A Framework For Security-Oriented Distributed Programming

submitted by

Manuel Reinert

on May, 9th, 2012

Supervisor
Dr. Matteo Maffei

Advisor
Kim Pecina

Reviewers
Dr. Matteo Maffei
Prof. Dr. Michael Backes
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, .......................................................... ..........................................................
(Datum / Date) (Unterschrift / Signature)
Acknowledgements

This thesis would not have been possible without the direct and indirect support of many people, some of whom I would sincerely like to thank here.

First of all, I am grateful to Matteo Maffei for giving me the possibility to write my thesis about such an interesting research topic and for his supervision of this work.

My advisor, Kim Pecina, deserves special recognition for his always highly competent and helpful remarks on my work. His patience and calmness in supporting me make him a very good advisor.

Also I thank Lara Schneider for reviewing my thesis and giving me helpful comments.

I am deeply grateful to my parents for their support throughout my studies and their unshakable belief in me. They always come up with a good advice and make me reflect my actions.

Finally, I thank Angela for always being there and standing by me, be it in joyful times or in depressing times.
Abstract

Distributed system development challenges designers and cryptographers more and more. When designing distributed systems, the current state-of-the-art approach is to manually specify the whole system as opposed to use automated techniques. This is a complex and error-prone job, in particular if the system has to support sophisticated security properties. An indicator for this fact is the plenty of attacks on existing cryptographic protocols to enforce advanced security properties in distributed systems.

Recent work by Backes, Maffei, and Pecina presents a framework to automatically synthesize distributed systems that are able to enforce authorization policies and preserve user and data privacy. The central idea is to provide the programmer with a high-level declarative language for specifying the system and the intended security properties, abstracting away from cryptographic details. A compiler takes as input such high-level specifications and automatically produces the corresponding cryptographic implementations. They verify that the resulting systems have the desired security properties.

However, authorization and privacy are often not enough. Usually principals want to make sure that other users perform certain actions only a limited number of times. For instance, in e-voting schemes, voters are allowed to vote only once. Such linearity constraints are often implicitly present in the system specification but not expressible in frameworks like the one mentioned above. However, achieving linearity constraints while preserving the privacy of users is difficult since linearity constraints require the identification of a principal and her linkability to the actions that should be limited. If a principal does not reveal her identity, this is hard to achieve.

This thesis presents a new cryptographic mechanism called service-specific pseudonyms that enables system developers to enforce linearity constraints while preserving the privacy of users. We exemplify their usage and show how to cryptographically implement them via a combination of digital signatures and zero-knowledge proofs. Additionally, we show that service-specific pseudonyms can be integrated smoothly into the existing framework and how to generate code automatically.

We developed a tool that implements the framework of Backes, Maffei, and Pecina as well as service-specific pseudonyms. This tool enables programmers to specify distributed systems in a high level manner. It comprises two APIs: one to generate, modify and verify proofs and signatures which we need to achieve the required security properties. The other API allows for transmitting data on a distributed network. The user can choose between non-anonymous and anonymous communication primitives.

Several case studies and a performance evaluation of our implementation conclude this thesis.
# Contents

1 Introduction ........................................... 5  
   1.1 Related Work ........................................... 10  
   1.2 Outline .................................................. 12  

2 Privacy-Aware Proof-Carrying Authorization ........... 13  
   2.1 PA-PCA Zero-Knowledge Proof Deduction System .......... 13  
   2.2 Validity of Formulas ...................................... 15  
   2.3 Privacy-Aware Evidential DKAL .......................... 17  
   2.4 Cryptographic Implementation ........................... 20  
       2.4.1 Bilinear Maps ....................................... 20  
       2.4.2 Commitment Schemes ................................ 20  
       2.4.3 Groth-Sahai Zero-Knowledge Proof Scheme ........... 21  
       2.4.4 Re-randomizable Zero-Knowledge Proofs .............. 21  
       2.4.5 Automorphic Signatures .............................. 23  
       2.4.6 Zero-Knowledge Proofs of Signatures ................. 24  

3 Service-Specific Pseudonyms ............................ 27  
   3.1 Overview ................................................. 27  
   3.2 Deduction Rules ......................................... 28  
   3.3 Cryptographic Implementation ........................... 29  
       3.3.1 Properties of Pseudonyms ........................... 30  
   3.4 Application of Service-specific Pseudonyms ............. 33  

4 Implementation ........................................... 35  
   4.1 Proof API ................................................. 35  
       4.1.1 Overview ............................................ 35  
       4.1.2 Details ............................................... 38  
       4.1.3 Optimization ......................................... 41  
   4.2 Sender API ............................................... 42  
       4.2.1 JXTA ................................................ 42  
       4.2.2 Tor .................................................. 45  
       4.2.3 Functionalities ....................................... 46  
   4.3 Implementation of the Feedback System ................... 50
Chapter 1

Introduction

Most of today’s online services like Facebook, Google, or e-commerce, are centralized. Centralized here means that the system operator or the service provider are central authorities that manage the system and store all relevant data. Users have to trust the system operators in handling sensitive data confidentially. But even when they seem to handle sensitive data confidentially, it might be the case that they share it in anonymized form with third parties. For instance, Facebook states explicitly that it shares anonymized user data with third parties in order to personalize the advertisements even better [FbP]. However, Narayanan and Shmatikov [NS09] have shown that it is possible to de-anonymize such anonymized networks with small error rates which causes actual threats to privacy.

In contrast, decentralization takes another approach. There is no central authority since users are equal participants in a distributed system. Furthermore, data is not stored on a server in a centralized manner but locally on each user’s device. Despite the benefits of decentralization in comparison to centralization (no trusted authority, locally stored data, no bottlenecks), designing and implementing distributed systems with sophisticated security properties is very difficult and error-prone, in particular without a trusted authority. An indicator for this is the number of attacks on existing cryptographic protocols [Dou02, CWC05, LKXR05, SNDW06]. These attacks exploit flaws in the protocol design, which allow attackers residing on the network to break the intended security properties by eavesdropping, polluting the network with malicious files, or changing the content of sent messages. Amongst others, a main reason for this problem is that there does not exist any standard for developing distributed systems with advanced security properties.

