invaluable contribution towards the attainment of our shared goal.

In addition, I would like to thank Mrs. Dr. Silke Jandt for her professional conversations and remarks that have helped me discover new aspects and approaches on the way to the successful completion of my dissertation. She offered me an idea which I then transformed into the subject of my research.

A good working environment is particularly important for the successful and fruitful work, so I am also thankful to my colleagues, Mrs. Anette Kaiser-Brüger, Mr. Henner Heck, Mr. Nils Kopal, and Mrs. Inken Poßner, who I have worked with, especially for a good teamwork, a cooperation, and a friendly atmosphere in the team.

I extend particular thanks to my family, and especially to my daughter Johanna, who was my greatest stimulus and very significant party also involved in the working process. I express to her my deepest love and I am very grateful for her patience and understanding throughout the time of dissertation writing.

I feel also obliged to say my warm thanks to everyone else who has helped and supported me all this time and, for whatever reason, probably not mentioned by name here.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

"Knowing is not enough; we must apply.
Being willing is not enough; we must do."
Leonardo da Vinci

A great variety of different Internet services are offered in this day and age. As a result, many users of the Internet, knowingly or unknowingly, increasingly share their personal data. With the current nature of the Internet, this data stays on the Internet forever, since it can be copied and republished again and again. With other words, people lose control over the distribution of their data on the Internet. Until recently, most people had no problem with lack of control over the processing and dissemination of their own data once published on the Internet. For the time being, however, this practice undergoes significant changes, and Internet users start to exercise more caution about which data they share. The most crucial reason is the lately invasions of privacy by government security agencies and companies that have caused attention of the society and evoked a strong public resonance. Internet users became aware of massive surveillance and data storage programs. Furthermore, many evidences were uncovered showing that some Internet providers misuse personal user data [71].

Besides, people often make their decisions impulsively or for emotional reasons and can change their opinion and views at any time throughout their life. In doing so, everyone may want to be forgotten by the society about his or her past. Today, in the Internet era, people increasingly publish their personal content on social media networks. Due to a revised opinion, whoever shared personal data at some point in the past could feel regret about having done so. In that regard, there is a need for removing this data from the Internet in the present. For instance, a young student posted a party-picture on a social network. At that point in time, she had no worries regarding her public appearance. Years later, after finishing all studies, the former student applies for a top position at a famous company. While her opinion and attitude changed over time, the published picture did not and can still be found. Even worse, this may influence the decision of the personnel manager. To prevent this, she would like to completely remove this picture from the Internet. However, the Internet is a distributed system. Therefore, it is almost impossible to detect all locations and servers where the picture has already been copied and stored – the user has lost control over her own data. There are various others imaginable situations and reasons where it would be preferably that the Internet could also be able to "forget" data. In general, Internet users would like to retain control over their own data even after publishing it on the Internet. The Court of Justice of the European Union fully shared this view and declared this in its

judgment of May 13, 2014 (Case C-131/12) [68]. The judgment states that natural persons have the right to demand removing their data from the results of search engines. Since this judgment was rendered, Google faces the need to handle thousands of user requests for data deletion every day. Although the removal of links from search results does not actually delete the data, people use this possibility for lack of a better alternative.

Deleting data on the Internet is not only a legal but also a technical challenge. The legal provision is aimed not only to protect and to maintain individuals' rights to privacy, but also to take into account the public interests. Although individuals may be interested in public oblivion of their certain information, the interests of the society, on the contrary, may lie in storage of this information. To comprehensively protect the privacy of Internet users, the European Commission has developed the General Data Protection Regulation (GDPR) [20], which is mandatory for the entire European Union (EU). This regulation came into force on 4 May 2016 and is applicable from 25 May 2018. It is also binding on all foreign companies with subsidiaries doing business in the EU. The GDPR stipulated that *"Regulation applies to the processing of personal data in the context of the activities of an establishment of a controller or a processor in the Union, regardless of whether the processing takes place in the Union or not."* [20]. The key objective of this regulation is targeted to legalize the right of individuals to deletion of their personal data on the Internet. Since the issuance of the first draft of the GDPR in 2012 [19], experts had the opportunity to analyse the technical feasibility before the final adoption of the Regulation. For example, in [27], the authors considered the technical solutions to ensure the above mentioned right in information systems and specified technical challenges. In [45], we also discussed problems for service providers with respect to this draft. Specifically, in accordance with Article 17 of the GDPR, which introduces the notion of so-called "right to erasure", service providers shall *"take 'reasonable steps', including technical measures, to inform third parties of the fact the individual wants the data to be deleted"*. Thus, the primary challenge for service providers is to track every access to be able to adequately inform any and all third parties. However, there are currently no available technical solutions in order to meet this legal requirement with reasonable effort. As a result, this makes Article 17 practically inapplicable because the service providers are entitled to refuse implementation of legal requirements on grounds of their technical unachievability.

Both in general and especially within the requirements of Article 17 of the GDPR, finding a technical solution for deleting data on the Internet is a challenging task. From the technical point of view, there are two main challenges in the way of achieving secure data deletion on the Internet – locating all copies and deleting those copies. Due to the open nature of the Internet and the variety of its services, technologies for deleting data in closed systems cannot by applied to the Internet. Instead, innovative technologies should be designed, which are specifically tailored towards deleting on the Internet to protect user's privacy and confidentiality of identifiable personal information. In this doctoral dissertation, we analyse the feasibility for a technical implementation of data deletion and propose a new approach, which combines the technical concept with the need to meet the legal requirements of the GDPR. The solution proposed in this research can be termed as an "Internet service" based on a decentralized network architecture and particularly designed to protect user's privacy. The research area covered by this doctoral dissertation is depicted in Figure 1.1.

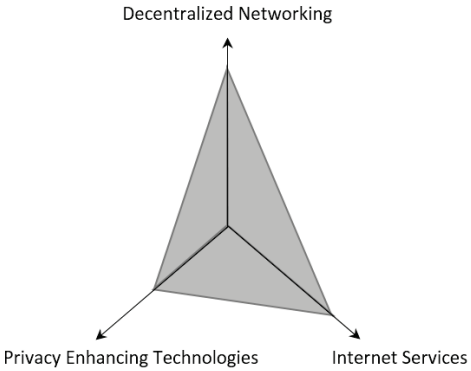

FIGURE 1.1: Area of Research

1.1 Deleting Data on the Internet

In general, existing technical approaches for controlling published data objects on the Internet can be divided in two classes: (1) deleting or blocking and (2) hiding the copies. With the approaches of the first class, data objects are no longer available after their deleting or disabling. That can be achieved by actual deleting data objects on the server or by disabling them. In both cases, on the one hand, it is necessary to request the provider for deleting a certain data directly and individually. On the other hand, the provider should follow the deleting request. Technically, deleting a data object means that you cannot recover this data object after its deletion. In contrast, blocking a data objects connotes disabling this data object for a time, i.e., it is not accessible during the blocking time. However, after revoking the blocking, the data object is accessible again. Technically, blocking a data object forever corresponds to its deletion. Another possibility to achieve deleting is to encrypt the data object before publishing it on the Internet and making the encryption key available for a certain time or only for authorized users. However, as long as the encryption key is available, users can access the data object, copy it, and spread it on the Internet. With the second class of approaches, i.e., hiding the copies, search engines do not show certain links in their results. Since nowadays most information is found by using search engines, hiding (filtering) links causes that linked data objects cannot be found. However, these data objects are still on the Internet and can be accessed by other means, e.g., directly through their URL or with other search engines. Furthermore, search engine providers need to manually manage these filtering lists, as each user request must be verified for its legitimacy before being applied.

Even in a closed system, it is not trivial to completely delete a data object as copies are created in any system, e.g., temporary or for backups. To delete the data object completely, all those copies must be located. This may be associated with a lot of effort. On the Internet, it is all the more difficult, if not impossible, to find all copies and then deleting them on other people's computers. Due to the state of the art, deleting data on third party computers is very hard to achieve and only possible with additional hardware, e.g., with smart cards. This additional hardware would enable the execution of deleting in such a way that the owner of the computer could not intervene in this process. This problem is commonly known as hostile host problem: a program is running on a computer which is under full control of the adversary, and we assume that she is trying to

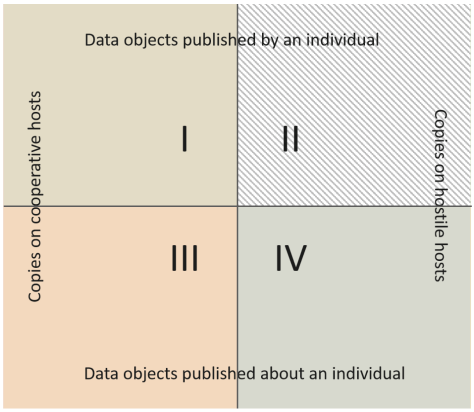

FIGURE 1.2: Classification of Published Data Objects

manipulate the execution of the program [41]. Approaches with the goal to actually delete data on third party computers have had no success (cf. Chapter 7).

Furthermore, on the Internet, data objects can be spread fast and widely. That means that after publishing a data object its copies may be saved on numerous different servers within a short time. Nowadays, to delete a certain data object on the Internet, we first have to find all its copies (e.g., with search engines) and then request each provider of a copy to delete it (e.g., by contacting provider's support). However, with this manual procedure, we are not able to capture all copies. Moreover, this approach does not scale for any number of deleting requests. We need an automatized solution to design deleting data objects on the Internet scalable.

In this dissertation, a data object is an information published on the Internet, e.g., a picture, a web page, or a PDF file. We differentiate between data objects published *by an individual herself* and data objects published *about an individual by a third party*. Moreover, we divide data objects according to their location: saved on cooperative or on uncooperative servers. With cooperative servers, we assume that the provider follows user's deleting requests. Respectively, providers of uncooperative servers ignore user's deleting requests. Hence, we derive four different classes for data objects published on the Internet (cf. Figure 1.2):

- Class I: data objects published by an individual and located on cooperative servers;
- Class II: data objects published about an individual by a third party and located on cooperative servers;
- Class III: data objects published by an individual and located on uncooperative servers;
- Class IV: data objects published about an individual by a third party and located on uncooperative servers.

In our solution for deleting data on the Internet, we consider data objects of Class I, i.e., we present an approach for the case where an individual publishes data objects on the Internet self-willed and consciously. For that, we followed our idea to create a possibility to notify the

providers about individual's revocation requests automatically and simultaneously. Figuratively, a person pushes the button "revoke my data object", and, subsequently, each affected provider is automatically informed about the revocation request. By realizing that idea, we, on the one hand, help the individuals to control their published data objects. On the other hand, we support the providers to identify data objects that must be revoked. This in turn enables the provider to implement the Article 17 of the GDPR. With this service, there is no need for the provider to track every data access in order to potentially inform third parties about the revocation requests. If most of the providers obey the individual's revocation request, they prevent the dissemination of these revoked data objects with the effect that the data objects eventually get extinct.

We realize this idea as an Internet service, which we call Data Revocation Service (DRS). To avoid a single point of failure or a single point of control and to design the service scalable, we use a distributed hash table (DHT) based on a structured Peer-to-Peer network (P2P) to implement DRS. In this DHT, individuals store notifications for service providers. An individual can anytime update the notification for any of her data objects. In an ordinary DHT, everybody can read and write to it, i.e., each user of our service would be able to alter notifications of other users. Furthermore, this circumstance would open the door for censorship. To guarantee that only authorized users must be able to update the notifications, we extended the ordinary DHT by access control mechanisms. Additionally, we design the access control mechanisms in such a way that they do not reveal any information about users to ensure their privacy.

1.2 Contribution

In this dissertation, we present a distributed and decentralized Internet-wide data revocation service. With this service, on the one hand, the service providers can automatically verify whether there are removal requests. On the other hand, the user can notify all service providers that her data objects should be deleted. Thus, our solution offers both parties a technical instrument to support Article 17 of the GDPR. Since we use a DHT for storing information for the service providers, we must secure the access control to the DHT entries. For the DRS, we must protect the write access to avoid malicious manipulating of revocation notifications as well as the read access to protect sensitive data. However, anyone should be allowed to read the revocation notifications. Furthermore, to allow the controlled access also for closed groups, we need a mechanism to grant permissions fine-granularly. Therefore, we generalize the access control in such a way that it can be used for protecting read and/or write operations in any DHT for arbitrary scenarios. Thus, the two core contributions of this dissertation are the **design of the DRS** and the **design of the access control for a DHT**. More specifically, all contributions of this work are as follows:

- We derive and analyse the requirements for a system architecture for deleting data objects on the Internet.
- We analyse different possibilities to realize such a system and identify two main principles to implement it: (1) to *push* the revocation requests to providers or (2) providers *pull* the revocation requests by themselves.
- We elaborate the system design for the DRS. Hereby, the resulting system architecture is applicable for both implementation approaches, i.e., push and pull.
- We evaluate both approaches and compare them with each other to illustrate their advantages and disadvantages.

- We develop the DRS as a new possibility for individuals to simultaneously inform all providers whether her data objects can still be used or must be deleted.
- With the design of the DRS, we support the users in privacy protection of their personally identifiable information.
- Since the DRS is built on a distributed and decentralized network structure, we analyse the design for an access control in a DHT.
- Based on that analysis, we extend the ordinary DHT with a privacy-aware fine-grained access control scheme. With our approach, the read or write access to each index in the DHT can be regulated individually. Additionally, our approach also allows the delegation of access rights to other users.
- We strengthen our access control with resilience against malicious peers in a P2P network. That allows a reliable regulation of read or write access to any DHT item even under the assumption that a certain number of peers is subverted by an attacker.
- We develop three novel access control mechanisms for a distributed system with the analogue reliability as with access control in closed systems by using existing access control mechanisms based on public-key cryptography, zero-knowledge-proofs, and cryptographic hashes.

We complement these contributions with a detailed performance evaluation of both the DRS as a whole system and of our access control scheme in particular.

1.3 Structure of this Dissertation

This dissertation is organized as follows: In Chapter 2, we present our system model, the requirements for a technical solution for data revocation on the Internet, and the underlying security goals. Based on our main idea of a revocation service, we propose two different approaches to realize that service, the one is pull-based and another is push-based. After considering the design rationale for these approaches, we develop the system architecture for the revocation service. Hereby, we identify three main system components, i.e., the unique identifier ID to identify protected data objects, the DRS, and the access control scheme. In Chapters 3–6, we elaborate these components in detail. Specifically, in Chapter 3, we first consider possibilities to identify data objects that are protected with the DRS. Next, we present our first core contribution. As we rely on a distributed and decentralized network to realize the DRS, we consider three different ways to realize the access control in such a system to enforce protection against impermissible revocations or manipulations. We describe and evaluate them in Chapter 4. After that, in Chapter 5, we analyse a specific DHT protocol, namely Kademlia, to verify its suitability for the DRS. In Chapter 6, we describe our second core contribution. Here, we consider the DRS by applying separately both the push and pull approaches. Thereby, we present different methods to implement these approaches. Furthermore, we evaluate the DRS regarding its security properties, performance and usability, and compare the two approaches with each other. In Chapter 7, we describe other existing approaches for removing previously published data from the Internet and compare them to our solution. Additionally, we discuss existing access control mechanisms for P2P networks and compare them to our access control. Finally, we conclude this dissertation with a summary and provide an outlook on future research in Chapter 8.

2

System Design

Our goal is to provide Internet users a technical possibility to delete their previously published data from the Internet and to support providers to follow the Article 17 of the GDPR. Specifically, we aim to design a system that allows Internet users to notify all providers simultaneously about deleting requests for their data objects. Providers that obey the GDPR must follow user's demand and delete those data objects. To achieve our goal, we have considered different possible methods to realize such a notification system. In consequence, we identified two main implementation principles – to push notifications to providers, or providers pull the notifications by themselves. Hereby, the system can be imagined as a gathering point, like a post office, where the users deposit their notifications. Afterwards, notifications can be delivered to providers by the system (i.e., push notifications), or providers can fetch them from the system by themselves (i.e., pull notifications). Hence, the system acts as an intermediator between Internet users and service providers. We visualize our idea in Figure 2.1.

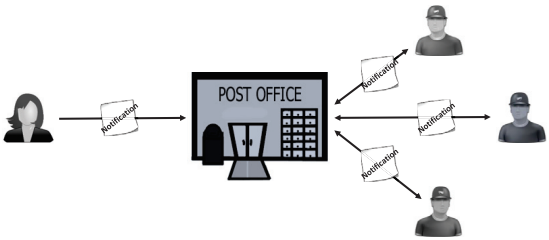

FIGURE 2.1: System as a Notification Depot

In this chapter, we present the system design to realize the data object revocation on the Internet for both the push and the pull approaches. For that, we first introduce the system model that forms the frame for our proposal in Section 2.1. After that, in Section 2.2, we define the requirements that the system must fulfil. Based on these requirements, in Section 2.3, we identify the security goals that must be considered while the system design. For both push and pull approaches, we analyse different methods to implement them and present the corresponding design rationale in Section 2.4. Finally, we showcase the system architecture for both approaches in Section 2.5.

2.1 System Model

Nowadays, arbitrary data is easily spread on the Internet. The spreading of data is effected by the fact that, on the one hand, there are Internet users willing to publish their data objects. On the other hand, there are Internet users interested in obtaining these published data objects. We refer to the first as *owners* and to the second as *users*. For publishing a data object, the owner uses one or multiple services offered by service providers on the Internet. With the usage of at least one of these services, the owner authorizes the provider to publish her data objects on the Internet. Hence, an owner is characterized by the ownership of a data object that she is going to publish on the Internet. Within our approach, other persons have no claim of ownership for this data object. We acknowledge that in reality (and also legally) the ownership of a data object is a much more complex issue. Nevertheless, this simplification allows for a clear (technical) determination of a responsible entity for a data object – which we call the owner. A *data object* can be any arbitrary information, e.g., a picture, a video file, or a document. The technical realization of the data object publication is performed by the *service provider* (in the following, abbreviated as provider). While the publication process, the provider usually saves this data object locally. Furthermore, the published data might also be replicated by other providers. Usually, different providers are under different administration controls, i.e., one provider cannot control the stored data of another. For their users, providers offer the retrieval of published data with service-specific clients, e.g., web browsers. Users are able to store the retrieved data objects locally. They also are able to alter the retrieved data objects for personal use.

Similarly to the real world, there are also legal regulations on the Internet. In particular, the new GDPR of the EU regulates the right of an owner to delete her previously published data. More specifically, the providers are required to delete data objects upon request of their owners. We, therefore, assume that all providers bound to this law within the EU act correctly and delete the data objects upon requests. Providers outside of this law's jurisdiction might not obey it. Furthermore, users and owners might act maliciously – independently of any legal regulations. For instance, some users might try to steal data objects from their owners and distribute them on the Internet. Even more, they might assume ownership of the stolen data objects.

In Figure 2.2, we summarize the participants of our system model. Depending on the scenario, an Internet user can act as an owner or as a user.

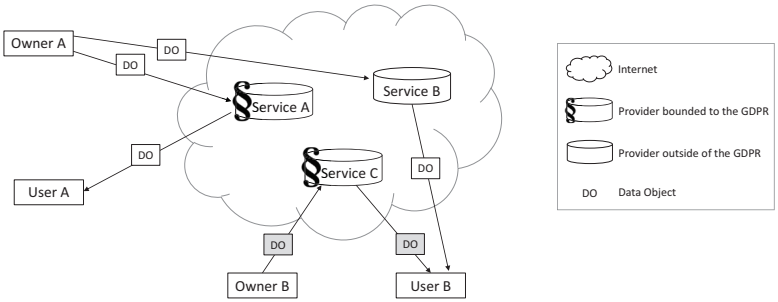

FIGURE 2.2: System Model

2.2 Requirements

With our idea as depicted in Figure 2.1 in mind, the presented system model leads to a number of requirements that must be fulfilled by a system for revoking data objects on the Internet. In the following, we describe and motivate these requirements.

Availability: The system must allow owners to delete their own data objects on the Internet at any time, i.e., the revocation system must be able to process revocation requests from owners at any time. It is acceptable that the actual revocation of a data object through the providers is performed at a later time. This time can be a fixed timespan, or whenever a user requests the respective data object in the future.

No censorship: We require that our system does not provide a way for censorship, e.g., by government authorities. This means that only the owner herself is able to initiate the revocation of her data objects – and nobody else. Thus, it should not be possible for governments, even with legal jurisdiction over the providers, to use our service for revoking data objects selectively.

Privacy: A crucial requirement is to protect the user's privacy and anonymity on the Internet. Specifically, it must be impossible to deduce the owner from the data objects with our system, i.e., our system must not introduce *new* ways to deduce the owner of a data object. If the owner is already known to the provider or can be deduced from the content of the data object, our system cannot (and is not intended to) prevent this. Furthermore, the system must not provide a way for any entity to find all data objects of a certain owner. In general, our system must not introduce *new* privacy risks for anyone.

Scalability: There are already a lot of data objects on the Internet. With every second, numerous data objects are added on the Internet worldwide [74]. Hence, a data revocation system must scale with respect to the number of data objects under its control. We assume that in the future almost all data objects on the Internet are under the control of this system. Therefore, it must cope with a huge amount of data objects.

Usability: In general, one of the most important acceptance factors of systems is the usability for all participants. Hence, we require that our system causes no changes for any user accessing a service on the Internet. Additionally, the burden for owners must be kept to a bare minimum when publishing new data objects. The same must be true for the provider with respect to software upgrades and maintenance. Finally, also the system itself should work with a minimum of resources, financial as well as human. In ideal case, the system works out-of-the-box with zero configuration.

2.3 Security Goals

It is commonly agreed that security for any system must be considered during the design of a system and cannot be added later [47]. To provide a reliable system, we have to identify the security goals before starting to design the system. The defined security goals are of great importance in the selection of methods for the realization of certain functions in the system. Additionally, we use them in the evaluation phase to verify whether the completed system is reliable regarding these goals. From the above presented requirements (cf. Section 2.2), we derive the following security goals:

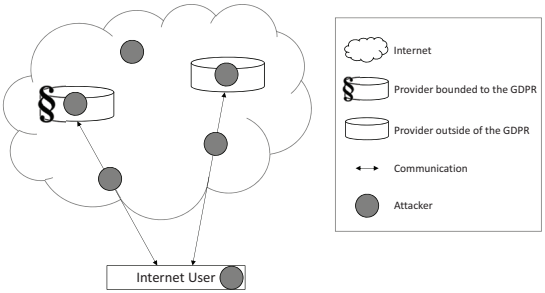

FIGURE 2.3: Attacker Model

1. **Secure against privacy violations**: It should be impossible to gain information about an owner only by using the system.
2. **Secure against malicious revocations**: It should be impossible that someone other than the owner can revoke her data objects.
3. **Secure against hindered revocations**: It should be impossible that someone could hinder the owner to revoke her data objects.

To achieve the security goals, we must identify which attacks might threaten the system. For that, we analyse what kind of attackers are possible with the given system model and what their goals might be. We consider that any participant of the system might act as an attacker. Additionally, there are attackers placed on the communication channels between the system participants. Furthermore, we consider that authorities, e.g., secret services or governments, use our system for their purposes that might contradict with our goal to underpin owner's privacy, e.g., for surveillance or censorship. Accordingly, the attackers have different abilities depending on the resources that are available to them for achieving their goals. Further, attackers might consolidate to achieve own goals. For constructing the attacker model, we simplified the system model from Figure 2.2 by merging owners and users to one group, i.e., Internet users, since owners and users are similar regarding their resources and only differ in their relation to a certain data object. We depict the resulting attacker model in Figure 2.3.

The main goal of our approach is to prevent the dissemination of a data object *after* the owner has notified the providers about her request to delete this data object. With our approach, we do not prevent copying and republishing data objects by Internet users *before* owners have initiated their deleting. Therefore, we consider only attacks possible after the owner triggers the revocation of her data object.

Contrary to our security goals, the goals of attackers are (1) to violate the owner's privacy, (2) to censor data objects protected by our system, (3) to deny deleting and disseminate data objects despite revocation notifications by owners. We classify the attackers based on these goals and by regarding attackers' abilities as follows:

- *The passive eavesdropping attacker (**Eavesdropper**)*: This attacker collects knowledge about owners and their data objects protected with our system. Her intention is to eavesdrop data transfer between a certain owner and the system. She might also aim to eavesdrop as many as possible interactions between arbitrary owners and the system. By collecting this information, the attacker is able to sort data objects by owners. Further, this attacker eavesdrops the communication between the system and the providers to analyse the spreading of certain data objects over the Internet. The ability of this attacker is the interception of the communication between the owners and the system, or between the providers and the system. Her strength is proportional to the area available for eavesdropping, i.e., the larger this area the stronger the attacker. For instance, a weak attacker is able to only eavesdrop the communication between an owner and the system. The intermediate attacker is able to eavesdrop which providers interact with the system. In contrast, the strong attacker can eavesdrop the whole communication with the system.

- *The active intruding attacker (**Censor**)*: The attacker's goal is to gain the ownership of data objects to be able to delete them on the Internet. Her particular aims might be to delete (1) a specific data object, (2) any data objects of a certain owner, or (3) as many as possible data objects of all owners. To achieve her goal, the Censor is able to eavesdrop the communication with the system, to alter eavesdropped information and inject manipulated messages into the communication within the system. The Eavesdropper and the Censor have in common that they eavesdrop the communication with the system. In contrast to the Eavesdropper, the Censor uses the intercepted information to revoke data objects instead of the owner, i.e., she is able to manipulate owner's notification for providers and inject it into the system. Hence, this attacker class subsumes the passive eavesdropping attacker class.

- *The active malicious attacker (**Denier**)*: The goal of this attacker is to disturb the functionality of the system in such a way that it cannot be supportive in deleting data objects on the Internet as intended. Her particular aims might be to deny deleting of (1) a particular data object, (2) any data objects of a certain owner, or (3) as many as possible data objects of arbitrary owners. The individual attackers of this class have a common goal, however, they differ in their abilities. Hence, regarding the resources, we divide attackers of this class in two types. The first attacker type, namely the *Rough Denier*, is capable to hinder the communication between the system and owners or providers by attacking the system infrastructure to prevent that providers can be notified about deleting requests. The second type, namely the *Sneaky Denier*, has the same ability as the Censor. Hence, she is able to drop deleting notifications by manipulating the communication between an owner (or a provider) and the system. The abilities of both types make an attacker of this class a powerful denial-of-service attacker.

In Figure 2.4, we present an overview of the attacker classes with their goals and abilities. While each of the attacker types is able to eavesdrop the communication, they differ in their goal and additional resources. These additional resources define the attacker's power. For instance, by the ability to impersonate one owner, the attacker can only manipulate this owner's data objects. In contrast, a malicious provider is able to influence deleting notifications for all data objects under her control.

	Eavesdropper	Censor	Denier	
			Rough	*Sneaky*
Goal	Privacy Violation	Censorship	Denial-of-Service	
Ability	Eavesdropping	Eavesdropping/Injection	Interruption	Eavesdropping/Injection

FIGURE 2.4: Attacker Classification

2.4 Design Rationale

Due to the hostile host problem [41], the solutions to directly delete data on third party computers are impractical, since we must rely on additional tamper resistant hardware on those computers. However, according to our usability requirement, we aim that the Internet users do not need additional hardware or software for requesting data protected by a system based on our approach. Therefore, we desist from deleting directly on foreign computers and choose a novel way to achieve our goal – we combine the information technology with the duty to follow the law. With the GDPR, we have a legal instrument with respect to publishing providers: they are bound by law to adhere to the owners' deleting requests. Under this assumption, deleting a data object is no longer a challenge. However, the providers are additionally required to inform all other parties who retrieved the owner's data object in the past. Due to the nature of the Internet, it is usually very hard for the providers to identify all other parties who accessed the corresponding data object. Hence, the main challenge for applying the GDPR is of how to locate all copies of a data object, i.e., to inform all third parties about the owner's deleting request. Based on our idea to realize deleting data on the Internet with a "notification depot", we derive two approaches to achieve technically that notifications reach the affected providers, namely the *push approach* and the *pull approach*. They differ in the way how the notifications are distributed to the providers.

With the push approach, each time when an owner demands a revocation of her data object with the system, the revocation notification is pushed to the provider. According to our analogy with a post office, a postman delivers the notification to the provider, i.e., the provider *receives* the notification. For that, the postman has to know the addresses of the affected providers. Having the list with addresses of these providers, the notification delivery can be performed by the owner or by the system. Hence, we have two different ways to push notifications, i.e., the *owner push method* and the *system push method*.

In contrast, with the pull approach, the provider *fetches* the notification by herself before delivering a requested data object to the users. By analogy with the post office, the provider regularly contacts the post office to inform herself whether there is a revocation notification for that data object. Also with pull, we can use two different methods to implement the notification. One method would be to store with the system a status message which gives the provider the information whether the protected data object can be delivered to the users or not. Accordingly, we call this the *status pull method*. With the second method, the owner encrypts her data object before publishing it and stores the corresponding encryption key with the system. Afterwards, she publishes the encrypted data objects with an Internet service. To revoke her data objects, she deletes the encryption key from the system. Consequently, the provider requests the encryption key from the system to decrypt the data object before delivering it to her users. If there is no encryption key available, we consider the data object as revoked, since it cannot be decrypted anymore. We call this the *key pull method*. Additionally, with this method, the owner is able to build closed groups and distribute the encryption key to the authorized users with the DRS.

However, as long as the encryption key is available, malicious users might republish the corresponding data object even after its revocation. The challenging part with this approach is to prevent the authorized users from caching the key. The key pull method essentially resembles the currently existing approaches, e.g., [12], [18], or [35] (cf. Section 7.2).

Overall, we identified two main approaches to realize the revocation of data objects on the Internet. For each of them, we distinguish two methods to implement the approach. We accentuate the main difference between the push and pull approaches as well as of their variants in Figure 2.5. In this dissertation, we pursue both approaches to be able to compare them and to analyse which approach suits best to our requirements and security goals.

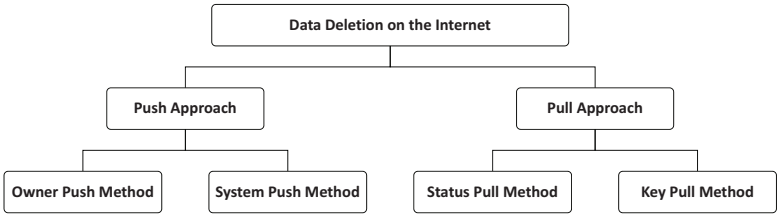

FIGURE 2.5: Implementation Approaches of Data Deletion on the Internet

For both approaches, we propose to realize the system as an Internet-wide data revocation service. As already mentioned, we call this service the Data Revocation Service (DRS). The owner uses this service for submitting her deletion request to the affected service providers. For any of the four revocation methods (cf. Figure 2.5), the DRS must resemble a sort of a database that is available at any time to manage the owner requests. This database contains the information needed to notify the providers about owner's revocation of a certain data object. Regarding our goals of avoiding censorship and protecting owner's privacy, the DRS must not have a central authority. By doing so, no single company has exclusive control, i.e., the ability to revoke arbitrary data objects wilfully. This could be achieved by using a hierarchical structure, similar to the well-known DNS [62] for the Internet. However, with a hierarchical structure, a compromised server has negative impact on the entire system. Another drawback of this approach is that such a system usually has to be set up and maintained manually by administrators. That contradicts our usability requirement. Therefore, we rely on a semi-distributed database in terms of a distributed and decentralized peer-to-peer (P2P) architecture. In general, the P2P network participants (i.e., peers) have equal rights and share their resources with each other, while they interact within the network both as clients and as servers. Further, all tasks or data are distributed to all peers. Therefore, with a P2P network structure, we can avoid both a central authority and a single point of failure.

Another argument for a P2P network is the ability of such networks to scale. Assuming our service will be widely used, we have to cope with a multitude of simultaneous requests. The owners will produce a lot of requests for storing and updating their notifications. Since each single provider offers many data objects and supplies a number of Internet users, the providers will also produce many requests with the DRS.

With a P2P network, we additionally are able to achieve the required usability. From existing products as the anonymizing network Tor [24] or the digital payment system Bitcoin [66], we know that a system based on a P2P architecture can be implemented self-organized. Hence,

its users (regardless of whether owners or providers in our case) do not have to configure the corresponding software for joining the network – to participate in the system, a user needs only to start the software on her computer.

Usually, an owner publishes multiple data objects with different Internet services. Besides that, providers generally handle many data objects from various sources. Therefore, if an owner wants to revoke a certain data object, she must be able to clearly point to this data object after its publishing on the Internet. That means we need to identify this data object afterwards. To achieve this, we assign a unique identifier (ID) to each protected data object. We use this identifier as a unique reference to this data object within the DRS.

With the DRS, we store the information needed either to allow delivering a certain data object or to execute its revocation on owner's demand. To refer to this information regardless of what revocation method is currently considered, we give it a generalized label *statement*. The exact content of this *statement* depends on how the process of provider notification occurs, i.e., by the push or the pull approach. With the push approach, we store per data object a list with the contact addresses of the affected providers. We consider this list as sensitive, since its content is crucial for the successful revocation for the corresponding data object. Hence, we must regulate the access to that list to avoid its misuse, e.g., by manipulating it or by using it for censorship. More precisely, we need to control both the write access and the read access. Also for the pull approach, the stored message or the encryption key are sensitive data that must be protected against unauthorized accesses to avoid its misuse. With the status pull method, we only need to protect the write access, since every provider should be able to read the owner's message. However, with the key pull method, we need both write and read protection to ensure that only the owner is allowed to store or update the encryption key, and only authorized third parties can readout the encryption key. Conclusively, we need an access control scheme for the DRS. To protect owner's privacy, the access control must be realized in such a way that it is impossible to draw inferences how many and which data objects belong to the same owner.

For our approach, we assume that the system participants behave within the law, i.e., they do not remove the unique identifiers from protected data. Thus, our goal is not to technically enforce the usage of our service or to delete data. We rely on law enforcement to use our service and provide a mechanism for the law-abiding participants in order to follow Article 17 of the GDPR.

2.5 System Architecture

In the previous section, we reasoned the main components that a system based on our approach must consist of (1) a unique identifier for each data object to be able to identify it after its publication, (2) a distributed database for providing owner's notifications to service providers (i.e., the DRS), and (3) an access control scheme for regulating the access to the DRS. We use this system architecture for both the push and pull approaches. In this section, we present how the system components interact with each other to delete data objects on the Internet with both approaches. A detailed description and analysis of these components follows in the next chapters.

We generate for each data object protected with the DRS a new database entry. Within this entry, we store the *statement* of the corresponding data object (more details in Chapter 6). To implement the DRS distributed and decentralized, we use a Distributed Hash Table (DHT). The DHT is a convenient way to utilize the power of highly scalable P2P networks. Currently, there exist numerous ways to build such a DHT, e.g., Chord[81], CAN[73], Kademlia[60]. Although

DHTs are no real distributed databases, their functionality is sufficient for our system. Generally, DHTs offer the following two operations:

- put (key, value) - for adding a key-value pair or modifying the value of a certain key,
- get (key) - for retrieving the value for a certain key.

In the rest of this dissertation, by referring to the DHT key, we use the term *index* instead of *key* to avoid confusion with an encryption key.

With the DRS, the ID of a data object is also the index for the DHT entry where the value *statement* associated with that data object is to be stored. Specifically, a DHT entry is reserved for exactly one data object, and all information regarding this data object is stored within this entry. Consequently, we need an access control scheme which offers access protection for each single DHT entry. In general, the scalability and the architecture of DHTs are well researched. However, security issues like access control or storing confidential data in a DHT are still mostly unsolved. While the existing DHT implementations are suitable for currently existing products, there is no solution with a fine-grained access protection for put and get operations. Especially, there is no access control mechanism to regulate access to a single DHT entry (cf. Section 7.1). With a DHT, the responsibility for the entries is distributed among all participating peers. Hence, each peer is responsible for a certain subset of entries. This implies that the values of those entries could be manipulated arbitrarily by the responsible peer. In an open system, we must consider that some of the peers might act maliciously by manipulating the values under their control. Thus, to cope with up to k malicious peers, we require that there are $2k+1$ responsible peers for each DHT entry. Consequently, each DHT operation (put or get) always operates on $2k+1$ peers. To determine the correct value upon a put operation, we apply the majority voting, i.e., the decision for a correct value is made when over half of the verified $2k+1$ values are the same. By doing so, an attacker would need at least $k+1$ subverted peers to successfully alter a value. As part of this dissertation, we developed a novel access control with individual restriction for both the read and write access to a single DHT entry in consideration with up to k malicious peers (cf. Chapter 4).

Before publishing a data object protected by the DRS, the owner embeds a unique identifier into the data object and registers this identifier with the DRS. Although we use the same system architecture for both approaches, they differ in how owner's revocation demands are submitted to the affected providers with the DRS. Specifically, the registration procedure of a data object as well as the notification procedure are different for push and pull approaches. In the following, we consider these differences for both approaches separately.

Push approach

The idea of this approach is that each time when an owner demands a revocation of her data object, a notification is pushed to the affected providers. Hence, the system needs to know who of the providers offers the revoked data object. For that, we store the contact information of the affected providers as follows: Whenever the provider publishes a protected data object, she registers its publication with the DRS. During this registration, the provider adds her contact information to the *statement* item of the corresponding DHT entry. Having a list of affected providers, we can pursue two different ways to push the notifications:

- *Owner push method* – To revoke her data object, the owner requests the DRS for a list of providers offering that data object using a get operation. Then, the owner notifies these providers by herself. With this method, we must ensure that a provider actually offers the protected data object and, therefore, is allowed to add (or to update) her contact to *statement*, i.e., to perform a put operation. Furthermore, only the owner must be allowed to read the contact list.

- *System push method* – When the owner notifies the DRS about her revocation demand, the DRS determines which providers are affected and sends them the deleting request for the corresponding data object. Similar to the above method, we must protect *statement* against malicious write operations. Additionally, we must ensure that the revocation demand is performed by the owner before sending notifications to the affected providers.

Pull approach

Every provider publishing a protected data object is required to check with the DRS whether the data object can still be published or must be deleted. For that, we store with the DRS within *statement* a permission for delivering the corresponding data object. Hereby, only the owner of a data object must be able to change its *statement*. As mentioned before, we consider the status pull method or the key pull method to realize the pull approach. Following, we point out the differences between these revocation methods:

- *Status pull method* – The parameter *statement* contains a status with the information for the providers regarding whether they are allowed to deliver the corresponding data object or not. Every time, when a user requests a certain data object, the provider checks the latest status of this data object with the DRS by using the get operation. With this method, we must protect only the write operation for updating the status, since everyone is allowed to readout the status of a data object. The status can be implemented as a flag where 1 corresponds to “allowed for delivering”, and 0 to “must be deleted”. Alternatively, it can be a string, e.g., “active” conversely “revoked”, or a revocation date which specifies the time from which a data object should no longer be delivered. For the sake of consistency, we use “active” or “revoked” by referring to the status in the rest of this dissertation. If the status allows the delivering, the provider transfers it to the requesting user. Otherwise, the provider must deny its delivering and additionally delete it from the own storage. Consequently, the owner can change the status of a data object from “active” to “revoked” only once. With the status change in the opposite direction, i.e., from “revoked” to “active”, providers that have already deleted that data object from their servers would not be able to republish it without its renewed publication by the owner. To provide more flexibility with the status updating, we introduce an additional status, namely “inactive”. According to this status, the provider denies the delivering but does not delete the corresponding data object from her own storage in case the owner changes the status back to “active”.

- *Key pull method* – The owner encrypts her data object and stores the corresponding encryption key with the DRS by executing the put operation. Afterwards, whenever a user demands this data object, the provider requests the encryption key by using the get operation, decrypts the data object, and delivers it to the user. To revoke the data object, the owner deletes the encryption key by performing a put operation. Hence, we must protect the write access to prevent the misuse of the encryption key. However, to enable closed groups, we also must protect the read access to allow only the authorized users to read the key. Additionally, we must enforce that an authorized user requests the encryption

key from the DRS each time when she accesses the protected data object. This way, we achieve that she cannot decrypt the data object if the owner revoked her right to access that data object. To cope with the hostile host problem, we must use a trusted execution environment to ensure that the currently authorized user cannot store the encryption key locally for accessing the corresponding data object after the owner revoked her access right (cf. Section 6.3.2).

Depending on the approach and its particular revocation method, the overhead for providers and owners varies. In Chapter 6.5.2, we compare all four notification methods regarding their advantages and disadvantages for different scenarios.

2.6 Summary

In this chapter, we systematically considered different ways to design a system on the basis of our core idea to create an interface between owners and providers similarly to the postal service. First, we identified the system model and formulated the requirements. Next, we identified our security goals and considered which attacker types might threaten the achievement of these goals with the given system model. By building on this input, we motivated our design rationale to realize our idea and to simultaneously design the system secure against the identified attackers. Hereby, we developed two different approaches, push and pull, for notifying the providers about the revocation of data objects by their owners. Finally, we described the system architecture that consists of three basic components, i.e., the ID, the DRS, and the access control scheme. With this architecture, we are able to realize both approaches without the need to modify the architecture for one of them. However, depending on the approach, we store a different content within *statement* in the DRS, and design the submitting procedure of owner's revocation demand differently. Therefore, we have to adapt the access control scheme to the particular notification methods. These two topics, i.e., the DRS design and its access control scheme, are the focus of this dissertation. In the following chapters, we comprehensively analyse and evaluate each system component.

3

Unique Identifier

To be able to identify the data object that should be revoked on the Internet after its publication, we assign a unique identifier (ID) to each data object protected by the DRS. Hereby, we must consider that a data object published with different providers might be stored on their servers in different formats under different file names. In this chapter, we first define the requirements for an ID suitable for our revocation approach. Then, we discuss different mechanisms to realize the ID. Finally, we reason our decision for one of them, namely a random value from a large ID space.