For the development of such a standard technique we focus on three security properties: authorization, privacy, and linearity. Authorization is necessary to ensure that only authorized users gain access to protected resources and that only authenticated users take part in the system; privacy is necessary to ensure that users can hide sensitive data and their identity itself; linearity is necessary to be able to limit the number of

---

1Here, anonymized means that all personal data as well as data related to personal information is deleted.
accesses to certain resources or services. Before we elaborate on these three requirements in more detail we discuss a general design choice: the aim to get interoperable distributed systems.

**Interoperability.** As today’s systems are most often clusters of services, we want the standard to produce only systems that are interoperable. This property makes systems easily extensible, that is, new services can demand resources that are already used by other services, and can hence be aggregated seamlessly. Systems that offer interoperability among components are also called open-ended. For instance, suppose a patient visits the doctor. Upon this visit the patient receives a certificate stating that she has been at the doctor and that she is ill. This certificate can serve for different purposes which we can see as services. Firstly, the patient can use the certificate to tell her employer that she is ill and wants to stay at home. Secondly, she can go to the pharmacy to order a medicine with the certificate. And thirdly, she can ask her health insurance for sick pay as long as she is ill. So this sole certificate is used by three different services and the system providing this feature is open-ended. It would, however, not have been open-ended if the doctor had to give three different certificates to the patient, knowing in advance the services in which such certificates are used.

In the remainder of this section we discuss existing techniques to enforce authorization and privacy in distributed systems and we show their limitations.

**Authorization.** In order to manage authorization in distributed systems, the idea is to come up with *authorization policies* that protect the access to sensitive data and let a *reference monitor* check if a principal requesting access to certain data (also called *requester*) has sufficient permissions according to the authorization policy or not. Depending on the complexity of the policy and the number of requests, a bottleneck can arise that may cause, for example, a denial of service, since the reference monitor cannot process all incoming requests in a reasonable amount of time. However, it is much easier for the requester to prove upfront that she has sufficient permissions according to the known authorization policy. The most successful framework to enforce authorization in distributed systems based on this idea is the proof-carrying authorization (PCA) framework introduced by Appel and Felten [AF99]. The idea is to formalize the authorization policy logically and to let a requester construct a formal proof which shows that she has sufficient permissions to access the resource protected by the policy. The task of the reference monitor is just to verify if the incoming proof is correct and access is grant depending on these verifications. In order to build authorization policies Appel and Felten use a says-modality, which is illustrated below.

**Example 1.1.** Assume that a university professor Prof wants to get feedback from her students for her course after the lecture period. The professor gives every student Stud a certificate of the form

\[ \text{Prof says Reg(Stud, } \ell \text{ec)} \]  

stating that the professor Prof says that the student Stud is registered for the course \( \ell \text{ec} \).
There is great consent among cryptographers that the says-modality can be implemented via digital signatures. For instance, the above certificate is implemented as a digital signature of the form

\[
\text{Sign}(\text{Reg}(\text{Stud}, \ellec))_{sk_{Prof}}
\]

issued by \(\text{Prof}\) on the (bit-string encoding of the) predicate \(\text{Reg}(\text{Stud}, \ellec)\) using the secret signing key \(sk_{Prof}\).

Using this says-modality, system developers can easily specify authorization policies in a logical form. For instance, the professor could state that only registered students may provide feedback for her course.

\textbf{Example 1.2.} The authorization policy stating this requirement is as follows:

\[
\forall x, y, z. \text{Prof} \text{ says } \text{Reg}(x, y) \land x \text{ says } \text{Feedback}(y, z) \rightarrow \text{Rate}(y, z).
\]  

(1.3)

In more detail, the policy says that for all students \(x\), all courses \(y\), and all feedbacks \(z\), if the professor admitted student \(x\) to the course \(y\) and \(x\) gives the feedback \(z\) for course \(y\), then this feedback is counted on the rating board.

\textbf{Authorization and privacy.} Considering our example from above again, we can observe that the students reveal their identity while providing feedback for the lecture. In typical evaluation systems, this is not desirable since several privacy issues can occur. Their consequences are manifold, for instance, a bias in the grading of exams due to a critical feedback, or a bad impression in the professor’s mind. Students should submit their feedback anonymously in order to protect themselves from such consequences. Since \textsc{PCA} does not preserve any kind of privacy we cannot use it to fix the problem. The reason why it cannot preserve privacy is that digital signatures themselves always reveal the signed message and the signer. But we want to hide both in the case of the student.

To summarize, we want to build systems that achieve both authorization and privacy at the same time. These two properties are seemingly conflicting since we have to decide whether a person who does not reveal her identity is authorized to perform a certain action or not.

Maffei and Pecina [MP11] present an extension of \textsc{PCA}, the privacy-aware proof-carrying authorization (\textsc{PA-PCA}) framework, that gives the desired properties, that is, authorization and privacy at the same time. They establish privacy on the logical side via existential quantification of sensitive data.

\textbf{Example 1.3.} Every student participating the course has obtained a certificate of the form \(\text{Prof} \text{ says } \text{Reg}(\text{Stud}, \ellec)\). \textsc{PA-PCA} enables the student to create a statement of the form

\[
\exists x. \text{Prof} \text{ says } \text{Reg}(x, \ellec) \land x \text{ says } \text{Feedback}(\ellec, gr').
\]  

(1.4)

The existential quantification hides the identity of the student while convincing the professor that the student was registered. This statement says that there is some student \(x\)
that is registered for the course by the professor and that this student $x$ evaluates the course with grade $gr$.

Existential quantification preserves the anonymity of the student in this case. We observe that it is very convenient to express any kind of hiding sensitive information via existential quantification. However, the cryptographic implementation of existential quantification turns out to be very challenging. Digital signatures are not applicable since we cannot hide any information inside a signature and still expect the signature verification to succeed. The idea is to use a powerful combination of digital signatures and zero-knowledge proofs of knowledge\(^2\) of such signatures. Considering Example 1.3 again, the implementation is a zero-knowledge proof for a statement of the form

$$\exists s_1, s_2, x. \text{ver}_{\text{sig}}(s_1, \text{vk}_{\text{Prof}}, \text{Reg}(x, \ell \text{ec})) \land \text{ver}_{\text{sig}}(s_2, x, \text{Feedback}(\ell \text{ec}, gr)).$$

This statement expresses that there are two signatures $s_1$ and $s_2$ and a student $x$ such that $x$ knows both signatures, $s_1$ is a signature on the predicate $\text{Reg}(x, \ell \text{ec})$ issued by the professor, and $s_2$ is a signature on the predicate $\text{Feedback}(\ell \text{ec}, gr)$ issued by the student.

In addition to the framework specification, Backes et al. [BMP12] show how to transform their framework into a tool, which automatically synthesizes distributed systems that protect user privacy, the privacy of data, and allows for authorization constraints. They did, however, not build this tool so far.

**Authorization, privacy, and linearity.** Up to now, our evaluation system preserves the student’s privacy and the professor gets only feedback from students that actually attended the lecture. However, students are not restricted in the number of feedbacks they submit. They can essentially submit as many feedbacks as they like biasing the overall evaluation result. We want students to submit at most one feedback for each course they participated in. PA-PCA is not able to solve this problem. A student, once hidden via existential quantification in a feedback, can never be linked to a student in another feedback even if both students are the same person.

To the best of our knowledge, so far, there does not exist any general purpose solution to develop open-ended systems that enforce authorization policies, user privacy and the privacy of data, and linearity constraints at the same time.

Our solution builds on the concept of service-specific pseudonyms. They behave like pseudonyms, that is, their owner’s identity is protected, and additionally they provide two service-specific properties: uniqueness, that is, every user can have exactly one pseudonym per service, and unlinkability across different services, that is, it is not possible to link actions performed for different services. However, linking actions within a specific service is possible.

\(^2\)A zero-knowledge proof combines two seemingly contradictory properties. First, it is a proof of a statement that cannot be forged, i.e., it is impossible, or at least computationally infeasible, to produce a zero-knowledge proof of a wrong statement. Second, a zero-knowledge proof does not reveal any information besides the bare fact that the statement is valid [GMW91]. A non-interactive zero-knowledge proof is a zero-knowledge protocol consisting of one message sent by the prover to the verifier. A zero-knowledge proof of knowledge additionally ensures that the prover knows the witnesses to the given statement.
Example 1.4. Let $fb$ be the service for which students want to create service-specific pseudonyms. Additionally to the already existing statement, the student concatenates the following pseudonym predicate for the pseudonym $psd$:

$$SSP(psd, vk_{Stud}, fb).$$

Instead of submitting the former statement the student submits the conjunction of the former one and the above predicate, that is,

$$\exists x. Prof\ says\ Reg(x, lec) \land x\ says\ Feedback(lec, gr) \land SSP(psd, vk_x, fb).$$

The professor can now check if $psd$ already submitted an evaluation or not. If she already submitted one, then the feedback is rejected. Otherwise the feedback is counted on the rating board and the professor stores the pseudonym for further checks.

Our Contributions. We present a framework that aids system developers and programmers to conveniently synthesize distributed systems with complex security properties. In contrast to the today’s development practice where a system is developed without verification in mind, this framework takes a different approach: the user provides a declarative high-level specification of the desired system including the security and privacy properties as well as the intended data distribution on the network. The high-level specification language abstracts away all cryptographic details, that is, system designers do not need to have expert knowledge in advanced cryptography; they just have to state the desired properties. Concerning the data distribution, they do not specify how data is distributed but they only specify the start and endpoint in the data transmission. An automated compiler translates the specification into an executable distributed system which is guaranteed to have the specified security and privacy properties and that the sending and receiving of data is managed correctly.

We extend PA-PCA with the novel concept of service-specific pseudonyms. We show how to use them, how to logically characterize them, and how to implement them using the cryptographic primitives that also PA-PCA uses. Finally, we end up with a powerful framework that is able to provide all three desired security properties, that is, authorization, privacy, and linearity.

The second contribution of this thesis is the practical implementation of the extended PA-PCA framework as a library in Java. This implementation comprises a programming
language and two APIs: the programming language Asplada\(^3\) is a formal specification language: the programmer specifies the logical policies required to fulfill the desired security and privacy properties of the system. Asplada hides all cryptographic details. Asplada code can be incorporated into Java code. Listing 1.1 depicts the implementation of Example 1.3 in Asplada. A compiler translates this meta-code into valid Java code that interacts with the first API; an API that implements the complete PA-PCA framework extended with service-specific pseudonyms. Developers can also use the API without Asplada; they are able to construct and modify proofs and statements operating on logical representations that hide all cryptographic details. The second API provides the network functionalities. It allows users to send data on a distributed network, anonymously or not.

We can imagine the implementation on three different levels of abstraction. The lowest level is a cryptographic library that implements zero-knowledge proofs and digital signatures. The level in the middle is our proof API that abstracts completely from cryptographic details except for a few functions. The highest level is Asplada where users do not have to mind about cryptography at all.

In order to show the feasibility of our implementation and the usability of our APIs we carry out several case studies, show their implementation in our APIs, and present a performance evaluation.

### 1.1 Related Work

The pioneering works on authentication and access control in distributed systems by Abadi et al. [LABW91, ABLP93] paved the way for several languages and logics managing authorization like Aura [JVM\(+08\)] and its confidentiality extension AuraConf [Vau11], PCML\(_5\) [ADH10], PCAL [CG09], Binder [DeT02], SecPal [BFG07], and F* [SCF\(+11\)]. Except for AuraConf, these logics do not include any kind of privacy in their specification. AuraConf enforces privacy via encryption of sensitive data with the public key of the intended reader. This, however, makes it impossible to realize open-ended applications since we cannot reuse encrypted data for principal \(A\) in a protocol where the data should be read by \(B\).