3.1 Assumptions and Requirements

Based on Chapter 2, we assume that owners use the DRS to revoke their published data objects on demand. Consequently, we assume each owner generates an ID each time when she intends to publish a new data object on the Internet. In turn, providers use the DRS to follow owners' revocation requests. Hereby, regardless of the particular revocation approach, we assume that a provider reads the ID of a data object each time when she processes a user request, e.g., publishing a new data object or sending a requested data object to the user. From these particular assumptions and the general requirements for the DRS (cf. Section 2.2), we define the following requirements for the ID:

Uniqueness: A data object must be clearly identifiable to be found on the Internet. Hence, the ID must be unique, i.e., there are no two data objects with the same ID.

Privacy: From the ID, it should not be possible to deduce whom the corresponding data object belongs to. Specifically, the ID must not introduce *new* ways to deduce the owner of a data object.

Usability: The burden for the owner with the ID generation and its assignment to the data object must be kept to a bare minimum. Regardless of the particular revocation approach used with DRS, the ID must be publicly available to enable the identification of a data object. Hereby, the ID must be readily identifiable for each involved party, e.g., the providers. Additionally, reading the ID from the data object should work with a minimum of resources. This is important especially for the providers, as they read the ID each time a user requests a data object.

Backward compatibility: Not all data objects on the Internet are protected with the DRS. Hence, the ID should be backward compatible so that the processing of not protected data objects is not affected in any way.

3.2 ID Mechanisms

In the following, we consider two mechanisms to produce an ID, namely by hashing the corresponding data object or choosing a random number. Additionally, we discuss their advantages and disadvantages regarding our requirements from above.

3.2.1 Hash Value

With the hashing approach, we can derive the ID directly from the content of the data object. We distinguish between cryptographic and robust hashing approaches, which we describe bellow.

Cryptographic Hash Value In general, hash functions provide compression and ease of computation. The cryptographic hash functions have additionally the following properties:

- Pre-image resistance: For all outputs y, it is computationally infeasible to find an x such that $h(x) = y$;
- Second pre-image resistance: For a given x, it is computationally infeasible to find any second input x' with $x \neq x'$ such that $h(x) = h(x')$;
- Collision resistance: It is computationally infeasible to find any pair (x,x') with $x \neq x'$ such that $h(x) = h(x')$

Hence, a cryptographic hash value is unique, and, therefore, it identifies a data object uniquely. With this approach, the ID is directly intertwined with the data object and, thus, implicitly present. However, this also means that we derive a completely different ID when we change even a single bit in the data object. While this is the intended behaviour in most cases, there are situations when similar data objects should be identified with similar or even identical IDs. This applies for instance to images which might have been modified. For instance, after a user uploads an image to publish it on the Internet, the provider resizes it to offer a smaller image for mobile devices. This can cause that the owner cannot revoke her data object if she uses the cryptographic hash value of the original data object as ID in the revocation process. In this specific case, we can use a robust hash function [87] to identify all modified derivatives of an image with the same ID.

Robust Hash Value With a robust hash function, similar data objects have the same ID. For this, a data object is first reduced to its essential features, and the hash, also called fingerprint, is calculated therefrom. With other words, the details of the data object are ignored for the hashing procedure. This results in similar or even identical hash values for similar data objects. This way, the hash values are robust against operations in which the content of the corresponding data object is not changed fundamentally, e.g., scaling or colour modifying with images.

To compare data objects, the difference between two hashes is computed by using a distance measurement and setting a threshold value. For the distance computing, we can use the Hamming distance. For example, the distance of 1 indicates that the verified data objects are the

same or very similar, the value 5 indicates small differences, and the distance of 10 detects the data objects as different. The threshold value is used for determining when the verified data objects are different. Nowadays, there are robust hash functions available for video (e.g., [21]), audio (e.g., [38], and images (e.g., [87]). In the following, we exemplarily show the simplified Average Hash Algorithm [54] for computing the robust hash value of images:

1. Scale the image to 8 x 8 pixels;
2. Convert the image to grayscaling representation;
3. Compare the brightness of each pixel with that of the overall image and define the colour mean value;
4. Generate the hash value: 1 for lighter pixels, and 0 for darker pixels.

Consequently, the ID based on the robust hash values requires a database which supports similarity search requests. Additionally, the similar or identical data objects retrieved for a certain ID must be processed reliable during the revocation process. For instance, we must avoid collisions occurred with similar images of different owners. Such collisions would lead to an undemanded revocation.

3.2.2 Random Value

A simple possibility to implement a unique ID would be to take a sufficient large ID space, e.g., 512 bits, and choose for each data object a random value from this space. With such a large ID space, the probability of a collision, i.e., two different data objects with the same ID, is very low. Randomized IDs are used, for instance, for identifying peers in P2P networks. The workflow for generating an ID by using a random value is straightforward as presented in Listing 3.1.

```
1 //bitLength: specified ID length
  object owner_generateID(bitLength){
3     id = new Random(bitLength);

5     return id;
  }
```

LISTING 3.1: Generate ID

In comparison to the hashing methods presented above, this approach is not based on sophisticated algorithms and is, therefore, resource-efficient. However, with this approach, the ID is independent of the data object content. This results in the possibility to assign different IDs to the same data object. Consequently, with the DRS, two individuals might be owners of replicas of the same data object due to the different IDs. However, each of them is able to control only the replica assigned to the ID under her ownership.

3.3 ID Embedding

After the ID for a certain data object is computed, it must be embedded in this data object to allow the providers to identify protected data objects. In the following, we discuss methods to achieve this.

3.3.1 Watermark

Digital watermarks are a visible or a hidden additional information embedded in data objects to achieve specific goals, e.g., authentication, copyright protection, or data integrity. Generally, digital watermarks can be classified into robust and fragile [26]. With a robust watermark, the embedded information is retained even after the data object has been altered, e.g., format changing, analog-to-digital conversions, scaling, or clipping. In contrast, the fragile watermark is destroyed with every slightest change of the data object. This allows to verify the originality of data objects. Fragile watermarks are used, for example, with surveillance cameras to be able to use the recordings in court as evidence. Furthermore, the digital watermarks can be characterized by a number of properties and must fulfil a variety of requirements for a certain scenario. The most important properties are the following:

- Robustness: This property describes how stable the watermark is against changes in the data objects, i.e., resistance to general changes such as scaling.
- Security: Without knowing the secret key, the watermark cannot be read, modified, or destroyed without making the file itself unusable. The difference to robustness is that security refers to targeted attacks.
- Capacity: This property describes how much information can be embedded with a watermark in the data object.
- Transparency: This property describes how perceptible the watermark is.
- Performance: In practice, the embedding and reading process must be performed sufficiently fast.

For embedding watermarks into the data object, there are various approaches for different media types and carrier signals that depend on the requirements which a watermark must fulfil for a certain scenario. Hereby, the basic principles are replacing the least significant bit, rendering watermarks by pseudo noise, or changing statistical characteristics of the carrier signal [25]. Furthermore, a secret information is used to prevent unauthorized extraction of the watermark, i.e., to guarantee robustness and security. However, according to the state of the art, there is still no approach available which can provide robustness and security against any possible change.

The watermark extracting process is similar to the embedding process: The embedded watermark is detected according to the used algorithm. Additionally, if the watermark is secured against unauthorized access by using cryptography, the corresponding key is needed to decrypt the watermark after its extraction from the data object. Finally, the information provided with the watermark can be read. In our case, it would be the ID of the data object.

As presented in [37], regarding the computational complexity, the watermark processing requires in dependence of the used algorithm and software $0.77 - 351.46$ seconds for its embedding and $0.55 - 39.25$ seconds for its extracting. The authors implemented five audio watermarking algorithms in MATLAB 5.3. under Linux, and in ANSI C using Visual C++ 6.0 and Windows NT 4.0. They evaluated these algorithms on an Intel Pentium PC with 166 MHz.

FIGURE 3.1: Example of Exif Metadata in a Picture

3.3.2 Metadata

Adding the ID as meta-information (i.e., metadata) to the corresponding data object is another possibility to link the ID with the data object. The general purpose of metadata is to find data objects according to the given tags. Usually, there are universal (e.g., file name or file size) as well as type-specific (e.g., height or width with images) metadata properties available to describe the corresponding data object more detailed. Furthermore, it is possible to add custom properties. A metadata property can be, for instance, a name-value pair or a tag. Thus, we can embed the ID in the data object by extending its metadata with a new name-value pair. For instance with images, the "DRSid" could simply be another tag in the already available exchangeable image file format (Exif) data of the image [11]. In Figure 3.1, we show a snippet of Exif metadata embedded into an image. Another example are websites, where we can integrate the additional meta-tag "DRSid" in their headers, e.g., < meta name = "DRSid" value = "123456789" > (cf. Figure 3.2).

```
<html class=" js" xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <meta name="DRSid" value="123456789">
        <base href="https://www.uni-kassel.de/eecs/">
        <style type="text/css">
    </head>
    <body>
        ...
    </body>
</html>
```

FIGURE 3.2: Example of Metadata in a Web Site

Consequently, to identify a data object protected with the DRS, the provider verifies the metadata of the corresponding data object for the specified tag as exemplarily shown in Listing 3.2. For that, we assume the operations *getMetadata(do, name)* for reading and *setMetadata(do, name, value)* for writing the metadata of any data object type. Hereby, *do* is the data object whose metadata information we intend to process, i.e., to read or write, while *name* and *value* are the

parameters for the required metadata property.

```
//do: data object requested by a user
//tagName: name of the metadata property, e.g., ''DRSid''
object provider_checkForID(do, tagName){
    metadata = getMetadata(do, tagName);
    result   = false;//initial value

    if(metadata.contains(''DRSid''){
        result = true;
    }

    return result;
}
```

LISTING 3.2: Metadata Verification for the ID Tag

3.4 Random ID as Metadata

For the ID in our approach, we decided to use a random value from a large ID space, and to link it to the data object within its metadata. In the following, we compare the presented ID techniques and justify our decision by referring to the requirements presented in 3.1.

By deriving the ID directly from the content of a data object using a cryptographic hash function, we can avoid adding it as additional meta-information to the data object. Thereupon, the provider computes the cryptographic hash value by processing each user request for any data object. An issue with this approach is that a malicious user can claim the ownership of public goods on the Internet and revoke them afterwards. An attacker has this possibility especially with the status pull approach (cf. Section 2.5). For instance, she downloads an image of the national flag of Germany, computes its cryptographic hash value, and registers it with the DRS. From this point, she is the owner of this image. When she revokes it, the providers will not deliver this image to the users and, additionally, delete it from their servers. As the identification of revoked data objects is linked to the hash value, the revocation will be valid for all image replicas with the same hash value. As a consequence, the identification by the hash value does not provide the backward compatibility, because it will affect all data object independent whether they are protected with the DRS or not. Thus, a provider needs to be able to distinguish between protected and unprotected data objects. To enable this, we must add meta-information to the protected data objects. In this case, the meta-information can simply be a flag or a set of instructions that define the algorithm for deriving the ID. Then, the provider first verifies the metadata to identify whether it is protected. Only when it is protected, she computes its hash value. This way, we achieve the backward compatibility and, consequently, prevent the hostile acquisition of the ownership.

Furthermore, using a cryptographic hash value, we must prevent a different ID in cases when the data object is changed, e.g., due to the bit errors during its transfer, size modification, or format conversation. For this, we can add the ID in the meta-information. Then, the provider does not have to compute the hash value, as she can extract it from the meta-information. However, the advantage of a hash value for the ID is lost, as we still need to pre-compute the ID to add it to the meta-information. Hence, its main property becomes a disadvantage for our approach. Moreover, due to the compression with any hash function, we must cope with collisions. Although they are unlikely with cryptographic hash functions, nevertheless, we can handle the detected

collision by changing a bit of the affected data object. This will lead to a completely different hash value. However, this solution in combination with the need to add metadata can be considered as similar to random values. Therefore, we do not need to waste resources to compute the hash value. Instead, we can choose the ID randomly. Additionally, using a random value, the ID is independent of the data object content – the revocation will affect only the replicas with the certain ID. In case we detect a collision when registering a new data object, we simply generate a new random value.

Although the cryptographic hash value is not suitable to be used as ID with our revocation approach, we can use them to generate a random value for the ID as they provide a reliable distribution. In this case, we still use the notation "random value" to avoid confusion with properties of the cryptographic hash values.

Regarding the robust hash value approach, the processing of the database requests based on this approach requires a complex workflow but does not guarantee a reliable identification. This contradicts our goal of revoking a certain data object with a specific ID. Moreover, the issue with collision caused by similar data objects of different owners is with robust hashes stronger than with cryptographic hash values. We cannot simply alter some bits of the data object or apply other modifications to prevent collisions – the purpose of robust hash functions is to recognise similar data objects. Hence, by applying strong modifications that are able to prevent unmeant collisions, we eliminate the main property of robust hash functions. Therefore, we disregard this approach.

For linking the ID with the corresponding data object, we use metadata as presented in 3.3.2, because it is a resource-efficient method in comparison to the digital watermarking approach. We could use watermarking aiming that the ID cannot be removed or altered. However, the underlying algorithms require a secret information for embedding the watermarking. Accordingly, this secret is also needed for extracting the watermark. Since the ID must be public, we do not benefit from the security property of a digital watermarks. We could benefit from its robustness property, i.e., readable and stable. However, there is no approach available that guarantees the robustness against any possible changes. Moreover, the complexity of embedding and extracting digital watermarks contradicts the *Usability*-requirement. Our goal is to provide an ID that is simply to process with a minimum of resources, while any attack aiming the ID deletion are out of our scope.

```
//do: data object to be registered with the DRS
void owner_setID(do){
    //bitLenght is a system parameter for the ID length, e.g., 160 bit
    bitLenght = 160;
    //tagName is a system parameter for the ID metadata, e.g., ''DRSid''
    tagName   = ''DRSid'';

    //1)generate a random value for the ID
    id = new Random(bitLength);
    //2)assign the ID to the metadata
    setMetadata(do, tagName, id);
}
```

LISTING 3.3: ID Setting

For our approach, we exemplarily implement the generating and the assigning of an ID with the operation *setID* as presented in Listing 3.3. Again, we assume the operations *getMetadata* and *setMetadata* for any data object type. The setting of a new metadata property within our system

means that this property is stuck to the data object and also available outside our system for other applications.

3.5 Summary

With a random ID placed in the metadata of the data object, we can achieve all our requirements. By using the ID space of 160 bits, there are 2^{160} different possible IDs. Currently, it is estimated that there is over 1 exabyte (i.e., 2^{63}) of data stored online [65]. Assuming, we must provide IDs for 2^{63} single data objects. Then, according to Equation 3.1, the average number of collisions by a random choice with 2^{63} IDs in a space of 2^{160} possible values is $\approx 2,91 \cdot 10^{-11}$. Hence, even for a such exaggerated amount of data objects, the collisions are improbable, i.e., we can achieve our *Uniqueness*-requirement. Nevertheless, if a collision occurs, we simply generate a new random value.

$$\mathbb{N}(n,k) \approx \frac{k(k-1)}{2 \cdot n} \tag{3.1}$$

Furthermore, as each ID is a random value, it does not contain any information about the owner of the corresponding data object. By analysing IDs, any protected data object technically belongs to another owner. Hence, we fulfil the *Privacy*-requirement.

To determine the average time for generating a random value, we executed the Java SE 7 operation `new BigInteger (160, new Random())` one Million times on a workstation with Intel i7-4900 MQ, 2.8 GHz, 32 GB RAM. According to the results, the average time is 0.84 μs and can be considered as a negligible burden for the owner. Assigning the ID to the metadata of a data object is also a simple operation regarding resource consumption. Moreover, the ID is publicly available and uncomplicated to access, as it must be only read from the metadata. Therefore, we achieve the *Usability*-requirement.

Finally, assigning the ID to the metadata does not affect data objects that are not protected with the DRS. The only difference between protected and not protected data objects is just the one additional metadata property with the ID for the protected data objects. Not involved parties, e.g., providers that are not obliged to follow owners' revocation requests, are not hindered to process data objects in the usual way (*Backward compatibility*-requirement).

4

k-Resilient Access Control for DHT

As presented in our system design (cf. Chapter 2), we use a DHT for storing the information needed to enable the data revocation on the Internet, i.e., *statement*. With *statement*, the owner informs the providers when her data object should be deleted. Hence, we must control the access to *statement* in order to prevent its misuse, e.g., malicious revocation by an attacker.

Access control is usually defined as the combination of authentication and authorization [80]. The purpose of the authentication is to verify whether the user that accesses the system is the one she claims to be. In contrast, authorization regulates the access to various system resources. Previous work on distributed access control for DHTs has focussed mostly on authenticating DHT participants (cf. related work in Section 7.1). However, after authentication, each participant is free to access any resources in the network without further restrictions. In the literature, this is sometimes referred to as a *coarse-grained* access control.

However, such a coarse-grained access control is not sufficient for our approach, as we additionally need to separate user permissions for reading or writing each single entry. This separation provides the flexibility to use all introduced revocation methods. For instance, with the owner push method, the provider must have write access to add her contact information into *statement*. However, only the owner should be able to read *statement*. In contrast, with the system pull method, the provider needs read access for requesting the status of a certain data object, but only the owner should be able to update this status. Furthermore, we should provide the owner a possibility to authorize/deautorize certain users to revoke data objects on her behalf.

Implementing such an access control for a DHT is a challenge due to the overall architecture of a DHT. Without further modification, any peer can read or write any DHT value via the operations *put* and *get*. Furthermore, as the responsibility for the set of all indexes is distributed among all participating peers, they have full access to entries under their own control. This means that a peer could arbitrarily manipulate values under its control. We must consider that some of the peers might act maliciously by manipulating these values. Hence, we need an access control that is also able to tolerate a number k of malicious DHT participants without compromising data item security.

Hereby, we cannot rely on classical approaches used in centralized systems, e.g., the owner authentication with username and password. In this case, each user who wants to publish protected data objects must first register with the DRS. However, this would be a contradiction to the privacy requirement with the DRS as it identifies the owner of a data object. Even if we use

pseudonyms as usernames, the DRS would still be able to identify all data objects of the same user. Therefore, this approach is not a suitable access control mechanism for the DRS. Another possibility is the usage of shared secrets. In this case, the owner stores a different shared secret for each data object registered with the DRS. For all subsequent DHT operations regarding a specific data object, the DRS needs to verify whether the user provides the correct shared secret. Instead of storing the shared secret itself, it would be sufficient to store its cryptographic hash value – similar to storing a password when using authentication with username and password. However, with this approach malicious peers have access to the shared secret and could steal the ownership of this data object. Those peers could impersonate the owner and change *statement* of the corresponding data object at will.

Although there exist approaches to implement an access control scheme for P2P networks (cf. Section 7.1), currently available DHTs allow arbitrary clients to access any data item stored in the DHT. Consequently, if only one participant is subverted by an attacker, the whole system fails. Therefore, for the DRS, we need an access control which combines the following features:

- individual restrictions for both the read and write access to a single DHT entry,
- revocation of write or read access for a certain user,
- delegation of administration for write and read access,
- safeguards against malicious peers.

To provide an access control for a generic DHT with all these features, we extend the coarse-grained access control with the ability to manage individual access rights for each stored data item by separating between read, write, and administration rights per data item and participant. We call this *fine-grained* access control. In addition, our approach is able to tolerate a number k of subverted DHT participants without compromising data item security. We call this fine-grained *k-resilient* access control (k-rAC). In the following, we describe our approach in detail. For that, we first discuss our system model and provide an overview of our approach in Section 4.3. After that, we present the access control scheme k-rAC in Section 4.4. Finally, we analyse its performance in different scenarios in Section 4.5.

4.1 System Model

Our system is a P2P-based network forming a DHT based on any of the existing approaches, e.g., Kademlia [60]. This DHT can cope with churn, scalability, availability, persistence, consistence and routing. With other words, we do not propose a new DHT but an extension to an arbitrary existing one. We assume that the DHT already implements a coarse-grained AC scheme like [67], i.e., an authenticated participant has full access, anybody else has no access. The DHT space comprises a certain number of *DHT entries*. Each DHT entry is stored on at least one *responsible peer*. A DHT entry consists of an *index* and a *value*. The index uniquely identifies an entry. The value contains the data that the user wants to store in the DHT. A DHT entry may contain additional *metadata* about the data value, e.g., its owner.

While the overview of the whole system model is given in Figure 4.1, we describe the participants, the DHT operations, and the attacker model in the following sections.

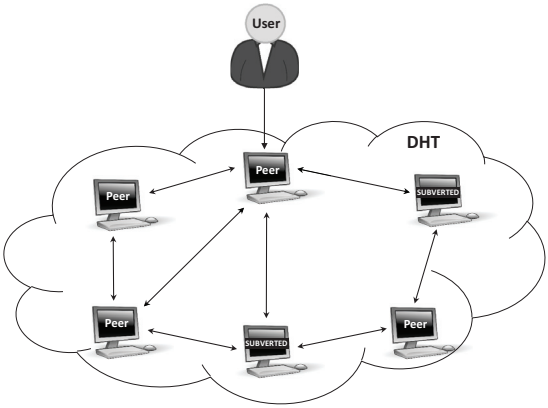

FIGURE 4.1: System Model

4.1.1 Participants

We distinguish between two different types of participants: peers and users. A *peer* models the software that implements part of the DHT and provides the two following API operations:

- *put (index, value)* for storing a DHT entry, i.e., write a value;
- *get (index)* for retrieving the value for a certain index, i.e., read a value.

Through these API operations, it is possible to read or write any DHT entry. Peers interact with other peers via messages over an arbitrary network, e.g., the Internet. Communication between peers is assumed to be secure, i.e., confidentiality, integrity, and authenticity of the messages is ensured. This is achieved by the coarse-grained access control scheme guaranteeing that only authenticated peers can participate in the network. Each peer has a unique ID determining its position in the DHT space. We assume that the ID is determined by the system, and the peer cannot influence it. This can be achieved by, e.g., hashing the IP address.

A *user* models the application software that uses the DHT. It is executed on a different device than the peers. The user can access one or more arbitrary peers to store or retrieve values (cf. Figure 4.1). For that, she accesses the DHT through the peer API, i.e., by calling one of the two DHT operations *put* or *get* on a peer. To do so, the user sends a message to a *requesting peer*, which represents the user's entry point to the DHT. The requesting peer routes this message through the network – via multiple *forwarding peers* – until it reaches a *responsible peer*. A responsible peer holds a copy of the value and is able to execute the requested action on it. Similar to communication between peers, we assume secure communication channels between users and peers, e.g., based on SSL/TLS.

4.1.2 DHT Operations

For the realisation of storing and retrieving of values in a DHT, we assume some general operations with similar functionality in any particular DHT implementation. Accordingly, there is an

operation for sending a message to the peer responsible for a specific DHT index – we call this operation *send_p2p(event, index, value)*. For this operation, the first parameter *event* instructs the responsible peer which action should be performed, i.e., *store* or *retrieve* a value for the given index. The second parameter, i.e., *index*, is the index for the DHT. It depends on the specific implementation of the DHT how this index is mapped and routed to the responsible peer. For the rest of this work, we assume that the operation *send_p2p* always delivers the message to the peer responsible for the given DHT index. The last parameter *value* is an arbitrary data object. The return value of the operation *send_p2p* depends on its parameter *event*: For *store*, the return value is an acknowledge message to signal whether the value storing was successful (*ack*) or not (*nack*). For *retrieve*, the return value is the value stored in the entry under the requested index.

For any asynchronous event like receiving a message, there is a so-called *event operation* to handle it. Accordingly, whenever the responsible peer receives a message, it executes the operation *on_receive_store* for a store message, or *on_receive_retrieve* for a retrieve message. The parameters for the event operations are *sender* (i.e., from which peer the message was received), and the same parameters which are sent within the message for the operation *send_p2p*, i.e., *index* and *value*. Locally, each peer maintains a variable *localstore* where all values the peer is responsible for are held. It can be implemented as a dictionary or a hash table indexed by the DHT index.

To send the result of the event operation to the requesting peer, the responsible peer uses another fundamental operation, namely *send_direct(receiver, 'type_reply', index, result)*. In contrast to *send_p2p*, this operation requires the destination peer *receiver* and the type of the reply *type_reply* as the additional parameters, i.e., the response to the store or retrieve event is send directly to the requesting peer. The operation *send_direct* has no return value. Here, we assume that this operation is always executed with success. In the reality, we would verify it. Finally, the requesting peer returns the result from the responsible peer to the user via the operation *response('type_reply', index, result)*, where the parameter *type_reply* indicates the reply to the messages *store* or *retrieve*.

In the following, we specify the abovementioned DHT operations with pseudocode. Hereby, we use their simplest form without any error checking or validation. Additionally, for reasons of clarity and comprehensibility, we omit the waiting routine needed in the interval between sending the message to the responsible peer and receiving its response. In the reality, we would additionally use a monitor which regulates the further execution until a result is available, e.g., the requested value or a timeout notification. Furthermore, we use the notation of the operation presented in a listing to denote who is its executor, i.e., *executor_operation_name*. Accordingly, in Listing 4.1, the put operation is executed by a user, and it represents a wrapper for the operation *send_p2p('store', index, value)*. With this wrapper, the user initiates the storing routine: She sends the message *store* to the requesting peer. In turn, the requesting peer forwards the received message to the responsible peer according to the protocol of the used DHT.

```
//index: unique identifier, e.g., 160 bit value
//value: arbitrary data object
object user_put(index, value)
{
    //send the value to the responsible peer and
    //wait for the acknowledgment message from the responsible peer
    result = requestingPeer_send_p2p('store', index, value);
    return result;
}
```

LISTING 4.1: DHT API Operation *put*

When the responsible peer receives a message with the message *store*, it executes the operation *on_receive_store* for the provided DHT index as shown in Listing 4.2. As a result, this peer stores the received value under the given index in its local store. To signal to the requesting peer the result of the storing operation, the responsible peer sends a reply message with an *ack*-signal in case of a successful storing, and a *nack* when an error hindered storing the provided value. For sending the result, it uses the operation *send_direct*.

```
//sender: destination address of the requesting peer
//index: unique identifier, e.g., 160 bit value
//value: arbitrary data object
void responsiblePeer_on_receive_store(sender, index, value)
{
    //storing routine with localstore, e.g., HashMap
    try {
        localstore.put(index, value);
        result = true;//ack for success
    }
    catch (Exception ex){
        result = false;//nack for failure
    }
    //signal the result to the requesting peer
    send_direct(sender, 'store_reply', index, result);
}
```

LISTING 4.2: DHT Operation on Receiving a Store Message

In Listing 4.3, we present a simplified form of the get message. Similar as with *put*, the user initiates the retrieving routine by executing the API operation *get* on the requesting peer. Then, this peer sends the message *retrieve* via the forwarding peers to the responsible peer. Receiving this message, the responsible peer executes the operation *on_receive_retrieve* to process the routine for loading the requested value from the local store and sending it to the requested peer. For sending the value, it uses the operation *send_direct*. We summarize this routine in Listing 4.4.

```
//index: unique identifier, e.g., 160 bit value
object user_get(index)
{
    //retrieve the value from the responsible peer and
    //wait for the reply with the value from the responsible peer
    result = requestingPeer_send_p2p('retrieve', index, null);
    return result;
}
```

LISTING 4.3: DHT API Operation *get*

```
//sender: destination address of the requesting peer
//index: unique identifier, e.g., 160 bit value
void responsiblePeer_on_receive_retrieve(sender, index)
{
    //get the current value from localstore and
    //send the value back to the requesting peer
    value = localstore.get(index);
    send_direct(sender, 'retrieve_reply', index, value);
}
```

LISTING 4.4: DHT Operation on Receiving a Retrieve Message

In Figure 4.2, we visualize the communication flow between the affected participants when processing a message for storing or retrieving a value. The overview of the fundamental DHT operations and the API operations *put* and *get* is given in Table 4.1. While the specific implementation of the fundamental operations of various DHTs might differ, every DHT offers those or very similar operations.

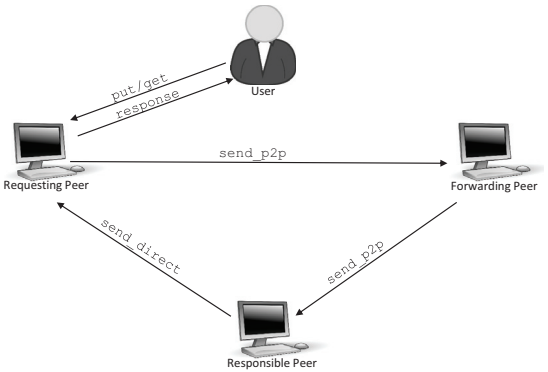

FIGURE 4.2: Communication Flow Between the DHT Participants

Type	Operation	Return Value	Description
API	`put` (index, value)	ack/nack	invoke the store event
API	`get` (index)	value	invoke the retrieve event
Fund.	`send_p2p` (event, index, value)	ack/nack/value	retrieve or store a value
Fund.	`on_receive_store` (sender, index, value)	–	store the given value
Fund.	`on_receive_retrieve` (sender, index)	–	provide the requested value
Fund.	`send_direct` (receiver, 'type_reply', index, result)	–	send the result to the requesting peer
Fund.	`response` ('type_reply', index, result)	ack/nack	return the result to the user

TABLE 4.1: API and Fundamental Operations of a DHT

4.1.3 Attacker Model

We assume the presence of one or more *attackers* with the goal to gain unauthorised access to DHT entries. An attacker may compromise and take over one or more arbitrary peers and/or users. A subverted peer is under the attacker's full control, i.e., she can send, receive, discard, replay, and forge messages on behalf of the peer. Moreover, a subverted peer can manipulate DHT entries that are stored on it. However, we assume that the total number of peers in our DHT is much larger than the number of subverted peers.

4.2 Requirements

In this section, we describe and motivate the requirements for our novel access control scheme.

Access: A user must be able to perform only the actions if and only if she has the appropriate rights.

Ownership: We require a mechanism to uniquely determine the owner of a certain DHT entry. Once the ownership is set, it must be impossible for an attacker to steal the ownership. Only the owner must be able to revoke her ownership.

Granularity: The owner must be able to manage read and write right separately for each of her entries in the DHT. These rights can be assigned to individual users. Additionally, the owner must also be able to delegate the handling of read and write rights to other users.

Privacy: It must be impossible to deduce the owner from the DHT entries. Specifically, our scheme must not introduce *new* ways to deduce the owner of a DHT entry. Further, the access control scheme must not provide a way for any entity to find all DHT entries of a certain owner. In general, our scheme must not introduce *new* privacy risks for anyone.

Scalability: DHTs are built with a high scalability in mind, i.e., they can handle a multitude of simultaneous read and write operations (get/put). Any access control scheme added on top of such a DHT must not degrade this property. Thus, the communication and processing overhead for any operation in the DHT must not increase significantly.

***k*-Resilience:** In our P2P network, the attacker can compromise at most k arbitrary peers. Therefore, we require that our access control scheme remains functional even if the attacker controls up to k peers. Specifically, even in this worst case, our other requirements, i.e., Access, Ownership, Granularity, Privacy, and Scalability, must still be fulfilled.

4.3 Design Rationale and Overview

Before presenting our approach in detail, we first provide a brief design rationale to explain our main design choices. As discussed before, our system model includes a coarse-grained access control scheme that is already available. This ensures that only authenticated participants can access the DHT. To attack the system, an attacker must compromise one or more participants, i.e., users and/or peers. Therefore, we must provide mechanisms to safeguard against compromised users and against compromised peers, and to enforce access rights against them. In the following, we first discuss how to handle compromised users. After that, we discuss how to handle compromised peers.

4.3.1 Compromised Users

To safeguard against compromised users, the responsible peer must authenticate the accessing user and enforce her access rights with a suitable authorization scheme. For that, we extend the standard DHT operations with an additional parameter for the user authentication, named *auth*. We discuss three different approaches for implementing the *auth*-parameter in Section 4.4. For the authorization, we use access control lists (ACLs). We enforce the ACL on the responsible peer, since it can store the access rights as metadata in the corresponding DHT entry.

In addition, we must determine the ownership of a DHT entry. Since a DHT is a decentralized data structure, there is no central administration to assign the ownership. Therefore, we use the approach from [89] for determining the ownership of a DHT entry: The user who first writes a value under an index becomes its owner and has sole access to it. Afterwards, she can delegate the access rights to other users.

Using the *auth*-parameter, the ACL on the responsible peer, and the owner concept as presented above, we can cope with compromised users. However, an attacker can also compromise peers. We discuss this second – and more difficult – case in the next section.

4.3.2 Compromised Peers

Due to the fact that a compromised peer has full access to the DHT entries under its control (cf. Section 4.1), the attacker can ignore the ACL, the *auth*-parameter, and also the ownership. Therefore, we need additional safeguards to protect the DHT entry against unauthorized write and read access.

To counter unauthorized write access, we use replication and majority voting. For that, we store each value on $2k+1$ different responsible peers. Hereby, we map the index of this value to $2k+1$ different new indexes. We compute these indexes with the Equation 4.1 for all i with $1 \leq i \leq (2k+1)$ where $i \in \mathbb{N}$, and h is a cryptographic hash function.

$$index_i := h(index|i) \tag{4.1}$$

With the concatenation of the initial *index* and i, the user first calculates $2k+1$ different storing DHT positions. However, these positions are consecutive and probably handled by the same peer. Hence, she additionally applies a cryptographic hash function $h(x)$. The hash function is used to create indexes that are evenly distributed over the DHT space. Consequently, each of these replicas is stored on a different peer. To write a value, a user performs $2k+1$ put operations, one for each new index. To read these replicas, we also need to perform get operations on all the $2k+1$ indexes. As long as we have no more than k subverted peers, we can guarantee that this will return at least $k+1$ correct values. Using a majority function over received values, we determine the correct value. Even if the attacker compromises more than k peers, our approach provides a graceful degradation. We discuss this in more detail in the security analysis (cf. Section 4.5.1).

To counter unauthorized read access, the owner encrypts the value with a randomly generated symmetric key before storing the $2k+1$ replicas in the DHT. This key is shared with all users that have read access. All other participants – including compromised peers – cannot read the value. In this case, the responsible peers do not have to verify the authorization of the requesting user – without the encryption key, nobody can decrypt the data anyway. Therefore, we can omit the authentication parameter *auth* for the get operation.

The above described safeguards rely on replicating a value to $2k+1$ different peers. This is true if the system contains a sufficient number of peers. The minimum number of peers that must be available depends on the specific DHT implementation. As an example, with Kademlia [60], we can ensure the distinctiveness of storing peers if there are at least $2k+1$ peers online. This is possible due to Kademlia's way to determine the responsible peer for a specific index (cf. Section 5.1).

So far, the system would be vulnerable against compromised requesting peers. If the user accesses the DHT with such a compromised requesting peer, the peer could drop all operations. This is countered by connecting each user to $2k+1$ randomly selected requesting peers. In addition, we must ensure that there are disjoint paths from these requesting peers to the $2k+1$ responsible peers. Therefore, the attacker would need to compromise in total more than k requesting, forwarding, or responsible peers to ensure that a user cannot communicate with at least $k+1$ benign responsible peers.

4.4 *k*-rAC

After giving a brief overview of the approach for a new access control and explaining our main design choices, we now describe *k*-rAC in more detail. For that, we first present the general integration of *k*-rAC in a DHT. Hereby, we introduce a data structure for a DHT entry to enable a controlled access to it and present the main *k*-rAC components. After that, we show the authorization mechanisms for both write and read accesses. Finally, we consider three different user authentication mechanisms.

4.4.1 System Architecture

To realize the approach presented in the previous section, we extend the DHT API with the new operation *set*. With this operation, we enable the owner to define access rights on the DHT value in her entry. Hereby, the put and the set operations are closely related, as we could manage the access rights also with the put operation by including an *acl* parameter. However, for the sake of simplicity, we preferred a clear separation of managing the value and the ACL.

The ACL of an entry is stored together with the value in the local storage of the responsible peer. A single ACL item contains the user's authentication *auth*, and the access rights received with the parameter *acl*. To implement this, we extend the DHT entry with metadata as shown in Listing 4.5. An empty entry has no ACL and is not owned by anyone. The first access via a put operation to a previously empty entry determines the owner of this entry. This means, the peer stores in the ACL *auth* of the user who first accessed this entry, and sets her access right to *owner*. Therefore, the smallest possible ACL is a list with the owner as a single item, whereby the stored *auth* depends on the used authentication mechanism and will be described in Section 4.4.3.

```
1 dht_entry {
    index: BigInteger;
3   value: object;          //the stored data object
    acl:   list<acl_item>;//a list with acl items
5 }
  acl_item {
7   auth:  authenticator;//for authentication
    right: {o,a,w,r};
9 }
```

LISTING 4.5: Access Control Data Structures

For the *right*-parameter within an ACL item, we define the following access rights:

- read (r) – the right to read the value,
- write (w) – the right to write the value,
- admin (a) – the right to change, add, or remove the read or write access of other users,
- owner (o) – this right implies r, w, and a. Additionally, only the owner can provide admin rights to other users.

While the owner is set by the system on first access via a put operation, the other rights can be arbitrary managed by the owner, or users with the admin right for a specific entry. This implies the access hierarchy $o > a > (r|w)$, where $>$ is defined as 'includes right'.

In conjunction with the set operation, we introduce an additional event message *setacl*, and the new DHT operation *on_receive_setacl(sender, index, acl, auth)* to handle the reception of such a message. Using this operation, the responsible peer sets the access rights for the corresponding entry according to the received ACL items within the parameter *acl*.

With these extensions, we can still rely on the ordinary DHT API operations for our approach – we only must add the *auth*-parameter to achieve the controlled access to an entry. In summary, we provide the three following API operations:

- put (index, value, auth) – stores the *value* at the *index* in the DHT, i.e., a write access. The user is authenticated with the *auth* parameter.
- set (index, acl, auth) – modifies the ACL without modifying the *value*, i.e., a write access.
- get (index, [*auth*]) – retrieves a *value* from the given *index*, i.e., a read access. The authentication parameter *auth* is optional, since the read access control is enforced by distributing the encryption key. However, it can be used as the identifier for the corresponding ACL item in some scenarios (e.g., with the PK approach as shown in Section 4.4.2.2).

From the safeguards presented in the design rationale, we determine the following four fundamental components of *k*-rAC: replication to the $2k+1$ responsible peers, authentication, authorization, and the majority function. Hereby, the replication and majority voting are implemented on the user side, while the authentication and the authorization are handled by the responsible peers.

4.4.1.1 User Side

For the replication and majority voting, we use the API operations $2k+1$ times, once for each of the corresponding $2k+1$ indexes. The authentication and the authorization are based on the information provided with the *auth*-parameter. In consequence, the $2k+1$ indexes as well as the *auth*-value must be computed by the user before performing the write or read access. Furthermore, as the value in a DHT entry can be any data object, we must provide it with the universal properties to implement a generally applicable authentication independent of the data object type in a single DHT entry. We achieve that by using metadata. For that, as described in Section 3.5, we assume the operations *getMetadata* and *setMetadata* for any data object type. Additionally, we introduce the object *Credentials*. The user creates an instance of this object before storing a value with the DHT, i.e., on the initial access via the put operation. This object

contains values for the particular authentication mechanism. Hereby, we differentiate between its private and public part. The user keeps the private credentials secret, and uses them only for calculating the *authentication value* required for the *auth*-parameter. The public credentials are the values needed by the responsible peer to verify the user's authentication value. The user sends them to the responsible peers inside the *auth*-parameter on the initial write access. The responsible peers store the public credentials in an ACL item, and use them on subsequent accesses to verify the received authentication value, i.e., to authenticate the user.