Au et al. [AKS12] recently introduced BLACR, a system that allows service providers to blacklist misbehaving users so that they cannot access the system anymore while preserving their privacy. In BLACR, the user generates a private key, which gets authenticated by a group manager. In order to access a service, the user generates a ticket (comparable to our pseudonyms) using her private key. This ticket is revealed to the service provider. Service providers maintain reputation lists, that is, blacklists and whitelists containing pairs of tickets and scores. A score is a positive or negative number describing the severity of the misbehavior and it is assigned to a category that represents the accessed service. In reputation lists, negative scores refer to bad behaviors and positive scores characterize good behaviors. In order to access a service, the user has to prove her

\(^3\) Automated synthesis of privacy-, linearity- and authorization-aware distributed applications
1.1 Related Work

reputation to the service provider via a zero-knowledge proof based on $\Sigma$-protocols. This proof reveals a score-based property of the tickets that the user has generated with her private key and that belong to the service provider’s list. However, the proof hides the tickets, which is crucial to guarantee unlinkability of user actions. The main differences to our approach are the presence of a group manager and the application scenario: in our system, there is no central authority and every service provider can authenticate users on her own, as opposed to BLACR, where a central authority, the group manager, authenticates users; the purpose of BLACR is the anonymity and unlinkability of actions within and across different services, whereas the purpose of our system is to be linkable within a certain service but to be un linkable across different services in order to enforce linearity constraints. Therefore, our system reveals the pseudonym (corresponding to the ticket in BLACR) and the service provider checks if the plain pseudonym is allowed to access the service or not, as opposed to BLACR, in which a user has to construct a proof for a complete list of tickets, showing to have a certain reputation.

Lu et al. [LHL+08] show with Pseudo Trust how to generate pseudonyms to achieve anonymity in peer-to-peer systems. Pseudo Trust creates a pseudonym for an identity, based on a one-way hash function together with a well-formedness proof. Pseudonym systems like [WH09, CPHL07] are based on encryption schemes and involve the presence of a trusted third party. However, in all of these systems, it is possible to generate several pseudonyms at a time (this is also true for [ACK+10]).

Schartner and Schaffer [SS05] generate unique pseudonyms via RSA encryptions without relying on a trusted third party. However, they did not study how to integrate pseudonyms and authorization.

Martucci et al. [MRM11] use a trusted third party only to register the real identity. Upon that, users generate pseudonyms on their own. A pseudonym is unique for a given context and a user is linkable for actions performed within this context. Pseudonyms are generated by means of group signatures. The problem with group signatures is that the trusted authority that creates the initial identity is able to extract the signature creator.

All of the above pseudonym systems fulfill at least a few of the properties that we require for our framework. But most of them rely on a trusted third party which contradicts our approach to have a fully decentralized system. Building pseudonyms via encryptions destroys the aim to synthesize open-ended applications. Finally, most of these systems enable users to generate multiple pseudonyms which contradicts our constraint for linkability within a certain service.

Brickell et al. [BCC04] presented the concept of direct anonymous attestations that offers a pseudonymous-attestation functionality. Users can authenticate their trusted platform module (TPM) with a service provider via a pseudonym. This pseudonym is derived from the TPM’s secret value and a base value chosen by the service provider. A trusted third party, called issuer, signs the secret value. Our design does not rely on trusted third parties and is more flexible since it does not require any special protocol format. This makes it well-suited for open-ended applications since principals can receive certificates from other principals and combine them to realize sophisticated authorization proofs.
1.2 Outline

This thesis is organized as follows: Chapter 2 reviews the privacy-aware proof-carrying authorization framework [MP11, BMP12]. In Chapter 3 we present a new protocol called service-specific pseudonyms that solves linearity issues while preserving authorization and privacy properties. We show in Chapter 4 how we implemented our proof and sender APIs and exemplify their usage. We present further case studies and a performance evaluation in Chapter 5 and we conclude this thesis and give directions for further research in Chapter 6. Appendix A presents the zero-knowledge proof system by Groth and Sahai [GS08] and the digital signature scheme by Abe et al. [AFG+10]. Appendix B gives a detailed description of our high-level-language to interact with the proof API.
Chapter 2

Privacy-Aware Proof-Carrying Authorization

This chapter presents the privacy-aware proof-carrying authorization (PA-PCA) framework [MP11, BMP12]. We explain the framework in Section 2.1. Section 2.2 considers the validity of formulas constructed by PA-PCA, Section 2.3 explains the extension of the evidential distributed knowledge authorization logic (E-DKAL) to privacy-aware E-DKAL, and finally, Section 2.4 shows how we cryptographically implement the logical framework for the specification of privacy-aware authorization systems.

2.1 PA-PCA Zero-Knowledge Proof Deduction System

In this section, we formally introduce the privacy-aware proof-carrying authorization framework. Before detailing its components, we introduce the notation and conventions that we use here and throughout this thesis. We let $M$ range over cryptographic messages like signatures or zero-knowledge proofs, $m, n$ over names, that is, bit strings, $x, y, z$ over variables, $u$ over names and variables, $F$ over authorization formulas, and $E$ over quadratic equations in $\mathbb{Z}_p$. In Table 2.1, $ap$ ranges over all atomic predicates. Atomic predicates can either be signature verifications $\text{ver}_{\text{sig}}(u_s, u_k, F)$ or quadratic equations $E$. A signature verification $\text{ver}_{\text{sig}}(\text{sig}, \text{vk}, F)$ holds if the verification of sig with vk is true and the message signed in sig was $F$. Zero-knowledge statements, ranged over by $S$, are composed of atomic predicates and logical conjunction, disjunction, and existential quantification thereof.

The functions $[\cdot] : M \mapsto F$ and $[\cdot]_{zk} : S \mapsto F$ establish the mapping from cryptographic messages to logical formulas.

**Example 2.1.** Let $M = \text{Sign}(\text{Reg}([\text{Stud}, \ell \text{ec}])_{\text{sk}_{\text{prof}}})$ be the signature from (1.2). Applying the function $[\cdot]$ to $M$ yields its logical representation from (1.1):

$[M] = \text{Prof says RegStud, \ell \text{ec}}$.