Additionally, the user creates an instance of the object *Authenticator* according to the used authentication mechanism for using it as *auth* in the succeeding API operations. Thereupon, the user calculates the 2k+1 indexes. Hereby, we use the unique identifier of the DHT value (e.g., its hash value) as *index* to derive from it the new $2k+1$ different indexes. Each $index_i$ is mapped to one of the $2k+1$ value replicas. For each index, the user executes the put API operation. As confirmation, she gets $2k+1$ responses from the responsible peers, whereby a single response contains an acknowledgement *ack* or a negative-acknowledgement *nack*. In Listing 4.6, we summarize these steps in the operation *user_k-rAC_wrapper_put*, which represents a wrapper for the API operation *put*. Hereby, we assume each user maintains a variable *wallet*, where she holds the metadata of her DHT values, e.g., her credentials.

```
1  //dObj: a data object to be stored with the DHT
   //tagName: name of the metadata property, e.g., ''DRSid''
3  void user_k-rAC_wrapper_put(dObj, tagName){
       id = getMetadata (dObj, tagName):
5
       //1)verify whether it is the initial put for this data object
7      if (id == null){
           //it is an initial DHT access
9          //generate the ID and assign the metadata according to Listing 3.3.
           setID(dObj);
11
           //create credentials according to the used authentication mechanism
13         credentials = new Credentials();

15         //save credentials locally, e.g., in HashMap<BigInteger, Object>
           wallet.put(getMetadata(dObj).getValue(tagName), credentials);
17     }

19     //2)create auth according to the used authentication mechanism
       auth = new Authenticator();
21     //differentiate between auth for the initial and subsequent accesses
       if (id == null){
23         //compute auth for the initial access
           ...
25         //now update id, as we need it in 3)
           id = getMetadata(dObj).getValue(tagName);
27     } else {
          //compute auth for the subsequent access
29        ...
       }
31
       //3)calculate the 2k+1 indexes according to Equation 4.1
33     results[] =  new object[2k + 1];//k is a system parameter
       for (int i = 0; i++; i < (2k + 1)){
35         index_i = calculateIndex(id, i);
           //execute put and collect peer responses in results[]
37         results[i] = DHTapi.put(index_i, dObj, auth);
       }
```

```
39
    //4)compute majority function over the received results
41  result = computeMajority(results[]);//ack or nack
}
```

LISTING 4.6: *k*-rAC Wrapper for API Operation *put* (Template)

With the other two API operations, we modify the ACL (*set*) or retrieve the value (*get*) for a certain index. Hereby, the user must also compute the $2k+1$ indexes and the *auth*-parameter. While the parameter *auth* is optional for the get operation, it is mandatory for *set*. We exemplarily show user's steps for *set* in Listing 4.7, and for *get* in Listing 4.8. We describe these steps in detail in Section 4.4.2.

```
//id: unique identifier, e.g., 160 bit value
2 //acl: list of acl items
void user_k-rAC_wrapper_set(id, acl){
4   credentials = wallet.get(id);
    //1)compute auth according to the used authentication mechanism
6   auth = ... ;

8   //2)calculate the 2k+1 indexes according to Equation 4.1
    results[]   = new object[2k + 1];//k is a system parameter
10  for (int i = 0; i++; i < (2k + 1)){
        index_i    = calculateIndex(id, i);
12      //3)execute set and collect the peer responses in results[]
        results[i] = DHTapi.set(index_i, acl, auth);
14  }

16  //4)compute majority function over the received results
    result = computeMajority(results[]);//ack or nack
18 }
```

LISTING 4.7: *k*-rAC Wrapper for API Operation *set* (Template)

```
//id: unique identifier, e.g., 160 bit value
2 void user_k-rAC_wrapper_get(id){
    //1)compute auth, optional
4   auth = ... ;

6   //2)calculate the 2k+1 indexes according to Equation 4.1
    results[]   = new object[2k + 1];//k is a system parameter
8   for (int i = 0; i++; i < (2k + 1)){
        index_i = calculateIndex(id, i);
10      //3)execute get and collect peer responses in results[]
        results[i] = DHTapi.get(index_i,[auth]);
12  }

14  //4)compute majority function over the received results
    result = computeMajority(results[]);//value
16
    //5)decrypt the value to read it
18 }
```

LISTING 4.8: *k*-rAC Wrapper for API Operation *get* (Template)

4.4.1.2 Responsible Peer Side

Complementarily, on the peer side, we include the controlled access to an entry in the work processes of the used DHT. Hereby, we also use a *k*-rAC wrapper to include the verification whether the requested entry is empty, the user authentication, and the authorization of the access before executing an event operation. In Listing 4.9, we sketch the steps a responsible peer performs for the access control before executing a DHT operation according to the requested access. As mentioned above, the read access control is enforced by the value encryption. Therefore, our focus is the access control for write accesses, i.e., for the API operations *put* and *set*. We describe these steps in the following sections. Particularly, to specify the distinctiveness of the three authentication mechanisms for write accesses on the peer side, we introduce the operation *authenticate_user(acl_item, auth, [value])* (cf. Listing 4.10). Hereby, the parameter *acl_item* is the user's ACL item for the requesting index, and *auth* is user's authentication value received by the responsible peer with the event message. Further parameters are optional; they depend on the particular authentication mechanism.

```
//sender: destination address of the requesting peer
//event: requested action, i.e., store, setacl, or retrieve
//index: unique identifier, e.g., 160 bit value
//value: arbitrary data object, optional
//auth: user's authentication value, optional
void responsiblePeer_k-rAC_dispatcher(sender, event, index, [value], [auth]){
    switch (event) {
        case ``store'' or ``setacl'':
            //1)verify if the entry is occupied
            if (event == ``store'' && entry empty){
                //access allowed, the user becomes owner
            } else {
                //2)select user's ACL item from the ACL of the entry
                //3)authenticate the user, cf. Listing 4.10
                //4)verify user's access rights
                //5a)execute event operation upon put or set
            }
        case ``retrieve'':
            //5b)execute event operation upon get
    }
}
```

LISTING 4.9: *k*-rAC Dispatcher for incoming Requests on Peer Side (Template)

```
//acl_item: user's ACL item
//auth: user's authentication value
//value: data object of the accessed entry, optional
object responsiblePeer_authenticate_user(acl_item, auth, [value]){
    //do calculations to authenticate
    //if authentication was successful, return true
    //else, return false

    return result;
}
```

LISTING 4.10: Authentication by Responsible Peer (Template)

As depicted in Listing 4.9, the control of a write access upon *put* starts with the verification whether the accessed entry is empty, i.e., *verifying the ownership*. The following steps depend on this result: If the entry is empty, the user becomes its owner, and the responsible peer performs the requested write operation. Otherwise, the user must be first authenticated, and then

authorized for the requested access. Hence, we define that the authorization consists of two parts: verifying the ownership and checking the access rights. The write access upon *set* is slightly different: By definition, there is no initial access to an entry via a set operation, i.e., it is not possible to modify access right for a value which does not exist. Therefore, the responsible peer does not verify whether the entry is already in possession. In the rest, the access control upon *set* ist the same as upon *put*.

With the listing templates from above, we explain the general integration of *k*-rAC in a DHT. In the next section, we use these templates to show the authorization process according to the received event message and the access rights in the ACL item of the requesting user. Furthermore, we use them to specify the differences between the three proposed authentication mechanisms.

4.4.2 User Authorization Mechanisms

As described in Section 4.1, on receiving an event message, the responsible peer is instructed by its parameter *event* which action should be performed. We differentiate between the events for a write action, i.e., *store* or *setacl*, and the event for a read action, i.e., *retrieve*. Below, we consider how to authorize the user for the write access via the API operation *put* or a *set*, and for the read access via the API operation *get*. To showcase the authorization characteristic upon receiving the individual event message, we fully specify the operation *responsiblePeer_k-rAC_dispatcher* (cf. Listing 4.9). In the following, we describe each case presented in this template in detail.

4.4.2.1 Write Access

Upon the user's write access via *put* or *set* on $2k+1$ peers, each of the responsible peers receives a corresponding event message, i.e., *store* or *setacl*. Thereupon, a responsible peer starts with processing the first part of the authorization as follows: It first verifies whether it is the initial or a subsequent store access to the given index, i.e., if there is already an owner or not. If it is the initial access, the responsible peer creates a new *dht_entry* data structure and initializes its *acl* field with the minimal ACL, i.e., a single ACL item for the requesting user with the *o* right. After that, the responsible peer stores the new entry locally. Finally, it executes the corresponding DHT operation *on_receive_store* to store the received value in the entry (cf. Listing 4.11).

```
//sender: destination address of the requesting peer
//event: requested action, i.e., store, setacl, or retrieve
//index: unique identifier, e.g., 160 bit value
//value: arbitrary data object, optional
//auth: user's authentication value, optional
void responsiblePeer_k-rAC_dispatcher(sender, event, index, [value], [auth]){
   switch (event) {
     case ''store'' or ''setacl'':
        //it is a write access
        dataset ds   = localstore.get(index);
        //verify if the entry occupied
        if (event == ''store'' && ds == null){
            //no value stored; hence, create a new dht_entry
            ds = new dataset();
            ds.setOwner(auth);//current user becomes owner
            localstore.put(index, ds);//store the new entry locally
```

```
18              //store the value
                on_receive_store(sender, index, value, auth);
20          } else {
                if (event == ''store''){
22                  on_receive_store(sender, index, value, auth);
                } else if {event == ''setacl''){
24                  //in this case, value contains an ACL,
                    on_receive_setacl(sender, index, value, auth);
26              }
            }
28      case ''retrieve'':
            //it is a read access
30          on_receive_retrieve(sender, index, [auth]);
    }
32  }
```

LISTING 4.11: *k*-rAC Dispatcher Snippet: Authorization – Verifying the Ownership

Otherwise, it is a subsequent write access, and the responsible peer executes the DHT operation according to the requested access, i.e., *on_receive_store* for storing a value, or *on_receive_setacl* for modifying the ACL. However, before performing the write action, it verifies the authenticity of the requesting user with the provided *auth* accordingly to the used authentication mechanism (cf. Section 4.4.3). If the authentication fails, the user's request is rejected. Upon a successful authentication, the responsible peer performs the second part of the authorization, i.e., checking the access rights. Hereby, we include the authentication into the DHT operations *on_receive_store* and *on_receive_setacl* to adapt them to *k*-rAC. For that, we extend them with the parameter *auth*. In the following, we consider the steps for enforcing the authorization policy for storing the value and modifying the ACL separately.

Store a Value

Before storing the received value, the responsible peer checks whether the authenticated user has the right to write to this index, i.e., at least the *w*-right. For that, it first selects from the ACL the item associated with the requesting user by using the received *auth*. If the user has the write right, the responsible peer overwrites the locally stored value in the corresponding DHT value with the received value. Otherwise, the responsible peer rejects the request. In Listing 4.12, we show the adaptation of the operation *on_receive_store* (cf. Listing 4.2) to provide an authorized write access to the DHT value.

```
//sender: destination address of the requesting peer
2 //index: unique identifier, e.g., 160 bit value
//value: arbitrary data object
4 //auth:  user's authentication value
void responsiblePeer_on_receive_store(sender, index, value, auth){
6     dataset ds    = localstore.get(index);
      object result = nack;//initial value
8     //get user's ACL item
      acl user_acl  = ds.acl.getUserAcl(auth);
10    if ((user_acl != null)
        && ((authenticate_user(user_acl, auth, [value]) == true)
12      && (user_acl.rights == ['a'|'w'|'o'])){
          //write access allowed
14        ds.value  = value;
          result    = ack;
16    }
```

```
    //signal the result to the requesting peer
    send_direct(sender, 'store_reply', index, result);
}
```

LISTING 4.12: k-rAC Operation on Receiving a Store Message

Modify the ACL

While a user uses *put* for storing or modifying a data object in the entry field *value*, with the new operation *set(index, acl, auth)*, she modifies the ACL. We use this operation to provide the owner with a mechanism to delegate the read, write, and admin rights to other users. As presented above, the access control for *set* is also handled by the responsible peer, as with *put*. To perform the authorization part, the responsible peer executes the new DHT operation *on_receive_setacl*. Hereby, the peer first verifies whether the requesting user has the appropriate right to change the ACL of this index. While to update the read or write access for other users the requesting user needs at least the admin right, to update the admin right, she must be the owner of this entry. We summarize the steps of the ACL updating routine in Listing 4.13.

```
//sender: destination address of the requesting peer
//index: unique identifier, e.g., 160 bit value
//acl: list of acl items
//auth: user's authentication value
void responsiblePeer_on_receive_setacl(sender, index, acl, auth)
{
    dataset ds    = localstore.get(index);
    object result = nack; //initial value
    //get user's ACL item
    acl user_acl  = ds.acl.getUserAcl(auth);
    //verify user's access rights
    if ((user != null)
        && (authenticate_user(user_acl, auth, [value]) == true)
        &&(user.rights == ['a'|'o']) ) {
          // ACL update allowed
          ds.acl.modify(user.rights, acl);
          result   = ack;
    }
    //signal the result to the requesting peer
    send_direct(sender, 'setacl_reply', index, result);
}
```

LISTING 4.13: *k*-rAC Operation on Receiving a Setacl Message

With the set operation, we can also revoke the access of a user to a specific entry by removing her from the ACL. Any subsequent write access by the revoked user will be denied by the responsible peers. To revoke a read access, we need to re-encrypt the value with a new key which is not known to the revoked user.

4.4.2.2 Read Access

A read access is mapped to the execution of *get(index)* on $2k+1$ responsible peers. Each of these operation returns the DHT value for the given *index*. Upon read access, there is neither authentication nor authorization required, i.e., any user can retrieve the value for any index in the

DHT. The read access control is enforced by encrypting the value with a randomized symmetric data encryption key (k_d), and distributing it to the intended users. This can be done out-of-band, e.g., per email or instant messaging. Since in this case the DHT is not involved, we do not need any additional mechanism to protect the read access. Therefore, the responsible peer uses the DHT operation *on_receive_retrieve(sender, index)* without any adaptation (cf. Listing 4.4). Hence, in contrast to the write accesses, the authorization for a read access is handled on the user side. For that, the user retrieves the value from $2k+1$ different indexes as shown in Listing 4.8. After receiving the $2k+1$ replicas, the user calculates the majority voting over all of them, i.e., she compares them and uses the value which was received at least $k+1$ times. To optimise this routine, the user could evaluate the majority whenever a new reply is received. Thus, in best case, if the first $k+1$ values are already identical, the user can proceed with the value decryption without waiting for the remaining replies.

As mentioned above, we assume that the authorized user receives the corresponding data encryption key k_d out-of-band. Another possibility to distribute k_d with the DHT is by using public key cryptography. For that, each user needs a public/private key pair. The key k_d is encrypted with each public key of users who should have read access. As a result, each authorized user gets her individual decryption key. To distribute these individual decryption keys, we extend the ACL item from Listing 4.5 with a *key*-field. When a user performs the get operation for an entry, she gets the DHT value including the corresponding ACL with the encrypted k_d. Thereupon, she first uses her own private key to decrypt k_d, and then she uses k_d to decrypt the value. Thus, the read access to the DHT entry is enforced by having access to k_d or not. An optimisation for the public key approach would be to include the *auth* parameter with the get operation, i.e., *get(index, auth)* as shown in Listing 4.14. With this additional information, the responsible peer can search for the user's encrypted k_d and reply it together with the value within the DHT entry. For that, we adapt the DHT operation *on_receive_retrieve* as shown in Listing 4.15. Without *auth*, the user needs to search for her individual key-item after receiving the DHT entry.

```
1  //index: index of the entry with the requested value
   void user_k-rAC_wrapper_get(index){
3      credentials = wallet.get(index);
       //1)compute auth according to the used authentication mechanism
5      auth = ...;

7      //2)calculate the 2k+1 indexes according to Equation 4.1
       results[]   =  new object[2k + 1] //k is a system parameter
9      for (int i = 0; i++; i < (2k + 1)){
           index_i = calculateIndex(index, i);
11         //3)execute get and collect peers' responses in results[]
           results[i] = DHTapi.get(index, auth);
13     }

15     //4)compute majority function over the received results
       entry           = calculateMajority(replies);
17     encrypted_value = entry.object;
       encrypted_key   = entry.acl.k_d;
19
       //5)decrypt the individual decryption key with own private key
21     key = decryptKey(encrypted_key, credentials.keypair.sk);

23     //6)decrypt the value to read it
       value = decryptValue(encrypted_value, key);
25 }
```

LISTING 4.14: *k*-rAC Wrapper for API Operation *get* with Integrated Key Distribution using Public Key Approach

```
//sender: destination address of the requesting peer
//index: unique identifier, e.g., 160 bit value
//auth: user's authentication value
void responsiblePeer_on_receive_retrieve(sender, index, auth)
{
    //get the current value from localstore
    entry = localstore.get(index);

    //reduce the ACL to the user's individual encrypted key
    reduceACL(entry.acl, auth);

    //send the value back to the requesting peer
    send_direct(sender, 'retrieve_reply', index, entry);
}
```

LISTING 4.15: *k*-rAC Operation on Receiving Retrieve Message

Alternatively, the read access can be realized based on Shamir's secret sharing algorithm [77] as done in [44]. However, we decided to use the above described method with encryption, as it suits better for the modular access control with *k*-rAC. Finally, if no read access control is required, we can omit the encryption of the data object altogether (e.g., for the DRS with the status pull method).

4.4.3 User Authentication Mechanisms

As discussed before, to enforce access control, we need to authenticate a user that requests access to a DHT entry. To do so, we analysed a number of existing authentication mechanisms and adapted three of the most promising ones to our use case. We found that each mechanism had specific advantages that depend on the actual usage scenario. Therefore, we decided to implement all three mechanisms. We also structured the access control such that a system designer can select and use the mechanism that is best suited for her scenario. We analyse the specific advantages and disadvantages of all three authentication mechanisms in Section 4.5. Common to all three mechanisms is the usage of the *auth*-parameter to authenticate the user and their resilience against up to *k* subverted peers.

In the following, we specify the operation *user_k-rAC_wrapper* to emphasize the characteristics of the particular authentication mechanism on the user side. Additionally, we use the template of the operation *authenticate_user* from *responsiblePeer_k-rAC_wrapper* (cf. Listing 4.10) to specify the distinctiveness of the authentication mechanisms on the peer side.

4.4.3.1 Public-Key Cryptography

With the public-key (PK) authentication mechanism, we authenticate the user by verifying her signature. For that, each user generates a public/private key pair (pk, sk). Hereby, the user keeps her private key *sk* secret and uses it only for calculating the authentication value for the *auth*-parameter, i.e., her signature. In contrast, she sends the public key *pk* inside the *auth*-parameter to the responsible peers, as they need it to verify the user's signature. Besides *pk*, *auth* contains a message ID *mid* and a signed hash of the value concatenated with the message ID, i.e., $auth := \{pk, mid, sign\}$ where $sign := \text{encrypt}_{sk}(h(value|mid))$, and $h(x)$ is a cryptographic hash function. We use the message ID to prevent replay attacks. Hereby, we rely on the standard approach with a sliding window, similarly to IPsec [32]. With this approach, the user increments

the message ID *mid* with each update of the DHT entry, i.e., by executing *put* or *set*. Thus, for the subsequent write accesses to the corresponding DHT entry, the user must calculate the new signature accordingly to the currently valid message ID. For that, she stores the currently valid message ID locally. In Listing 4.16, we exemplarily show the adaption of the operation *user_k-rAC_wrapper_put* for the PK approach. Hereby, the user does not sign the value for the initial access, as the corresponding public key is in the same message, and, therefore, the responsible peers cannot securely verify the provided signature. Hence, we set the *sign*-value on the initial access to *null*, as the responsible peers ignore it anyway on the initial access.

```
//dObj: data object to be stored with the DHT
//tagName: name of the metadata property, e.g., ''DRSid''
void user_k-rAC_wrapper_put(dObj, tagName){
    id = getMetadata(dObj, tagName);
    //1)verify whether it is the initial put for this data object
    if (id == null){
        //it is an initial DHT access
        //generate the ID and assign the metadata according to Listing 3.3.
        setID(dObj);
        //generate public/private key pair
        credentials             = new Credentials();
        credentials.keypair     = new AsymmetricKey();
        //prevent replay attacks with sliding window
        credentials.window      = SlidingWindow.generateSlidingWindow();
        credentials.window.mid = SlidingWindow.setMessageID();
        //save credentials locally
        wallet.put(getMetadata(dObj).getValue(tagName), credentials);
    }

    //2)create authenticator, i.e., auth := (pk, mid, sign)
    auth = new Authenticator();
    auth.pk             = credentials.keypair.pk:
    //differentiate between auth for the initial and subsequent accesses
    if (id == null){
        //compute auth for the initial access
        auth.mid        = credentials.window.mid;
        auth.sign       = null;
        //update id, as we need it in 3)
        id = getMetadata(dObj).getValue(tagName);
    } else {
        auth.sign       = calculateSignature(credentials.keypair.sk,
                              dObj, credentials.window.mid);
        credentials.window.mid ++;
    }

    //3)calculate the 2k+1 indexes according to Equation 4.1 and
    //execute the intended API operation for each index, e.g., put
    results[] =  new object[2k + 1] //k is a system parameter
    for (int i = 0; i++; i < (2k + 1)){
        index_i     = calculateIndex(id, i);
        //collect results for majority voting
        results[i] = DHTapi.put(index_i, dObj, auth);
    }

    //4)compute majority function over the received values
    result = computeMajority(results[]);
}
```

LISTING 4.16: *k*-rAC Wrapper for API Operation *put* with PK Approach

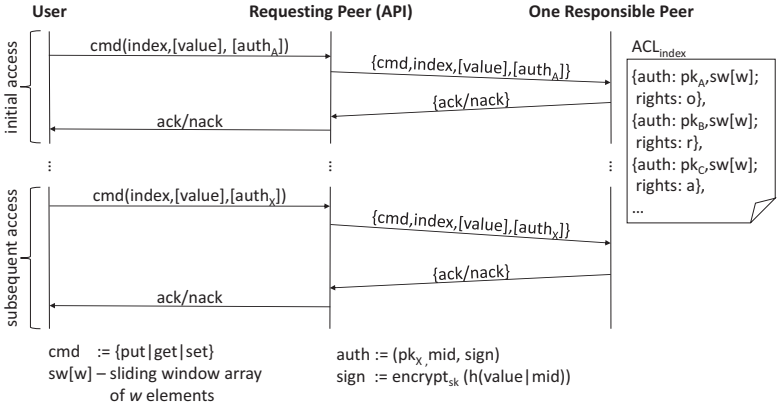

FIGURE 4.3: PK Message Flow

Complementary, the responsible peers store upon initial access the user's public key *pk* in the *auth* field of the ACL item. By doing so, this public key is now pinned to that index, and the user is defined as its owner. As mentioned above, the responsible peers ignore the parameter *sign* on the initial access to the requested index. In subsequent requests, they verify the validity of the provided signature using the public key *pk* which they stored upon the initial access (cf. Listing 4.17). Only the user in possession of the right private key is able to generate a valid signature. The public key is also included in *auth* on subsequent accesses. However, the responsible peers use it only as the identifier by selecting the corresponding ACL item in the local storage. To prevent replay attacks, the responsible peers store at most *w* old message IDs, where *w* is the window size. Combining the value signing with the sliding window approach, the user is authenticated on subsequent requests with a valid signature, and the freshness of the message is ensured. We summarize the message flow of the PK authentication in Figure 4.3.

```
//acl_item: user's ACL item
//value: data object, sent by the user with the requested operation
//auth: received authenticator
object responsiblePeer_authenticate_user(acl_item, auth, value){
    //calculate the signature
    //and compare it with the signature received within the auth-parameter
    sign = calculateSignature(acl_item.pk, value, auth.mid);
    if (sign == auth.sign){
        result = true;//the user is authenticated
    } else {
        result = false;//the authentication failed
    }

    return result;
}
```

LISTING 4.17: Authentication by Responsible Peer with PK Approach

Since the ACL does not contain personal information about the user, this approach does not leak any information about user's identity. However, if a user owns multiple DHT indexes, an attacker could determine all her DHT entries by comparing the stored public keys. In some

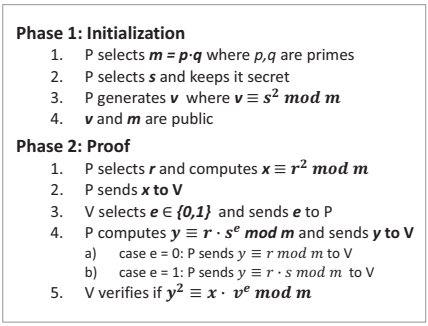

FIGURE 4.4: Overview of Mathematical Calculations in Fiat-Shamir Protocol (P = Prover, V = Verifier)

scenarios, especially with the DRS, this is not desirable and must be prevented. Here, the user should use different public/private key pairs for each index (as already done in Listing 4.16). This way, no connection between any two indexes can be established, even with an all-powerful attacker who could read the entire DHT.

4.4.3.2 Zero-Knowledge Proof

With a zero-knowledge proof (ZKP) [36], a prover proves to a verifier that she possesses a secret without revealing it. It works as a challenge/response system, where the prover has a chance of 50% to cheat in any single round. Therefore, the proof must be performed n times to achieve a high confidence.

We applied the ZKP to a DHT by building on the Feige-Fiat-Shamir protocol [30]. In Figure 4.4, we give an overview of the mathematical calculations for this protocol. We adapt these calculations to our approach as follows: For the initial write access to an entry, the user generates a secret random number s. The square of this secret, i.e., $v \equiv s^2 \bmod m$, is stored along with the modulo m as her authenticator on first access in the ACL item for the requested index. The responsible peers need m for the modular arithmetic in the authentication process. To authenticate herself on subsequent accesses, the user selects n random numbers r_i, where $1 \leq i \leq n$, and calculates their squares, i.e., $x_i \equiv r_i^2 \bmod m$. The resulting vector with the n x-values is part of the *auth*-parameter, i.e., $auth := \{m, v, [x_1, ..., x_n]\}$. As on the first access there is no authentication required, the user does not generate the x-vector, instead she fills it with, e.g., zeros. Otherwise, the generation of the x-vector would cause an unnecessary effort for the user, because the responsible peers ignore it on the first access to the requested index anyway. We summarize these steps in Listing 4.18.

The responsible peers handle the content of the *auth*-parameter depending on the access type to the requesting index, i.e., the first or a subsequent access. Upon first access, the responsible peers store the values m and v in the *auth* field of the ACL item, but, as mentioned above, they ignore the x-vector. On subsequent accesses, the responsible peers use the values m and v only to identify user's ACL item in the local storage. Furthermore, on subsequent accesses, they use the vector with the received x-values in the authentication process for the challenge/response procedure. Hereby, a single x represents a different random number for each of the n challenges.

To authenticate, each responsible peer generates its own n challenges and sends them to the requesting user. If the user replies with n correct responses, she is authenticated (Listing 4.19). After the authentication, the responsible peers discard the x_i values, as they are valid only for a single authenticated k-rAC API operation.

```
//dObj: a data object to be stored with the DHT
//tagName: name of the metadata property, e.g., ''DRSid''
void user_k-rAC_wrapper_put(dObj, tagName){
    id = getMetadata(dObj, tagName):

    //1)verify whether it is the initial put for this data object
    if (id == null){
        //it is an initial DHT access
        //generate the ID and assign the metadata according to Listing 3.3.
        setID(dObj);

        //generate secret, cf. Figure 4.4
        credentials         = new Credentials();
        credentials.modulo = computeModulo();
        credentials.secret = new ZKPSecret(credentials.modulo);
        credentials.square = computeSecretSquare(credentials.secret,
                                    credentials.modulo);

        //save credentials locally, e.g., in HashMap<BigInteger, Object>
        wallet.put(getMetadata(dObj).getValue(tagName), credentials);
    }

    //2)create authenticator, i.e., auth := (m, v, [x_1, ..., x_n])
    auth                = new Authenticator();
    auth.m              = credentials.modulo;
    auth.v              = credentials.square;

    //differentiate between auth for the initial and subsequent accesses
    if (id == null){
        //compute auth for the initial access
        auth.x-values[] = fillWithZeros();

        //update id, as we need it in 3)
        id = getMetadata(dObj).getValue(tagName);
    } else {
       //generate n random numbers and square them
       auth.x-values[] = calculateSquaredRandomNumbers
                            (credentials.modulo, n);
    }

    //3)calculate the 2k+1 indexes according to Equation 4.1 and
    //execute the intended API operation for each index, e.g., put
    results[] =  new object[2k + 1] //k is a system parameter
    for (int i = 0; i++; i < (2k + 1)){
        index_i    = calculateIndex(id, i);
        //collect results for majority voting
        results[i] = DHTapi.put(index_i, dObj, auth);
    }

    //4)compute majority function over the received values
    result = computeMajority(results[]);
}
```

LISTING 4.18: k-rAC Wrapper for API Operation *put* with ZKP Approach

```
//acl_item:  user's ACL item
//auth: received authenticator
object responsiblePeer_authenticate_user(acl_item, auth){
    //send challenges to the user
    user_responses[] = challengeUser(challenges[]);
    //verify user's responses
    for each response in user_responses{
        result = verify(acl_item, auth.x-values, response);//cf. Figure 4.4
        if (!result){
            break;//authentication failed
        }
    }

    return result;
}
```

LISTING 4.19: Authentication by Responsible Peer with ZKP Approach

Since each request is authenticated with new challenges/responses, the freshness of the messages is ensured, and no additional mechanisms to prevent replay attacks are needed. After a successful authentication, the peer proceeds with the requested operation if this user has the appropriate rights (cf. Section 4.4.2). In Figure 4.5, we present the message flow of the ZKP authentication.

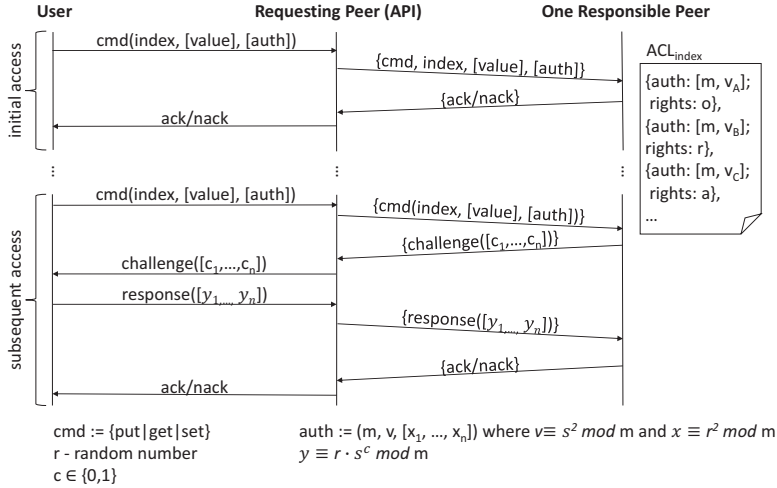

FIGURE 4.5: ZKP Message Flow

With this authentication scheme, the most costly part are the challenge/response messages for the authentication. As an optimisation, a peer could first check with the ACL whether the requesting user is authorized to perform the corresponding operation for the requested index. If the user does not have the appropriate rights, the peer suspends the requested operation without performing the challenge/response.

Also with this approach, there is no personal information leaking about the user. The only information an attacker could get is the squared number v used as the authenticator. However,

similarly to the public/private key approach, an attacker could determine all indexes belonging to the same user by comparing the authenticators v. Again, this can be avoided by choosing different secrets for different DHT indexes.

4.4.3.3 One-Time-Hash

The classical way to authenticate a user in client/server systems is with password hashes. In such an authentication scheme, the user authenticates with a secret, i.e., her password. On the server side, only the hash of the password is stored. To authenticate herself, the user sends her password to the server. Then, the server hashes and compares it with the stored hash. If they match, the user is authenticated. To prevent the usage of precalculated hash tables, an additional salt [64] is usually used. Salt is a random number, which is commonly appended to the secret before hashing. To still verify the password, the *salt* is stored in clear alongside the hash value.

This classical way is not applicable in our case, because the user must submit its secret to $2k+1$ responsible peers for the authentication process. Even by subverting only one of these peers, the attacker would get access to the user's password. To overcome this drawback, we extend the classical hash-based authentication by introducing an individual secret for each of the $2k+1$ peers, i.e., an individual authenticator for each of the responsible peers. For this, the user uses a master secret s with an HMAC-function [53] to create $2k+1$ individual secrets. Specifically, she calculates the individual secrets with $s_i = \text{HMAC}_s(index|i)$ for all i with $1 \leq i \leq (2k+1)$, $i \in \mathbb{N}$. We include the index in the calculation of these secrets to obtain a different secret for each index. The user hashes these individual secrets to create the $2k+1$ *auth*-parameters for each responsible peer, i.e., $auth_i := \{h(s_i|salt_{index}), salt_{index}\}$. Hereby, the user uses different *salt*-values for different indexes and stores them locally.

When the user creates a new DHT entry, she sends an $auth_i$-parameter to each of the $2k+1$ responsible peers, which store them in the corresponding ACL item. With any subsequent request, the user sends the individual secret s_i to each of the responsible peers to authenticate herself. Each responsible peer verifies this by hashing the received secret and comparing it to the stored hash (cf. Listing 4.21). It is impossible to determine the secret s_i from the stored hash $h(s_i|salt_{index})$. Hence, after the initial access and before the first authentication, no subverted peer is able to impersonate the user.

However, to authenticate, the user needs to send the secrets s_i to the responsible peers. In a P2P network, the responsible peers for a certain index might change over time. Therefore, some peers might still be able to collect some or all of the individual secrets. If the attacker manages to collect more than k individual secrets, she can still impersonate the corresponding user. To prevent this, we introduce the second extension, the so-called *one-time-extension*. With this, the individual secrets must only be used once, i.e., we update it with each authenticated operation. For that, the user generates a new individual secret for each operation by hashing the current one, i.e., $s'_i = \text{HMAC}_s(s_i)$.

Hence, the user includes in the $auth$ parameter the current secret s_i and the salted hash of the next secret s'_i, individually for each responsible peer. However, on initial access no current secret is required, as the responsible peers do not authenticate it. Nevertheless, to avoid different signatures for the initial and subsequent accesses, we define that the $auth$ parameter always comprises the three values, i.e., $auth_i := \{s_i, h(s'_i|salt_{index}), null\}$, while s_i is set to null on initial access. The responsible peers store the received s_{salt} only upon the initial access, because its value does not change afterwards. Since we do not need to send s_{salt} within $auth$ afterwards

again, we set it to *null* on subsequent accesses. Alternatively, we can use the salt value in *auth* on subsequent accesses as the identifier for the affected ACL item. Finally, the user stores the current individual secrets s_i along with the master secret s. In Listing 4.20, we exemplarily show the pseudocode for the operation *user_k-rAC_wrapper_put* with the one-time-hash (OTH) authentication mechanism. Hereby, we adapt the step sequence from Listing 4.6 by integrating the calculation of the individual secrets and the execution of the API operation in one `for`-loop (cf. line numbers 32 – 47).

```
1  //dObj: data object to be stored with the DHT
2  //tagName: name of the metadata property, e.g., ''DRSid''
3  //master_key: master key used for the HMAC function
4  //k: system parameter for the resilience level
5  void user_k-rAC_wrapper_put(dObj, tagName, master_key, k){
6      id = getMetadata (dObj, tagName):
7      //1)verify whether it is the initial put for this data object
8      if (id == null){
9          //it is an initial DHT access
10         //generate the ID and assign the metadata according to Listing 3.3.
11         setID(dObj);
12         //generate credentials, i.e., salt and hashes
13         credentials                  = new Credentials();
14         credentials.salt             = generateSalt();
15         credentials.individual_secrets = new HashMap();
16         //derive 2k + 1 new DHT indexes (cf. Equation 4.1)
17         //and calculate the initial individual secrets
18         for (i = 0; i < 2k +1; i++){
19           index_i   = calculateIndex(getMetadata(dObj).getValue(tagName), i);
20           s_i       = generateIndividualSecret(master_key, index_i, i);
21           credentials.individual_secrets.put(index_i, s_i);
22         }
23         //save credentials locally
24         wallet.put(getMetadata(dObj).getValue(tagName), credentials);
25     }
26
27     //2)create authenticator, i.e., auth := (s_i, hash, salt)
28     auth = new Authenticator();
29     auth.salt = credentials.salt;
30
31     results[] =  new object[2k + 1] //k is a system parameter
32     for (entry : credentials.individual_secrets.entrySet(){
33         //differentiate between auth for the initial and subsequent accesses
34         if (id == null){
35           //compute auth for the initial access
36           auth.s_i  = null;
37           auth.hash = calculateHash(entry.getValue(), auth.salt);
38         } else {
39           auth.s_i  = entry.getValue();
40           new_s_i   = newIndividualSecret(master_key, entry.getValue());
41           auth.hash = calculateHash(new_s_i, auth.salt);
42           wallet.put(entry.getKey(), new_s_i);//update locally
43         }
44         //3)execute the intended k-rAC API operation, e.g., put
45         results[i]  = DHTapi.put(entry.getKey(), dObj, auth);
46     }
47
48     //4)compute majority function over the received values
49     result = computeMajority(results[]);
50 }
```

LISTING 4.20: *k*-rAC Wrapper for API Operation *put* with OTH Approach

```
//user_acl: user's ACL item
//auth: received
//index: unique identifier, e.g., 160 bit value
object responsiblePeer_authenticateUser(user_acl, auth, index){
    //calculate the hash value with the received secret
    hash = calculateHash(auth.s_i, user_acl.salt);

    //compare it with the one stored locally
    if (hash == user_acl.hash){
        result = true;//user is authenticated

        //store hash of the next secret received within auth
        //for the next authenticated access
        localstore.get(index).acl.getUserAcl(auth).hash = auth.hash;
    } else {
        result = true;//the authentication failed
    }

    return result;
}
```

LISTING 4.21: Authentication by Responsible Peer with OTH Approach

The one-time-extension also provides protection against replay attacks, as each authenticated message uses new secrets. The message flow of the OTH authentication is shown in Figure 4.6.

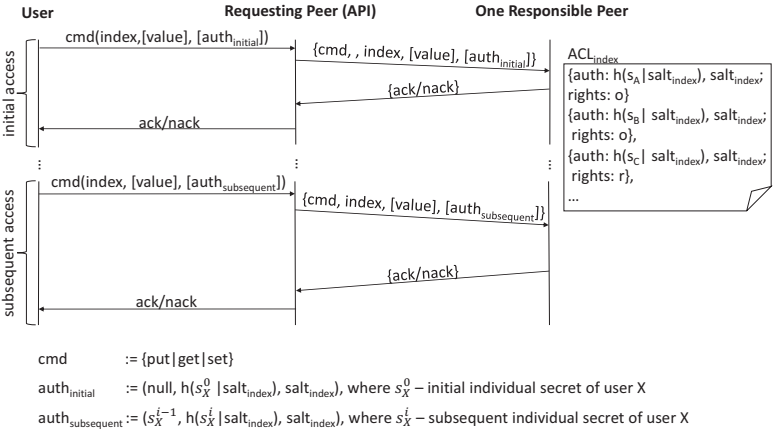

FIGURE 4.6: OTH Message Flow

4.4.4 Properties

With the above presented access control scheme, we achieve all requirements from Section 4.2. By using one of the three proposed authentication mechanisms together with an ACL for each index, we achieve the *Access*-requirement. The read protection is achieved by encrypting the value, and the write protection is achieved by replicating the data to $2k + 1$ responsible peers

and using majority voting. Malicious peers might still write in conflict with the ACL, but their actions remain inert with respect to the entire system.

With our approach, the owner of a DHT entry is uniquely defined and cannot be altered by the attacker (*Ownership*-requirement). To take ownership of an DHT entry, the attacker would need to circumvent the access control or insert herself as the owner during the initial access. However, taking ownership during initial access is secured by sending the request to $2k+1$ peers. If the attacker is not able to subvert the majority of these peers, she cannot modify the initial access with success.

With an individual ACL for each index, we achieve the fine-granularity, i.e., the *Granularity*-requirement. We introduced four different access rights (*o*, *a*, *w*, and *r*) and the new API operation *set* for delegating the rights to other users. The access to the set operation is handled with the same mechanisms as with the put or get operation. Additionally, with the set operation, we can revoke access rights of a user to a certain entry. With the PK approach, we achieve the access revocation by using the put and the set operation. Specifically, the user chooses a new encryption key and re-encrypts the value, which she then stores with a put operation in the DHT. Afterwards, she uses the set operation to set the new rights and the new encrypted data encryption key k_d for each authorized user, i.e., for all as before except for the revoked one. Also with the ZKP and the OTH approach, the value must be re-encrypted and stored in the DHT using a put operation. However, there is no integrated key exchange with the ZKP and OTH approaches. Thus, the user must re-distribute this key out-of-band to all remaining authorized users. The costs for revoking a user are the summed effort of a put and a set operation, which we evaluate in the next section.

To fulfil the *Privacy*-requirement, we use a different authenticator for each index as described above with the authentication mechanism. This way, the attacker cannot derive any information about the owner of an entry or associate entries with persons.

The scalability (*Scalability*-requirement) of our approach mainly depends on the total number of messages used. As we show in the next section, *k*-rAC uses a constant factor ($\approx 2k \ldots 4k$) of additional messages with respect to a DHT without any access control. As this factor does not depend on the number of peers, our approach scales in the same way as the underlying DHT architecture.

We achieve the *Resilience*-requirement by replicating all DHT values to $2k+1$ different indexes. When reading a DHT entry, we perform a majority voting. Hence, a value can only be modified by altering the majority of all replicas.

4.5 Evaluation

In the following, we discuss the security properties and present an analytical model of our approach. To compare the different authentication mechanisms, we determine the overhead by simulating the involved cryptographic operations.

4.5.1 Security Analysis

For the security analysis of *k*-rAC, we use the attacker model described in Section 4.1. As mentioned there, the goal of the attacker is to circumvent the access control. In general, for

any k-rAC API operation, the user accesses $2k+1$ arbitrary but different peers to send requests to $2k+1$ different indexes in the DHT. The $2k+1$ indexes are calculated by a cryptographic hash function that distributes them evenly over the entire DHT space. Hence, the probability for housing these $2k+1$ indexes on different peers increases with the total number of peers online. That is, assuming there are enough peers online, each replica is stored on a different peer. For any DHT, the lower bound to achieve peer distinctiveness is $2k+1$. Whether this can be achieved depends on the specific DHT implementation. The Kademlia protocol can be adapted such that this lower bound is possible (cf. next Chapter). Additionally, the $2k+1$ requests from the requesting peers to the respective responsible peers must be routed over disjoint paths. This disjoint routing can be achieved with the protocol extension of S/Kademlia [15]. Hence, we only need to ensure that there exist enough disjoint paths. We showed in [39] that with Kademlia there are at least c peer disjoint paths, where c is the adjustable parameter for the bucket size. Thus, for k-rAC, we need to set $c \geq 2k+1$. To manipulate a single request, the attacker would need to subvert the corresponding responsible peer or a forwarding peer. If the attacker can subvert at most k different peers, she can modify at most k requests (or the corresponding value). However, as each k-rAC API operation always operates on $2k+1$ indexes, there are always at least $k+1$ of indexes which the attacker cannot modify. Due to the majority voting, only those values with more than k replications are considered valid, rendering the actions of the attacker inert.

Even with more than k subverted peers, our approach gracefully degrades. To manipulate a value, the attacker needs to control the majority of the responsible peers for the corresponding index. However, she cannot freely choose the DHT index for which she is responsible; it becomes unlikely that she can subvert exactly the 'right' peers for a specific index. Thus, she needs to subvert a much higher number of peers to manipulate at least $k+1$ DHT entries.

The PK approach offers also integrity of the transmitted value. By using the signature, the attacker cannot modify the value in transit through the P2P network. This decreases the attacker's possibilities: to modify a value, she necessarily needs to subvert the responsible peers for a specific index – subverting forwarding peers is no longer sufficient. Hence, with the PK approach, we achieve an even better graceful degradation when the attacker can subvert more than k peers.

4.5.2 Performance

Below, we establish an analytical model to determine the performance of our access control scheme with respect to time, message, storage, and computational overhead. We compare our approach, especially our three authentication schemes (AS), with a classical DHT without any access control (DHT w/o AC). To quantify our analytical results, we exemplarily apply well-known and suited cryptographic algorithms to our approach. For this, we use performance data from the literature and from own measurements.