PA-PCA comes with a zero-knowledge proof deduction system depicted in Table 2.2. This system describes which proofs a principal can create, based on proofs and signatures.
Table 2.1 Cryptographic evidence of authorization formulas.

\[
\begin{align*}
ap & := \text{versig}(u_s, u_A, F) \mid E \\
S & := ap \mid S_1 \land S_2 \mid S_1 \lor S_2 \mid \exists x. S
\end{align*}
\]

\[
[M] = \begin{cases} 
\text{vk}_A \text{ says } F & \text{if } \text{versig}(M, \text{vk}_A, F) \\
[S]_{\text{zk}} & \text{if } \text{verZK}(M, S)
\end{cases}
\]

\[
[S]_{\text{zk}} = \begin{cases} 
u_k \text{ says } F & \text{if } S = \text{versig}(u_k, F) \\
[S]_{\text{zk}} \land [S']_{\text{zk}} & \text{if } S = S_1 \land S_2 \\
[S]_{\text{zk}} \lor [S']_{\text{zk}} & \text{if } S = S_1 \lor S_2 \\
\exists x. [S']_{\text{zk}} & \text{if } S = \exists x. S'
\end{cases}
\]

Table 2.2 Rules from the PA-PCA zero-knowledge deduction system.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-S-VER</td>
<td>( M \in \Gamma ) \text{ versig}(M, \text{vk}, F)</td>
<td>( \Gamma \vdash_{\text{S}} \text{versig}(M, \text{vk}, F) )</td>
</tr>
<tr>
<td>I-S-&amp;</td>
<td>( \Gamma \vdash_{\text{S}} S_1 ) ( \Gamma \vdash_{\text{S}} S_2 )</td>
<td>( \Gamma \vdash_{\text{S}} S_1 \land S_2 )</td>
</tr>
<tr>
<td>I-S-\lor-1</td>
<td>( \Gamma \vdash_{\text{S}} S_1 )</td>
<td>( \Gamma \vdash_{\text{S}} S )</td>
</tr>
<tr>
<td>I-S-\lor-2</td>
<td>( \Gamma \vdash_{\text{S}} S_2 )</td>
<td>( \Gamma \vdash_{\text{S}} S )</td>
</tr>
<tr>
<td>I-ZK-S</td>
<td>( \Gamma \vdash_{\text{S}} S ) \text{ verZK}(M, S)</td>
<td>( \Gamma \vdash_{\text{ZK}} S )</td>
</tr>
<tr>
<td>I-ZK-\lor-1</td>
<td>( \Gamma \vdash_{\text{ZK}} \exists \bar{x}, S_1 \land S_2 )</td>
<td>( \Gamma \vdash_{\text{ZK}} \exists \bar{x}, S_1 \land S_2 )</td>
</tr>
<tr>
<td>E-ZK-\lor-2</td>
<td>( \Gamma \vdash_{\text{ZK}} \exists \bar{x}, S )</td>
<td>( \Gamma \vdash_{\text{ZK}} \exists \bar{x}, S_1 \land S_2 )</td>
</tr>
</tbody>
</table>

There are two judgments: the \( \Gamma \vdash_{\text{S}} M \) judgment is used to compose digital signature verification statements in conjunctive and disjunctive form, while the \( \Gamma \vdash_{\text{ZK}} S \) judgment is used to hide values, to combine the zero-knowledge statements in conjunctive form, and to split zero-knowledge statements in conjunctive form apart.

A principal can prove signature verification statements for signatures already contained in \( \Gamma \) using rule I-S-VER. She can combine signature verification statements with conjunction (I-S-\&) and disjunction (I-S-\lor-1, I-S-\lor-2). In order to operate no longer on the signature level she can transform the signatures into zero-knowledge proofs using rule I-ZK-S. She can also project the statements of the zero-knowledge proofs in \( \Gamma \) with rule I-ZK-\lor and can combine these statements using logical conjunction (I-ZK-\&) and existential quantification (I-ZK-\exists). Splitting zero-knowledge conjunctions is also possible via rules E-ZK-\lor-1 and E-ZK-\lor-2. Notice that we allow for disjunctions only in the judgment \( \Gamma \vdash_{\text{S}} M \) since we are not aware of any technique to build a zero-knowledge proof for a disjunction given only one of the proofs as a witness.
Example 2.2. In order to derive the formula to access the professor’s rating board in (1.4) we assume that \textit{Stud} has two signatures \( M_1 \) and \( M_2 \) (contained in \( \Gamma \)) such that \( \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(\text{Stud}, \ell ec)) \) and \( \text{ver}_{\text{sig}}(M_2, \text{vk}_{\text{Stud}}, \text{Feedback}(\ell ec, gr)) \) hold. At first, \textit{Stud} introduces these two signature verifications using rule I-S-Ver. The rule instantiations are depicted in Table 2.3a and 2.3b. Next, \textit{Stud} combines these two proofs via I-S-\& and obtains the statement \( \text{Prof says Reg(Stud, \ell ec)} \wedge \text{Stud says Feedback(\ell ec, gr)} \). This statement is further transformed into a zero-knowledge statement using rule I-ZK-S. As a last step, \textit{Stud} existentially quantifies her name and obtains the final statement as given in (1.4) using rule I-ZK-\exists. The complete deduction is depicted in Table 2.3c.

\begin{table}[h]
\centering
\caption{Deduction of the statement \( \exists x. \text{Prof says Reg}(x, \ell ec) \wedge x \text{ says Feedback}(\ell ec, gr) \) from (1.4).}
\begin{tabular}{ll}
(a) & \begin{align*}
\text{I-S-VER} & \quad M_1 \in \Gamma \quad \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(\text{Stud}, \ell ec)) \\
& \quad \Gamma \vdash \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(\text{Stud}, \ell ec))
\end{align*} \\
(b) & \begin{align*}
\text{I-S-VER} & \quad M_2 \in \Gamma \quad \text{ver}_{\text{sig}}(M_2, \text{vk}_{\text{Stud}}, \text{Feedback}(\text{Stud}, \ell ec)) \\
& \quad \Gamma \vdash \text{ver}_{\text{sig}}(M_2, \text{vk}_{\text{Stud}}, \text{Feedback}(\ell ec, gr))
\end{align*} \\
(c) & \begin{align*}
\text{I-ZK-S} & \quad \Gamma \vdash \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(\text{Stud}, \ell ec)) \wedge \text{ver}_{\text{sig}}(M_2, \text{vk}_{\text{Stud}}, \text{Feedback}(\ell ec, gr)) \\
\text{I-ZK-\exists} & \quad \Gamma \vdash \text{ZK} \exists x. \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(x, \ell ec)) \wedge \text{ver}_{\text{sig}}(M_2, \text{vk}_{x}, \text{Feedback}(\ell ec, gr))
\end{align*}
\end{tabular}
\end{table}

2.2 Validity of Formulas

We use the rules depicted in Table 2.4 to establish the binding between cryptographic objects and their logical interpretation. Using Ver-Sig, we can prove judgments of the form \( \text{vk}_A \text{ says } F \) from a signature \( M \) on \( F \) that successfully verifies using \( \text{vk}_A \). In particular, every signature generated by an honest principal can be verified correctly using this rule.