4.5.2.1 Response Time

We assume that a put or a get operation takes on average t ms for a single operation to succeed in a DHT w/o AC. Our access control scheme does not change this, as we build on top of these operations. Especially, we do not change anything about the underlying mechanisms of the specific DHT implementation, e.g., routing. In fact, for our *Resilience*-requirement, we must execute at least $2k+1$ classical operations for any authenticated operation. However, these operations are executed in parallel. Hence, the average time to complete an authenticated operation should not vary significantly from the average time t ms of a single operation in a

AS	Response Time $\mathbb{T}_{AS}$ (ms)	Overhead $\Delta\mathbb{T}_{AS}$
DHT w/o AC	$t \approx 100 \ldots 700$	-
PK	$t \approx 100 \ldots 700$	0
ZKP	$2t \approx 200 \ldots 1400$	$t \approx 100 \ldots 700$
OTH	$t \approx 100 \ldots 700$	0

TABLE 4.2: Response Times

DHT w/o AC. This is only true for the PK and OTH approaches, i.e., $\mathbb{T}_{PK} = \mathbb{T}_{OTH} = t$ ms. With the ZKP approach, we need an additional request/reply for the challenge/response. Hence, the expected average time for ZKP is $\mathbb{T}_{ZKP} = 2t$ ms. Two well-known implementations of DHTs are Kademlia and Chord. Kovacevic et al. [52] analysed the average time t for these implementations yielding $t_{Kademlia} = 100 \ldots 250$ ms and $t_{Chord} = 450 \ldots 700$ ms. We summarize the results in Table 4.2. Accordingly, the PK and the OTH approaches have the same response time as a DHT w/o AC, while the ZKP requires twice as many.

4.5.2.2 Message Overhead

We assume that y messages are required for *put* or *get* in a DHT w/o AC scheme. The PK and OTH approaches use the same number of messages from the requesting peer to a single responsible peer as a put or get operation in a DHT w/o AC. However, since we need to send the request to $2k+1$ different responsible peers, we require in total $m_{PK} = m_{OTH} = (2k+1) \cdot y$ messages. Thus, the overhead of these approaches can be estimated with $\Delta m_{PK} = \Delta m_{OTH} = m_{PK} - y = 2ky$. Since the ZKP approach requires an additional request/reply pair of messages, the total number of messages is $m_{ZKP} = (2k+1) \cdot 2y$. Therefore, its overhead is $\Delta m_{ZKP} = m_{ZKP} - y = 4ky + y$. These results are shown in Table 4.3. Comparing our approaches, PK and OTH use the same number of messages, while ZKP uses roughly twice as many.

The next metric is the message size overhead. In our access control scheme, this is the additional *auth*-parameter in each operation. In the PK approach, it contains a public key, a message ID, and a digital signature, i.e., $auth = \{pk, mid, sign\}$ (cf. Section 4.4.3). We use a 32 bit message ID *mid* for preventing replay attacks. The sizes of the public key and the signature depend upon the used asymmetric cryptographic algorithm. Exemplarily, we use RSA 2048 and ECC (Elliptic Curve Cryptography) 224, which provide roughly the same security. Here, the key lengths refer to the bit lengths of the moduli (RSA) and the subgroup (ECC). However, the actual number of bits required to store a public key for both algorithms is larger, as the public key contains additional parameters [56]. To determine the size of the *auth*-parameter, we used OpenSSL [88] to generate a new key pair for both cryptosystems and stored the public key in the binary DER format [7]. This yields 294 bytes for the RSA key and 80 bytes for the ECC key. The signature in the *auth*-parameter is an encrypted hash of the value concatenated with the message ID, i.e., $sign := \text{encrypt}_{sk}(h(value|mid))$. The size of the signature also depends on the specific hash function used. Here, we used SHA-256 (32 bytes) exemplarily and encrypted the resulting hash with both asymmetric algorithms. This results in 256 bytes for RSA and 63 bytes for the ECC signature. Hence, our total overhead is $\Delta s_{RSA} = 4 + 294 + 256 = 554$ bytes and $\Delta s_{ECC} = 4 + 80 + 63 = 147$ bytes. With the ZKP approach, we only consider the final message when calculating the size overhead. The first and the second message are new messages and already included in Δm_{ZKP}. Additionally, these messages are small: the first message contains the initialization of the protocol, and the second message contains the challenges from the responsible peer which can be represented as n bits. A sound value for n is, e.g., 20. In comparison, a put or a get

AS	Number Δm_{AS}	Size Δs_{AS} (B)
DHT w/o AC	y	-
PK with RSA	$2ky$	554
PK with ECC	$2ky$	147
ZKP	$(4k+1)\cdot y$	$n\cdot 83$
OTH	$2ky$	64

TABLE 4.3: Message Overhead

message contains at least the DHT index, e.g., 160 bits. The final message contains n responses to the challenges from the second message (cf. Figure 4.5). A single response represents a large number, big enough to make the calculation for the discrete logarithm infeasible, e.g., 200 decimal places or 665 bits $\approx$ 83 bytes. This yields an overhead of $n\cdot 83$ bytes. With $n=20$, the overhead is 1660 bytes. Finally, with the OTH approach, each message contains two additional hashes. By using again SHA-256 as the hash function, we get an overhead of $2\cdot 32 = 64$ bytes. We summarize the results of this metric in the second column of Table 4.3. Comparing our approaches, OTH and PK with ECC use the smallest overhead with respect to message size. The overhead of ZKP is significantly larger for realistic n. All are independent of k.

4.5.2.3 Storage Overhead

With respect to a DHT w/o AC, the storage overhead differs for the responsible peers and users. We assume that the used DHT uses a 160-bit hash function (e.g., SHA-1). Hence, for storing the index to a specific value in the DHT w/o AC, the storage requirement for any user is 20 bytes. Thus, we do not include these 20 bytes in the calculation of the storage overhead for k-rAC. Furthermore, we assume that each responsible peer uses a data structure (e.g., a hash table) to locally store the DHT values.

User's Storage Overhead We assume, that the user has access to i different DHT entries. Her storage overhead is comprised of a constant and a per-entry overhead.

Using the PK approach, the user stores a different public/private key pair for each entry she has access to. Similarly to our analysis of the message size overhead, we used OpenSSL to determine the size of a RSA and a ECC key pair in DER format, i.e., 1192 bytes for RSA and 113 bytes for ECC. For each DHT entry, she also stores a message ID for the freshness of the messages. We assume a 32-bit (4 bytes) message ID. Hence, the total storage overhead for the user is $\mathbb{S}_{RSA} = 1192+4 = 1216$ bytes or $\mathbb{S}^U_{ECC} = 113+4 = 137$ bytes.

With the ZKP approach, the user stores her individual secret and the associated modulo for each DHT entry she has access to. For a secure secret, we require $\approx$ 45 bytes. As the modulo m is a product of two large numbers, we require $\approx$ 83 bytes to store it. Hence, the storage overhead for a single index is $\mathbb{S}^U_{ZKP} = 45+83 = 128$ bytes.

With OTH, the user stores once the master key (we assume 32 byte). For each DHT entry she has access to, she stores also a salt (16 byte) and $2k+1$ current secrets (each 32 bytes). Hence, the total storage overhead is $\mathbb{S}^U_{OTH} = 32+i\cdot(16+32\cdot(2k+1))$ bytes. Hereby, the storage overhead for a single index is $\Delta\mathbb{S}^U_{OTH} = 16+32\cdot(2k+1) = 17+64k$ bytes.

In Table 4.4, we summarize the storage overhead in dependence of the resilience k for a user having access to i different DHT entries. In terms of storage overhead for the user, the ZKP

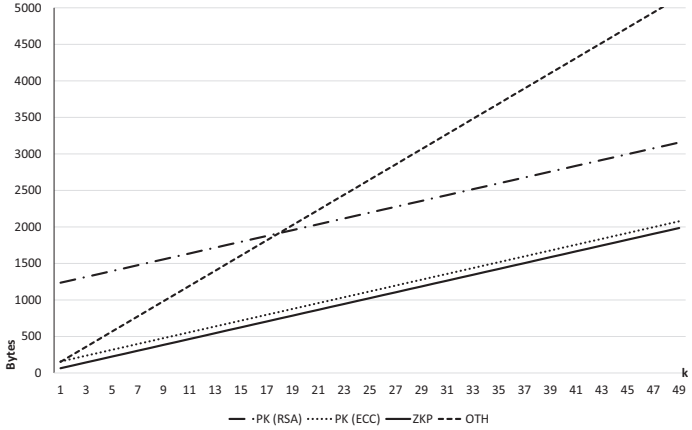

FIGURE 4.7: Comparison of User's Storage Overhead $\Delta\mathbb{S}^{U}_{AS}$

approach is the best choice: The overhead is smallest and independent of k. The second best choice is the PK approach with ECC. Here, the storage overhead is also independent of k, and it grows linearly with the amount of DHT entries the user has access to. However, it also includes the key distribution, making it the better choice for some scenarios. On the other hand, the PK approach with RSA should be avoided due to the overall higher storage requirement. Finally, OTH has the largest overhead, as it grows linearly with both parameters, i.e., the resilience k and the amount of entries. For $k > 18$, it even requires more storage than the PK approach with RSA (cf. Figure 4.7).

Peer's Storage Overhead With the PK approach, the responsible peer stores additional meta information for each authorized user in the ACL according to Listing 4.5: the encrypted symmetric key, the user rights, and the *auth*-parameter. We assume a symmetric key of 128 bits (16 bytes) and one byte for the user rights. Encrypting the symmetric key yields 256 bytes for RSA and 64 bytes for ECC. The size of the *auth*-parameter depends on the used authentication mechanism. With the PK approach, *auth* contains the public key and a sliding window of accepted message IDs *mid*. We used 32 bits for *mid* and a window of 32 entries, i.e., $4 \cdot 32 = 128$ bytes. As before, the size of the public key is 294 bytes for RSA and 80 bytes for ECC. Hence, the storage overhead for each ACL item is $\Delta\mathbb{S}^{P}_{RSA} = 256 + 1 + 128 + 294 = 679$ bytes for RSA and $\Delta\mathbb{S}^{P}_{ECC} = 64 + 1 + 128 + 80 = 273$ bytes for ECC.

With the ZKP approach, the peers do not store a symmetric encryption key, since they exchange it out-of-band. Hence, with the ZKP approach, the peers store the square of the secret, the modulo, and the user rights, i.e., $\Delta\mathbb{S}^{P}_{ZKP} = 83 + 83 + 1 = 165$ bytes for each ACL item.

Similarly, with the OTH approach, the peer stores a hash value (23 bytes), a salt value (16 bytes), and the user rights (1 byte). Thus, the storage overhead is $\Delta\mathbb{S}^{U}_{OTH} = 32 + 16 + 1 = 49$ bytes.

In Table 4.4, we summarize the storage overhead for a peer with an ACL containing u items. The storage overhead for the peers is independent of k and scales linearly with the number of ACL items. We get the lowest overhead with the OTH and the ZKP approaches, while the PK

AS	$\Delta\mathbb{S}^{U}_{AS}$ (B)	$\Delta\mathbb{S}^{P}_{AS}$ (B)
PK with RSA	$i \cdot 1216$	$u \cdot 679$
PK with ECC	$i \cdot 137$	$u \cdot 273$
ZKP	$i \cdot 128$	$u \cdot 165$
OTH	$32 + i \cdot (17 + 64k)$	$u \cdot 49$

TABLE 4.4: Storage Overhead

with ECC approach use up to six times more storage, and the PK with RSA uses even up to fourteen times more storage.

4.5.2.4 Computational Overhead

To evaluate the computational overhead, we differentiate three metrics: First, we analyse the initial effort for the user when creating a DHT entry, i.e., $\mathbb{U}^{0}_{AS}$. Secondly, we determine the effort for the user during any subsequent authenticated access, i.e., $\mathbb{U}^{i}_{AS}$. Finally, we analyse the computational overhead of the responsible peers, i.e., $\mathbb{P}_{AS}$. For each of these metrics, we consider the API operations *put*, *get*, and *set* separately as they cause different effort.

User's Initial Effort Initially, every DHT entry is empty and does not have an owner. Albeit a get operation for an empty entry is not forbidden, it returns a *null* value. Hence, in our system an initial get operation causes no overhead with respect to a DHT w/o AC. Similarly, there is no overhead for the set operation on initial access, as it is by definition not possible.

For a put operation using the PK approach, the user initially creates a key pair for the accessed index. This key generation operation (KG) depends on the used asymmetric cryptographic algorithm, e.g., with RSA, it involves finding two large primes. For the initial access, we do not sign the data object, as there is no way for a peer to securely verify this signature (the corresponding public key is in the same message). We achieve the security of the initial put operation with the k-resilience property of our approach. For the read access, the value is encrypted with a symmetric encryption algorithm (SO). For a different ACL items, we encrypt this symmetric encryption key with the public key of each authorized user (AO_{pk}). At the very least, the corresponding encryption key is encrypted with the owner's public key. Due to our clear separation of managing data and managing the ACL, this prepared ACL must be sent with a subsequent set operation. Hence, the initial complexity of a put operation for the user is $\mathbb{U}^{0}_{PK} = KG + SO + a \cdot AO_{pk}$.

For an initial put operation using the ZKP approach, the user calculates $m = p \cdot q$, generates a random number, and calculates its modular square (cf. Figure 4.4). Assuming that the effort for the multiplication of two numbers and generating a random number is negligible, we only consider the modular operations (MO). The value is encrypted with an SO operation. Hence, the user's initial effort for *put* is $\mathbb{U}^{0}_{ZKP} = MO + SO$.

With the OTH approach, the user initially calculates $2k+1$ individual hashes (HO) for *put*. Similarly as before, the value is encrypted with an SO operation. In summary, the user's initial effort for the put operation is $\mathbb{U}^{0}_{OTH} = (2k+1) \cdot HO + SO$.

User's Subsequent Effort For any subsequent get operation, the user does not have to authenticate, since the read access control is based on the encrypted value. She only has to retrieve the value and decrypt it. Hence, the user's subsequent effort for a get operation includes for all three authentication schemes an SO operation for decrypting the value. With the ZKP and the

AS	User $\mathbb{U}^0_{AS}$
PK	$KG + SO + a \cdot AO_{pk}$
ZKP	$MO + SO$
OTH	$(2k+1) \cdot HO + SO$

TABLE 4.5: Initial User Computational Overhead for *put*

AS	User $\mathbb{U}^i_s$		
	get	put	set
PK	$AO_{sk} + SO$	$SO + a \cdot AO_{pk} + HO + AO_{sk}$	$a \cdot AO_{pk} + HO + AO_{sk}$
ZKP	SO	$SO + (2k+1) \cdot n \cdot 1.5 \cdot MO$	$(2k+1) \cdot n \cdot 1.5 \cdot MO$
OTH	SO	$SO + (2k+1) \cdot HO$	$(2k+1) \cdot HO$

TABLE 4.6: Subsequent User Computational Overhead

OTH approach, the decryption key is distributed out-of-band. Thus, there is no additional effort in these cases. However, for the PK approach, the user first needs to decrypt the encryption key using an asymmetric cryptographic operation with the her private key (AO_{sk}). Hence, the user's subsequent effort can be determined with $\mathbb{U}^i_{PK} = AO_{sk} + SO$ for the PK approach, and $\mathbb{U}^i_{ZKP} = \mathbb{U}^i_{OTH} = SO$ for the ZKP and the OTH approaches.

For a put operation with the PK approach, we encrypt the value with an *SO* operation. After that, the encryption key is encrypted with the public keys of all authorized users, i.e., $a \cdot AO_{pk}$. Again, this prepared ACL must be sent with a subsequent set operation. Here, each message also contains a signature which requires hashing (*HO*) and an asymmetric operation with the private key (AO_{sk}). Hence, the user's total subsequent effort can be determined by $\mathbb{U}^i_{PK} = SO + a \cdot AO_{pk} + HO + AO_{sk}$.

With the ZKP approach, we also encrypt the data object for a put operation. To authenticate the operation, the user calculates n responses to the challenges from each of the $2k+1$ peers. For this, she chooses a random number r and calculates its modular square (n MO). Depending on the peer's challenge (i.e., 1 or 0), the user replies with the modular product of her secret with r or only r. On average, there will be $n/2$ 1s in the challenge yielding additional $n/2$ MOs. This involves on average $n \cdot 1.5 \cdot (2k+1)$ modular operations (*MO*). Thus, the user's subsequent effort for a put operation is $\mathbb{U}^i_{ZKP} = SO + (2k+1) \cdot n \cdot MO$.

For *put* with the OTH approach, we also require to encrypt the value. For the authentication, we need to calculate $2k+1$ individual hashes. Hence, the user's subsequent effort for a put operation is $\mathbb{U}^i_{OTH} = SO + (2k+1) \cdot HO$.

In all three approaches, the only difference for *set* is the lack of the value encryption. Furthermore, the user performs the majority voting over all received results. However, this operation is negligible with respect to computational complex operations like modular arithmetic or hash calculations. We summarize our analytical results of the user's subsequent effort in Table 4.6.

Peer's Effort The final metric is the computational overhead for the responsible peers. Due to the encrypted value, we do not need to authenticate the get requests. Hence, there is no overhead for the responsible peers in comparison to a DHT w/o AC.

For a put and a set operation, the responsible peer needs to authenticate the user. For the peer, there is no difference whether it is a put or a set operation. With the PK approach, the peer must verify the signature to authenticate the request. This involves decrypting the signature with

AS	Peers $\mathbb{P}_s$		
	get	put	set
PK	0	$AO_{pk} + HO$	$AO_{pk} + HO$
ZKP	0	$n \cdot 1.5 \cdot MO$	$n \cdot 1.5 \cdot MO$
OTH	0	HO	HO

TABLE 4.7: Peer Computational Overhead

the public key and hashing the value. Hence, the resulting complexity is $\mathbb{P}_{PK} = HO + AO_{pk}$. With the ZKP and the OTH approaches, the integrity of the value is not verified (we rely on the k-resilience). Hence, there is no hash operation involved. The authentication with the ZKP approach involves calculating n times the modular square of the received response y (n MOs), and comparing it either with the modular product $v \cdot r^2$ or with r^2, i.e., $y^2 \equiv v^c \cdot r^2 \pmod{p}$ for $c \in \{0,1\}$ (cf. Figure 4.5). Whether we need to perform the modular product depends on the challenge c. Hence, assuming that on average there are about half of the challenge bits 1s, we need to perform $n/2$ MOs. This yields an average complexity of $\mathbb{P}_{ZKP} = n \cdot 1.5 \cdot MO$. Finally, with the OTH approach, a responsible peer only performs one hash calculation to verify the authenticity of the user, i.e., $\mathbb{P}_{OTH} = HO$. We summarize our results in Table 4.7.

4.5.2.5 Simulation

To quantify the results of our analytical analysis and to compare the three proposed authentication mechanisms, we implemented the cryptographic operations by using the built-in cryptographic engines from the cryptography architecture [2] of Java SE 7. We measured the time for each operation by taking the average of one Million executions of the operation on a workstation (Intel i7-4900 MQ, 2.8 GHz, 32 GB RAM). Although the absolute values will vary on different computer systems, the measurements allow to compare the proposed authentication mechanisms. For the ECC-measurements, we used the elliptic curve integrated encryption system with AES ("ECIESwithAES"), which uses ECC together with AES to build an asymmetric cryptosystem. In cases where only a signature is required, it would suffice to use the Elliptic Curve Digital Signature Algorithm (ECDSA). However, we evaluated this and found that ECDSA is slightly slower than ECIESwithAES. Therefore, we did not follow this further. The time for SO and HO depends on the number of bytes encrypted or hashed. We used 32 kB as an average size with AES 128 and SHA 256. We chose those two algorithms, as they provide strong security, are widely used nowadays, and well analysed [48]. We assume that values stored in a DHT are rather small and usually fit into a UDP packet (max 64 kB). However, even with bigger data objects, the time needed to perform *SO* or *HO* is small in comparison to other operations. This is probably caused by the fact that the used processor uses hardware acceleration for these operations (AES-NI). Nevertheless, we acknowledge that for big data objects (several megabytes to gigabytes) this operation can become a significant factor [34]. For *MO*, we used a big integer number with 200 decimal places, i.e., 665 bits. With this size, the calculation of the discrete logarithm is impractical. In Table 4.8, we summarize the measured times in microseconds.

There are three parameters in our system, namely n, k, and a. The parameter n is the number of challenges with the ZKP approach. With $n = 20$, there is only one in a Million ($1 : 2^{20}$) chance of fooling the AS. The parameter k is the number of tolerated subverted peers. This parameter highly depends on the possibilities of the attacker to add peers under his control to the network.

Operation	Time
KG (RSA 2048)	317092 μs
KG (ECC 224)	685 μs
AO_{sk} (RSA 2048)	5135 μs
AO_{sk} (ECC 224)	170 μs
AO_{pk} (RSA 2048)	148 μs
AO_{pk} (ECC 224)	380 μs
HO (SHA 256)	176 μs (32 kB)
SO (AES 128)	33 μs (32 kB)
MO (665 Bits)	4 μs

TABLE 4.8: Measurement Results

For our evaluation, we choose exemplarily $k = 20$. Finally, the parameter a is the amount of authenticated users in an ACL. Exemplarily, we used $a = 10$.

In Figure 4.8, we present the resulting computational overhead of our authentication mechanisms. For this, we used the measured times from Table 4.8 and the values for the parameters from above together with the analytical model. According to these results, the overhead for the peer varies from 100 μs to approximately 600 μs. This overhead is negligible in comparison to approximately 200 *ms* roundtrip times for DHT operations [52].

When comparing the computational overhead of the get operation, we must consider that the PK approaches include the key distribution. Thus, a higher overhead is acceptable as they offer additional features. Nevertheless, we see in Figure 4.8 that the overhead for the get operation is negligible for all authentication mechanisms. The highest overhead for the get operation is from the PK approach with RSA, where a get operation needs approximately 5 *ms*. Although this is still acceptable, the PK approach with RSA additionally generates a notably higher overhead for put or set during the initial access. Therefore, we suggest to use ECC instead of RSA and consider in the remainder of this analysis only the PK approach based on ECC.

For *put* and *set*, the OTH approach generates the highest overhead. This is an interesting result, as the overhead for this authentication mechanism is similar to the overhead of hash chains. Hash chains are usually preferred over asymmetric cryptography due to their lower computational overhead. However, with OTH, we need $2k+1$ individual secrets to tolerate k subverted peers. Therefore, its advantage disappears for a certain k. By ignoring the overhead for the key distribution using the PK with ECC approach (by setting $a = 0$), PK outperforms OTH already for $k \approx 0.5$. Also for the ZKP approach, we determined that for a $k > 1$ the PK approach with ECC outperforms ZKP for any subsequent access. However, the major advantage of ZKP is its lowest overhead during initial access. Additionally, on subsequent accesses, ZKP needs approximately 1.5 times less resources than OTH. However, this is just the computational overhead – ZKP requires an additional message pair (cf. Section 4.5.2.2).

4.5.2.6 Performance Summary

To sum up the performance results, none of our three schemes is optimal with respect to all our metrics. Overall, the PK approach with ECC is a very flexible approach that offers most features with acceptable overhead. Except for message overhead, it is independent of k. OTH has the smallest communication overhead and low storage overhead for responsible peers. However, the storage overhead for users is rather high. Therefore it is best suited for a scenario with small,

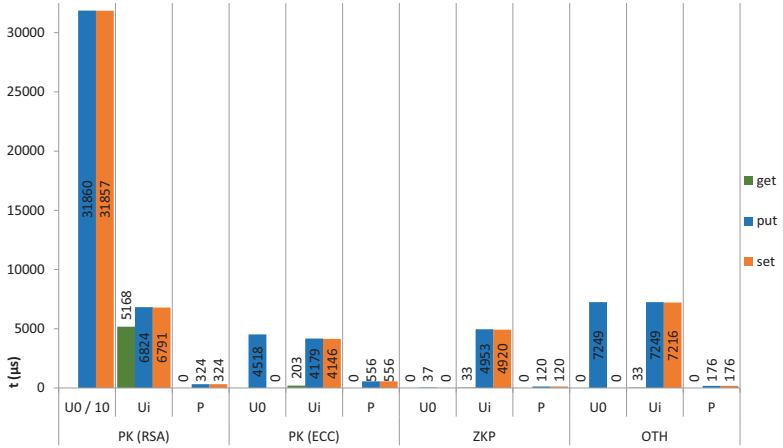

FIGURE 4.8: Comparison of Authentication Mechanisms

embedded peers that are connected wirelessly. ZKP is best suited for scenarios with few updates and many read accesses.

4.6 Summary

In this chapter, we presented *k*-rAC, a novel fine-grained *k*-resilient access control for DHTs. Currently existing DHTs cannot be used in privacy aware applications because of the lack of a reliable and practical access control. With our generic access control, we open the door for using a DHT in such applications. Specifically, we can rely on *k*-rAC to realize the DRS with any of the four proposed revocation methods. With *k*-rAC, the read or write access to each index in the DHT can be regulated individually. We determined the owner of a DHT entry by the initial access and provided privacy-aware mechanisms to delegate access rights to other users. The security of our approach is based on *k*-resilience, i.e., the access control cannot be circumvented as long as the attacker is not able to subvert more than *k* peers. Additionally, even in cases when the attacker is able to subvert more than *k* peers, our approach gracefully degrades. This is caused by the fact that the attacker needs to subvert specific peers in order to circumvent the access control for a specific index.

In our evaluation, we compared three different authentication mechanisms in terms of response time, message, storage, and computational overhead. We showed that the caused overhead for all three approaches is acceptable. Since the approaches have different advantages and disadvantages, it depends on the requirements of a specific scenario (in our case, of the particular revocation method) which of the authentication mechanisms should be used. The suitable authentication mechanism can be determined with the presented analytical model.

We published our generic approach for the *k*-resilient access control in a DHT in [51] and presented it at the 12th International Workshop on Frontiers in Availability, Reliability and Security (FARES 2017).

5

DHT Modifications

In Chapter 2, we motivate our decision to use a DHT for the DRS, where we store the variable *statement* to enable information exchange between owners and providers. To protect *statement* against unauthorized accesses, we developed *k*-rAC (cf. Chapter 4) – a generic approach for an access control in a decentralized and distributed architecture given with any DHT. However, the network structure in existing DHT protocols differ from each other. For instance, with Chord [81], the peers are arranged in a ring structure, and with CAN [73], they form a d-dimensional torus. With Kademlia [60], the XOR arithmetic, which is used for routing, forms an abelian group. From all existing DHT protocols, Kademlia is the only one used in software applications, thereby proving its applicability. It is already successfully used in various products, e.g., the file distribution system BitTorrent [4], the instant-messaging and video-calling platform Tox [5], or the portal creation software Osiris [3]. Therefore, due to its proven applicability in existing applications, we decided to analyse the suitability of Kademlia for the DRS. In the following, we first present the general characteristics of Kademlia. Then, we analyse the required modifications that must be applied to fulfil the requirements for *k*-rAC.

5.1 Kademlia

With Kademlia [60], each peer and each stored data object is identified by a numerical ID[1] with the fixed bit-length b. These identifiers are generated from the network address of a peer or the data object respectively using a cryptographically secure hash function to equally distribute the identifiers in the identifier space. Each peer maintains a routing table with identifiers and network addresses of other peers, the so-called *contacts*. The routing table consists of b so-called c[2]-buckets to store the contacts of the peer. The buckets are indexed from 0 to $b-1$, and the contacts are distributed into these buckets depending on the distance of their identifiers ID_i. The distance between two identifiers is computed using the XOR metric, meaning that for two identifiers ID_a and ID_b the distance is $dist(ID_a, ID_b) = ID_a \oplus ID_b$, interpreted as an integer value. The buckets are populated with those contacts ID_i fulfilling the condition $2^i \leq dist(ID, ID_i) < 2^{i+1}$ with i being the bucket index. This means that the bucket with the highest index covers half of the id space, the next lower bucket a quarter of the id space, and so on. The maximum number

[1] This ID corresponds to *index* in *k*-rAC.

[2] The authors of Kademlia use character k for this parameter. However, we refer to it as c to avoid confusion with our k which we defined for *k*-rAC.

of contacts stored in one bucket is c. Next to b and c, another defining property of a Kademlia setup is the request parallelism α, which determines how many contacts are queried in parallel when a peer locates another peer or retrieves/stores a data object. Greater values of α can speed up the operation while at the same time increasing the resulting network load. The staleness limit s determines how often in a row the communication with a contact must fail, so that it is considered stale and is removed from the routing table. Greater values of s delay the removal of actually stale peers by waiting for more failed communication attempts, while small values might lead to a frequent removal of non-stale peer due to a disturbed communication channel. The Kademlia authors set the default values $b = 160$, $c = 20$, $\alpha = 3$, and $s = 5$.

The peers of a Kademlia network locate resources (other peers, data objects) by means of their ID. Given a target ID, a peer queries α peers from its routing table closest to this ID. Those, in turn, answer with their own list of closest peers, which can be used in new queries. This way, the requesting peer iteratively gets closer to the target ID. This process ends when a number of c peers have been successfully contacted, or no more progress is made in getting closer to the target ID. For the protocol, the authors of Kademlia introduce four remote procedure calls (RPCs): *ping* for probing whether a peer is online, *store* for instructing a peer to store an index-value pair, *find_node* for retrieving c peers closest to the target ID, and *find_value* for retrieving the value for a given ID. Hence, they differentiate between procedures for locating c closest peers (*peer lookup*) and locating a value (*value lookup*). Furthermore, in Kademlia, the peers periodically republish their ID-value pairs to prevent that values become unavailable due to the network churn. The second purpose of the re-republishing is to cover new joined peers that are closer to the indexes of the republished values.

5.2 Requirements to Kademlia Based on k-rAC

As reasoned in the previous chapter (cf. Section 4.3), we rely on the k-resilience by designing the access control for a generic DHT. To provide the k-resilience in a DHT reliably, we must perform the DHT operations on $2k+1$ different responsible peers. Furthermore, we require that each request is routed to the responsible peers over disjoint paths. Based on these two general requirements to a DHT, we identify the specific goals for gaining a reliable access control with the k-rAC approach in Kademlia as follows:

Disjoint paths: All requests must be sent to the $2k+1$ responsible peers over $2k+1$ disjoint paths.

Disjunct peers: The $2k+1$ responsible peers must be indeed disjunct.

Reduction of requests: Since in Kademlia each request is sent to multiple peers per default, this routing characteristic should be used as part of the $2k+1$ requests needed for k-rAC.

5.3 Adaptation of Kademlia to k-rAC

In the following, we identify which modifications must be applied to Kademlia for achieving the $2k+1$ disjoint paths, the $2k+1$ disjunct peers, and the reduction of requests. Hereby, we assume that there are more than $2k+1$ participating peers in the network.

5.3.1 Disjoint Paths

To achieve that a request is sent to all $2k+1$ responsible peers over $2k+1$ disjoint paths, we must ensure that there is a sufficient number of paths available in the network, i.e., the network connectivity κ is sufficiently high. In [39], we analyse the connectivity of the overlay network Kademlia in multiple simulated scenarios. Hereby, we transfer its network structure into the domain of graph theory by creating a connectivity graph. The representation of the network structure as a connectivity graph enables the application of concepts and algorithms from graph theory to analyse properties of the network. To simulate Kademlia, we use the network simulation software PeerSim [63]. We added to it own software components to provide functionality for creating network churn (addition and removal of peers) as well as for requesting data objects and disseminating information into the network.

In a simulation, we persist the connectivity graph of a network at pre-defined time stamps to calculate the graph connectivity over time. For that purpose, we interrupt the simulation and save the current contents of the routing tables of all network peers to disk into a snapshot file. We use this snapshot file to transform the connectivity graph by using the mathematical foundations (specifically, we use Even's algorithm [28]).

To determine how different environments and protocol parameters influence the connectivity of the network, we devise in total eight dimensions for the simulations, i.e., network size, network churn, network traffic, message loss, the Kademlia bucket size c, the parallelism factor α, the bit-length b, and the staleness limit s. Regarding k-rAC, we are especially interested in the influence of the bucket size c on the network connectivity. In Kademlia, the bucket size is directly responsible for the number of contacts a peer can keep in its routing table. To determine its effect on the network connectivity, we use four different values for c, i.e., $c \in \{5, 10, 20, 30\}$. Hereby, we analyse the connectivity in two networks of different sizes: in a small network with 250 peers and in a large one with 2500 peers. In fact, assuming most of the providers use the DRS and their servers are part of the DHT, we anticipate a network with tens of thousands peers. However, to simulate such a large network is not feasible, as even our simulations in a network with 2500 peers take weeks. Therefore, we perform our simulations in the networks with 250 and 2500 peers to extrapolate a trend from the results. Furthermore, we consider the impact of different churn scenarios in these two networks. In the scenario (1/1), we add one peer and remove one peer every minute. Similarly, in the scenario (10/10), we remove ten peers and add ten peers per minute. The add/remove actions happen at random points in time within each minute range. We chose these high churn rates to get a clear indication of effects related to churn in our simulations. A numerical comparison of the 1/1 and 10/10 churn scenarios is given in Table 5.1 . It shows the mean and the relative variance (RV), i.e., Mean/RV, of the minimum connectivity during the churn phase for the simulations with data traffic for all four c values. As the RV values in Table 5.1 show, the increase in churn from 1/1 to 10/10 leads to an increased RV in all simulations. The exception is the network size 2500 with $c = 5$, where the minimum connectivity is zero throughout the whole churn phase for both churn scenarios.

The results from our simulations show that the network connectivity of Kademlia κ strongly correlates with its bucket size c. In most simulations, κ was equal to or higher than c. In stressful scenarios like depicted in Fig. 5.1 where we simulate the 10/10 churn, it dropped slightly below c. To achieve a certain resilience level k for the DRS using k-rAC with Kademlia, we require a network connectivity $\kappa > 2k+1$. With our results, we determined that the bucket size needs to be set to a value greater than $2k+1$, i.e., $c > 2k+1$. Nevertheless, especially for scenarios with strong churn, the resilience level cannot be guaranteed. In situations with no or few peers joining the network, the network connectivity was equal or greater than c. Due to the peer

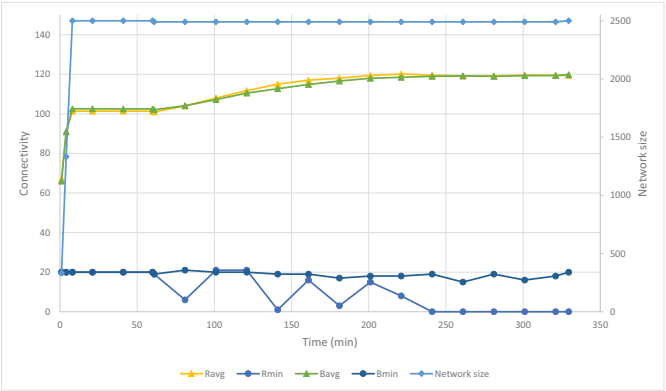

FIGURE 5.1: Connectivity. Network Size 2500, c=20, Churn 10/10 per min, with Data Traffic

Size	c	Churn	Mean	RV
250	5	1/1	3.49	0.63
		10/10	1.93	0.75
	10	1/1	10.12	0.17
		10/10	9.22	0.23
	20	1/1	22.22	0.36
		10/10	20.53	0.39
	30	1/1	32.84	0.34
		10/10	32.78	0.62
2500	5	1/1	0.00	0.00
		10/10	0.00	0.00
	10	1/1	9.30	0.13
		10/10	7.38	0.21
	20	1/1	22.06	0.07
		10/10	16.62	0.16
	30	1/1	31.35	0.10
		10/10	25.73	0.24

TABLE 5.1: Mean as Relative Variance of Connectivity for Various Bucket Sizes in Combination with Different Churn

lookup procedure, which supports the learning of new contacts, the presence of network traffic greatly enhances the network connectivity, both in terms of absolute connectivity and the time to reach this connectivity. However, the churn can negatively influence the connectivity: While the churn can even have a positive effect on the average connectivity, the minimum connectivity drops significantly below c with stronger churn and shows greater variance relative to its mean. Hence, to ensure a reliable connectivity, we have to provide a network with a low churn.

In summary, assuming we have a network with a low churn, we can achieve that the $2k+1$ requests associated with a DHT operation are sent over $2k+1$ disjoint paths. To implement the disjoint routing to the responsible peers, we can rely on the protocol extension of S/Kademlia [15].

5.3.2 Disjunct Peers

In Kademlia, each storing request starts with the peer lookup by querying in parallel α peers to get close to the target ID (cf. Section 5.1). Each queried peer responds with a list of c peers that have, from its perspective, the closest ID to the given target ID. The requesting peer recursively queries the α new learned peers from these lists. The peer lookup procedure terminates when the requesting peer has queried and gotten responses from the c closest peers it has seen. We use this protocol property to ensure that a value is stored with different responsible peers. In line with this, we consider the lookup procedure of closest peers in Kademlia as corresponding to the determination of the responsible peers in k-rAC, i.e., determining the peers that will store the value during this procedure. To achieve our goal for disjunct peers by storing a value using the peer lookup procedure, we must set the value for the c-parameter so that it matches our requirement of a resilience against up to k malicious peers.

As described in Section 5.1, the value lookup is similar to the peer lookup, but it terminates immediately when a peer returns the requested value. However, for the majority voting with k-rAC, we need at least $2k+1$ responses from $2k+1$ disjunct peers. To achieve this with Kademlia, we modify the value lookup in such a way that it terminates when the requesting peer receives at least $2k+1$ values. Additionally, while processing the responses, we choose from all received responses the $2k+1$ peers so that no peer ID is repeated, i.e., we accept each ID only onetime. For that, as with the peer lookup, we must set the c-parameter appropriately.

From this perspective, we are able to achieve our requirement for disjunct responsible peers, as we can control which peers stores the given value and from which peers we accept received values by checking their IDs. To achieve a sufficient number of disjunct peers while the storing and retrieving procedures, we set c appropriately. However, at this point, we do not know whether the Kademlia routing protocol enables reaching all the c value replicas when storing them on c peers. Hence, to set c appropriately, we must know how many replicas do we receive by retrieving a certain value from at most c peers after we store it on c peers with Kademlia. In this context, we introduce the so-called *replica return ratio* (RRR) to denote the ratio between the stored value replicas upon putting a value ($\mathbb{N}_{put}$) and returned value replicas upon getting this value ($\mathbb{N}_{get}$). Thus, to calculate the RRR, we use Equation 5.1:

$$RRR = \frac{\mathbb{N}_{get}}{\mathbb{N}_{put}} \tag{5.1}$$

Accordingly, a RRR of 100% corresponds to c received value replicas upon storing the value on c peers. To determine how to set c to achieve a required number of disjunct peers for a certain k resilience level, we need to analyse RRR in Kademlia. In the next section, we evaluate Kademlia regarding the RRR and deduce from the evaluation results the suitable c for k-rAC.

5.3.3 Reduction of Requests

In a generic DHT, we cannot control which peer is responsible for a certain ID. For instance, with Chord [81], values with adjacent IDs are likely to be stored on the same peer. However, in Kademlia, we can deterministically control this. With our modification from the previous section, we prevent that some replicas of a value are stored on the same peer multiple times. Thus, we implicitly ensure that each replica is stored on a different peer. Additionally, the peers that store replicas of a certain value cannot belong to the same entity, as it is not possible to

influence the peer ID assignment due to the ID generation using a cryptographic hash function. This way, an attacker cannot create many IDs in the same area to control multiple replicas – she cannot influence where peers under her control are placed in the Kademlia network. Moreover, if a large network segment with consecutive IP addresses fails (e.g., the network of an Internet provider), we can still rely on the robustness of the Kademlia network – replicas are distributed randomly, and the failure of some peers does not necessarily affect all replicas of a value. Therefore, with Kademlia, we do not need to additionally hash the indexes as proposed in Section 4.3.2. Kademlia provides per default features that we need for our approach – a reliable distribution of IDs and a robust network.

On the one side, in *k*-rAC, we perform each API operation $2k+1$ times and provide a general approach to determine the $2k+1$ responsible peers which is applicable for any DHT (cf. Equation 4.1). Hence, when a user executes a single put, get, or set API operation, *k*-rAC performs this operation $2k+1$ times to achieve the *k*-resilience. On the other side, in Kademlia, a user stores a value on *c* peers that are closest to the ID of this value. Hereby, a put operation involves the RPCs *find_node* and *store*: After the requesting peer located *c* closest peers to the target ID using *find_node*, it sends each of them a store RPC. Bringing together the both components *k*-rAC and Kademlia without any DHT modifications would lead to the multiplied replication: The total number of operations for storing a value $\mathbb{N}_{store}$ is the multiplication of $2k+1$ put operations (i.e., API operations) needed with *k*-rAC by *c* storing operations performed with a DHT (i.e., DHT operations) upon a single put operation, i.e., $\mathbb{N}_{store} = (2k+1) \cdot c$.

Furthermore, a get operation triggers the value lookup procedure by using the RPC *find_value*. This procedure is also performed recursively the same way as the peer lookup [60], but it terminates as soon as a peer returns the requested value. As described in the previous section, we modify the termination condition of the value lookup for the majority voting, i.e., it terminates after requesting *c* peers. Therefore, the total number of operations for retrieving a value consists of $2k+1$ get operations needed with *k*-rAC multiplied by *c* retrieve requests that are needed with the modified Kademlia, i.e., $\mathbb{N}_{retrieve} = (2k+1) \cdot c$.