Before we explain the rule Ver-ZK, we describe one possible problem of the structure of zero-knowledge statements. The next example shows that not all zero-knowledge statements that we can deduce using the system are meaningful.

Example 2.3. Suppose that a principal receives a proof of the following statement:

\( \exists x_m, x_p. \text{ver}_{\text{sig}}(x_m, x_p, \text{Reg}(\text{Stud}, \ell ec)). \) \hspace{1cm} (2.1)

At first sight, we would believe that the predicate \( \exists x_p. x_p \text{ says Reg(Stud, \ell ec)} \) holds. However, neither the principal creating this signature is revealed, nor is there any proof
that \( x_p \) is a principal of the system. It is even possible that this proof was created by an attacker that picked a fresh key-pair.

To overcome this issue we introduce the notion of trustworthy keys. We assume that principals sign only honest verification keys, that is, keys belonging to principals of the system. We call a key \( \text{vk} \) trustworthy if either \( \text{vk} \) is revealed and known to belong to a principal of the system or is signed by a trustworthy key of the system. This notion in mind, we can directly conclude that \( x_p \) in (2.1) is not trustworthy whereas \( x \) in (1.4) is trustworthy since it is signed by \( \text{Prof} \) who is a well-known principal of the system.

Due to this fact we allow for introducing zero-knowledge statements into the system (cf. rule I-ZK-Ver) only if the statement \( S \) is well-formed. Informally, a statement is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]

We do not know which of the disjuncts is true. Therefore, this statement should not be well-formed. In order to fix this problem, we consider formulas in disjunctive normal form; we say that a formula is well-formed if every key is trustworthy. However, disjunction is a problem when defining well-formedness. Consider the proof

\[ \exists x_p, x_m. \text{ver}_{\text{sig}}(x_m, x_p, F) \lor \text{ver}_{\text{sig}}(x_m, \text{Stud}, F). \]
statements as explained above would succeed without any problem. Hence, we would believe that such statements are correct although they are not. To illustrate the usage of this rule we derive the validity of the zero-knowledge proof statement constructed in Table 2.3c as follows:

\[
\frac{\Gamma \vdash F}{\Gamma \vdash \exists x. \text{ver} \text{sig}(M, vk, \text{Reg}(x, \ell ec)) \land \text{ver} \text{sig}(M, vk, \text{Feedback}(\ell ec, gr))}
\]

### 2.3 Privacy-Aware Evidential DKAL

**E-DKAL.** The evidential distributed knowledge authorization logic (E-DKAL) is based on infor logic which addresses the notion of information rather than the validity of formulas. This is a fundamental difference to classical logics. E-DKAL handles information distribution among principals and decides, based on the received information, if access is granted or not. E-DKAL incorporates the so called knows-modality. It expresses which information a certain principal has at disposal. A principal can use information that she knows to deduce more information. Table 2.5 presents selected rules from E-DKAL. As before, \( \Gamma \) denotes a database storing all information that a principal has collected.

Rule **Ensue** says that if a principal knows an environment \( \Gamma \) and she can deduce \( F \) using information in \( \Gamma \) then she also knows \( F \). **P-A** says that if a principal asserts \( (A : F) \) some formula \( F \), then she knows this formula. **P-S** is used to establish the binding between a signature \( M \) and its corresponding logical representation \( vk_A \text{ says } F \): a principal can assert an atomic predicate by writing \( A : vk_A \text{ says } F \); this together with the signature verification, that binds a signature \( M \) to this atomic predicate, yields the knowledge of the verified signature \( M \). Knowledge can also be combined via **Msg-\( \cup \)**. To wrap up, whenever a principal wants to create a signature signed with her signing key or wants to introduce a formula, she can do so via assertions.

Rule **COMM-J** synchronizes the communication between two principals \( B \) and \( A \) by means of the two communication assertions

\[
\begin{align*}
\text{if } F_B & \text{ then } B \text{ sends } F \text{ to } p \\
\text{if } F_A & \text{ then } A \text{ receives } F' \text{ from } q
\end{align*}
\]
Table 2.6 Core Rules of Privacy-aware Evidential DKAL.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-ZK</td>
<td>$A$ knows $\Gamma \vdash_{ZK} S$ $\text{ver}_{ZK}(M, S)$ then $A$ knows $M$</td>
</tr>
<tr>
<td>Comm-A</td>
<td>if $F_B$ then $? \text{ sends } F$ to $p$</td>
</tr>
<tr>
<td></td>
<td>if $F_A$ then $A \text{ receives } F'$ from $?$</td>
</tr>
</tbody>
</table>

In order to use this rule, $B$ must be aware of an instance of $A$’s precondition $F_A$, that is, the substitution $\eta$ applied to $F_A (F_A \eta)$, and $A$ must know an instance of $B$’s condition $F_B$, that is, $F_B \theta$ for some $\theta$ and $\eta$. Additionally, $B$ must know to whom he sends his information, that is, it must hold that $p \eta = A$ and also $A$ must know whom she receives information from, that is, $q \theta = B$. Furthermore, $[M] = F$ must hold, that is, $B$ has to know the cryptographic evidence $M$ for the formula $F$ that he sends and $A$’s received formula $F'$ must correspond to this $F$ via $\theta$, that is, $F' \theta = F$. When this rule is fired, $A$ knows $M$, the cryptographic evidence for $F$. Whenever the condition on which the assertions are fired ($F_A$ and $F_B$) are true, we omit them and simply write $B$ sends $F$ to $p$ and $A$ receives $F'$ from $q$.