With both API operations *put* and *get*, the multiplied number of DHT operations increases the effort unnecessarily, as we could merge them and, hence, reduce the total number of operations needed per one API operation. With the Kademlia modifications from Section 5.3.2, we store and retrieve a value on/from *c* disjunct peers. In the ideal case for our approach, the RRR in Kademlia is 100%, and the parameter *c* corresponds to the $2k+1$ of *k*-rAC. Then, we would need to perform an API operation only one time instead of $2k+1$ times. For the put operation, it would mean that we execute only one put operation to store a value, but it is still stored at $2k+1$ different peers. Similarly for the get operation, we would perform only one get operation to receive $2k+1$ replicas from $2k+1$ disjunct peers. However, due to the distributed network architecture and the related routing, this case is rather improbable. We must assume that $\mathbb{N}_{get} \leq \mathbb{N}_{put} \leq c$. To determine the real RRR, we have to analyse the behaviour of Kademlia by performing put and get operations and evaluate the ratio between $\mathbb{N}_{get}$ and $\mathbb{N}_{put}$. Evaluating Kademlia, our goal is to determine how we should choose the value for the *c*-parameter to achieve a certain *k*-resilience level, i.e., on how many peers we should store a value to retrieve thereupon $2k+1$ replicas.

5.3.3.1 Evaluation

In the following, we evaluate the put and get operations in a simulated Kademlia network. Specifically, we evaluate on how many peers a value is stored upon a single put operation, and

how many replicas of this value we can afterwards retrieve with a single get operation. Hereby, we first describe the evaluation environment and the simulations scenarios. Then, we present the achieved results and discuss them.

Environment

To simulate Kademlia, we use the network simulation software PeerSim [63] extended by own software components to simulate storing and retrieving processes in simulations with network churn as well as without it (cf. Section 5.3.1). Furthermore, we implemented the modification of the termination condition in the value lookup procedure (cf. Section 5.3.2): A get operation terminates after the requesting peer successfully contacted c peers. A successful contact means a received response from the contacted peer. That response contains either the requested value or a notification that this peer does not have this value. In this connection, a requested peer additionally sends a list of closest peers to the given ID within its response. This way, the requesting peer iteratively gets closer to the target ID (similar as with the peer lookup procedure).

We ran the simulations on a dual socket system with Intel Xeon E5-2690 CPUs (2.6 GHz), each with 14 cores plus hyper-threading. To build the network, the initial bootstrap procedure is done sequentially: A new peer joins every 180 milliseconds until the intended network size is reached. Hereby, a new peer chooses the bootstrap peer randomly from all already joined peers. Any peer can be removed from the network during a simulation with network churn. For our evaluation, we use five iterations for building the network with random start values. Per iteration, we execute each simulation scenario 300 times, i.e., each scenario is executed $5 \cdot 300 = 1500$ times altogether. In each simulation, the time for executing the put operation is randomly chosen within a time slot of three hours. A peer that executes a put or a subsequent get operation is also chosen randomly.

Simulation Scenarios

We proceed from the fact that different parameters may affect the RRR in Kademlia. To evaluate the RRR as comprehensively as possible, we identified the following simulation parameters:

- The bucket size c defines how many peers should store a value upon a put operation. It also determines how many peers are requested for a value upon a get operation. We evaluate the implication of this parameter by setting it to different values. In our simulation, we use the values 5, 10, 15, 20, 25, 30, 35, 40, 45, 50.
- With the parameter *lookup*, we analyse the number of the received values by increasing the number of contacted peers only for the get operation (i.e., the put operation is not affected). For this parameter, we use the values c, $c+5$ and $c+10$. During a get operation for each c value, we simulate $lookup_c$, $lookup_{c+5}$, and $lookup_{c+10}$. Accordingly, we consider three different value lookup procedures, as a get operation terminates after successfully contacting c, $c+5$, or $c+10$ peers respectively. In sum, per a get operation, we evaluate 30 different values for the c-parameter.
- To extrapolate a trend for large networks, we perform our simulations in the networks with 250 and 2500 peers (cf. Section 5.3.1).

- With *network churn*, we evaluate the impact of peer churn on the RRR. Hereby, we define scenarios where no churn is given (0/0), where we add one peer and remove one peer every minute (1/1), and where we add and remove ten peers every minute (10/10).
- With Kademlia, the peers propagate their values in the network after a certain time. To determine whether this Kademlia property affects the RRR, we introduce the parameter *time*. This parameter defines the time in seconds between executing a put operation and the subsequent get operation for the same ID. For its values, we use 60 s, 300 s, 1800 s, 3600 s, and 18000 s.

From these values, we determine 60 different scenarios for the simulations, where we combine the values of the simulation parameters c, network size, and network churn, i.e., $10 \cdot 2 \cdot 3 = 60$ combinations. We simulated all combinations to determine the impact of the affected parameters on the RRR. In a simulation, we perform one put operation and execute the subsequent get operations at different times according to the values of the *time*-parameter. Additionally, within a simulation, we vary the termination condition for these get operations according to the parameter *lookup*. Hereby, a get operation also terminates if the requesting peer knows no more peers to contact, even although the required number of replicas is not achieved. Thus, in each scenario, we perform 12 get operation: 4 get operations to cover all *time* values for c, $c+5$, $c+10$ respectively.

Results

To analyse the storing behaviour in Kademlia upon a put operation, we count how many of the required c peers we successfully contacted during the peer lookup to request the value storing. To determine the average number of peers ($\mathbb{N}_{put}$) that stored the given value in a single simulation scenario, we summarize all produced results in the 1500 iterations and divide the sum by 1500. In Table 5.2, we show the average $\mathbb{N}_{put}$ from the simulation scenarios for c values 5 to 50. Accordingly, the number of peers that store a given value correlates with the parameter c. It slightly deviates from c in cases with network churn. Probably, the peers which were located for storing during the peer lookup procedure were removed from the network before they have received the store RPC.

To analyse the retrieving behaviour in Kademlia upon a get operation, in each iteration, we count how many replicas we receive after a termination condition is achieved upon every 12 get operations in each scenario. Hereby, we are especially interested in the smallest number of the received replicas (*minval*), as we cannot rely on the best case and, therefore, need to analyse $\mathbb{N}_{get}$ in the worst case. To evaluate the impact of the parameter *lookup*, we count the individual $\mathbb{N}_{get}$ for c ($\mathbb{N}_{get}^{c}$), $c+5$ ($\mathbb{N}_{get}^{c+5}$), $c+10$ ($\mathbb{N}_{get}^{c+10}$) for every scenario. For that, we select the *minval* number from results for each get operation at every *time* value and calculate their average for c, $c+5$, $c+10$ respectively for each scenario. In general, as our measurements show, a greater time interval between storing and requesting improves $\mathbb{N}_{get}$. For instance, in a scenario with network size 2500, churn 1/1 and $c = 20$, we receive on average 12.39 replicas when performing a get operation after the put operation with the time interval of 60 s. In contrast, we receive 17.32 replicas after 18000 s. We assume this is due to the value propagation provided by Kademlia: more peers store a certain value in the course of time than directly after executing the put operation. Therefore, we receive more answers by increasing the value for the parameter *time*.

Network Size	c	$\mathbb{N}_{put}$ (0/0)	$\mathbb{N}_{put}$ (1/1)	$\mathbb{N}_{put}$ (10/10)
250	5	5.00	4.98	4.98
	10	10.00	9.96	9.95
	15	15.00	14.93	14.92
	20	20.00	19.91	19.90
	25	25.00	24.88	24.88
	30	30.00	29.85	29.86
	35	35.00	34.82	34.82
	40	40.00	39.82	39.79
	45	45.00	44.78	44.78
	50	50.00	49.80	49.74
2500	5	5.00	5.00	5.00
	10	10.00	10.00	9.99
	15	15.00	15.00	15.00
	20	20.00	19.99	20.00
	25	25.00	24.99	24.99
	30	30.00		29.98
	35	35.00	34.98	34.99
	40	40.00	39.99	39.98
	45	45.00	44.98	44.96
	50	50.00	49.98	49.98

TABLE 5.2: Average Count of Stored Value Replicas Upon a Put Operation for Different Churn

During a complete value lookup procedure upon a get operation for c, $c+5$, $c+10$, the resulting $\mathbb{N}_{get}$ can be between 0 and $c+10$. Therefore, we additionally count the number of successful requests at completion of a get operation ($\mathbb{N}_{get}^{final}$) to evaluate the difference between successful and lost requests. Having $\mathbb{N}_{get}$, we calculate the RRR for each scenario using Equation 5.1. In Figures 5.2 - 5.5, we present the RRR (left y-axis) for all captured $\mathbb{N}_{get}$ in networks with churn. Furthermore, on the right y-axis, we show how many peers were contacted in total (*Effort*) for a certain RRR-value, i.e., including the requests without a response upon a get operation.

Below, we omit scenarios with 0/0 churn, because network churn is common in real networks. As we can see in Figure 5.3, the churn in a small network has a significant impact on the RRR. When 10 peers leave at the same time, the probability that these peers have the requested value is rather high in a network of 250 peers, as 10 peers account for 4% of the total network. The value propagation cannot compensate this, because there are not enough peers to contact for re-storing. In contrast, the churn has no such significance in larger networks (cf. Figure 5.5). On the one hand, in a network with 2500 peers, the percentage of leaving peers is only 0.4. Thus, the probability that the requested value is stored on these peers is significantly lower. On the other hand, in a large network, there are more peers available for the value propagation. Hence, even when some peers leave, there are still multiple replicas available in the network.

In general, we receive less answers for smaller c. Especially for $c \leq 10$, the RRR is under 30% in any scenario. By increasing the c-value, the RRR becomes higher, except in a small network with a high churn (cf. Figure 5.3). However, as our measurements show, Kademlia does not

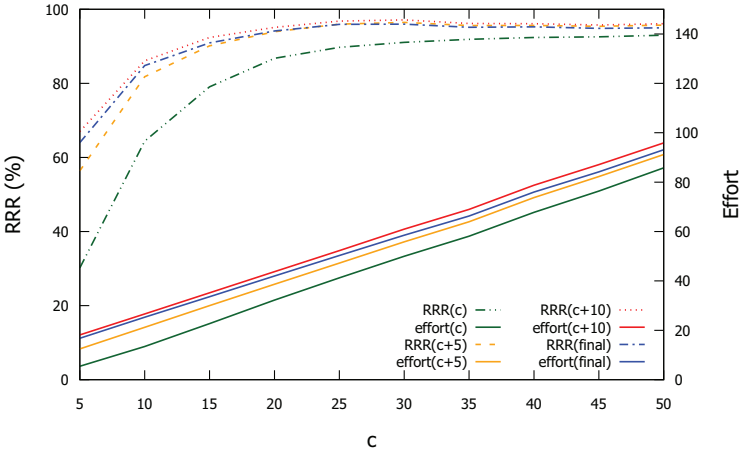

FIGURE 5.2: RRR. Network Size 250, Churn 1/1 per min

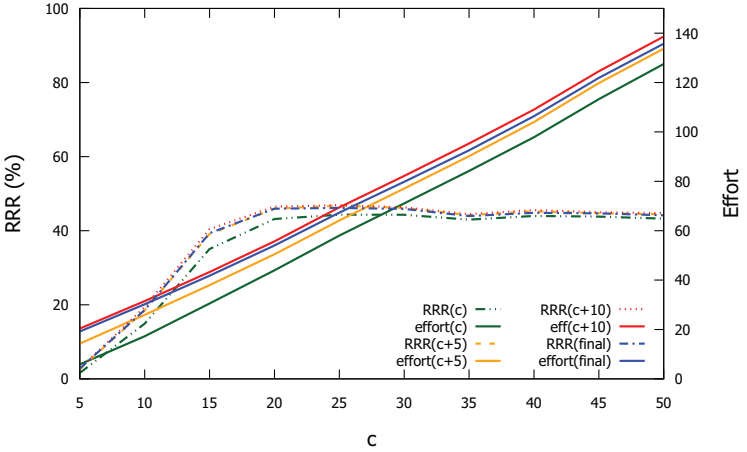

FIGURE 5.3: RRR. Network Size 250, Churn 10/10 per min

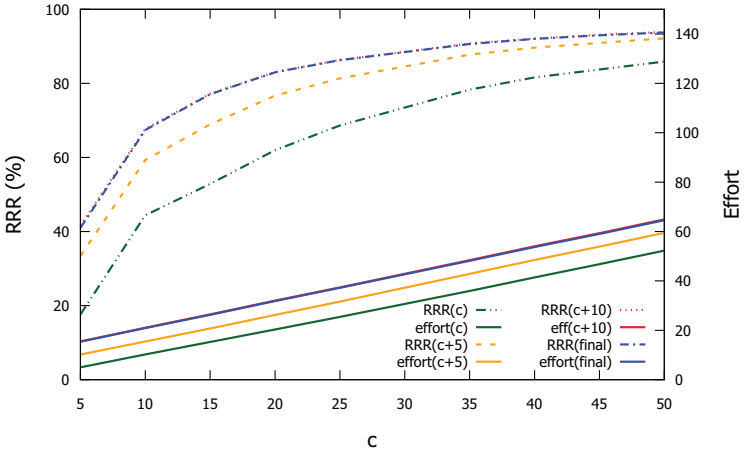

FIGURE 5.4: RRR. Network Size 2500, Churn 1/1 per min

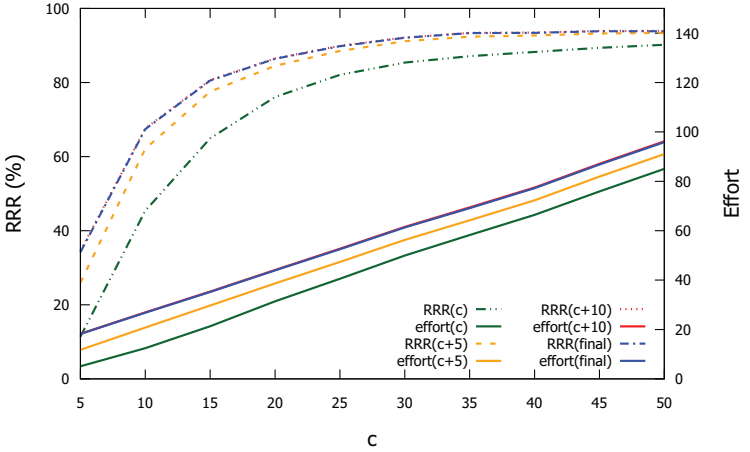

FIGURE 5.5: RRR. Network Size 2500, Churn 10/10 per min

provide an RRR of 100% in no one of the scenarios, since the number of returned values upon a get operation deviates stronger from the c parameter than upon a put operation. Even when we increase the value for *lookup*, we do not achieve the RRR of 100%, albeit it becomes better.

To determine whether the produced effort is reasonable to the achieved improvement, we compare the *minval* results of each simulation scenario regarding $\mathbb{N}_{get}$ with the three different values for *lookup*-parameter, i.e., with c, $c+5$, and $c+10$. For that, we consecutively set $\mathbb{N}_{get}$ to values from the range 1 to 60 and evaluate with which *lookup* we achieve the current $\mathbb{N}_{get}$. This way, we determine how many *find_value* RPCs were performed in total upon receiving a certain $\mathbb{N}_{get}$ in the worst case. For instance, to receive 11 replicas in a small network with 1/1 churn and $c = 20$, we contact 20 peers (cf. y-axis in Figure 5.6). More precisely, we achieve $\mathbb{N}_{get}$ in a range $11-16$ with the same c, because we use the values for c in intervals of five. This is similar for all the remaining scenarios: We receive a range of different $\mathbb{N}_{get}$ using a certain *lookup*. In Figures 5.6 - 5.9, we graphically specify the difference of achieved $\mathbb{N}_{get}$ using a particular *lookup*-value. In general, $\mathbb{N}_{get}$ is linear with respect to the required c for all three *lookup*-values. The corresponding graphs differ from each other only by a constant factor. To determine this factor for each scenario, we calculate it using the *fit*-command of the graphing utility Gnuplot [1]. This command uses the nonlinear least-squares Marquardt-Levenberg algorithm [72]. The resulting factor for each *lookup*-value as well as the associated graphs are also presented in Figures 5.6 - 5.9. Indeed, this factor represents the difference between c contacted peers and the achieved $\mathbb{N}_{get}$, i.e., $\Delta c = c - N_{get}$.

Having Δc, we can, on the one hand, mathematically determine how to set the value for c to achieve a given $\mathbb{N}_{get}$ in a certain network setting using Equation 5.2. For instance, we use $lookup_c$ in a small network with a moderate churn (cf. Figure 5.6) and need 13 replicas for our application. Then, we calculate the required c according to the corresponding Δc: $13+6.93 = 19.93$. However, we round up Δc-values, as we must request an integer number of peers. Hence, to receive 13 replicas in this network setting, we must use $c = 20$.

$$c = N_{get} + \Delta c \tag{5.2}$$

On the other hand, with Δc, we are able to compare the obtained $\mathbb{N}_{get}$ using different *lookup*-values regarding the additional effort and achieved improvement. Considering a small network with moderate churn (cf. Figure 5.6), the improvement achieved by the additional requests with $lookup_{c+5}$ and $lookup_{c+10}$ is negligible by comparing to $lookup_c$. We see this both from the graphs and from the values of Δc. Specifically, for all three *lookup*-values, we must additionally contact 7 peers to achieve a given $\mathbb{N}^c_{get}$, as rounded up $\Delta c = 7$ in all three cases. With other words, we obtain the same number of replicas regardless of whether we contact c, $c+5$, or $c+10$ peers. Hence, the additional effort is useless in this scenario. In Figure 5.7, we see that failing of plenty peers in small networks with a high churn makes it almost impossible to achieve a specific $\mathbb{N}_{get}$ ($\Delta c \geq 33$). In contrast, the network churn in large networks does not have such a significant impact on results – for both churns 1/1 and 10/10, we get comparatively similar results (cf. Figures 5.8 - 5.9). With the high churn, Δc still remains almost constant and low for all three *lookup*-values – we need 7 additional requests in each case. With the moderate churn, we identify a clear difference comparing the three *lookup*-values: $\Delta c = 10$ with $lookup_c$, $\Delta c = 6$ with $lookup_{c+5}$, and $\Delta c = 4$ with $lookup_{c+10}$. For example, to achieve $\mathbb{N}_{get} = 13$ using $lookup_c$, we store and retrieve a value on/from 23 peers, as the corresponding formula is $c = \mathbb{N}_{get} + 10$. In contrast, with $lookup_{c+5}$, we store a value on 19 peers with respect to $c = \mathbb{N}_{get} + 6$, but we request it from 24 peers due to the extended value lookup procedure by 5 additional contacts. Similarly with $lookup_{c+10}$, we store a value on 17 peers but request it from 27 peers with respect to the

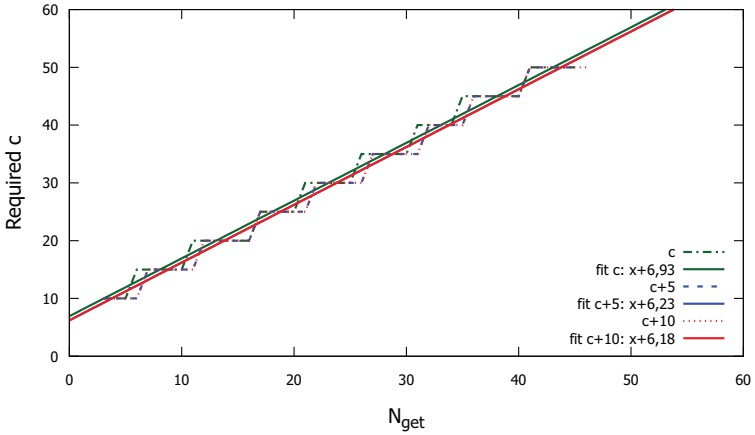

FIGURE 5.6: Δc. Network Size 250, Churn 1/1 per min

corresponding formula $c = \mathbb{N}_{get} + 10$. Using the extended value lookup procedure, we, on the one side, increase $\mathbb{N}_{retrieve}$ upon a get operation. On the other side, we reduce $\mathbb{N}_{store}$ upon a put operation. Comparing to $\mathbb{N}^{c}_{store}$ and $\mathbb{N}^{c}_{retrieve}$ using $lookup_c$, $\mathbb{N}^{c+5}_{store}$ is 17.4% lower and $\mathbb{N}^{c+5}_{retrieve}$ is 4.3% higher, while $\mathbb{N}^{c+10}_{store}$ is 26.4% lower and $\mathbb{N}^{c+10}_{retrieve}$ is 17.4% higher. It depends on a specific application which *lookup* is more suitable. For instance, in an application where upon a single put operation follow multiple get operations, the additional *find_value* RPCs could negatively impact the network traffic. Therefore, in such an application, $lookup_c$ would be preferable. In another scenario, where write operations prevail read operations, one of the extended lookup procedure would be more suitable.

In conclusion, to achieve a certain $\mathbb{N}_{get}$ in small networks, Δc is between 7 and 33 depending on the used *lookup* and the available network churn. In large networks, Δc is between 4 and 10 and only depends on the used *lookup*. Therefore, we consider a large network as a good choice to achieve a specific $\mathbb{N}_{get}$.

5.3.3.2 Implications

From the evaluation results, we deduce that the DRS should be a large network. Then, despite network churn, we can achieve a reliable k-resilience level by determining the suitable c-value.

By storing a value, we achieve $\approx 99\%$ of the required c with the original Kademlia (cf. Table 5.2). With other words, by performing a put operation, the given value is stored at least on $c-1$ different peers. In the scenarios without churn, we even achieve 100% of c. However, no churn in real networks is unlikely. To ensure a certain k-resilience level, we must only consider scenarios with worst results. According to these results, we still do not need to perform $2k+1$ single put operations to achieve the required k-resilience. Considering only the results upon the put operations, we could calculate the sufficient c-value by equating 99% of $\mathbb{N}_{put}$ to 100% of $2k+1$, i.e., $c-1 = 2k+1$. Hence, to determine the appropriate c for a specific k-resilience level,

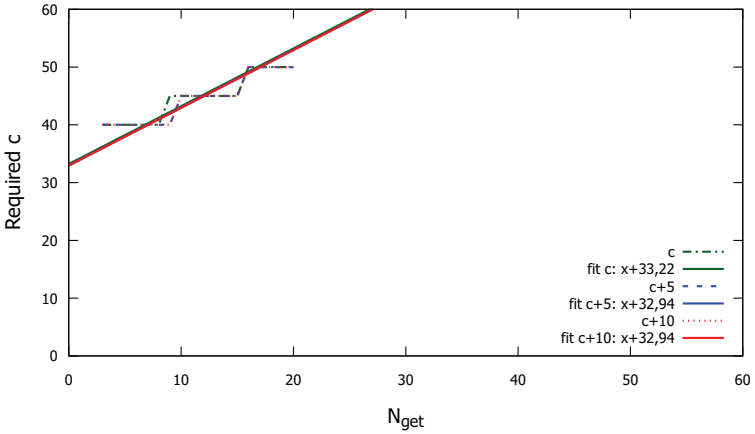

FIGURE 5.7: Δc. Network Size 250, Churn 10/10 per min

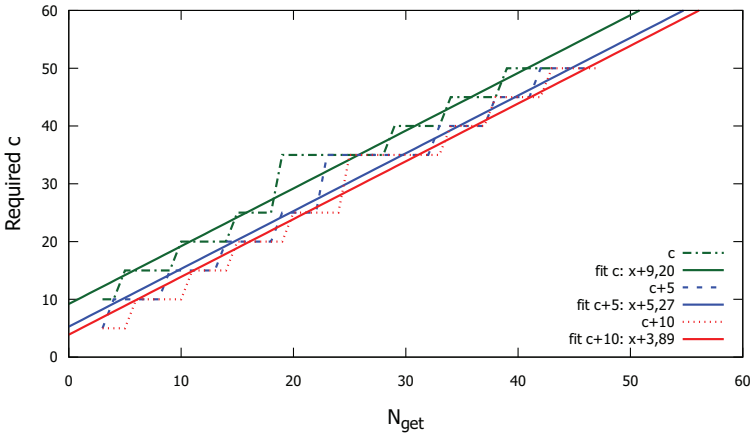

FIGURE 5.8: Δc. Network Size 2500, Churn 1/1 per min

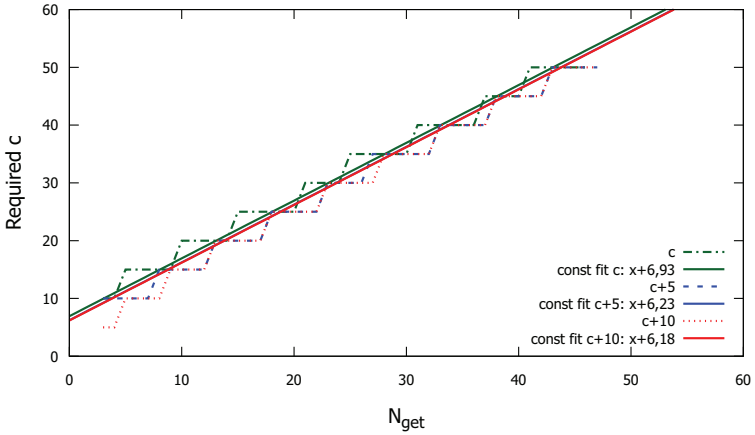

FIGURE 5.9: Δc. Network Size 2500, Churn 10/10 per min

we would use Equation 5.3. For instance, to achieve the k resilience level of 13, we need to store 27 replicas. Accordingly, we would set c to 28.

$$c = 2k + 2 \tag{5.3}$$

However, to achieve the same level upon a get operation, we have to consider $\mathbb{N}_{get}$, as we do not obtain a $RRR = 100\%$ in any scenario. To cope with this, we always must store more replicas than we need to receive. Hereby, $\mathbb{N}_{get}$ corresponds to $2k+1$ needed for majority voting to achieve a specific k-resilience level. While we achieve similar results for $\mathbb{N}_{put}$ in all scenarios, it depends on an individual scenario setting (i.e., used *network size*, *network churn*, and *lookup*) which Δc must be used to calculate the appropriate c for a specific $\mathbb{N}_{get}$. For example, in a large network with a moderate churn and $lookup_c$, we calculate the appropriate c for $k = 13$ with Equation 5.2 and obtain $c = 27 + 10 = 37$, where 27 is $\mathbb{N}_{get}$ that we need for the majority voting, and 10 is Δc.

In general, $\mathbb{N}_{get}$ is the decisive factor for calculating the associated c-value to achieve a certain k-resilience level. Applying this to the DRS, we consider two application scenarios:

- *major put* – users execute considerably more put operation than get operations. This scenario corresponds to the push approach.
- *major get* - a put operation is followed by multiple get operations. This corresponds to the pull approach.

For *major put*, it is preferable to use an extended value lookup procedure: Using a smaller c, we can reduce $\mathbb{N}_{store}$ upon put operations at cost of increased $\mathbb{N}_{retrieve}$ upon subsequent get operations. However, as get operations are not performed as often as put operations, we reduce the network load in the end. To take the example with $\mathbb{N}_{get} = 27$ further, we would store a

value on 33 peers and request it from 38 peers using $lookup_{c+5}$. Assuming, the ratio between a number of put operations and get operations in the network is 1 to 100. Then, in comparison to $lookup_c$, we would reduce the network load $\mathbb{N}_{store}$ by 9.6%, as there are $1 \cdot 37 + 100 \cdot 37 = 407$ total requests with $lookup_c$ and $1 \cdot 38 + 100 \cdot 33 = 368$ with $lookup_{c+5}$. Using $lookup_{c+10}$, we would store a value on 31 peers and request it from 41 peers. This would yield a 13.8% savings of total requests in the network.

With *major get*, we assume a less number of put operations in comparison to the number of get operation. Referring to the pull approach, the owner does not often change *statement* of her data object, but providers request it relatively often (cf. Section 6.5.2.1). Then, it is more efficient to produce occasionally more effort while storing a value using $lookup_c$ and, thereby, keeping $\mathbb{N}_{get}$ as low as possible. Thus, in this scenario, we achieve the optimal total number of requests in the network by using $lookup_c$.

In summary, we can reduce the number of operation upon *get* and *put*: It is sufficient to perform a single API operation with Kademlia and still achieve that a value is received from $2k+1$ different peers to enable the majority voting, i.e., to achieve k-resilience. For that, we calculate the appropriate c-value with Equation 5.2 according to the used network parameter. With the original Kademlia (i.e., $c = 20$ and $lookup_c$) used in a large network, we can achieve the resilience level of 6: With $c = 20$, the maximum odd $\mathbb{N}_{get}$ is equal 13. As 13 represents $2k+1$, we calculate $13 = 2k+1$ and get the resilience level $k = 6$.

5.4 Summary

To implement the DRS with the access control based on k-rAC, we must ensure k-resilience. As Kademlia is the only DHT used successfully in practice, we analysed and evaluated its suitability for the DRS. In general, Kademlia provides important features required with k-rAC. Due to the assigning of peer identifiers by using cryptographically secure hash function, the identifiers are equally distributed per default over the Kademlia DHT space. Moreover, locating of resources in Kademlia is closely linked to the bucket size c. During a lookup procedure, a requesting peer contacts up to c peers. This allows us to consider c as related to our resilience parameter k. Furthermore, we identified and applied modifications in Kademlia to achieve the required resilience for k-rAC. Particularly, as the value lookup procedure terminates after receiving one replica in original Kademlia, we extended this procedure to enable receiving multiple replicas.

We evaluated Kademlia regarding disjoint paths, disjunct peers, and the number of requests upon put and get operations in multiple scenarios, where we combined simulations parameters c, network size, and network churn. Hereby, we determined that the network connectivity κ of Kademlia strongly correlates with its bucket size c. Thus, by controlling the value of c, we can achieve that the requests associated with a DHT operation are sent over disjoint paths. To ensure that disjunct peers store or return the given value, we verify their IDs during the peer and value lookup procedure and accept each ID only onetime. To achieve a sufficient number of disjunct peers while the storing and retrieving procedures, we evaluated the Return Replica Ratio (RRR) of Kademlia. For that, we stored a value on c peers and retrieved it from c, $c+5$, $c+10$ peers. As measurement results show, the number of returned replicas is less than number of storing peers in any simulated scenario, i.e., $\mathbb{N}_{get} < \mathbb{N}_{put}$. Accordingly, to calculate the appropriate c, we use Equation 5.4.

$$c = N_{get} + \Delta c \tag{5.4}$$

Thereupon, we calculated the value of Δc for different network settings using the fitting function of Gnuplot [1]. With Δc, we are able to determine the value for c for a given $\mathbb{N}_{get}$ in a certain Kademlia network setting using Equation 5.4.

In summary, evaluating the connectivity, we determined that c must be larger than $2k+1$ to ensure disjoint paths. Evaluating the RRR, we determined the factor how large c must be to receive $2k+1$ replicas needed for majority voting to achieve a certain k-resilience level. Finally, the method for locating resources used in Kademlia allows us to reduce the number of operations upon *get* and *put*. While with a generic DHT, we need to perform $2k+1$ API operations to store or retrieve a value, using Kademlia, it is sufficient to perform a single API operation to achieve the required k-resilience.

We published our first results of the Kademlia connectivity evaluation in [39] and presented them at the 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2016). Furthermore, we elaborated the evaluation results in [40] and presented them at the 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017).

6

Data Revocation Service

In Chapter 2, we designed a system architecture that is suitable for both push and pull approaches. For each approach, we identified two different implementation methods. For the push approach, these are the owner and the system push methods. For the pull approach, these are the status pull and the key pull methods. Irrespective of which approach we consider, the system entities are the same, i.e., the DRS, owners, providers, and users. However, depending on the particular revocation method of the proposed approaches, we store different data with the DRS. Furthermore, the procedure of provider notification is different for the four proposed revocation methods. In this chapter, we first present the general properties of the DRS and, then, describe the interaction of the system entities for each particular revocation method. In general, we need an access control for the DRS independently of the considered revocation approach. The access control scheme is different for each revocation approach. With *k*-rAC (cf. Chapter 4), we have the possibility to combine the access control mechanisms to an appropriate access control scheme for each revocation method. In the following, we also explain how we use *k*-rAC with the four proposed revocation methods.

6.1 Properties

The DRS is the central component in our system for data revocation on the Internet. As introduced in Section 2.5, the DRS is an Internet service operating with no central authority by using a DHT structure. Its main task is the notification of providers about owner's revocation demands. Within the DRS, we store the information that is needed to execute the revocation of protected data objects. We manage one protected data object per a single DRS entry. Depending on the particular revocation method, certain system entities request or modify this information, i.e., the variable *statement* (cf. Section 2.5). These system entities can be owners, providers, or users.

For the protection against unauthorized accesses and modifications, we use *k*-rAC to control access to each single entry in the DRS (cf. Chapter 4). With *k*-rAC, the first access to an empty DHT entry defines its owner. The owner has the right to modify the value stored in this entry and to delegate particular access rights to other users. Depending on the assigned access rights, an authorized user can execute the same or some of the operations as the owner. In the following, when we use the term "owner", we refer to both the owner and users that are authorized to perform the same operation. Due to *k*-rAC, a *statement* is replicated to multiple DHT indexes

during a write operation. Even if up to k of the responsible peers are malicious and might manipulate it, the integrity of the read access is ensured by the majority voting. Besides the above general DRS features, each of the proposed notification methods has its specific requirements regarding the access control. Therefore, we apply the particular authentication and authorization mechanisms to the specific needs of a single revocation method based on k-rAC.

To replicate a DRS entry to multiple peers via different network paths, we need a sufficient number of peers participating in our P2P network. To achieve this, we propose providers' web servers to be part of the P2P network. This is sound, since a reliable DRS is also in the interests of providers to comply with the law (cf. Chapter 1). As a consequence, the DRS service is comprised of stable peers with very low churn, as providers' web servers usually have a low churn rate in comparison to a P2P network only consisting of personal computers. Involving providers' web servers, we can achieve a high number of participating peers and additional stability. Furthermore, we propose to use the modified Kademlia DHT for the DRS as elaborated in Chapter 5. This way, we increase the performance of k-rAC through reducing its message overhead to a minimum by taking advantage of the resource locating mechanisms built-in in Kademlia.

To secure all communications with the DRS, we use SSL/TLS [23]. This prevents man-in-the-middle attacks during the registration of new data objects. We cannot enforce encryption for the communication between Internet users and providers. However, this does not open any new attack possibilities for the system, as an eavesdropper could read only the unique ID from this communication, but the ID does not contain any relevant information for an attacker (cf. Chapter 3).

As introduced before, we propose two different approaches, *push* and *pull*, to realize the data revocation. With the presented DRS architecture, we can apply both approaches without the need to modify the DRS architecture. The implementation difference between these approaches is what is stored in *statement* and the way it is accessed. In contrast, their application difference is in how the notification about a revocation request occurs, i.e., the provider informs by herself (*pull*) or she is informed about it (*push*). Each of these approaches can be realised in two ways. Hence, we distinguish between four different methods to implement the revocation notification, i.e., the *owner push*, the *system push*, the *status pull*, and the *key pull* methods. In the following subsections, we highlight the differences of specific revocation methods by considering their protocol flow. From this flow, we identify four main protocol phases that are common for each method. These phases are:

- Registration: In this phase, the owner registers her data object with the DRS.
- Publication: In this phase, the owner publishes her protected data object on the Internet. Depending on the particular method, this phase also includes steps taken by the provider to inform the DRS about the publication.
- Distribution: With this phase, we describe the steps which the provider takes to ensure that she delivers only those data objects which are not revoked.
- Revocation: This phase comprises the actions that the involved entities perform to revoke a data object.

Following, we present the general description of the two revocation approaches and a detailed protocol flow for their implementation methods.

6.2 Push Approach

With the push approach, the provider is notified about a revocation request of a certain data object as soon as the owner demands it. To achieve that, the owner registers her data objects with the DRS. Additionally, the provider registers each protected data object, which is available with her service, with the DRS. This way, the system collects within *statement* the contact information of providers that are affected in case of a revocation request for a certain data object. With this information, it is possible to execute the notification by the owner (*o-push*) or by the DRS *(s-push)*.

6.2.1 Access Control

Depending on which particular push method is used, we need an appropriate access control mechanism to protect *statement* against unauthorized accesses. Using o-push, we must ensure that only the owner can initiate the revocation process and readout *statement*. Similar with s-push, only the owner should be able to initiate the revocation of her data object. However, since in this case the notification is performed by the DRS, the system must be able to readout *statement*. Due to the *Privacy*-requirement, no one else should be able to get the entire list of affected providers for a certain data object. Following, we consider the access control for both push methods in detail.

6.2.1.1 Access Control with the Owner Push Method

For the revocation with o-push, we need to control both (1) the read access to *statement* for protecting the confidentiality and (2) the write access for avoiding manipulations on *statement*.

To perform the revocation process, the owner executes the get operation to obtain *statement* with the contacts list of affected providers, i.e., *get(index)*. Hence, we must ensure that only the owner is able to readout *statement*. As the read access is protected by encryption (cf. Section 4.3), we store the contacts encrypted. We have to consider that while different providers add their contacts to a common list, we must ensure that still only the owner can decrypt the entire contact list. To achieve that, we encrypt each single item independently from other items inside *statement*, i.e., the *statement* itself is not encrypted, but it consists of single encrypted items. For that, we use the PK approach of *k*-rAC, as it is the most resource-efficient authentication mechanism in this case. With this approach, we can achieve both the authentication of the owner and the distribution of the encryption key (owner's public key), while only the owner remains in possession of the decryption key (owner's private key). In contrast, with ZKP or OTH, we would need additional mechanisms to distribute the key distribution. However, any additional step would complicate the system unnecessarily.

With the PK approach, the provider uses the owner's public key to encrypt her own contact and sends it encrypted to the DRS. As a consequence, no peer is able to collect the contact information of the affected providers for her own purposes. Before storing the received contact, the DRS verifies it in order to prevent invalid contacts. Only if the contact is valid, the systems adds it to *statement*. We discuss how to verify contacts despite their encryption in Section 6.2.2.

To reduce the system complexity (cf. the *Usability*-requirement in Section 2.2), we only allow adding of new contacts to *statement* and do not provide a process for updating them, i.e., the

provider cannot modify her contact afterwards, as it is encrypted with the public key of the owner. Therefore, only the owner is able to decrypt the contacts of providers that must be requested to revoke a certain data object. For the case when the contact modification should be possible, we would need to ensure that a provider is able to modify only her own contact and cannot rewrite contacts of other providers available in *statement*. We can achieve that by using additional public key cryptography for the verification that a provider is authorized to update a certain list item in *statement*. To keep the system lightweight, we refrain from considering this case further.

6.2.1.2 Access Control with the System Push Method

By revoking data objects with s-push, we must ensure that only the owner is able to initiate the revocation. Moreover, only the owner should be able to readout the corresponding *statement*. Since the revocation principle is the same as with o-push and only differs in who sends the notification to the providers, also with this method, the most resource-efficient authentication mechanism is the PK approach.

Furthermore, we introduce the new API operation *revoke* to clearly separate the revoke event executed by the owner from write accesses to the value *statement* by providers. While the put operation is used for updating *statement*, the revoke operation is reserved for an authenticated request to execute the revocation of a certain data object. Hence, the owner executes the operation *revoke(index, auth)* to initiate the revocation of her data object with the ID *index*. Complementary, we introduce a DHT operation *on_receive_revoke (sender, index, auth)* for processing the message *revoke*.

The owner proves her right to revoke a specific data object by signing its ID with her private key, i.e., $auth := \{sign\}$ where $sign := \text{encrypt}_{sk}(h(index|mid))$ and $h(x)$ is a cryptographic hash function. With the message ID *mid*, we prevent replay attacks as described in Section 4.4.3. At this time, the DRS has the corresponding public key and is able to verify the signature (cf. Section 6.2.2).

The challenge with this method is that *statement* must be accessible for the system in the revocation phase. That means the responsible peers must be able to readout *statement*. If for that we would omit encryption of *statement* and would instead protect it by disabling the get operation, the involved peers would be able to access the provider contacts. However, assuming not all peers act as required by the system, one malicious peer would be sufficient to manipulate the contact list or execute the revocation without the owner has initiated the event *revoke*. Therefore, we propose that the provider encrypts her contact equally as with o-push. To enable the revocation by the system anyway, the owner first requests *statement* with the get operation and decrypts it the same way as with o-push. Afterwards, she sends the decrypted contacts to the DRS with the revocation operation, i.e., *revoke(index, contacts, auth)*. This time, *auth* is a signature where $sign := \text{encrypt}_{sk}(h(contacts|mid))$. Consequently, the system gets access to *statement* only at the time when the owner demands the revocation. To prove to the provider that the DRS is authorized to request the revocation on her behalf, the owner assigns to each contact in the decrypted *statement* a provider specific signature (cf. Section 6.2.2). The DRS uses each of these signatures to authorize the revocation request with the corresponding provider.

6.2.2 Protocol Flow

In Figures 6.1 and 6.2, we demonstrate the protocol flow for both push methods. As we see from these figures, the registration, the publication, and the distribution phases are identical for both methods. They only differ in the revocation phase. Therefore, we first describe these three phases together for both methods. Afterwards, we consider the revocation phase for o-push and s-push separately.

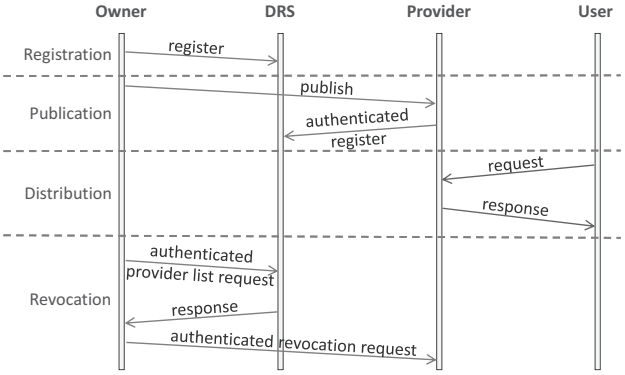

FIGURE 6.1: Protocol Overview – Owner Push Method

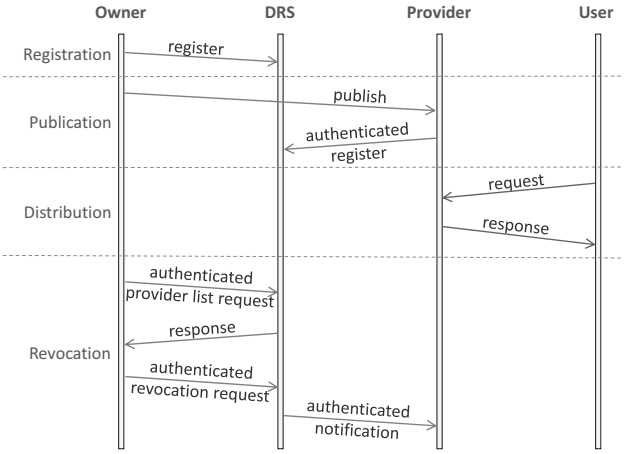

FIGURE 6.2: Protocol Overview – System Push Method

Registration Phase

Technically, the registration means an initial write access to an empty DHT index. With the push approach, the owner registers her data object by storing a dataset (*data*) with the DRS via a put operation, i.e., *put(index, data, auth)*. With other words, she stores *data* in a DHT entry, which is defined by *index*, and protects the access to that entry with *auth*. For that, the owner assigns an ID to her data object, generates *data*, and computes the *auth*-value before registering it with the DRS.