Example 2.7. Consider again our feedback example. The professor starts by providing the student with a registration certificate. So we use rule P-S. The professor keeps the assertion $\text{Prof} : \text{vk}_{\text{Prof}} \text{says Reg}(\text{Stud}, \ell ec)$ and a signature on $\text{Reg}(\text{Stud}, \ell ec)$ issued by $\text{Prof}$. Then $\text{Prof}$ knows $M_1$. To visualize this we have a look at the rule instantiation:

$$\text{P-S} \quad \text{Prof} : \text{vk}_{\text{Prof}} \text{says Reg}(\text{Stud}, \ell ec) \quad \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Prof}}, \text{Reg}(\text{Stud}, \ell ec)) \quad \text{Prof} \text{ knows } M_1.$$ (2.2)

In order to share this signature with the student we have two communication assertions

$$\text{Prof} \text{ sends } \text{Prof} \text{ says } \text{Reg}(\text{Stud}, \ell ec) \text{ to } \text{Stud}.$$ (2.3)

and

$$\text{Stud} \text{ receives } \text{Prof} \text{ says } \text{Reg}(\text{Stud}, \ell ec) \text{ from } \text{Prof}.$$ (2.4)

Together with the knowledge that $\text{Prof}$ knows $M_1$ from (2.2) we can deduce that $\text{Stud}$ knows $M_1$ using Comm-J:

$$\text{Comm-J} \quad (2.3) \quad (2.4) \quad (2.2) \quad \text{Stud} \text{ knows } M_1.$$ (2.5)

Note that both substitutions $\eta$ and $\theta$ are the empty substitutions. Now $\text{Stud}$ can use the new knowledge to construct her zero-knowledge proof for the feedback (cf. Table 2.3).
Privacy-aware E-DKAL. So far, E-DKAL is not aware of privacy. One of the reasons is the rule Comm-J because therein, the receiver always knows who the sender is. However, when the sender existentially quantifies her name in the sent statement then she does not want the receiver to know her identity. [BMP12] extend E-DKAL with two rules which solve this problem.

These two rules are depicted in Table 2.6. Intuitively, P-ZK establishes the binding between zero-knowledge proofs and their corresponding logical representations; it says that if A knows a store Γ, A can prove S from Γ in the PA-PCA deduction system, and the zero-knowledge verification $\text{ver}_{\text{ZK}}(M, S)$ holds for the zero-knowledge proof M, then A knows M. The anonymous communication rule Comm-A is analogous to the rule Comm-J with the difference that the receiver does not know the sender. The anonymity is established by replacing the sender’s identity in the assertions with a question mark.

Example 2.8. We are ready to give the full deduction of our feedback example. In order to do that, we need a few statements:

$S_1 = \exists x. \text{Prof says } \text{Reg}(x, \ell_{ec}) \land x \text{ says } \text{Feedback}(\ell_{ec}, gr)$

? sends $S_1$ to Prof  \hspace{1cm} (2.6)

Prof receives $S_1$ from ?  \hspace{1cm} (2.7)

Stud : Stud says Feedback($\ell_{ec}, gr$)  \hspace{1cm} (2.8)

$\text{ver}_{\text{sig}}(M_2, \text{vkStud}, \text{Feedback}(\ell_{ec}, gr))$.  \hspace{1cm} (2.9)

Table 2.7 depicts this deduction. Stud uses rule P-ZK to derive the knowledge of $M_{zk}$ – the cryptographic evidence for $S_1$ – from the previous reception of $M_1$ (cf. (2.5)), the knowledge of $M_2$ which she can establish on her own with P-S using the signature assertion (2.8) and the signature verification (2.9), and the zero-knowledge verification of $M_{zk}$ for the statement $S_1$ which was derived in Table 2.3. After deriving the knowledge of $M_{zk}$ the corresponding zero-knowledge statement $S_1$ is sent to the professor anonymously using rule Comm-A with the communication assertions (2.6) and (2.7).

It is worthwhile mentioning that the interaction between privacy-aware E-DKAL and the zero-knowledge deduction system is established as follows: every statement provable in the zero-knowledge deduction system can be introduced into privacy-aware E-DKAL using the P-ZK rule. Γ is filled with cryptographic messages a principal knows, deduced by Ensue.
2.4 Cryptographic Implementation

In this section we present the zero-knowledge proof system by Groth and Sahai [GS08] and the digital signature scheme by Abe et al. [AFG¹⁰].

More specifically, after we have presented the necessary notions like bilinear maps (cf. Section 2.4.1) and commitment schemes (cf. Section 2.4.2), we show how to construct zero-knowledge proofs for different kinds of equations in the Groth-Sahai proof scheme (cf. Section 2.4.3). Section 2.4.4 elaborates the fact that Groth-Sahai proofs are re-randomizable [BCC⁺⁰⁹] and explains the benefits for our framework.

In Section 2.4.5 we present the digital signature scheme by Abe et al. [AFG⁺¹⁰]. They define the notion of automorphic signatures. We use these kind of signatures in our implementation because they enable users to sign verification keys. This property is crucial in order to implement the idea of trustworthy keys.

Finally, in Section 2.4.6 we explain how the two schemes, the Groth-Sahai and the digital signature scheme, can be combined to prove the knowledge of signatures. We use the resulting proofs to implement the PA-PCA framework since we can deploy existential quantification and all other necessary operations on them.

2.4.1 Bilinear Maps

Definition 2.9 (Bilinear Map). Let $G_1, G_2$ and $G_T$ be cyclic groups of order $n$. Then $e : G_1 \times G_2 \to G_T$ is a bilinear map if the following holds for all $a, b \in \mathbb{Z}_n$, $X \in G_1$ and $Y \in G_2$:

- $e$ is bilinear, that is, $e(aX, bY) = e(aX, Y)^b = e(X, bY)^a = e(X, Y)^{ab}$.
- $e$ is non-degenerate, that is, if $G$ generates $G_1$ and $H$ generates $G_2$ then $e(G, H)$ generates $G_T$.

In the following sections we will use an algorithm SetupBM that generates on input $1^k$ (a security parameter) the tuple $(n, G_1, G_2, G_T, e, G, H)$ where $n$ is the order of the groups $G_1, G_2, G_T$, $e$ is the bilinear map, and $G$ and $H$ are the generators of $G_1$ and $G_2$, respectively.