With *data*, we provide a data structure for storing providers' contacts in the publication phase, i.e., a list object. Accordingly, for *data*, the owner generates an empty list to store it in *statement* with the DRS. Here, in contrast to the pull approach (cf. Section 2.5), we cannot additionally use an expiration date by storing it in *statement*. Due to the encryption for the read access control, the DRS does not have the right for performing the revocation as a function of a value stored in *statement*.

The parameter *auth* is used by the DRS in the revocation phase to ensure that only the owner is able to initiate the revocation of the corresponding data object. The content of *auth* depends on the used authentication mechanism. As we use the PK approach, *auth* is the owner's public key. In general, the registration workflow on the owner side corresponds to the *k*-rAC wrapper *user_k-rAC_wrapper_put* by using PK approach 4.4.3.1. We give an overview of the main steps performed by the owner during registering a data object with the DRS in Listing 6.1 and refer to Listing 4.16 for more details.

```
//dObj: data object to be registered with the DRS
//tagName: name of the metadata property, e.g., ''DRSid''
void owner_register(dObj, tagName){
    //1)Prepare Registration
    //assign ID according to Listing 3.3
    setID(dObj);
    id                      = getMetadata(dObj).getValue(tagName);

    //generate public/private key pair and prevent replay attacks
    credentials             = new Credentials();
    credentials.keypair     = new AsymmetricKey();
    credentials.window      = SlidingWindow.generateSlidingWindow();
    credentials.window.mid  = SlidingWindow.setMessageID();
    //save credentials locally
    wallet.put(id, credentials);

    //create list object
    data                    = new List<ProviderContact>();

    //create authenticator
    auth                    = new Authenticator();
    auth.pk                 = credentials.keypair.pk;
    auth.mid                = credentials.window.mid;
    auth.sign               = null;//due initial access

    //2)Execute registration with DRS
    DRSapi.put(id, data, auth);
}
```

LISTING 6.1: Push Approach – Owner Registration Routine

According to the access control scheme, the DRS first verifies whether the entry for the given *index* is empty. If for this *index* there is already a DHT value present, the registration is rejected. Otherwise, the DRS stores the received *data* in *statement*. Additionally, as we extended the classical DHT value by the ACL to manage the access rights, the DRS stores in the ACL item the received *auth* and sets the access right to *o* (cf. Listing 4.11).

This results in a fully anonymous registration as no information from the stored dataset includes any personal information about the owner. First, *index* is independent from its owner (cf. Chapter 3). Second, using a different key pair for each data object as proposed with *k*-rAC, the value *auth* also does not provide any information about the owner.

Publication Phase

While registering a new data object with the DRS, the owner embeds the metadata about the ID into the data object, e.g., as proposed in Chapter 3. Afterwards, she can publish this protected data object with an arbitrary provider on the Internet. To do so, she is free to use any way offered by the provider, i.e., this is independent from the DRS.

During the publication process, the provider checks whether the data object is protected by the DRS using its metadata (cf. Listing 6.2). If the data object is protected, the provider registers with the DRS that she published this data object. Similar to the registration of a data object by the owner, the provider stores a dataset with the DRS. Again, the dataset is identified by the ID of the data object. In this case, the dataset contains the provider's contact. This contact is composed of two details as presented in Listing 6.3. The first detail is the URL (*c-url*) under which the provider manages the incoming revocation requests. The second detail is a random number *nonce* generated and used by the provider to verify the authorization of a revocation request in the revocation phase. Finally, to prove that she indeed published the corresponding data object, the provider uses as *auth* the URL (*dObj-url*) under which the published data object is accessible. Only if this URL is valid, the provider's contact will be registered with the DRS.

```
//dObj: data object to be registered with DRS
void provider_publish_data_object(dObj){
    //publish the data object as usual and return its URL
    URL dObj_url = publish(dObj);

    //for case dObj is protected by DRS, check its metadata for item "DRSid"
    id = getMetadata(dObj).getValue("DRSid");
    if(id != null) {
        //retrieve the corresponding public key from DRS
        DRSreplies[]  = DRSapi.register(id);
        send_contact_to_DRS(DRSreplies[], id, dObj_url);//see Listing 6.4
    }
}
```

LISTING 6.2: Push Approach – Provider's Publication Routine

```
ProviderContact{
  c_url: URL;      //provider's API for revocation requests
  nonce: integer; //used for revocation request verification
}
```

LISTING 6.3: Push Approach – Data Structure of a Provider Contact

Since the provider encrypts her contact before storing it with the DRS, the registration of a data object publication consists of two steps: (1) requesting the corresponding public key from the DRS and (2) sending the encrypted contact to the DRS. Hence, it is not possible to wrap the whole registration procedure in one put operation. Therefore, we divide it in two separate operations. We could use the get operation for requesting public keys. In this case, we would need to introduce a new parameter to distinguish a *statement* request from a public key request. Although that does not cause a significant effort for the implementation, for reasons of clarity and comprehensibility, we introduce the new operation *register* to distribute public keys. Consequently, the provider first uses the register operation for getting the corresponding public key and then performs the put operation for storing her encrypted contact in *statement*.

The procedure to register the publication is as follows: The provider initiates the registration of a certain data object publication by executing the register operation, i.e., *register(id)* where *id* is the ID of the corresponding data object extracted from its metadata (cf. Listing 6.2). The DRS returns the requested public key. As with any API operation extended by *k*-rAC, the request for the public key via the register operation is sent to $2k+1$ responsible peers. Consequently, the provider receives $2k+1$ responses and calculates the majority over them.

Next, the provider adds *nonce* to her *c-url* and encrypts this composite contact with the received public key. She stores *nonce* and owner's public key locally for verifying the revocation request later in the revocation phase. Finally, the provider sends her encrypted contact to the DRS by executing the put operation, i.e., *put(id, data, auth)* where *id* is the ID of the corresponding data object, *data* is provider's encrypted contact, and *auth* is the *dObj-url*. The steps of this routine are shown in Listing 6.4.

```
//id: ID of data object to be registered with DRS
//dObj_url: web address of data object which is to register
//DRSreplies: an array with 2k+1 public key replicas from DRS
void provider_send_contact_to_DRS(id, dObj_url, DRSreplies){
    //calculate majority over received 2k+1 public key replicas
    pk      = calculate_majority(DRSreplies);

    //generate and encrypt own contact information with pk
    contact = new ProviderContact();
    data    = encrypt(contact, pk);

    //save the dObj metadata locally, e.g., in HashMap
    drsObjects_store.put(id, [contact.nonce, pk]);

    //complete registration with the DRS
    DRSapi.put(id, data, dObj_url);
}
```

LISTING 6.4: Push Approach – Sending Contact Information to DRS by Provider

Before storing the received contact, we must verify that the provider indeed published the given data object. This verification is twofold. First, we must ensure that the given data object is accessible under the provided *dObj-url*, i.e., we verify the publication. Second, we verify whether the provider has control over the given domain. Otherwise, the attacker could pose as a provider and store any number of entries in the contact list to negatively affect the system. To prevent this attack, we implement the verification as a challenge/response procedure similar to the Automatic Certificate Management Environment (ACME) [14]. The ACME is used for the authentication of domain names. Hereby, the prover receives a challenge from the verifier, which she should store on her webspace to prove in this way that the domain is under her control.

After the DRS successfully verified the publication, it adds the received *data* to *statement* under the given *index*. Specifically, each responsible peer adds the incoming provider contact to the provider contact list in *statement*, which is stored in its local storage. The verification routine is presented in Listing 6.5: As usual, when the DRS receives a store request, it first verifies whether the requested entry is empty or not. Hereby, it is crucial to differentiate the put operation executed by a provider from the one executed by an owner. The owner uses the put operation only once, namely to register her data object with the DRS, i.e., the corresponding DHT entry must be empty at this time to make the registration possible. When a provider executes a put operation to register her publication of a protected data object, it is always a subsequent access to this DHT entry, i.e., it is already occupied by the owner of the corresponding data object. Hence, we assume that if the DHT entry is not empty, the put request is executed by a provider. Even if it is not from a provider (e.g., in case of a collision), we catch it via an exception.

```
1  //sender: executor of the put operation (an owner or a provider)
   //index: ID of the published data object
3  //value: encrypted contact of the provider
   //auth: web address of data object which is to register
5  void drs_on_receive_store(sender, index, value, auth){
       if (localstore.get(index) == null ) {
7          //no value stored, hence no owner
           //register the owner (cf. Listing 4.11)
9      } else {
           //it is a provider registration request
11         //verify the publication by accessing it via URL given in auth
           try{
13             dataset ds = localstore.get(index);
               verify(auth);
15             ds.data.add(value);
           }
17         catch (MalformedURLException | OccupiedDHTEntryException ex){
               return nack(sender, ex);
19         }
       }
21 }
```

LISTING 6.5: Push Approach – Verification of Publication by DRS

Distribution Phase

In this phase, when a user requests a protected data object, the provider sends it to the user the same way as before without using the DRS. Hence, the DRS causes no burden neither for the user on requesting a data object nor for the provider on delivering the requested data object.

Revocation Phase

As mentioned before, the revocation procedure for o-push and s-push is different. Following, we discuss these differences in detail.

Revocation with o-push: In this case, the owner notifies the affected providers by herself about her revocation request for a certain data object. For that, the owner needs the contact list of the affected providers stored within *statement* with the DRS. To obtain this contact list, the owner

executes the get operation with *get(id)*, where *id* is the ID of the data object that should be revoked. The DRS returns *statement* for the requested *id*, i.e., the owner receives $2k+1$ replicas of the provider contact list. Subsequently, the owner performs the majority voting over the received replicas to determine the valid list. As described before, each single item in *statement* is encrypted with the public key assigned to the data object. The owner decrypts each contact URL with her private key assigned to this data object. Then, she sends a revocation request to each listed provider. Hereby, she signs each request with the private key of that data object by using the nonce which is contained in the provider's contact (cf. Listing 6.6).

```
//id: ID of the data object which should be revoked
//enc_contacts: list with encrypted provider contacts after majority voting
void owner_revoke(id, enc_contacts){
    credentials = wallet.get(id);

    for each enc_contact in enc_contacts{
        Contact contact = decrypt(credentials.keypair.sk, enc_contact);
        //sign nonce with private key to authenticate request
        auth    = sign(contact.nonce, credentials.keypair.sk);
        //send revocation request to provider
        ProviderAPI.revoke(contact.c_url, id, auth);
    }
}
```

LISTING 6.6: Owner Push Method – Revocation Request by Owner

Every notified provider verifies the signature of the revocation request with the locally stored public key for the provided *id*. Only if the signature is valid, the provider follows the request and deletes the data object from her server (cf. Listing 6.7).

```
//id: ID of the revoked data object
//auth: signed nonce to prove the right to revoke
void provider_on_receive_revocation_request(id, auth){
    //load metadata for the provided id from local store
    metadata = drsObjects_store.get(id);
    //verify signature of nonce with stored public key
    if (verify_request(auth, metadata[nonce], metadata[pk])){
        //delete requested data object
        //signal successful deleting to requester, i.e., ack
    } else {
        //reject deletion, i.e., nack
    }
}
```

LISTING 6.7: Owner Push Method – Verification of Revocation Request by Provider

Alternatively, we can extend the access control to *statement* by introducing the owner authentication before delivering her *statement*. For that, the owner sends with the get operation a signature for proving her owner right, i.e., *get(index, auth)* where $auth := \{sign\}$ and $sign := \text{encrypt}_{sk}(h(index|mid))$ as described in k-rAC for the PK approach. Only if the signature is valid, the DRS returns *statement*. However, that would be an additional effort for the system without a real advantage on the security, because *statement* is protected by encryption anyway.

Finally, if a revocation request at a provider fails, e.g., due to data message loss, the owner repeats sending her request until the provider reports the successful deleting.

Revocation with s-push: By using s-push, the DRS notifies the provider about the revocation of a certain data object on owner's behalf. In contrast to o-push, when a revocation request fails, the DRS sends the revocation request until the provider reports the successful deleting of the given data object.

To initiate the revocation, the owner first executes the get operation with *get(id)*. The DRS returns the corresponding *statement* with the encrypted contact list. To send back to the DRS the decrypted list of affected providers, the owner must decrypt the contacts and assign to each contact a signature valid for the corresponding provider. For that, the owner uses *nonce* that is contained in a contact, i.e., $sign := \text{encrypt}_{sk}(h(nonce))$. The DRS uses this signature to prove to the corresponding provider its authorization to revoke the data object on behalf of its owner. Next, the owner adds the decrypted contacts to a list, where a single item contains the provider's contact URL and the signature of her *nonce*. Finally, she initiates the revocation by executing *revoke(id, contacts, auth)* as described in Section 6.2.1.2. We summarized the revocation procedure on the owner side in Listing 6.8.

```
1  //id: ID of data object to be revoked
   //enc_contactlist: encrypted provider contact list returned by DRS
3  void owner_prepare_revocation(id, enc_contactlist)
   {
5      credentials       = wallet.get(id);
       contacts          = new List<ProviderContact>();
7
       for each enc_contact in enc_contacts{
9          contact       = decrypt(credentials.keypair.sk, enc_contact);
           //sign nonce with private key to authenticate request with provider
11         contact.auth = calculateSignature
                              (contact.nonce, credentials.keypair.sk);
13         contacts.add(contact);
       }
15     //create Authenticator, i.e., auth:=(pk, mid, sign)
       auth              = new Authenticator();
17     auth.pk           = credentials.keypair.pk;
       //sign contact list to authenticate revocation request to DRS
19     auth.sign         = calculateSignature(credentials.keypair.sk, contacts,
                              credentials.window.mid);
21     credentials.window.mid++;

23     DRSapi.revoke(id, contacts, auth);
   }
```

LISTING 6.8: System Push Method – Revocation Preparing by Owner

To process a revocation request by the DRS, we introduce a new event message *revoke*. On receiving a *revoke* message, the system verifies the provided signature (i.e., *auth*) to ensure that the request was sent by the owner of the corresponding data object. If the signature is valid, the system contacts each provider listed in the received dataset *contacts* to request the revocation of the data object with the provided ID. Specifically, each responsible peer authenticates the owner's request and notifies the affected providers according to the received provider contact list (cf. Listing 6.9).

Receiving this notification, the provider first verifies the received signature with the corresponding public key and *nonce* that she stores locally. If the signature is valid, the provider complies with the notification and deletes the corresponding data object from her server. Due to k-resilience, each affected provider gets up to $2k+1$ revocation notifications. In this case, the

provider must not calculate the majority voting: Even one valid notification is sufficient to delete the corresponding data object – a peer is only able to obtain the valid authenticator for revocation request when the owner herself initiated the revocation for the corresponding data object. We solve the multiple revocation notifications for the same data object by managing the local store for protected data objects *drsObjects_store* as follows: For every incoming revocation request, the provider first verifies if there is an entry available in the local store for the given ID. If it is available, the provider checks the signature. If the signature is valid, she deletes the corresponding data object locally from her servers, i.e., the data object is no longer available with her service on the Internet. Additionally, she deletes the associated entry in *drsObjects_store*. For all subsequent revocation notifications for this data object, the verification for the corresponding item in *drsObjects_store* will fail, i.e., this data object is already deleted. Hence, the provider can ignore all subsequent revocation notifications for this data object. We show the steps of this routine in Listing 6.10.

```
//sender: ID of the requesting peer
//index: ID of the revoked data object
//contacts: list with providers that must be notified
//auth: owner's authentication value
void drs_on_receive_revoke(sender, index, contacts, auth){
    //load DHT value for given index from local store
    dataset ds = localstore.get(index);
    object result = nack;//initial value
    //get user's ACL item
    acl user_acl  = ds.acl.getUserAcl(auth);

    if((user_acl != null)
       && ((authenticate_user(user_acl, auth, contacts) == true)
       && (user_acl.rights == ['o'])){
         //revocation access allowed
         for each contact in contacts{
             ProviderAPI.revoke(contact.c_url, index, conact.auth);
         }
         result    = ack;
    }
    //signal the result to the requesting peer
    send_direct(sender, 'store_reply', index, result);
}
```

LISTING 6.9: System Push Method – Revocation Request by DRS

```
//id: ID of the revoked data object
//auth: signed nonce for verifying the right to revoke
void provider_on_receive_revocation_notification(id, auth){
    if(drsObjects_store.get(id) != null){
        //load metadata for the requested data object from local store
        metadata = drsObjects_store.get(id);
        //verify signature of nonce with stored public key
        if (verify_request(auth, metadata[nonce], metadata[pk])){
            //delete requested data object system-widely
            delete(path);
            //delete the associated entry
            drsObjects_store.delete(id);
        } else {
            //reject deletion
        }
    }
}
```

LISTING 6.10: System Push Method – Handling of Revocation Notification by Provider

6.3 Pull Approach

The main characteristic of the pull approach is that the provider informs herself whether she is allowed to deliver a certain data object or not. As introduced in Section 2.4, we identified two methods to realize the pull method: the status pull (*s-pull*) and the key pull (*k-pull*) methods. With both methods, the owner uses the DRS for storing some information describing the delivering permission of her data object. However, the realization of the revocation for these methods differs significantly. With s-pull, the owner publishes her data object open and relies on the law-abiding acting of providers, i.e., providers regularly retrieve the latest delivering status from the DRS and act according to this status. In contrast, with k-pull, the owner encrypts her data object before publishing it and stores the encryption key with the DRS.

In the following, we analyse the details of both methods. As their implementation differs in each protocol phase, we consider these methods in separate sections.

6.3.1 Status Pull Method

With s-pull, we use the DRS as a unidirectional communication channel from an owner to the providers. Within *statement*, the owner stores a message that represents the delivering status of her data object (cf. Section 2.4). With this status, the owner communicates to the providers whether the corresponding data object is allowed to be delivered or not.

Below, we first describe how we use *k*-rAC for the access control with the s-pull method. Afterwards, we consider each phase of the protocol flow in detail.

6.3.1.1 Access Control

The reliability of the data object revocation with s-pull is based on the authentic status within *statement*. To revoke her data object temporarily or permanently, the owner changes its status “active” to “inactive“ or “revoked”. Hence, we must ensure that only the owner can set and update the status of her data objects, i.e., to perform a write operation on *statement*. Beside the status, there is no further information stored within *statement*. As the providers inform themselves about the revocation notification, the read access to *statement* must be possible for everyone. Therefore, we do not need any access control for the read operations. i.e., *statement* is stored unencrypted.

To regulate the write access, we can use any of the three authentication mechanisms proposed with the *k*-rAC. While with the push approach we prefer the PK approach due to the integrated key distribution, in this case, our choice for a certain authentication mechanism is based on its performance. In Section 4.5.2, we evaluate the tree authentication mechanisms regarding the time response, the message, storage and computational overhead. According to the results, OTH produces the most overhead with respect to all metrics and, therefore, is not suitable for our approach. Comparing the PK and ZKP authentication mechanisms upon a write access, it is not unambiguous which one is the best choice, because they have different benefits depending on the particular metric. While ZKP provides the best results regarding storage and computational overhead on the initial write access, PK is better regarding response time, message, and computational overhead upon subsequent write accesses. The difference in computational overhead with these two authentication mechanisms, namely 4484 μs, is negligible, as it is not a significant burden for modern computers. Furthermore, the difference in the storage requirement of 8

bytes on the user side and 9 bytes on the peer side is not relevant. However, considering some system participants communicate via a mobile connection, then, the authentication mechanism with the lowest message overhead is preferable. Thus, we use the PK approach also for the status pull method.

6.3.1.2 Protocol Flow

Whenever a user requests a protected data object, the provider retrieves the status for this object from the DRS. That is the most expensive part of this revocation method both for the DRS and for the providers with respect to the number of requests. Due to its design as a P2P network, the DRS scales and is able to cope with a high number of requests. However, it would still decelerate the Internet noticeable: The request/response communication would take additional time for each and every data object on the Internet. Users could feel the delay and might get annoyed. To avoid this, we introduce a caching time in the protocol. Using caching time, we mitigate the strict *Availability*-requirement which stipulates that the owner should be able to revoke her data object at any time. To fulfil this requirement, the revocation must be instantaneous. However, if we allow a weaker *Availability*-requirement by letting the data object be revoked within, for instance, 24 hours, we are able to introduce a caching time. With this caching time (TTL), if the provider has already requested the status of a certain data object, we allow her to request it again only after the TTL has elapsed.

On the one hand, the provider might deliver already revoked data objects during the caching time. On the other hand, the provider does not need to contact the DRS as long as the TTL is valid. This approach, which is also used by the Domain Name System [62], drastically reduces the number of requests to the DRS. Furthermore, each request which is not done to the DRS has a double effect. The one effect is that the user has not to wait; the second effect is that there is no communication overhead for any request which is not done. We present a detailed evaluation of the number of requests using TTL in Section 6.5.2.1.

In Figure 6.3, we depict the protocol flow of the status pull method and describe its phases in the following.

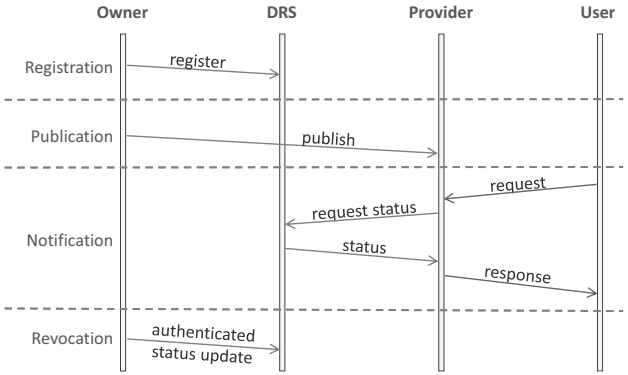

FIGURE 6.3: Protocol Overview – Status Pull Method

Registration Phase

In this phase, the owner embeds a unique identifier ID into the data object and registers this data object with the DRS by storing the corresponding *statement* with the DRS. In this case, *statement* is the delivering status (*status*) of the data object. During the registration, the owner sets the status to "active" for immediate release. Alternatively, she can set it to "inactive" to postpone the distribution of this data object to a later time.

To register her data object and to store its initial status, the owner executes *put(id, status, auth)* where *id* is the ID of the data object. The content of *auth* depends on the used authentication mechanism. Since we use the PK approach, *auth* contains a public key and a value for preventing replay attacks (cf. Section 4.4.3). We present the registration routine on the owner side in Listing 6.11.

```
1  //dObj: data object to be registered with the DRS
   //tagName: name of the metadata property, e.g., ''DRSid''
3  void owner_register(dObj, tagName){
       enum Status {ACTIVE, INACTIVE, REVOKED};
5      //1)Prepare Registration:
       //assign ID according to Listing 3.3
7      setID(dObj);
       id          = getMetadata(dObj).getValue(tagName);
9
       //create credentials according to the used authentication mechanism
11     credentials = new Credentials();
       ...
13     //save credentials locally, e.g., in HashMap<BigInteger , Object>
       wallet.put(id, credentials);
15
       //set status
17     status      = Status.ACTIVE; //or Status.INACTIVE

19     //create auth according to the used authentication mechanism
       auth        = new Authenticator();
21     ...
       //2)Execute registration with DRS
23     DRSapi.put(id, status, auth);
   }
```

LISTING 6.11: Status Pull Method – Owner Register Routine

Upon receiving this put operation, the DRS verifies whether there is already a value stored in the addressed DHT entry. If there is already a value present, the registration is rejected. Otherwise, the system completes the registration by storing the new value under the given DHT index, i.e., *auth* is stored in the ACL, and *status* is stored as the data object (cf. Listing 4.12).

The registration is anonymous, because neither *status* nor *auth* stored with the DRS includes any personal information about the corresponding owner. The ID is independent from its owner. Similarly, the value *status* does not provide any information about the owner independently whether it is a flag, a string or a random value.

Publication Phase

After registering a new data object with the DRS, the owner publishes the data object using an arbitrary service on the Internet. As with the push approach, it is important that the published

data object contains the metadata about its ID. The provider needs this information to identify data objects protected by the DRS. In the rest, this process is independent from the DRS.

Distribution Phase

For any protected data object, the provider must request its status from the DRS before delivering this data object to a user. As long as a data object is not revoked, the provider delivers it to the requesting users. Otherwise, the provider must not deliver this data object anymore. Additionally, she must delete it from her own data storage if this data storage is in her own administrative domain. For the cashing time TTL, the provider maintains a local storage *ttlStore*. An entry of *ttlStore* contains a data object ID, the corresponding *status*, and the time *timeLastReq* of the last status request for this ID from the DRS. Upon the first access by a user to a protected data object, the provider adds to *ttlStore* an entry for the corresponding ID. She removes an entry from *ttlStore* along with deleting the corresponding data object from her data storage, i.e., by revocation.

```
//dObj: data object requested by a user
void provider_on_receive_user_request(dObj){
    //for case dObj is protected by DRS, readout its ID for item "DRSid"
    id = getMetadata(dObj).getValue(``DRSid'');
    if(id != null) {
        boolean validTTL = checkLastRequestTime(id);
        if (!validTTL) {
            DRSreplies = DRSapi.get(id);
            //calculate majority over received 2k+1 status replicas
            status     = calculateMajority(DRSreplies);
            switch (status) {
                case ``active'':
                    deliver(dObj);//send dObj to user
                case ``inactive'':
                    nack();//notify user, optional
                case ``revoked'':
                    nack();//notify user, optional
                    delete(dObj);//delete dObj locally
            }
        } else {
            deliver(dObj);//TTL valid, deliver dObj to user
        }
    } else {//data object is not protected
        deliver(dObj);//return dObj to user in the usual way
    }
}
```

LISTING 6.12: Status Pull Method – Status Request by Provider

When a user requests a data object, the operating sequence for the provider is as presented in Listing 6.12: First, the provider determines whether the requested data object is protected by reading its metadata. If it is not protected, the provider delivers it to the requesting user as otherwise customary. For a protected data object, the provider checks when its status was last requested: If there is an entry for the given ID in *ttlStore*, and the TTL is not elapsed (i.e., the locally stored status is still valid), the provider can deliver the data object to the requesting user. When the TTL is elapsed or there is no entry for the corresponding ID, the provider retrieves the current status for this data object from the DRS by executing the get operation, i.e., *get(id)*, and updates *timeLastReq* or creates a new entry in her local store, respectively. Upon the provider's request, the DRS returns the status for the given index, i.e., the provider gets responses with the

status values from $2k+1$ responsible peers. Next, the provider calculates the majority function over these values to determine the valid status. Finally, she delivers the requested data object to the user if its status is "active". If the status is "inactive", the provider does not deliver the data object. If the status is "revoked", the provider also does not deliver the data object and, additionally, deletes it from own data storage. Consequently, it is not useful to set *statement* to "revoked" during the registration, as providers would delete the corresponding data object after its publication already for the very first user request. Finally, according to the particular use case, the provider potentially returns the user a notification about the revoked data object. In the case of a search engine, the provider simply does not list the revoked data object in the search results.

Revocation Phase

In this phase, the owner changes the status of her data object by executing *put(id, status, auth)* (cf. Listing 6.13). As described above, only she is allowed to change the status.

```
//dObj: data object requested to be revoked
//tagName: name of the metadata property, e.g., ''DRSid''
//newStatus: value for updating the status
void owner_revoke(dObj, tagName, newStatus){
    //readout ID of dObj
    id = getMetadata(dObj).getValue(tagName);

    //set new value for status
    status = setStatus(newStatus);

    //create auth according to the used authentication mechanism
    //by using credentials stored locally
    auth   = new Authenticator();

    //store the new status with DRS
    DRSapi.put(id, status, auth);
}
```

LISTING 6.13: Status Pull Method – Revocation Request by Owner

The DRS uses the received *auth* to verify whether the user is authorized to update the status. For that, each responsible peer uses the locally stored ACL of the DHT entry under the given index (i.e., the ID of the data object). Thereupon, the responsible peer authenticates the owner. If the authentication was successful, the responsible peer updates the stored status value with the new value (cf. Listing 4.12). That means that the providers are notified about the new status of that data object with the subsequent get operations in the distribution phase. If the initial status was "inactive", by updating it to "active", the owner allows the distribution of the corresponding data object by providers. To revoke the data object, she sets the status to "revoked". She changes the status from "active" to "inactive" for the case the owner wants to revoke her data object with the possibility to allow its distribution later again. In this case, the data object is not deleted but hidden for the users, as the providers do not deliver it.

6.3.2 Key Pull Method

Using the k-pull method, the owner encrypts her data object before publishing it on the Internet and stores the decryption key (*key*) with the DRS. To revoke the data object, the owner deletes

the key from the DRS. However, k-pull is not a reasonable method to notify providers about owner's revocation requests for the following reasons. By implementing this revocation method the same way as with the s-pull method, all processes and optimizations of s-pull can be applied to k-pull. Accordingly, the provider retrieves the corresponding key each time when a user requests a protected data object. If the key is available, the provider decrypts the data object and delivers it to the user. To optimize this process, the provider caches the key for the allowed caching time to reduce the number of requests to the DRS. This way, the provider obtains access to the decrypted data object and even stores the key locally for a certain time. We assume that providers follow the law and do not misuse both the decrypted data object and the key. However, we have this assumption also with the s-pull method. That is, we achieve the same purpose also with s-pull but without additional effort caused by the encryption. Hence, there is no benefit from the k-pull method in this version of use.

However, the k-pull method is suitable for the scenario when only certain users should get access to the protected data object. In this case, it is an advantage that the provider operates with the encrypted data object, because the owner wants to protect it also against the provider when she does not belong to the authorized users (*closed group*). To distribute the key only to the closed group, the owner transmits it out-of-band (e.g., via secure instant messaging) or stores it with the DRS. In the latter case, she has the possibility to revoke her data object by deleting the key from the DRS. Hence, a provider does not have to contact the DRS at all and delivers the encrypted data object directly to any requesting user. If the user has the key, she decrypts it by herself. To protect the key, we must ensure that only the users with appropriate rights are able to read or delete the key with the DRS. Providing such an access control, we also achieve our security goals for the given attacker model (cf. Section 2.3) with k-pull for closed groups, since no further user can obtain the key after its revocation with the DRS.

Although the s-pull and the k-pull methods are similar, they have different applications. With s-pull, the owner publishes her data object unencrypted, and providers transfer it worldwide over the Internet as long as the data object is not revoked. In contrast, with k-pull, the owner specifies who is allowed to get access to her data object by encrypting it. Though providers unrestrictedly deliver a data object worldwide also with this method, after all, it has the effect that providers only deliver it to a closed group of users. Here, our assumption is that an authorized user downloads the data object only for own use and does not transfer the protected data in its decrypted state to unauthorized users.

Due to the fine-grained access control possible with *k*-rAC, we additionally are able to realize k-pull in such a way that the owner can revoke her data object both for the whole group or for individual users. Hence, after the owner revoked the access for a certain user, the remaining group members can still access the key. With this method, the revocation refers to the deletion of the decryption key with the DRS. However, users that already are in the possession of the key can still decrypt the revoked data object. They also might disclose the key to unauthorized users both before as well as after the revocation. Since we aim to prevent the access to a data object after its revocation for new users, this type of attacks is beyond our attacker model. To still prevent such attacks, we must provide mechanisms similar to Digital Rights Management (DRM) schemes. Accordingly, we must avoid that users can save or copy the key before its revocation.

Considering the above, we deduce three security levels for the k-pull method:

- Level I – the revocation occurs for the whole closed group. After the revocation, the key is not accessible for new users. This level corresponds to our requirements.

- Level II – includes Level I. Additionally, the owner can revoke her data object for individual members of the closed group. After the revocation, the key is not accessible for the group members with the revoked access right and for new users.
- Level III – includes Level II. Additionally, the authorized users can decrypt the protected data object, but they cannot readout the key and, therefore, cannot copy or store it locally. The revocation has the same effect as with Level II. Besides that, no group member has a readable key replica stored locally.

In general, the key pull method is a generalized version of existing approaches for deleting data on the Internet, e.g., Vanish, EphPub, or X-pire! (cf. Section 7.2). Similar to the k-pull method, these approaches are based on encryption and differ from each other in proposed solutions for the key management. The achievement of an appropriate access control as required in Level III necessitate a trusted execution environment (TEE). The only difference to our approach using Level I is that we do not require additional hardware. Although this leads to the associated constraints, the Level I can be unproblematically achieved.

The presented security levels provide different challenges regarding the access control. We consider these challenges and analyse an appropriate access control for each level in the next subsection. Besides the differences in the access control, the revocation process is the same for all three levels. In the following, we particularize the protocol flow for Level I as it corresponds to our requirements and purpose.

6.3.2.1 Access Control

With the k-pull method, the owner uses the DRS to distribute the decryption key k_d to authorized users. Due to malicious peers, we cannot store k_d within *statement* unencrypted (cf. Section 4.3). We propose to use the PK approach of *k*-rAC, as it provides besides the authentication mechanism also the key distribution. With this approach, the key distribution is based on the ACL: each user can request the ACL assigned to a certain data object. Hereby, the ACL contains items where k_d is encrypted with the individual public key of each user who should have read access (cf. Section 4.4.2). Therefore, only a user that is in possession of a corresponding private key is able to decrypt k_d and, then, decrypt the protected data object. Consequently, for k-pull, the content of *statement* is irrelevant and can, therefore, contain a random number. We use this number to verify the owner's signature during the authentication process.

Regarding the access control with the DRS, common for all three security levels are the following properties: The owner is set by the system during the first access to the DHT entry for the given index via the put operation. She obtains the right to alter the corresponding ACL by default. Additionally, the owner can delegate the ACL administration to certain users (*admins*) by granting them the admin right. An admin has the permission to (1) add new ACL items for users with the read access and (2) to remove any ACL items except that which belong to the owner or other admins. The owner or an admin adds users to the closed group or removes them from the group by editing the ACL (cf. Section 4.4.2). Specifically, to build the closed group, the owner or an admin uses the set operation to assign the read right for intended users (*group members* or *members*). The closed group can be defined at once, i.e., the owner or an admin adds all corresponding ACL items within one set operation. Additionally, further group members can be added to the ACL later as required. Despite the owner and admins, the group members have only read access to the key. Hence, with the k-pull method, we provide rather an access control to a published data object on the Internet then a service for revocation notification.

In the following, we consider the differences for the appropriate access control with each k-pull level and point to the associated challenges. Since in this case *statement* is not relevant, we refer below only to the API operation *set*.

Level I

With this security level, the goal is to ensure that no new user can get k_d from the DRS after the data object was revoked. As mentioned above, we regulate the access to k_d by managing the ACL according to the PK approach in *k*-rAC. To revoke a data object with this level, the owner removes the ACL items of **all** group members, including these that belong to the admins. This way, we ensure that no new user can be authorized without owner's consent. Only the owner can republish the key if she decides to publish her data object again.

The main characteristic of this level is that we pursue a revocation of a data object and not of the access for a certain group member. When removing the ACL item belonging to a certain group member, this former member may already have k_d if she requested it from the DRS before her access right to it was revoked. Therefore, the revocation for a certain member is successful only when this member is not already in possession of the key. The same applies to the whole group: after the revocation of the data object, all members are still in possession of the corresponding k_d. However, under the assumption that group members use the data object for their needs only, no further user can decrypt the revoked data object.

Level II

For this security level, the goal is a dynamic closed group, i.e., it should be possible to add and exclude individual group members. Hereby, only the actual group members should be able to access the corresponding data object, i.e., to decrypt it. Adding of new group members can be realized the same way as with Level I. However, in contrast to Level I, the former group member should not be able to decrypt the data object even if she already decrypted it previously. Hence, we must prevent that an excluded member can use her k_d to decrypt the data object. To achieve that, the owner encrypts her data object with a new key k'_d. After that, she replaces the previously published encrypted data with the new one. To distribute the k'_d, the owner encrypts it with the individual public key of each user who still should have access to this data object. For subsequent requests for this data object on the Internet, the excluded member has no access to the new decryption key and cannot decrypt it. However, by storing locally the previously encrypted data object and the corresponding k_d, a group member can access the data object even after her exclusion from the group. Moreover, the member can store the data object locally after she decrypted it for the first time to be independent from a potential access revocation.

While this approach provides a conditional protection against subsequent accesses of an excluded member for dynamic data objects, e.g., text files in collaborative work, it is not reliable otherwise. Hence, this level works only under the assumption that group members have no local replica of the data object and only access it via the Internet. Another issue with this level is that the owner must republish the anew encrypted data object with each provider that had published the outdated replica. If the owner used more than one provider, her effort to republish a new version increases with each additional provider. To cope with the drawbacks of this level, we cannot rely only on the re-encryption and access control via DRS – we must consider additional protection mechanisms on the user devices as proposed with Level III.

Level III

The goal of Level III is similar to the goal of Level II. However, with Level III, the group members should not be able to access the k_d directly. Hence, the challenge with this level is that we must enable the decryption of the data object for members without revealing them the key.

While the access control on the DRS side as described above is valid also for this level without additional effort, we need a supplementary mechanism to protect the key on the user's device. This can be realized by using a trusted execution environment (TEE) [79]. With this technology, an application is executed in a secure area by using the device resources (e.g., main processor or memory) but isolated from the operating system. Specifically, a TEE provides a secure storage for data and cryptographic key, and prevents accesses to them by the operating system, other applications, the user, or attackers. Applying this technology to Level III, the API on the member's device requests the k_d and decrypts the data object but does not allow access to the k_d. After the group member closes the application which is protected with the TEE, she cannot access the data object, as it is in the encrypted state again. Due to the revocation, the k_d should not be stored on the member's device. Hence, the API must request the key from the DRS for each member's access to the encrypted data object. Consequently, if the key is revoked for the requested member, she cannot access the data object anymore – it is on her device but in the encrypted state.

Doing as described above, the distribution of the encrypted data object is independent of the distribution of the corresponding k_d. Specifically, the owner publishes her encrypted data object with one or more providers and manages the access to it only by updating the ACL with the DRS. In this case, no re-publishing of the re-encrypted data object as with Level II is needed. However, for this to work, user's device must embed a TEE technologies, e.g., Intel SGX [61] or ARM TrustZone [10]. There are alternative technologies to TEE, e.g., trusted platform modules or encrypted execution environments (cf. the review for secure and trusted execution in [78]). Regardless of the particular technology for secure execution, we need an appropriate hardware as well as the associated software components on each involved user device to provide a reliable revocation with this level.

6.3.2.2 Protocol Flow

In Figure 6.4, we present the protocol flow of the k-pull method for Level I and describe its phases in the following. Hereby, we assume that owner is in possession of public keys of the users to whom she intends to grant access to her data object.

Registration Phase

In this phase, the owner registers her data object with the DRS to get a DHT entry, where she can store k_d (analogous to a lockable box in a post office). Similar to other methods, the owner first assigns an ID to the data object to ensure its identification and to define its position in the DHT. Next, she generates a randomized symmetric encryption key k_d. Additionally, she must generate an authenticator *auth* for the access control to the obtained DHT entry. For that, due to the PK approach for the access control, the owner generates a public/private key pair. Finally, she creates a random number r to store it as *statement* in the DHT (i.e., DRS). The owner uses this random number in subsequent accesses to authenticate herself by signing it with her private

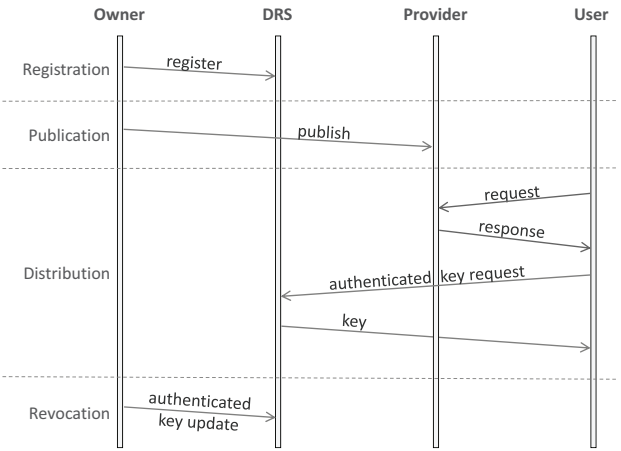

FIGURE 6.4: Protocol Overview – Key Pull Method

key. To register her data object with the DRS, the owner executes *put(id, r, auth)*. We present an overview of the registration routine in Listing 6.14.

```
//dObj: data object to be registered with the DRS
//tagName: name of the metadata property, e.g., ''DRSid''
void owner_register(dObj, tagName){
    //1)Prepare Registration
    //assign ID according to Listing 3.3
    setID(dObj);
    id                      = getMetadata(dObj).getValue(tagName);

    //generate public/private key pair and prevent replay attacks
    credentials             = new Credentials();
    credentials.keypair     = new AsymmetricKey();
    credentials.window      = SlidingWindow.generateSlidingWindow();
    credentials.window.mid  = SlidingWindow.setMessageID();
    //generate symmetric encryption key
    credentials.encKey      = generateEncryptionKey();

    //generate random random for statement
    credentials.r           = generateRandomNumber();

    //save credentials locally
    wallet.put(id, credentials);

    //create authenticator
    auth                    = new Authenticator();
    auth.pk                 = credentials.keypair.pk:
    auth.mid                = credentials.window.mid;
    auth.sign               = null;//due to initial access

    //2)Execute registration with DRS
    DRSapi.put(id, r, auth);
}
```

LISTING 6.14: Key Pull Method – Owner Register Routine

By receiving the store request, the DRS verifies whether the DHT entry for the requested index is empty. If it is already occupied, the system rejects the registration. Otherwise, it accomplishes the registration by storing *r* and *auth* under the given index (cf. Listing 4.11).

The owner can define a closed group directly after she has registered her data object or later, e.g., after the publication or in the distribution phase. For that, she first encrypts k_d with the public keys of the intended users. By building an ACL item for a group member, she also decides whether this member should get the admin right or only the read access. A group member with the admin right can decrypt the k_d with her own private key as well as other members, but she is also granted to distribute it to further users by extending the ACL with the associated items. In Listing 6.15, we demonstrate a routine for building ACL items for a given number of intended users with the same access right. Hereby, the fingerprint of the member's public key serves as her authenticator. In the distribution phase, the DRS uses it when responding to a user's retrieve message for k_d (cf. Section 4.4.2). After completing the ACL, the owner executes the set API operation to communicate the access rights to the DRS, i.e., *set(id, acl, auth)*.

```
//id: ID of the protected data object
//pk_members: array with public keys of the intended users
//right: the read or admin access right
void owner_add_group_member(id, pk_members, right){
    //get credentials of data object with given id
    credentials       = wallet.get(id);
    AclItem  aclitem;

    for each pk in pk_members{
        aclitem       = new AclItem();
        aclitem.auth  = pk.getFingerprint();
        //encrypt symmetric encryption key with user's public key
        aclitem.key   = encryptKey(credentials.encKey, pk);
        aclitem.setRight(right);
        acl.add(aclitem);
    }

    //create authenticator, i.e., auth:=(pk, mid, sign)
    auth              = new Authenticator();
    auth.pk           = credentials.keypair.pk;
    //sign random number to authenticate set request to DRS
    auth.sign         = calculateSignature(credentials.keypair.sk,
                              credentials.r, credentials.window.mid);
    credentials.window.mid++;

    //update the ACL in DRS, i.e., distribute the decryption key
    DRSapi.set(id, acl, auth);
}
```

LISTING 6.15: Key Pull Method – Granting Access to Encryption Key by Owner

In turn, the DRS verifies with the received *auth* whether the executor of the put operation is authorized to update the ACL for the corresponding index (cf. Listing 4.17). If the verification was successful, the system adds the items received with *acl* to the ACL under the given index. Otherwise, the ACL update request is rejected.

Also with k-pull, we provide an anonymous registration as neither the ID of the data object nor the random number *r* or the *auth*-value do not disclose any information about the owner. To keep the owner anonymous also by using the DRS for multiple data objects, the owner should generate for each of them a different public/private key and another random number.

Publication Phase

With the k-pull method, the owner publishes her data object encrypted with a symmetric cipher. Before publishing, the owner must ensure that the ID is embedded as metadata into the data object. Apart from that, the publication process is independent from the DRS – the owner can use any service on the Internet to publish her data object.

Distribution Phase

When a user (an arbitrary user or a group member) requests an encrypted data object, the provider sends it to the user in the same way as with unencrypted data objects. Hence, there is no additional effort for the provider with delivering data objects protected by the k-pull method.

After receiving the encrypted data object, the user readouts its ID to request the corresponding encrypted symmetric key k_d from the DRS. For that, she executes the get operation, i.e., *get(id, auth)* where *auth* is the fingerprint of her public key. On the basis of the received parameters, the DRS returns the corresponding ACL item as presented in Listing 4.14. If there is no ACL item belonging to the given *auth*, the system sends back a negative-acknowledgement signal to inform the user about the error. When receiving the ACL item, the user (in this case, a group member) decrypts the k_d with her private key. After that, she decrypts the data object using this k_d. The pseudocode for this procedure is shown in Listing 6.16.

```
//encryptedObj: protected data object
//tagName: name of the metadata property, e.g., ''DRSid''
//keyPair: member's public/private key pair
void member_decrypt_dObj(encryptedObj, tagName, keyPair){
    //readout ID of encrypted dObj
    id              = getMetadata(dObj).getValue(tagName);

    //request the encrypted decryption key from DRS
    DRSreplies      = DRSapi.get(id, keyPair.pk);//pk is public key

    //calculate majority over received 2k+1 replies
    encryptedKey    = calculate_majority(DRSreplies);

    //decrypt the symmetric encryption key with own private key
    symmetricKey    = decrypt(encryptedKey, keyPair.sk);

    //decrypt data object with the symmetric key
    dObj            = decrypt(encryptedObj, summetricKey);
    walet.put(id, key);//optional, store the key locally
}
```

LISTING 6.16: Key Pull Method – Request for Decryption Key by Member

As pointed out before, any user can request k_d for a certain data object. Hereby, it is irrelevant if she knows a valid public key fingerprint or brute forces a right one – without the corresponding private key, she cannot get access to k_d.

Revocation Phase

In this phase, the owner revokes her data object by deleting the ACL items of all group members except her own ACL item in the corresponding DHT entry. This way, she deletes the encryption key, which is needed to read the protected data object. For that, she updates the values of the affected ACL items. Then, she communicates the revocation to the DRS by executing *set(id, acl, auth)*, where *acl* contains the updated ACL. In Listing 6.17, we exemplarily set the value of the parameter *acl* to *null* to indicate the revocation request.

```
//id: ID of data object that should be revoked
void owner_revoke_key(id){
    //get credentials of data object with given id
    credentials = wallet.get(id);

    //mark as revocation request
    acl         = null;

    //create authenticator, i.e., auth:=(pk, mid, sign)
    auth                = new Authenticator();
    auth.pk             = credentials.keypair.pk;
    //sign random number to authenticate set operation with DRS
    auth.sign           = calculateSignature(credentials.keypair.sk,
                                credentials.r, credentials.window.mid);
    credentials.window.mid++;

    //update the ACL in DRS
    DRSapi.set(id, acl, auth);
}
```

LISTING 6.17: Key Pull Method – Revocation of Decryption Key by Owner

On receiving the event message *setacl* (cf. Listing 4.11), the DRS authenticates the requesting user and verifies whether she is authorized to update the ACL. If she has the owner right, the system deletes the ACL items as requested. In the case, an admin revokes the read access for the group members, we must ensure that she cannot delete the ACL items of other admins. Therefore, in the case the requesting user has the admin right, the system accepts ACL updating only for the ACL items limited to the read access. Finally, if the authentication failed, the ACL updating is rejected. In our example, we update the ACL by setting its value to null. Thereupon, based on the access right of the requesting user, the DRS decides which particular ACL items should be updated , i.e., all except the owner's or only those that belong to members. The pseudocode of this procedure is shown in Listing 6.18. In contrast to Listing 4.13 were we present the general ACL updating procedure, here, we integrate the deleting of the ACL items accordingly to the highest hierarchical access right of the requesting user, i.e., *o* or *a*.

```
//sender: ID of the requesting peer
//index: ID of the corresponding data object
//acl: ACL items for updating access rights
//auth: user's authentication value
void drs_on_receive_setacl(sender, index, acl, auth)
{
    //get requested DHT entry
    dataset ds = localstore.get(index);

    //get user ACL item from ACL
    acl user = get_user_acl(ds.acl, auth);
```

```
13      //the user has the owner right
        if((user != null) &&
15          (verify_auth(user) == true) && (user.rights == 'o') ){
            for each acl_item in acl{
17              if(acl_item.rights != 'o'){
                    acl.delete(acl_item);
19              }
            }
21      }
        //the user has the admin right
23      elseif((user != null) &&
            (verify_auth(user) == true) && (user.rights == 'a') ){
25          //delete all items except the one associated with owner or admin
            for each acl_item in acl{
27              if((acl_item.rights != 'o') && (acl_item.rights != 'a'){
                    acl.delete(acl_item);
29              }
            }
31      } else {
            //ACL update rejected
33      }
}
```

LISTING 6.18: Key Pull Method – ACL Update by DRS

6.4 Maintenance

With the DRS maintenance, we mean the release, of a DRS entry after its owner has revoked the associated data object, i.e., deregistration of a data object. Otherwise, over time, the DRS will contain unused entries. To recall, we use a DHT with a 160-bit ID space for the DRS, i.e., there are numerous empty entries for new registrations (cf. Section 6.5.1). Nevertheless, it is preferable to prevent a single responsible peer from managing unused DHT entries. In the following, we consider the possibilities to automatically maintain the DRS with the push and pull approaches separately.

Push approach: With both revocation methods o-push and s-push, after the revocation of a data object is performed, the affected providers do not offer it anymore, since they delete it from own servers. Hence, the further storing of their contact information with the DRS is not needed. Therefore, we propose to implement a "cleaning" function along with the revocation procedure. For that, we can use the set operation to signal the responsible peers that the corresponding entry can be deleted. Due to *k*-rAC, only the owner has the full control over her DHT entry. Accordingly, she is also allowed to assign any value to the ACL. We use the value *NULL* as an indicator for an unused entry as follows: The owner executes *set(id, acl, auth)* where *acl* has the value *NULL*. She authenticates herself for updating the ACL as usual with *auth*. After that, she sets the parameter *acl* to *NULL*. The extension of the procedure for the ACL updating by responsible peer with a new instruction can be realized as proposed in Listing 6.19: Before writing the new ACL value, the responsible peer checks whether the new value for the owner is *NULL*. If it is *NULL*, she deletes the affected DHT entry from her local store.

```
//sender: ID of the requesting peer
//index: ID of the corresponding data object
//acl: ACL items for updating access rights
//auth: user's authentication value
void drs_on_receive_setacl(sender, index, acl, auth)
{
    //get requested DHT entry
    dataset ds = localstore.get(index);

    //get user ACL item from ACL
    acl user = get_user_acl(ds.acl, auth);

    // the user has the owner right
    if ((user != null) && (verify_auth(user) == true)
        && (user.rights == 'o') ) {
        if (acl == NULL){
            localstore.deleteEntry(index);
        }
        //as ever (cf. Section 4.4)
    }
    //as ever (cf. Section 4.4)
}
```

LISTING 6.19: Deregistration with the Push Approach

Pull approach: With s-pull, we cannot delete the DHT entry during the revocation procedure – revocation notification works by propagating the status with the value "REVOKED". We assume that, over the time, the revoked data object will be no more requested. Therefore, we can realize the deregistration by introducing a new parameter *lastRequest* within a DHT entry. Each time when a provider requests the corresponding status, we update the value of *lastRequest* with the date of this request. Additionally, we must define a system parameter for the "cleaning time" that defines after which period of time it is allowed to delete the DHT entry of a revoked data object, e.g., 50 years. The peer checks periodically the entries that it is responsible for to find the elapsed entries, i.e., entries that have the status "REVOKED" and that were last requested more than 50 years ago. For k-pull, we also do not apply the deregistration within the revocation process, as the owner can decide to distribute the key for the corresponding data object later again. Therefore, we rely also with k-pull on the parameters *lastRequest* and "cleaning time".

6.5 Evaluation

In the following, we evaluate the DRS with respect to the requirements described in Section 2.2. First, we analyse the system regarding its security properties. Herewith, we consider the requirements *privacy* and *no censorship*. After that, we analyse the DRS regarding its *availability* and *scalability*. For that, we evaluate the performance for each revocation method by analysing their message, storage, and computational overheads. Finally, we analyse the *usability* of the DRS.

6.5.1 Security Analysis

Below, we analyse the reliability of the DRS regarding the security goals defined in Section 2.3. Accordingly, no privacy violations, no malicious and hindered revocations should be possible in our system. Based on the attacker classification given in Figure 2.3, we consider the attacker classes Eavesdropper, Censor and Denier in our security analysis separately.

6.5.1.1 Eavesdropper

The Eavesdropper eavesdrops passively the communication between the system participants with the goal to identify which protected data objects belong to whom, and how widely is a certain data object spread over the Internet. To achieve her goal, the Eavesdropper must be able to read the data transferred between the communication partners within our system. To protect the data in transit, we use SSL/TLS [23] for all communication with the DRS. Due to the encrypted communication, the Eavesdropper cannot determine neither the message type nor the ID involved in a particular interaction between the communication partners in the system. Even the strong Eavesdropper that is able to eavesdrop the whole communication in the system cannot analyse the traffic. She can only identify the users and providers that interact with the DRS. However, she cannot derive any information about the exchanged content to be able to sort data objects by their owners. To analyse the spreading of certain data objects, the Eavesdropper also needs to know which ID is affected by a particular interaction with the DRS. Since the DRS processes requests for multiple data objects, it is not possible to distinguish requests for different IDs by eavesdropping the communication with the system.

6.5.1.2 Censor

The Censor also eavesdrops the communication with the system. In contrast to the Eavesdropper, the Censor actively uses the eavesdropped information for message manipulation and message injection with the goal to revoke certain or arbitrary data objects instead of their owners. However, as the communication between the system participants is encrypted, she cannot use the intercepted information to manipulate the revocation notifications. Even when assuming that the Censor could perform a man-in-the-middle attack [17] to inject some manipulated messages into the communication, she still cannot trigger the revocation of a data object due to the access control with the DRS – she needs to authenticate herself with the DRS to initiate the revocation process. Solely with s-push, she can obtain the owner's authenticator, when the owner sends it to the DRS in the revocation phase. However, with the integrated safeguards, the owner herself sends it with the goal to revoke her data object. Hence, the Censor cannot successfully attack our system.

6.5.1.3 Denier

The Denier's goal is to prevent the revocation of a certain data object. In general, the attacker of this class must hinder that the owner's revocation notification reaches the provider to achieve her goal. For that, the Sneaky Denier manipulates the communication. In contrast, the Rough Denier attacks the infrastructure of the system by subverting the communication. Against the Sneaky Denier, we rely on the encrypted communication. To cope with the Rough Denier, we consider her possibilities to attack the revocation notification on its way (1) between the owner and the DRS or (2) between the DRS and the providers. For the first case, we rely on the k-resilience of the DRS. Accordingly, the owner sends her revocation request to $2k+1$ responsible peers over $2k+1$ disjoint paths (cf. Chapter 5). The Rough Denier must subvert the majority of these peers to interrupt the revocation of a certain data object (cf. security analysis in Section 4.5.1). To hinder as many as possible arbitrary revocation requests, the Rough Denier must interrupt the communication between owners and the DRS for any request. To achieve this goal, the Rough Denier needs a global view of the distributed system network. While a weak attacker (e.g., one or several common Internet users) is not a real threat for the DRS, a strong one (e.g., the

government) can deny the service of the DRS. Such attacker has access to any arbitrary point of the Internet at any arbitrary time in the area she has the authority. Hence, she can disconnect some users. However, assuming the DRS is distributed over the world, it would be even for the strong Rough Denier a hard task to completely interrupt the communication between owners and the DRS, e.g., in whole Europe or even worldwide.

For the second case when the revocation request is on its way between the DRS and the providers, the Rough Denier's handling is the same as with the first case. Additionally, she can attack the provider to hinder her to follow the revocation request for certain or arbitrary data objects. If the attacker is a provider, she can simply ignore the revocation requests. However, the GDPR requires that the provider must obey the users' deleting requests. Therefore, we assume that most of the providers are not interested in hindering the availability of the DRS. A strong Rough Denier, e.g., a secret service, can force multiple providers to deny the revocation requests. However, even she is not able to control providers of different countries.

Another possibility to deny the service of the DRS is to occupy all DRS entries. Then, no new data objects can be registered with the DRS and, therefore, the revocation service cannot be offered. For the DRS, we use a DHT with 2^{160} entries (cf. Section 2.5). To demonstrate the enormous effort needed for writing the entire DHT, we assume that the attacker has 100 years to finish this attack. To determine how many put operations she must execute per second in 100 years, we calculate $2^{160} \div (60 \cdot 60 \cdot 24 \cdot 365 \cdot 100)$. Then, the attacker would need to perform $\approx 4,6 \cdot 10^{38}$ put operations per second. Assuming, every Internet user available worldwide owns a device capable to register a data object with the DRS, i.e., $\approx 3,5$ billion users [8]. Further, assuming the Denier distributes the attack effort to all these users, the effort of $\approx 4,6 \cdot 10^{29}$ operations per second and per user still makes this attack unrealistic. Nevertheless, the usage of the "cleaning time" as proposed in Section 6.4 prevents the system against the filled DHT: The maliciously registered entries will not be associated with a published data object and, therefore, not requested by the system users. Hence, these entries will be free again after a certain time, e.g., after 50 years.

6.5.1.4 Summary

Our system provides security against the Eavesdropper and the Censor. Furthermore, not even the strong Denier can violate the owner's privacy, as our system does not store any personal information. There is no link between the owner and the data objects. Therefore, even if the attacker has a global view of the system, she will not gain any additional information. Particularly, only the data owner in possession of the correct secret can trigger the removal of a data object from all providers who serve it. Assuming that the DRS is distributed worldwide, it is also impossible for governments or agencies to trigger the removal of specific data objects, i.e., to use the DRS for censorship. The peers responsible for *statement* of a data object might not be located within the jurisdiction of a certain government. Since we require that each *statement* is always stored on $2k+1$ distinct peers, any government would need to gain access to at least $k+1$ peers. This can be made arbitrary difficult by increasing k. Surely, a government could contact the providers within their jurisdiction directly and request certain data objects to be removed. However, this is outside the scope of our approach, as it can also be done today without the DRS. Thus, our service does not introduce any new means for censorship of data objects.

Finally, the DRS does not introduce any new privacy risks. The owner is authenticated by means of an authenticator *auth* which contains no personal information. Therefore, the authentication

procedure does not leak any information about the owner's identity. Additionally, it is impossible to find or identify all objects from the same owner, since the owner uses a different *auth* for each data object. Thus, the publicly available information stored in the DRS does not provide any new information.

Note that the DRS does not prevent any malicious entity to download a protected data object, remove its ID, and re-upload it. Since there is no ID with this "new" data object, the provider will not treat it as a protected data object and, thus, will not delete it on the original owner's request. In this case, the original owner must contact the provider directly and use the classical way to delete her data object – if necessary with a court order based on the GDPR. Even though there will be cases in which our system is circumvented by malicious users, it will reduce the amount of manual requests for data removal for any provider.

6.5.2 Performance

In general, a P2P network is available as long as there are enough peers online. Hence, assuming enough peers in the systems, the owner can access the DHT at any time for changing *statement* of her data objects or for registering new data objects. Furthermore, also providers can access the DHT at any time to register publishing of a protected data object with the push approach, or to verify *statement* of a data object with the pull approach. Similarly, the requirement for a scalable service can be fulfilled by the underlying P2P network, as it gracefully scales with the number of peers. For instance, the DRS should handle five million data objects per minute, i.e., 2^{22}. Assuming these are all registration requests, the DHT will only become full after $6.6 \cdot 10^{35}$ years when using the index space of 160 bits. Assuming there are $10,000$ peers in the network, then, each peer must handle 8.3 requests per second. Hence, any number of requests can be handled if there are enough peers online.

Below, we evaluate the performance of the revocation methods elaborated in this chapter, i.e., the methods o-push, s-push, s-pull, and k-pull. Our goal is to identify the advantages and disadvantages of each revocation method by comparing them with each other. For that, we establish an analytical model to determine the performance of the particular method with respect to time, message, storage, and computational overhead for each system participant. By referring to the system model (cf. Section 2.1), the DRS, the owner, the provider, and the user are the system participants that we have to consider evaluating a single revocation method. In general, the DRS is a DHT with an access control based on k-rAC. Since we already evaluated k-rAC, we omit the overhead caused by the access control of the DRS to avoid duplicate calculations. Specifically, we do not consider all messages and calculations associated with the access control. Hence, in the following, we theoretically analyse only the overhead which is produced by using a particular revocation method. Hereby, we do not cover the overhead caused by other factors that also influence the overall overhead value, e.g., the used programming language or individual implementations. We focus on the parameters that are associated solely with the revocation methods to determine their lower bound.

6.5.2.1 Message Overhead

To evaluate the message overhead, we analyse the number Mn_{part}^{phase} and size Ms_{part}^{phase} of messages transferred in a single protocol phase for each revocation method. This way, we identify the communication effort for each system participant (in the following abbreviated as *part*).

Considering the protocol flow for all revocation methods, we distinguish between 8 different message types as follows:

- `put` – storing a value with the DRS,
- `get` – requesting a value from the DRS,
- `return` – returning a value by the DRS,
- `register` – registering the publication of a data object by the provider,
- `ack/nack` – respond to a write access by the DRS or a provider,
- `verify` – checking provider's publishing by the DRS,
- `prove` – proving publishing by the provider,
- `revoke` – revoking a data object by the owner or the DRS.

In Tables 6.1 – 6.4, we give an overview of message type and message number involved in each protocol phase for each revocation method by protecting a data object with the DRS. Additionally, we specify the size for every message depending on its parameters. The notation in a table cell reads as follows: "`message type`(parameter) / message size". Hereby, when a parameter is a sort of a collection, e.g., array, we designate it by the square brackets $[n]$, where n is the number of elements in the collection. From these tables, we can determine the effort for a particular system participant by summing up the amount of messages given in a row. In contrast, from the columns, we can derive the message overhead of a single protocol phase. In the following, we compare the overhead regarding the message number and message size separately.

Message Number Overhead

As shown in Tables 6.1 – 6.4, the message number overhead for each revocation method in the registration phase is the same: the owner sends a put message to register her data object, and the DRS sends back a response to inform whether the put operation was successful or not, i.e., 2 messages in total. There is no overhead for the provider and user in this phase.

In the remaining phases, we have to consider the push and pull approaches separately.

In the publication phase with the push approach, only the provider and the DRS produces message overhead by communicating with each other for registering the publishing of a data object. This results in 3 messages for each of them, i.e., 6 messages in total. In contrast, there is no message overhead for any system participant with the pull approach.

In the distribution phase, the push approach requires no additional message. With s-pull, there is an overhead of 2 messages, i.e., one for the provider and one for the peer while processing the status request. Similarly with k-pull, the user and the peer exchange in total 2 messages by retrieving the decryption key.

Part	Message Number Mn_{part}^{phase} / Message Size Mn_{part}^{phase} (Byte)			
	register	publish	distribute	revoke
Owner	put(id, contact[]) / 20	–	–	get(id) / 20 $p\cdot$ revoke(URL, id, sign) / $p\cdot 2166$
Peer	ack(flag) / 1	return(pk) / 80 verify(URL) / 2083 ack(flag) / 1	–	return(contact[p]) / $p\cdot 2189$
Provider	–	register(id) / 20 put(contact) / 2189 prove(id) / 20	–	ack(flag) / 1
User	–	–	–	–

TABLE 6.1: Message Overhead with O-Push

Part	Message Number Mn_{part}^{phase} / Message Size Ms_{part}^{phase} (Byte)			
	register	publish	distribute	revoke
Owner	put(id, contact[]) / 20	–	–	get(id) / 20 revoke($p\cdot$(URL, id, sign)) / $p\cdot 2166$
Peer	ack(flag) / 1	return(pk) / 80, verify(URL) / 2083 ack(flag) / 1	–	return(contact[p]) / $p\cdot 2189$ $p\cdot$ revoke(URL, id, sign) / $p\cdot 2166$
Provider	–	register(id) / 20 put(contact) / 2189 prove(id) / 20	–	ack(flag) / 1
User	–	–	–	–

TABLE 6.2: Message Overhead with S-Push

Part	Message Number Mn_{part}^{phase} / Message Size Ms_{part}^{phase} (Byte)			
	register	publish	distribute	revoke
Owner	put(id, status) / 28	–	–	put(id, status) / 28
Peer	ack(flag) / 1	–	return(id, status) / 28	ack(flag) / 1
Provider	–	–	get(id) / 20	–
User	–	–	–	–

TABLE 6.3: Message Overhead with S-Pull

Part	Message Number Mn_{part}^{phase} / Message Size Ms_{part}^{phase} (Byte)			
	register	publish	distribute	revoke
Owner	put(id, r) / 28	–	–	set(id, NULL) / 20
Peer	ack(flag) / 1	–	return(id, key) / 84	ack(flag) / 1
Provider	–	–	–	–
User	–	–	get(id) / 20	–

TABLE 6.4: Message Overhead with K-Pull

In the revocation phase, with the push approach, the overall overhead consists of $4+p$ messages, where p is the number of providers to be notified about owner's revocation request. Specifically, with o-push, the owner has the highest message overhead, as she notifies the p affected providers. In contrast, with s-push, the peer transfers the n notifications. With both push methods, the provider only sends one confirmation message. Using the pull approach, the message

number overhead is the same for its both methods: the owner and the peer exchange in total 2 messages while updating the ACL.

Message Size Overhead

The second metric of the message overhead is the message size. To determine the size of a particular message, we consider the parameters that are sent within it. In the following, we first determine the size of each single parameter. Based on these values, we evaluate the message size overhead for the particular revocation method. According to Tables 6.1 – 6.4, we handle 8 different parameters which we describe bellow:

- *id* represents the ID of a data object and, at the same time, the DHT index where this data object is stored with the DRS. As presented in Chapter 3, we randomly choose the ID from a large ID space. As with evaluating *k*-rAC, we use a space of 160 bits, i.e., *id* is **20 bytes**.
- *pk* is the public key as required for the access control with the PK approach in *k*-rAC. Its size depends upon the used asymmetric algorithm. Here, we rely on the evaluation of *k*-rAC (cf. Section 4.5.2.2). As we showed that the PK approach with ECC is preferable over RSA, we use **80 bytes** for *pk*.
- *sign* is a digital signature that is built by encrypting the hash value of a given data with the private key, i.e., $sign := \text{encrypt}_{sk}(h(data))$. The size of *sign* also depends upon the used asymmetric algorithm and, additionally, the specific hash function. As mentioned above, we use the PK approach with ECC for encryption. For hashing, we use SHA-256 (32 bytes) to be compliant with the evaluation of *k*-rAC. By encrypting the hash value with ECC, the size of *sign* is **63 bytes**.
- *key* is the symmetric encryption key which is encrypted with the public key of a group member. As for *k*-rAC evaluation, we assume for this key 128 bits (16 bytes). Encrypting it with ECC yields **64 bytes**.
- *flag* represents a signal to the communication partner about the success of the requested operation. We assume that it is implemented as an enumerated type of integers where each of them indicates a certain signal, e.g., 1 means success, 2 means an authentication error, 3 means an unreachable host error, etc. Hence, **1 byte** is a sufficient size for this parameter.
- *status* signals to the providers whether they are allowed to deliver the corresponding data object. We use for its implementation an enumerated type with values “ACTIVE”, “INACTIVE”, and “REVOKED”. As we optionally also use an expiration date within *status*, we reserve for this parameter **8 bytes**.
- *URL* is a reference to a web resource. In our system, it is used twofold with the push approach: (1) to communicate the provider’s interface where she receives the revocation requests, and (2) to prove the publishing of a data object by the provider. According to the specification of Hypertext Transfer Protocol (HTTP/1.1) [31], there is no limit placed on the URL length. However, the authors recommend supporting lengths of 8000 octets at a minimum. Practically, although the most widely used web browsers are not known to limit the URL length, some web browsers have a maximum URL length. For instance, the Microsoft Internet Explorer (IE) has a maximum length of $2,083$ characters. Since we

aim to reach users with any web browser, for our evaluation, we use the 2,083 characters as the average URL length, i.e., **2,083 bytes**.

- r is a nonce used for proving the revocation right by signing it with the corresponding private key. The nonce must be a random number sufficient large to avoid repetition. Otherwise, the use of the same number for different data objects by the owner might be exploited for replay attacks. We use 128 bits to prevent repetitions, i.e., the size of r is **16 bytes**.
- *contact* consists of the two parameters: the provider's URL and the nonce r associated with the corresponding data object. The provider encrypts *contact* with the owner's public key (pk) before storing it with the DRS. By encrypting the *URL* and r with pk using ECC, the size of *contact* is **2,189 bytes**.

Having the size of the individual parameters, we summarize the sizes of parameters sent with a particular message to determine the message size overhead. The results are presented in Tables 6.1 – 6.4. Hereby, we omit the size of the empty list object and the NULL value, as these values are negligible.

Comparison of the revocation methods regarding the message overhead

For both revocation approaches, the system-wide message overhead depends on the number of data objects protected with the DRS and the number of providers who published them. To compare the particular revocation methods, we must consider the distribution of message overhead to the system participants with respect to the load in particular protocol phases.

As we can see in Tables 6.1 – 6.4, after registering a data object with any revocation method, there is no burden for the owner until she decides to revoke it. The provider and the DRS have the most effort with the message exchange – they must cope with multiple protected data objects of numerous owners and, consequently, process many requests. The most expensive phase regarding the message number overhead is the distribution phase (abbr. 'dist'), as in this phase a data object can be accessed by any number of users via the Internet.

Under this aspect, the push approach is the most resource-efficient, as it requires no message transfer in the distribution phase. By using this approach, the message effort is shifted to the publication phase (abbr. 'pub'), i.e., there are 0 messages in the distribution phase but 6 messages in the publication phase for a single data object (3 for the peer and 3 for the provider). Hereby, we must consider that once the provider published the data object, it can be accessed by users numerous times without causing further message overhead for the provider, i.e., this overhead is constant. Extrapolating it to the arbitrary number of protected data objects and providers, the overall number message overhead Mn_{push}^{pub} depends on the number of published data objects n and the number of providers p, and it can be calculated as:

$$Mn_{push}^{pub} = 6 \cdot n \cdot p \tag{6.1}$$

Regarding the message size, we get per one data object and one provider the following overhead: $Ms_{push}^{peer} = 80 + 2083 + 1 = 2164$ bytes for a peer, and $Ms_{push}^{provider} = 20 + 2189 + 20 = 2229$ bytes for a provider. To get the overall message size overhead produced with the push approach in

the publication phase system widely, we apply the sum of Ms_{push}^{peer} and $Ms_{push}^{provider}$ to an arbitrary number of data objects and providers:

$$Ms_{push} = 4393 \cdot n \cdot p \tag{6.2}$$

In contrast, with the pull approach, there is no message overhead in the publication phase. In the distribution phase, there are in total only 2 messages to be exchanged per one protected data object. Specifically, with s-pull, the provider requests the status of a protected data object from the DRS. The frequency of these requests depends, on the one hand, upon how popular the data object is, i.e., how often users access it. On the other hand, it depends on the used caching time TTL, i.e., after the TTL is expired, the 2 messages must be exchanged again for a new user access to this data object. To analyse the effect of the TTL on the message number overhead for n data objects, we introduce the Equation 6.3. Using it for different caching times (t_{TTL}), we determine the number of requests to the DRS (r_{DRS}) depending on the requests (r_W) to the provider's web server for a certain number of protected data objects (n).

$$r_{DRS} = \min(\frac{n}{t_{TTL}}, r_W) \tag{6.3}$$

Since the DRS responds each request, we must multiply the result from Equation 6.3 by 2. Extrapolating the message overhead system widely, we must additionally consider all participating peers p to get the overall message number overhead:

$$Mn_{s-pull}^{dist} = 2 \cdot p \cdot \min(\frac{n}{t_{TTL}}, r_W) \tag{6.4}$$

In Figure 6.5, we exemplarily show the number of requests to the DRS for 100 protected data objects. Hereby, we assume a normal distribution of user accesses over time. Under this assumption, we can see that the number of requests to the DRS becomes constant and only depends from the TTL after a certain amount of requests to the web servers is reached. To underline the significance of the TTL, we depicted the influence of the caching time to the number of requests to the DRS in Figure 6.6. Accordingly, increasing the caching time drastically reduces the number of requests to the DRS as already claimed in Section 6.3.2.2. However, to decide the specific value for the TTL is beyond the scope of this work. From a technical perspective, the TTL should be a system parameter to avoid that owners set its value to 0 and, thereby, annihilating its purpose. Moreover, its value should be as high as possible but at the same time remaining acceptable to owners. Thus, the decision about the specific TTL value is rather political than technical.

Considering the message size values in the distribution phase with s-pull, we get per one status request the following message size overhead: $Ms_{s-pull}^{peer} = 28$ bytes for a peer, and $Ms_{s-pull}^{provider} = 20$ bytes for a provider. We apply these values to Equation 6.4 to determine the system-wide message size overhead produced with s-pull in the distribution phase:

$$Ms_{s-pull}^{dist} = 96 \cdot p \cdot \min(\frac{n}{t_{TTL}}, r_W) \tag{6.5}$$

Comparing to s-pull, the k-pull method produces in the distribution phase less message overhead, since requesting the key from the DRS for an encrypted data object affects only a closed group of users. Moreover, the provider delivers the data object directly to the requesting user,

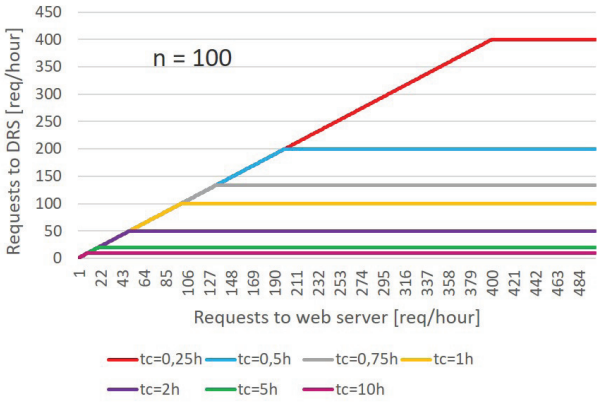

FIGURE 6.5: Requests from Web Server to DRS for 100 Data Objects with S-Pull

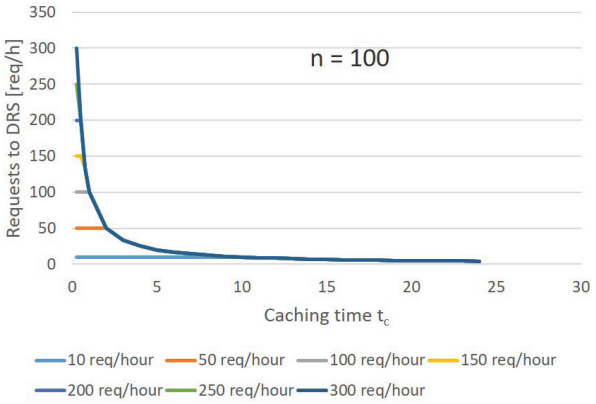

FIGURE 6.6: TTL Influence to Number of Requests to DRS with S-Pull

so the effort for the key request is by the user, i.e., not for the provider as with s-pull. However, k-pull cannot scale to be used Internet-widely. As already mentioned, it is better suited for controlling access to a published data object in a closed group.

Comparing Equation 6.1 and Equation 6.4, we identify that the factor for the message number overhead with the push approach is three times larger as with s-pull. For the message size overhead, it is even 45 larger. However, hereby, the effort for the provider and the peer per one data object is onetime. In contrast, with s-pull, there is a repeated effort as long as the provider offers the data object to users, and as long as users access it.

Part	Push		Pull
	publish	revoke	
Owner	–	$2p \cdot AO_{sk} + p \cdot HO$	–
Peer	–	–	–
Provider	AO_{pk}	$AO_{pk} + HO$	–
User	–	–	–

TABLE 6.5: Computational Overhead

6.5.2.2 Computational Overhead

With Tables 6.1 – 6.4, we show which messages are sent in each phase, and additionally give an overview of parameters whose values must be precomputed for these messages. Based on the given parameters, we determine the computational overhead per a single data object. Hereby, we do not consider the effort associated with the generation of random numbers, as it is negligible. Accordingly, we only consider the cryptographic operations which have not yet been addressed in the evaluation of *k*-rAC. Analysing the tables under this condition, we must only consider the computational effort produced with the push approach. With the pull approach, both methods cause no additional computational overhead.

In the following, we evaluate the computational overhead for the o-push and s-push methods. For both methods, the computational overhead is the same. Specifically, there is no additional effort for a user and a peer in any protocol phase. For an owner and a provider, there is computational overhead only in the publication or the revocation phase. Hence, to evaluate the computational overhead, we differentiate between the effort for the owner, i.e., O_{pull}^{phase}, and for the provider, i.e., P_{pull}^{phase}.

In the publication phase, the provider encrypts his contact information with the owner's public key (AO_{pk}). Hence, the computational overhead for the provider in this phase is $P_{pull}^{pub} = AO_{pk}$. For the owner, there is no computational overhead in this phase, i.e., $O_{pull}^{pub} = 0$.

In the revocation phase, after receiving the list with the p encrypted provider contacts from the DRS, the owner first decrypts the individual contacts by using her private key (AO_{sk}), i.e., $p \cdot AO_{sk}$. After that, she calculates the signature for each contact by using the corresponding nonce. Each signature requires a hashing (HO) and an asymmetric operation with the private key (AO_{sk}). Hence, the owner's total effort in the revocation phase is $O_{pull}^{rev} = 2p \cdot AO_{sk} + p \cdot HO$. By receiving a revocation notification, the provider verifies its signature to authenticate the request. For that, she decrypts the signature with the public key and hashes the corresponding nonce (cf. Section 6.2.2). Hence, the resulting complexity for the provider in this phase is $P_{pull}^{rev} = AO_{pk} + HO$.

In Table 6.5, we summarize the computational overhead for the individual system participant for a single data object. Accordingly, the pull approach outperforms the push approach regarding the computational overhead.

6.5.2.3 Storage Overhead

In the following, we compare the storage overhead S_{method}^{part} with the four revocation methods for each participant (i.e., the owner, the peer, the provider, and the user), by managing a single data

object. Hereby, we omit the storage space for the ID of a data object, as it is already taken into account by evaluating the access control with *k*-rAC. However, this was only done for the peer and the user, including the user with the owner right. As the provider is a new kind of participant in the system and, therefore, not covered in the evaluation with *k*-rAC, we must still add the ID when we calculate her storage effort. Based on the overview given in Tables 6.1 – 6.4, we can identify which parameters we must consider to determine the storage overhead for the individual revocation method. Accordingly, the user has no storage overhead with any revocation method, and the owner has it only with the s-pull method. By calculating the particular storage overhead for the remaining participants (i.e., the provider and the peer), we also refer to the sizes given in Tables 6.1 – 6.4.

For the push approach, we have the same storage overhead with both revocation methods. The provider stores the ID, the corresponding nonce r and the owner's public key per a single data object which is protected with the DRS. She needs these values to authenticate the requesting entity, when she receives a revocation request for the given ID. Accordingly, the storage overhead for the provider is $S_{push}^{provider} = 20 + 16 + 80 = 116$ bytes per a data object she has published. For a single data object, the peer stores the contacts of providers that published this data object. Therefore, the total storage overhead for the peer is $S_{push}^{peer} = p \cdot 2,189$ bytes where p is the number of providers that published this data object.

For the pull approach, we consider the storage overhead for both revocation methods separately. With s-pull, the peer only stores the status of a data object. Hence, its storage overhead is $S_{s-pull}^{peer} = 8$ bytes. As we use TTL to reduce the communication overhead in our system, the provider stores the ID of the protected data object and the time she last requested its status with the DRS. We use 8 byte for storing this time. Accordingly, the storage overhead for the provider with s-pull is $S_{s-pull}^{provider} = 20 + 8 = 28$ bytes. With k-pull, as the storage overhead is nearly covered with the evaluation of *k*-rAC, we only must consider the parameter r that is used by the owner and the peer in the authentication process (cf. Section 6.3.2.1). It is stored by both the owner and the peer. Hence, the storage overhead for the owner is $S_{k-pull}^{owner} = 16$ bytes, and for the peer also $S_{k-pull}^{peer} = 16$ bytes.

In Table 6.6, we give an overview for the overall storage overhead of the individual participant for n data objects. Hereby, except for the peer with the push approach, we solely must multiply the storage overhead given above by n, i.e., $S_{method}^{part} = n \cdot x$ where x is the amount of bytes presented above. For the peer with the push approach, we must consider that it stores a different number of provider contacts per one data object. Therefore, we introduce Equation 6.6 to determine the storage overhead of a peer for n data objects.

$$S_{push}^{peer} = \sum_{i=1}^{n} p_i \cdot 2,189 \tag{6.6}$$

As we can see, the most storage overhead is produced with s-push for a peer, as it depends on two parameters: the number of data objects the peer is responsible for, and the number of providers that published these data objects. With s-push, the provider also has a higher storage overhead. Applying the results from Table 6.6 system widely and considering k-pull not applicable Internet-widely, we can say that s-pull is the most resource friendly revocation method.

Part	Push	Pull	
		s-pull	k-pull
Owner	–	–	$n \cdot 16$
Peer	$\sum_{i=0}^{n} p_i \cdot 2,189$	$n \cdot 8$	$n \cdot 16$
Provider	$n \cdot 116$	$n \cdot 28$	–
User	–	–	–

TABLE 6.6: Storage Overhead (Byte)

6.5.3 Usability

We also achieve a high usability of the DRS: First, there is no burden at all for a common Internet user – she just browses the Internet as before, without the need to use additional hardware or install any additional software. The only exception is the k-pull method, as the user needs the symmetric encryption key k_d to get access to the protected data object. The key retrieving from the DRS can be solved with a browser plugin: when the authorized user downloads a protected object, it automatically requests the corresponding k_d from the DRS and decrypts the data object.

Second, the burden on the owner can be kept to minimum. The owner has to store a secret S for each published data object. Additionally, per a data object, she has to generate a unique ID and to embed it into the data object. However, the entire process can be automated by a software running on her computer. Even easier, this could be solved with a browser plugin: whenever the owner uploads some data object, the plugin transparently performs all necessary steps for her. One might argue that storing a secret for each protected data object still poses a burden for its owner. However, this can be mitigated by deriving the secret from the data object ID and a randomly generated master key K (e.g., a 256-bit key). In this case, the owner uses a hash-based message authentication code (e.g., HMAC [53]) with the master key and the data object ID to generate the required secret, i.e., $S = \mathrm{HMAC}_K(ID)$. By doing so, all secrets can be derived from the master key. Hence, only the master key needs to be stored. To prevent data loss, this master key can be stored – in an encrypted form – on some cloud storage, or even with the DRS itself.

Next, with exception of k-pull, the provider needs to register her publication of protected data objects (push) or to request the current status (s-pull). Both can be automated with a software extension on her web server. With the push approach, whenever an owner publishes a data object, the provider automatically registers this data object with the DRS. Additionally, she provides an API to receive the revocation notifications. Similarly with s-pull, whenever a user requests a data object, the software plugin verifies the current status of this data object. In [33], we implemented a prototype for an Apache web server module for the status requesting from a simulated DRS, and evaluated the processing and loading times upon a user request for a protected data object. The evaluation results show that the communication between the provider and the DRS can be performed fully automated and produces a low overhead of $\approx 5\%$ with respect to the processing time of a user request.

Finally, by using a P2P network, we can keep the configuration effort for setting up this service to a minimum. In most cases, this is just a matter of installing the software on a peer and executing it. The self-organizing nature of the P2P network makes it possible to start such a service with nearly zero configuration. Like with any other software, it must be possible to update the peer software if new versions or security updates are released. However, the update procedure can also be automated, e.g., similar as with the operating system Debian, that provides a package called UnattendedUpgrades to automatically update the system in the background [86].

6.6 Summary

In this chapter, we elaborated our push and pull approaches for data revocation on the Internet. With these approaches, we propose four different methods to prevent distribution of a data object after its revocation, i.e., the o-push, s-push, s-pull, and k-pull methods. With the push approach, the owner herself (o-push) or the system (s-push) notifies the providers, after the owner of a data object demands its revocation. With the s-pull method, the providers regularly request the status of the protected data objects before delivering them to the requesting users. According to their status, the providers deliver the data objects or not. By using the k-pull method, the owner publishes her data object encrypted and distributes the decryption key to the authorized users. In this case, the revocation is achieved by deleting the encryption key. Both push and pull approaches are privacy aware, since they do not require any personal information. Furthermore, the system does not require any additional software for the Internet users to use protected data objects. The k-pull method is the only exception, as the data objects are published encrypted and, therefore, must be decrypted by the user.

Finally, we evaluated the DRS regarding its security properties, performance, and usability. Especially, to be able to determine the suitability of a particular revocation method for a certain scenario, we analytically analysed the overhead produced with each revocation method. Hereby, we showed how the effort is distributed between the system participants in each protocol phases. In general, a provider offers a lot of data objects for her users on the Internet. Therefore, she has the most effort in managing protected data objects compared to an owner or a user by using the DRS.

With both push revocation methods, the most effort for a provider arises during the publication phase (i.e., communication overhead due to the registering the publication of a protected data object with the DRS), and during the revocation phase (i.e., computational overhead due to the verification of the revocation requests). Additionally, the provider must store parameters needed for the verification of the revocation requests (i.e., storage overhead). However, the effort with the push methods is onetime per a protected data object.

With the pull approach, only the status pull method is suitable for the Internet-wide revocation. Using it, the provider has no computational and no storage overhead. However, she has to request the status of a protected data objects as long as she offers it on the Internet. In comparison to s-pull, k-pull does not scale due to the required key distribution, but it suits better for closed groups.

We published the basic idea of the DRS in [49]. Furthermore, we extended the basic idea from [49] by elaborating the s-pull method and published it in [50]. We presented the DRS with the s-pull method at the International Workshop on Data Privacy Management DPM 2015.

7

Related Work

In this dissertation, we presented two technical solutions – a service for data revocation on the Internet (DRS) and a *k*-resilient fine-grained access control for a DHT (*k*-rAC). We elaborated this access control to protect the DRS against its misuse, i.e., its exploiting for censorship or for malicious revocations. Therefore, considering related work, we structure this chapter as follows: First, we compare *k*-rAC to related work. After that, we discuss existing approaches for removing previously published data from the Internet and compare them to the DRS.

7.1 Access Control Schemes for Distributed Systems

As described in Chapter 4, access control schemes are comprised of authentication and authorization. In general, we can divide the existing approaches for access control in P2P networks as follows.

First, there are *coarse-grained (CG)* approaches that focus on authentication and provide no authorization mechanisms. Authentication mechanisms for P2P networks have been already studied extensively, and there are reliable approaches available. The basic idea of these approaches is the usage of a public key infrastructure (PKI). Hereby, each peer uses a signed certificate from a certification authority (CA) to authenticate itself. Furthermore, it can be used to realize confidential communication between arbitrary peers. This coarse-grained access control is used, e.g., in [6]. However, this approach treats all peers equal and, thus, allows all authenticated peers to access any information in the system. In [84], the authors also propose an authentication-only approach but based on a distributed Merkle tree. They focus on achieving consistent updates in the DHT and resilience against replay attacks. The authors of [58] pursue an anonym authentication of peers and integrity of messages based on a Zero-Knowledge Proof [30]. With this approach, unforgeable and verifiable pseudonyms can be generated without a central authority. However, the authors focus on authentication and do not consider write and read accesses separately. Takeda et al. [83] propose a decentralized mutual authentication mechanism for each pair of nodes to avoid performance issues of a PKI approach. The authors use Web of Trust and a DHT to perform a distributed management of public keys, but they do not consider authorization. In [46], the authors build on the reputation-based trust management to realize the authentication of peers. Here, a DHT is used only for storing the trust levels of each user. The authorization part is solved by storing the data locally on the owner's computer.

The data owner requests the trust levels from the DHT to decide whether to allow the access to data on her computer for a certain user. In [16], the authors extend the PKI approach by trusted groups to regulate the access to resources based on group memberships. MacQuire et al. [59] propose an improved routing protocol for DHTs tailored for highly heterogeneous peers. The main idea of their approach is that peers with different capabilities have different roles in the DHT. For that, they extend the PKI-approach to include permissions for authenticated peers. In sum, the coarse-grained approaches do not consider the fine-grained protection of the get and put operations for individual DHT entries.

Second, the *tailored fine-grained (Tailored FG)* provide both authentication and authorization. However, the provided mechanisms are tailored for specific scenarios and are not applicable for DHTs. For instance, in [55], the authors propose OceanStore, an architecture for a global-scale persistence storage. Regarding the authorization, they mention that the data should be encrypted. Although they propose to use access control lists (ACL) for the reader and writer restrictions, they do not propose a specific solution how to protect this ACL from manipulation. In general, they provide only abstract details about the realisation of their access control, as their focus is on the global architecture. Another difference is that they offer read protection only out-of-band, i.e., the users need to distribute encryption keys. In contrast, we specified the details of our access control and offer an in-band key exchange for the read protection. The proposed fine-grained access control mechanism in [82] is built on a hierarchical model of peers. Here, the developers connect servers with dozens of users into a P2P network and do not consider home office computers. A single server manages the access control policies of its users and collaborates with other servers to delegate access control to other peers and users. Therefore, the proposed approach is closely tied to their scenario and requires a central server. In contrast, our approach is a general access control for a DHT and fully decentralized. Furthermore, the proposed approaches in [43, 85] are not applicable for regulating the fine-grained access control in a DHT, since they also use centralized components to manage access policies.

Next, the *fine-grained DHT (DHT FG)* approaches have similar goals as ours. However, they usually do not determine the owner or do not mention anything about the ownership of protected data. Furthermore, approaches of this group usually rely on trusted groups. On the one hand, this requires specialized peers, and, on the other hand, leads to a scalability issue. For instance, in P-Hera [22], the access control scheme allows data owners to specify fine-grained restrictions on who can access their data. For this, they use super nodes to manage access policies. In contrast, our approach does not rely on any centralized components. In [70], Palomar et al. propose a PKI-based access control scheme by extending certificates with authorization capabilities. However, their approach relies on trusted groups and does not allow for a fine-grained access control where individual permissions can be set for each entry in the DHT. In [67], they use ACLs for controlling the access to individual keys of a DHT. However, they protect these ACLs by using trusted groups, i.e., this requires a defined subset of peers to rely on. In contrast, with k-resilience, any subset of a given number of peers can be used. Further, in [76], the authors propose a protocol for delegating access control to intermediaries in such a way that requesters do not learn the access policy, and the intermediates do not learn the privileges. Although the authors' focus is privacy, they do not distinguish between put and get operation to secure the access to a single DHT entry.

Finally, the closest approach to k-rAC is *DECENT* [44], which proposes an architecture for enforcing access control in a decentralized online social network (OSN). Although based on a DHT, this architecture is specifically tailored to the OSNs. Similar to us, the authors aim to regulate the read and write access to a single DHT entry, and to delegate access rights to other users. To achieve that, they use public key cryptography and describe how to apply it for the

OSN scenario. We, in contrast, propose three different access control mechanisms: one is also based on the public key cryptography, the second is based on the zero-knowledge proof, and the third one is derived from password hashes. The three mechanisms are all generic for the usage in a DHT for arbitrary scenarios. Moreover, we compare them with each other and consider in which scenarios they suit better. To cope with malicious peers in DECENT, the authors rely on replication. However, they do not mention any further details or specify a protection method against malicious peers. With *k*-rAC, we propose a specific mechanism to enable resilience up to *k* malicious peers. Hence, with *k*, we introduce a parameter to achieve a certain security guarantee. Furthermore, we evaluate our access control scheme by detailing the effort for the three access control mechanisms. With our evaluation, researcher and engineers are able to decide which particular mechanisms is suitable for a specific scenario. Contrarily, the authors of DECENT evaluated only the public key approach for the OSN scenario.

In Figure 7.1, we compare *k*-rAC with the four types of related approaches with respect to the requirements we defined for access control in a DHT (cf. Section 4.2). In this context, we mark a fulfilled requirement with a green tick. In cases where we lack the information to fully determine whether the requirement is fulfilled, we use a yellow tick. For an unfulfilled requirement, we use a red cross.

Requirement	CG	Tailored FG	DHT FG	DECENT	*k*-rAC
Access	✓	✓	✓	✓	✓
Ownership	✓	✓	✓	✓	✓
Granularity	✗	✓	✓	✓	✓
Privacy	✓	✓	✓	✓	✓
Scalability	✓	✓	✓	✓	✓
k-Resilience	✗	✗	✗	✗	✓

FIGURE 7.1: Comparison of *k*-rAC with Related Work

In summary, the existing approaches are either limited to a certain scenario or do not offer a viable solution for fine-grained access control. In contrast, we propose a fine-grained access control for DHTs without the need of any centralized components, trusted groups, or super nodes.

7.2 Deleting Data on the Internet

In the past, there have been several technical approaches for controlling the availability of data objects after publishing them on the Internet. The ideal solution are self-destructing data objects which disappear or become useless after a specified time or on owner's demand. However, to delete a data object completely, all copies must first be found. This is a challenge even in a closed system. On the Internet, it is even more difficult or impossible to find all copies and, then, delete them on foreign computers. This is generally known as the hostile host problem [41], which states that it is theoretically impossible to force a foreign host into performing specific actions

if its administrative owner disagrees and interferes. Despite this theoretical impossibility, there are practical solutions which make it very hard for the administrative owner to interfere. The more secure ones rely on additional tamper resistant hardware like High Security Modules or smart cards. The existing technical approaches to control data objects after their publication on the Internet can be divided into the following classes: *protecting copies*, *hiding copies*, and *providing information about copies*.

Protecting Copies

The main characteristic of the approaches in this class is that the data objects are encrypted before they are published on the Internet. Additionally, an expiration date is assigned to the protected data objects. The expiration date specifies how long the associated encryption key may be delivered. Then, the encrypted data object is published, and the key is handed to a key management service. Anyone who wants access to the data object must get this key. When the expiration date passes, the key is removed from the key management service. This way, it is no longer possible to decrypt the data object, and, thus, the object has been effectively deleted.

The solutions for protecting copies aim to hinder automatically that protected data objects can be accessed after a certain time. The critical parts for these approaches are the key and the (decrypted) representation of the data object. Anyone who can intercept the key has no need to request the key again and, thus, can decrypt the data object at any time in the future. The protection can also be circumvented if it is possible to make a copy of the actual data object while it is decrypted. The best known example based on this principle are Digital Rights Management (DRM) systems [57]. Even though some of the DRM systems rely on tamper resistant hardware, they get circumvented eventually – it is usually just a matter of time.

While DRM systems usually aim for protecting content owned by industry, there are similar systems for arbitrary users. The individual solutions of this class differ from each other mainly in the key management and the deletion procedure. For instance, X-pire! [12] is a proprietary software solution for protecting pictures on the Internet. The user encrypts her picture, assigns an expiration date, and stores the corresponding key on a central server. With X-pire!, there are no restriction for the expiration date, i.e., the user is allowed to choose an arbitrary expiration date. To view the protected picture, other users need to install a web browser add-on that retrieves the key and decrypts the picture. With this approach, there are two main security risks. First, the key management on a central server requires users to trust that the company does not cease the service as well as does not misuse her access to the key [29]. Second, the decryption is done with a web browser add-on. As the browser environment is not protected from its own administrator, it took only a short time for the first successful attack against it [75]. To solve this problem in X-pire 2.0, the developers require a trusted execution environment (TEE) for requesting and storing the decryption keys as well as for displaying the decrypted content [13]. Hence, this approach is limited to TEE-enabled devices. Despite the hostile host problem, this approach does not scale for the data revocation on the Internet, since it is based on a central server infrastructure.

A similar concept are “self-destructing” data objects in Vanish [35]. In Vanish, the data object is also encrypted, but the key is stored in a DHT. This key gets lost after some time due to the used Vuze DHT [42]. In Vuze, peers remove from their local storage values whose store timestamp is more than 8 hours old. Hence, with Vanish, data objects automatically become inaccessible. Another advantage of this approach is that it has no single point of failure and single point of trust due to its decentralized and self-organized architecture. However, Vanish does not provide

flexible expiration times. To prolong the access for a certain data object, it must be re-published. Similar to X-pire!, Vanish requires a software component for retrieving the key to decrypt the protected data object. Furthermore, Wolchok et al. showed in [90] that the design of Vanish is insecure and can be circumvented even under the stricter assumptions made by the Vanish authors.

To overcome the attacks on Vanish, EphPub [18] uses the world-wide domain name system (DNS) [62] to store the key. With EphPub, the key is divided in single bits. Thereupon, each of these bits is stored along a different domain name. For instance, to store a key with a length of 128 bits, we need 128 different domain names. In turn, all these domain names must be requested to retrieve the key. For that, the user needs to install a software component. In general, EphPub uses an existing stable system without a central authority and is not proprietary. However, the expiration date depends on the TTL of the domain names. Thus, to provide a certain expiration date, e.g., seven days, we must search for domain names with such a TTL. To provide a data object longer, the user must re-publish it. Finally, EphPub probably does not scale, as the authors do not provide a large-scale evaluation on the DNS load. They acknowledge that a high number of users could cause a remarkable and unwanted traffic on the DNS cache resolvers.

A common aspect of the protecting approaches is that they are designed to prevent access to data objects after expiration time. Especially, they do not offer DRM-like mechanisms which prohibit anyone from copying or republishing content before the expiration time. Thus, there is an implicit assumption that also providers do not circumvent the system before a data object is expired. Additionally, to decrypt the protected data objects, they all require some software installed on the user's computer (e.g., browser extension). In our approach, we use the same assumptions to protect data objects after their expiration time. However, we do not use any data encryption, but rely on the cooperation of the providers. This reduces the complexity of our approach and does not require any changes on the computer of any Internet user (e.g., no browser extension is required).

Hiding Copies

A different approach for controlling the availability of data objects is practiced by Internet search engines. Nowadays, we find most of our information on the Internet with the help of search engines. Thus, if some data objects (here mostly websites) are removed from the result list of search engines, they can no longer be found by most Internet users. Google currently uses this approach to "delete" websites on request. With this approach, there is no additional burden for the Internet users, i.e., they request data the usual way and do not need any additional software components to view them.

However, when a user wants to delete a certain website, she must contact every search engine provider separately to request its "deletion" and, additionally, prove that she is authorized to "delete" it. Then, the search engine provider must manually verify each request and decide according to the local law if a "deletion" is acceptable or not. With other words, the search engine provider has the power to decide whether the request should be approved or not. Even if the search provider removes the site from its result list, the site itself does still exist and can be accessed with the correct URL. Even worse, it might still be found by a search engine not obeying this law. On the other hand, giving the search engine providers the power to filter the content, then, they finally create a mechanism which might be used for censorship.

Providing Information about Copies

Another approach to delete data object is to provide so-called information points, where providers can obtain information about data objects that must be deleted. For instance, the companies Facebook, Microsoft, Twitter, and YouTube propose to share a database that identifies "content that promotes terrorism" [9]. This approach is based on a central database where hashes of "bad" data objects are stored. Each company independently determines which data objects should be considered as promoting terrorism and deletes them from own services. By storing hashes of these data objects in the shared database, a provider notifies other providers about the potential terrorist content. Thereupon, other providers can use these hashes to identify such content on own services and remove it if it matches own removing policies. However, relying on a central database is a risky proposition, because the providers have the full control over the database and, therefore, any possibility for censoring. Furthermore, such methods cannot be transparent, as each provider has her own policy to decide whether a content promotes terrorism. Consequently, the users become dependent on the providers' policies. Moreover, to identify forbidden content, the provider must scan all data objects they offer with their services. This allows the providers to build a robust surveillance system for a (allegedly) good cause.

In [69], the authors propose a Personal Data Management Architecture (PDMA) that provides a user-centric consent management tool to regulate access to "personal data collection devices". They do not provide a specific realization for such an architecture. They rather propose an abstract idea of a privacy and identity assistant which is able to understand the context and point to data, or handle the interactions around it.

We can consider the DRS as such an information point to distribute user's revocation requests. In contrast to the approaches from above, we design the DRS by preserving user's privacy and avoiding a central authority. Additionally, we specify the system architecture and propose four different implementations, i.e., the o-push, s-push, s-pull, and k-pull revocation methods.

Summary

The existing attempts for deleting data objects on the Internet offer neither a real protection, nor a possibility to prevent the data propagation. In Figure 7.2, we compare the DRS to the related work discussed above with respect to our requirements that must be fulfilled by a system for revoking data objects on the Internet (cf. Section 2.2). Hereby, we use the notation *authentication* instead of *no censorship* to emphasize the mechanism which we use to prevent censorship.

We have the same overall goal as protecting approaches, e.g., X-pire!, EphPub, or Vanish. However, in contrast to them, we provide for the owner an instrument to keep control over her data object after publishing it. Moreover, we give the provider an instrument to obey the Article 17 of the GDPR. We achieve this with the requirement that the provider actively cooperates with our system. We assume this is sound for all providers under the jurisdiction of the EU. Other approaches only require the provider not to circumvent their protection during the lifetime of a data object. However, they achieve this by using encryption and, therefore, requiring some additional software components on any computer of the Internet users. We do not use any encryption of the data object for revocation methods with o-push, s-push, and s-pull. In turn, these methods do not require any additional software to be installed on computers of the Internet users. Furthermore, in contrast to Vanish and EphPub, we allow arbitrary expiration times until a data object expires, especially very long times. Additionally, with the key pull approach, we also

provide a possibility to share data objects within a closed group by encrypting them. This way, the data objects are also protected from unauthorized accesses by providers.

Using the hiding approach, Google fulfils none of the requirements. With our status pull approach, the owner is able to request the hiding of her data object by setting its status to "inactive". This way, we also provide the possibility to hide data objects and, though, achieve all our requirements.

Finally, with the DRS, we provide a shared database with a similar aim as Facebook, Microsoft, Twitter, and YouTube, namely to distribute information about data objects that should be deleted. However, their approach is not transparent regarding decisions which data object should be deleted and, hence, enables censorship. Nevertheless, this approach shows that a collaboration between providers for sharing information about revoked data objects can be achieved.

Requirement	X-pire!	EphPub	Vanish	Google	Facebook et al.	DRS
Availability	✓	✗	✗	✗	✗	✓
Authentication	✓	✓	✓	✗	✗	✓
Privacy	✗	✗	✓	✗	✗	✓
Scalability	✗	✗	✗	✗	✓	✓
Usability	✗	✓	✓	✗	✗	✓

FIGURE 7.2: Comparison of DRS with Related Work

8

Conclusion and Outlook

In this chapter, we summarize the important results of this dissertation and give an outlook of future work.

Nowadays, there is a great demand from privacy-aware users for technical solutions to remove previously published data objects from the Internet. This is complemented by the GDPR of the EU which regulates that any provider must delete data objects on owner's demand. Even more, the GDPR demands that providers must also inform all third parties who accessed those data objects. However, due to the openness of the Internet, it is in general very hard for the providers to track every access to all data objects. Hence, providers might simply be unaware of all parties who may have stored copies. This makes it impossible for them to comply with the regulation. We additionally argued that self-destructing data objects are a theoretical impossibility due to the hostile host problem. This means, there is no reliable way for a data owner to exercise any control over her data object once it is under the administrative control of one or more providers. This implies that data owners always require the cooperation of the providers. As the GDPR is legally binding, we do not have to encrypt a data object before publishing it on the Internet to be able to delete it afterwards on demand – providers must delete it on user's request from 2018. Hence, we can infer that at least the providers in the EU will cooperate. For that, providers need to know which data object must be deleted, when it must be deleted, and whether the request for its deleting is authorized. If the provider does not follow user's request, then there are legal sanctions against it. Thus, there is no need for a DRM-like solution. Instead, we need a notification service that informs all affected providers about a revocation request.

We, therefore, proposed the Data Revocation Service (DRS) as a new possibility for data owners to simultaneously inform all providers whether her data objects can still be used or must be deleted. With this service, there is no need for a provider to track every data access in order to potentially inform third parties about the deletion of the data object. For this, we introduced the concept of ownership. The entity who published the data object is its owner. The owner adds an ID to her data object and registers it with the DRS. The DRS is a point for exchanging information between users and providers. We propose two approaches for the exchange of information: With the *pull approach*, every provider can retrieve the status of any data object protected with the DRS on her own (i.e., the status pull method). Depending on its status, a provider delivers the data object to her users or not. This results in an intermediate step that can slow down user requests, so we use caching. Latest when the TTL expires, the revocation request reaches the affected providers. Based on this approach, we realize a second method to protect

data objects by encryption, i.e., the key pull approach for using in closed groups. With the *push approach*, we consider a different perspective: Whenever an owner publishes a data object via a service on the Internet, its provider stores her contact information with the DRS. This way, we collect providers that need to be notified in case of revocation. After a provider registered the publication of a protected data object, she delivers it to her users as usual. Hence, there is no delay for the Internet users. We provide two methods to implement this approach: o-push and s-push. With o-push, the owner retrieves the contacts of the affected providers from the DRS, and notifies these providers by herself. With s-push, the DRS notifies the affected providers about the revocation request on behalf of the owner. With both approaches, the system architecture is the same, so we can adapt it straightforward to each of the four revocation methods. Furthermore, we evaluated each revocation method as well as the three authentication mechanisms of our *k*-resilient fine-grained access control. With our evaluation, it is possible to choose any revocation method, to combine it with one of the three authentication mechanisms, and, then, to determine the entire overhead with the DRS.

With our approach, we protect data that an individual publishes self-willed (Class I, cf. Figure 1.2) and that are published with services within the EU jurisdiction. However, our approach is also applicable for data objects of Class II, i.e., data objects published about an individual by a third party and located on cooperative servers. For instance, it is determined by a legal process that an article about a certain individual is to be deleted. Applying this to the DRS, the publisher of that article is its owner. Consequently, the owner can revoke the article using our system.

Furthermore, our service can be used by search engine providers or forwarding providers (e.g., proxies). With the DRS, search engines can automatically filter the search results by omitting all "revoked" data objects (here mostly websites) from the final results list delivered to the user. This can only be used for data objects where the owner is the one who requested the revocation. It cannot help in cases where the revocation of a data object is requested due to different circumstances by others than its owner. However, at least in the first case, the time and effort for owners and search engine providers can be reduced to a minimum, since it can be automatized. In contrast, currently, every single data "deletion" must be processed and verified manually by the search engine provider, and the owner must proof her right, potentially in a legislative court.

In summary, deleting data on the Internet is a complex issue that cannot be solved by purely technical means. To solve it, we must combine technology and law. With the DRS, we provide a technical solution to prevent dissemination of data objects after their revocation based on the Article 17 of the GDPR. Thus, this service closes the gap between the legislation (i.e., inform third parties) and the technical possibilities (i.e., hard to track every access). Specifically, the DRS scales with the number of data objects, allows revocation at any time, and does not require Internet users to install any additional software component for viewing protected data objects. Moreover, the DRS cannot be used for censorship, as even powerful authorities cannot remove arbitrary data objects using it. One can claim that such authorities can directly request providers under their control to delete some data. However, they cannot use our system to automatically delete certain data objects. Furthermore, we propose that the providers build the P2P network for the DRS, and share the database of protected data objects. In ideal case, providers do not deliver revoked data objects. However, since it is likely that not every single service provider will be law-abiding, there will remain some providers who simply ignore the data owner's demand. Nevertheless, if search engine providers, proxies, and most of the providers who store the actual data object comply with the owner's wishes, the data object will eventually get extinct and increasingly harder to be found on the Internet. Hence, in some way the Internet "forgets" revoked data objects.

Even though the DRS provides an advancement over existing solutions, there are still some open research questions available that are worth exploring. For instance, we consider data objects of Class I in this dissertation. While we explain above how to use the DRS also for data objects of Class II, protection of data objects of Class III and IV is part of future work. To revoke data objects on uncooperative servers is very hard and requires additional tamper resistant hardware. An appropriate access control and a revocation method were not in the focus of this dissertation and, therefore, are a new research territory.

Furthermore, with our evaluation, we identify the overhead for each individual system participant. Another interesting aspect to be evaluated is the propagation of the revocation on the Internet in a real usage. For that, we could analyse the impact of providers that ignore users' revocation requests. The question which arises is what is the ratio between law-abiding providers and not cooperative providers to achieve "forgetting" of a revoked data object on the Internet.

Bibliography

[1] Gnuplot. http://www.gnuplot.info/, [Accessed: May, 12th, 2017].

[2] Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7.

[3] Osiris – Serverless Portal System. http://www.osiris-sps.org/, [Accessed: January, 11st, 2017].

[4] The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, [Accessed: January, 11st, 2017].

[5] The Tox Project. https://tox.chat/, [Accessed: January, 11st, 2017].

[6] Waste. http://waste.sourceforge.net.

[7] Information Technology – ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). X.690, August 2015. http://www.itu.int/rec/T-REC-X.690-201508-I/en, [Accessed: February, 29th, 2017].

[8] Internet Users in the World, December 2016. http://www.internetlivestats.com/internet-users/, [Accessed: December, 21st, 2016].

[9] Partnering to Help Curb Spread of Online Terrorist Content, December 2016. https://newsroom.fb.com/news/2016/12/partnering-to-help-curb-spread-of-online-terrorist-content/, [Accessed: May, 23rd, 2017].

[10] ARM. Security Technology Building a Secure System Using TrustZone Technology (white paper). *ARM Limited*, 2009.

[11] Camera & Imaging Products Association et al. Exchangeable Image File Format for Digital Still Cameras: Exif Version 2.3. Technical report, CIPA DC-008-2010 & JEITA CP-3451B Standard, 2010.

[12] Julian Backes, Michael Backes, Markus Dürmuth, Sebastian Gerling, and Stefan Lorenz. X-pire! – a Digital Expiration Date for Images in Social Networks. *arXiv preprint arXiv:1112.2649*, 2011.

[13] Michael Backes, Sebastian Gerling, Stefan Lorenz, and Stephan Lukas. X-pire 2.0: a User-Controlled Expiration Date and Copy Protection Mechanism. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1633–1640. ACM, 2014.

[14] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic Certificate Management Environment (ACME). *IETF Draft, May*, 2017. https://ietf-wg-acme.github.io/acme/, [Accessed: June, 1st, 2017].

[15] Ingmar Baumgart and Sebastian Mies. S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing. In *Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–8. IEEE, 2007.

[16] Karlo Berket, Abdelilah Essiari, and Artur Muratas. PKI-Based Security for Peer-to-Peer Information Sharing. In *Peer-to-Peer Computing, 2004. Proceedings. Fourth International Conference on*, pages 45–52. IEEE, 2004.

[17] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security Privacy*, 7(1):78–81, Jan 2009.

[18] Claude Castelluccia, Emiliano De Cristofaro, Aurelien Francillon, and Mohamed-Ali Kaafar. Ephpub: Toward Robust Ephemeral Publishing. In *Network Protocols (ICNP), 2011 19th IEEE International Conference on*, pages 165–175. IEEE, 2011.

[19] European Commission. Proposal for a Regulation of the European Parliament and of the Council on the Protection of Individuals with Regard to the Processing of Personal Data and on the Free Movement of Such Data (General Data Protection Regulation). January 2012. http://ec.europa.eu/justice/data-protection/document/review2012/com_2012_11_en.pdf, [Accessed: May, 20th, 2017].

[20] European Commission. General Data Protection Regulation. *Official Journal of the European Union*, May 2016. http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf, [Accessed: May, 13th, 2016].

[21] Baris Coskun and Bulent Sankur. Robust Video Hash Extraction. In *Signal Processing Conference, 2004 12th European*, pages 2295–2298. IEEE.

[22] Bruno Crispo, Swaminathan Sivasubramanian, Pietro Mazzoleni, and Elisa Bertino. P-Hera: Scalable Fine-Grained Access Control for P2P Infrastructures. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, volume 1, pages 585–591. IEEE, 2005.

[23] Tim Dierks and Eric Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol. *The Internet Engineering Task Force*, 2008.

[24] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. Technical report, DTIC Document, 2004.

[25] Jana Dittmann. *Digitale Wasserzeichen: Grundlagen, Verfahren, Anwendungsgebiete*. Springer-Verlag, 2013.

[26] Jana Dittmann, Martin Steinebach, and Ralf Steinmetz. Klassifizierung von Digitalen Wasserzeichen. In *Systemsicherheit*, pages 251–262. Springer, 2000.

[27] Peter Druschel, Michael Backes, and Rodica Tirtea. The Right to Be Forgotten – Between Expectations and Practice.

[28] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[29] Hannes Federrath, Karl-Peter Fuchs, Dominik Herrmann, Daniel Maier, Florian Scheuer, and Kai Wagner. Grenzen des "Digitalen Radiergummis". *Datenschutz und Datensicherheit-DuD*, 35(6):403–407, 2011.

[30] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-Knowledge Proofs of Identity. *Journal of cryptology*, 1(2):77–94, 1988.

[31] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. 2014.

[32] Sheila Frankel and Suresh Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. Technical report, 2011.

[33] Fabian Froelich. Bachelor Thesis: Design of a Data Revocation Module for the "Deleting" Data on the Internet. Master's thesis, Universität Kassel, 2015.

[34] William C. Garrison III, Adam Shull, Steven Myers, and Adam J. Lee. On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud. *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[35] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *USENIX Security Symposium*, pages 299–316, 2009.

[36] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 291–304. ACM, 1985.

[37] JD Gordy and LT Bruton. Performance Evaluation of Digital Audio Watermarking Algorithms. In *Circuits and Systems, 2000. Proceedings of the 43rd IEEE Midwest Symposium on*, volume 1, pages 456–459. IEEE, 2000.

[38] Jaap Haitsma, Ton Kalker, and Job Oostveen. Robust Audio Hashing for Content Identification. In *International Workshop on Content-Based Multimedia Indexing*, volume 4, pages 117–124. Citeseer, 2001.

[39] Henner Heck, Olga Kieselmann, and Arno Wacker. Evaluating Connection Resilience for Self-Organized Cyber-Physical Systems. In *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2016)*. IEEE Computer Society, September 2016.

[40] Henner Heck, Olga Kieselmann, and Arno Wacker. Evaluating Connection Resilience for the Overlay Network Kademlia. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*. IEEE Computer Society, June 2017.

[41] John H. Hine and Paul Dagger. Securing Distributed Computing against the Hostile Host. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 279–286. Australian Computer Society, Inc., 2004.

[42] Azureus Software Inc. Vuze wiki, 2012. `https://wiki.vuze.com/w/Distributed_hash_table#How_it_works`, (accessed Juny 5th, 2017).

[43] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. Privacy-Preserving P2P Data Sharing with Oneswarm. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 111–122. ACM, 2010.

[44] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia. DECENT: A Decentralized Architecture for Enforcing Privacy in Online Social Networks. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pages 326–332, March 2012.

[45] Silke Jandt, Olga Kieselmann, and Arno Wacker. Recht auf Vergessen im Internet - Diskrepanz zwischen rechtlicher Zielsetzung und technischer Realisierbarkeit? *Datenschutz und Datensicherheit (DuD)*, (4/2013):235–241, April 2013.

[46] Mohamed Jawad, Patricia Serrano-Alvarado, and Patrick Valduriez. Protecting Data Privacy in Structured P2P Networks. In *Data Management in Grid and Peer-to-Peer Systems*, pages 85–98. Springer, 2009.

[47] Olga Kieselmann, Nils Kopal, and Arno Wacker. Interdisziplinäre Sicherheitsanalyse. In *10. Berliner Werkstatt Mensch-Maschine-Systeme - Grundlagen und Anwendungen der Mensch-Maschine-Interaktion*, Berlin, Oktober 2013.

[48] Olga Kieselmann, Nils Kopal, and Arno Wacker. Ranking Cryptographic Algorithms. In K. David, K. Geihs, J. M. Leimeister, A. Roßnagel, L. Schmidt, G. Stumme, and A. Wacker, editors, *Socio-technical Design of Ubiquitous Computing Systems*, pages 151–171, Berlin, 2014. Springer.

[49] Olga Kieselmann, Nils Kopal, and Arno Wacker. "Löschen" im Internet. Ein neuer Ansatz für die technische Unterstützung des Rechts auf Löschen. *Datenschutz und Datensicherheit (DuD)*, (1/2015):31–36, Januar 2015.

[50] Olga Kieselmann, Nils Kopal, and Arno Wacker. *Data Privacy Management, and Security Assurance: 10th International Workshop, DPM 2015, and 4th International Workshop, QASA 2015, Vienna, Austria, September 21-22, 2015. Revised Selected Papers*, chapter A Novel Approach to Data Revocation on the Internet, pages 134–149. Springer International Publishing, Cham, 2016.

[51] Olga Kieselmann, Arno Wacker, and Gregor Schiele. *k*-rAC – a Fine-Grained *k*-Resilient Access Control Scheme for Distributed Hash Tables. In *12th International Conference on Availability, Reliability and Security (ARES 2017)*. ACM ICPS.

[52] Aleksandra Kovacevic, Kalman Graffi, Sebastian Kaune, Christof Leng, and Ralf Steinmetz. Towards Benchmarking of Structured Peer-to-Peer Overlays for Network Virtual Environments. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 799–804, 2008.

[53] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-hashing for Message Authentication. 1997. `https://tools.ietf.org/html/rfc2104`,[Accessed: May, 20th, 2015].

[54] Neal Krawetz. Looks Like It. Published online at HackerFactor.com, 2011. `http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html`, [Accessed: April, 10, 2017].

[55] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An Architecture for Global-Scale Persistent Storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[56] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. *Journal of cryptology*, 14(4):255–293, 2001.

[57] Qiong Liu, Reihaneh Safavi-Naini, and Nicholas Paul Sheppard. Digital Rights Management for Content Distribution. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003-Volume 21*, pages 49–58. Australian Computer Society, Inc., 2003.

[58] Li Lu, Jinsong Han, Yunhao Liu, Lei Hu, Jinpeng Huai, Lionel M. Ni, and Jian Ma. Pseudo Trust: Zero-Knowledge Authentication in Anonymous P2Ps. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1325–1337, 2008.

[59] Andrew MacQuire, Andrew Brampton, Idris A. Rai, Nicholas J.P. Race, and Laurent Mathy. Authentication in Stealth Distributed Hash Tables. *Journal of Systems Architecture*, 54(6):607–618, 2008.

[60] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[61] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[62] Paul Mockapetris. RFC 1034: Domain Names - Concepts and Facilities. 1987.

[63] Alberto Montresor and Márk Jelasity. PeerSim: A Scalable P2P Simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[64] Robert Morris and Ken Thompson. Password Security: A Case History. *Communications of the ACM*, 22(11):594–597, 1979.

[65] Julia Murphy and Max Roser. "Internet". Published online at OurWorldInData.org., 2017. `https://ourworldindata.org/internet/#content-and-communication`, [Accessed: April, 6, 2017].

[66] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[67] Rammohan Narendula, Zoltán Miklós, and Karl Aberer. Towards Access Control Aware P2P Data Management Systems. In *Proceedings of the 2009 EDBT/ICDT Workshops*, pages 10–17. ACM, 2009.

[68] Court of Justice of the European Union. Judgment in Case C-131/12. *Press Release No 70/14*, May 2014.

[69] Kieron O'Hara, Nigel Shadbolt, and Wendy Hall. A Pragmatic Approach to the Right to be Forgotten. 2016.

[70] Esther Palomar, Juan M. Estevez-Tapiador, Julio C. Hernandez-Castro, and Arturo Ribagorda. Certificate-Based Access Control in Pure P2P Networks. In *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on*, pages 177–184. IEEE, 2006.

[71] Lee Rainie, Sara Kiesler, Ruogu Kang, and Mary Madden. Anonymity, Privacy, and Security Online. *Pew Research Center*, 2013.

[72] Ananth Ranganathan. The Levenberg-Marquardt Algorithm. *Tutoral on LM algorithm*, pages 1–5, 2004.

[73] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A Scalable Content-Addressable Network*, volume 31. ACM, 2001.

[74] Real Time Statistics Project. Internet Live Stats. `http://www.internetlivestats.com/`, July 2016.

[75] Marc Ruef. Labs: Erfolgreicher Angriff gegen X-pire!, January 2011. `http://www.scip.ch/?labs.20110131`,[Accessed: May, 20th, 2015].

[76] Marc Sánchez-Artigas. Distributed Access Anforcement in P2P Networks: When Privacy Comes into Play. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2010.

[77] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[78] Carlton Shepherd, Ghada Arfaoui, Iakovos Gurulian, Robert P Lee, Konstantinos Markantonakis, Raja Naeem Akram, Damien Sauveron, and Emmanuel Conchon. Secure and Trusted Execution: Past, Present and Future–A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems. In *15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2016.

[79] GlobalPlatform Specification. TEE System Architecture, version 1.0, December 2011.

[80] Mark Stamp. *Information Security: Principles and Practice*. John Wiley & Sons, 2011.

[81] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[82] Christoph Sturm, Klaus R Dittrich, and Patrick Ziegler. An Access Control Mechanism for P2P Collaborations. In *Proceedings of the 2008 International Workshop on Data Management in Peer-to-Peer Systems*, pages 51–58. ACM, 2008.

[83] Atushi Takeda, Kazuo Hashimoto, Gen Kitagata, Salahuddin Muhammad Salim Zabir, Tetsuo Kinoshita, and Norio Shiratori. A New Authentication Method with Distributed Hash Table for P2P Network. In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 483–488. IEEE, 2008.

[84] Roberto Tamassia and Nikos Triandopoulos. Efficient Content Authentication in Peer-to-Peer Networks. In *Applied Cryptography and Network Security*, pages 354–372. Springer, 2007.

[85] Yang Tang, Patrick PC Lee, John CS Lui, and Radia Perlman. FADE: Secure Overlay Cloud Storage with File Assured Deletion. In *Security and Privacy in Communication Networks*, pages 380–397. Springer, 2010.

[86] Debian Wiki Team. Debian Wiki. UnattendedUpgrades., 2017. `https://wiki.debian.org/UnattendedUpgrades`, (accessed Oktober 5th, 2017).

[87] Ramarathnam Venkatesan, S-M Koon, Mariusz H Jakubowski, and Pierre Moulin. Robust Image Hashing. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 3, pages 664–666. IEEE, 2000.

[88] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. "O'Reilly Media, Inc.", 2002.

[89] Arno Wacker, Gregor Schiele, Sebastian Schuster, and Torben Weis. Towards an Authentication Service for Peer-to-Peer Based Massively Multiuser Virtual Environments. *International Journal of Advanced Media and Communication*, 2(4):364–379, 2008.

[90] Scott Wolchok, Owen S. Hofmann, Nadia Heninger, Edward W. Felten, J. Alex Halderman, Christopher J. Rossbach, Brent Waters, and Emmet Witchel. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs. *NDSS*.

Abbreviations

AC	**A**ccess **C**ontrol
ACL	**A**ccess **C**ontrol **L**ist
AO	**A**symmetric Encryption **O**peration
AS	**A**uthentication **S**cheme
DHT	**D**istributed **H**ash **T**able
DRS	**D**ata **R**evocation **S**ervice
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
HO	**H**ashing **O**peration
ID	Unique **ID**entifier
KG	**K**ey **G**eneration Operation
***k*-rAC**	***k*-r**esilient **A**ccess **C**ontrol
MO	**M**odular **O**peration
OTH	**O**ne-**T**ime-**H**ash Authentication Mechanism
P2P	**P**eer **2**(to) **P**eer
PK	**P**ublic-**K**ey Authentication Mechanism
RRR	**R**eplica **R**eturn **R**atio
SO	**S**ymmetric Encryption **O**peration
ZKP	**Z**ero-**K**nowledge **P**roof Authentication Mechanism

After publishing data on the Internet, the data publisher loses control over it. However, there are several situations where it is desirable to revoke published information. To support this, the European Commission has elaborated the General Data Protection Regulation (GDPR). In particular, this regulation requires that controllers must delete data on user's demand. However, the data might already have been copied by third parties. Therefore, Article 17 of the GDPR includes the regulation that a controller must also inform all affected third parties about revocation requests. Hence, the controllers would need to track every access, which is hard to achieve. This technical infeasibility is a gap between the legislation and the current technical possibilities. To close it, we provide a distributed and decentralized Internet-wide data revocation service (DRS), which is based on the combination of the technical mechanisms and the obligation to follow the legal regulations. With the DRS, the user can notify automatically and simultaneously all affected controllers about her revocation request. Thus, we implicitly provide the notification of third parties about the user's request.

ISBN 978-3-7376-0420-8