2.4.2 Commitment Schemes

Intuitively, a commitment scheme can do the following: assume that two persons are sitting down opposite each other at a table and one of them writes a message, packs it into an envelope, and seals this envelope before she puts it on the table. The other person can not open the envelope until the sealer allows for opening it. More formally, in a cryptographic commitment scheme, a principal can commit to a value $x$ (corresponding to the message) with randomness $r$. This algorithm produces a commitment $c$ (corresponding to the envelope) which the principal can send to another principal. At a given time, the principal that created the commitment can decide to open it by sending the message $x$ together with the randomness $r$. We call $x$ also the witness and $(x, r)$ also the opening
information in the remainder of this thesis. The principal receiving the commitment can then recompute $c$ from the opening information to check if the commitment corresponds.

Intuitively, a commitment scheme can be hiding, that is, once sealed, the envelope cannot be opened until the sealer allows to, and it can be binding, that is, once sealed, the envelope’s content cannot be changed. More formally, hiding means that a verifier in possession of $c$ cannot gain information about $x$, and binding means that $c$ can only be opened with a unique $x$ and $r$. We refer to [TPL+08] for a formal definition.

Here and throughout the rest of this thesis we denote by $C_x$ the commitment to $x$ and by $\| C \|$ the value committed to in $C$.

### 2.4.3 Groth-Sahai Zero-Knowledge Proof Scheme

We give a brief introduction to the Groth-Sahai [GS08] scheme. Appendix A.1 provides a complete explanation of proof construction and verification together with a correctness proof for the construction.

**Zero-knowledge.** We do not define non-interactive zero-knowledge proofs of knowledge formally since it is only important to know the idea behind this notion. Informally, a non-interactive zero-knowledge proof of knowledge has to fulfill the following requirements: if the proven statement is true, the principal verifying the proof will be convinced of this fact. If the statement is false, a principal verifying the proof will not be convinced of the opposite except with a small probability. If the statement is true, the proof reveals no more than this bare information. As we deal with proofs of knowledge, we require in addition to the previous properties that the one creating the proof also knows the witnesses.

Let $gk = (p, G_1, G_2, G_T, e, G, H)$ be a setup constructed with the SetupBM algorithm where $p$ is a large prime number, $G_1, G_2$, and $G_T$ are cyclic groups of order $p$, $e : G_1 \times G_2 \rightarrow G_T$ is a bilinear map, and $G$ and $H$ are the generators of $G_1$ and $G_2$, respectively. Today, $|p| = 224$ is considered to be secure in practice [NIS11].

Given $gk$, we can construct zero-knowledge proofs\(^1\) for solutions for sets of equations like those depicted in Table 2.8, where $A_i, T_1 \in G_1, B_i, T_2 \in G_2, a_i, b_i, \gamma_{ij}, t \in \mathbb{Z}_n$ and $t_T \in G_T$. The variables $X_i \in G_1, Y_i \in G_2, x_i, y_i \in \mathbb{Z}_n$ are the unknowns or witnesses of the equations. The proofs are formed over commitments to these witnesses in order to achieve zero-knowledge.

In order to construct the proofs of knowledge which we require for our framework, the implementation uses binding commitments. Appendix A.1 explains this in more detail.

### 2.4.4 Re-randomizable Zero-Knowledge Proofs

This section shortly explains the intuition behind re-randomizable zero-knowledge proofs [BCC+09]. Appendix A.2 presents a detailed explanation of re-randomization techniques.

\(^1\)Groth-Sahai proofs are in general not zero-knowledge but witness-indistinguishable which is weaker than zero-knowledge. However, the equations which we prove have a special form that makes the resulting proofs zero-knowledge.
Table 2.8 Equations for which we can construct witness-indistinguishability proofs.

Pairing product equations (PPE):

\[
\prod_{i=1}^{n} e(A_i, Y_i) \cdot \prod_{i=1}^{m} e(B_i, X_i) \cdot \prod_{i=1}^{m} \prod_{j=1}^{n} e(X_i, Y_j)^{\gamma_{ij}} = t_T
\]

Multi-scalar multiplication equation in \( G_1 \) (MSM1):

\[
\sum_{i=1}^{n} y_i A_i + \sum_{i=1}^{m} b_i X_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} y_j X_i = T_1
\]

Multi-scalar multiplication equation in \( G_2 \) (MSM2):

\[
\sum_{i=1}^{n} a_i Y_i + \sum_{i=1}^{m} x_i B_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} x_j Y_i = T_2
\]

Quadratic equations in \( \mathbb{Z}_n \) (QE):

\[
\sum_{i=1}^{n} a_i y_i + \sum_{i=1}^{m} x_i b_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} x_i y_j = t
\]

Re-randomization has many applications concerning the PA-PCA framework. It allows for selectively changing the randomness inside commitments. However, commitment modification alone is not enough since the zero-knowledge proofs using the commitments do not verify anymore. Therefore, whenever we change the commitments we also have to change the proofs. We show in Appendix A.2.2 how this modification can be done; furthermore, we prove the correctness of this modification (cf. Appendix A.2.4).

Example 2.10. Suppose that there are proofs for two statements \( A \) says \textit{Good}(\( B \)) and \( A \) says \textit{Good}(\( C \)), which \( A \) wants to combine in conjunctive form. Every identity in the statement is represented by a commitment in the proof. Our convention says that commitments referring to the same identity have to be equal in order to link commitments inside a proof. However, it is not necessarily true that the commitment for \( A \) in the first statement is equal to the one in the second statement. We have to ensure after conjunction that both commitments are equal. The reason is that we have to link \( A \) not only in the statement but also in the proof. This can be established via re-randomization. Suppose that the first commitment for \( A \) uses randomness \( r_1 \) and the second commitment uses randomness \( r_2 \). By re-randomizing the second commitment with randomness \( r_1 - r_2 \) we achieve that after re-randomization both commitments are equal. A simple calculation proves this fact: \( r_2 + (r_1 - r_2) = r_1 \).
2.4.5 Automorphic Signatures

Abe et al. [AFG+10] propose a signature algorithm operating on bilinear groups. The message space are all Diffie-Hellman pairs \( \{(vG, vH) | v \in \mathbb{Z}_p\} \). As we want to sign verification keys they need to have the same form \((sG, sH)\) so that they fit the message space.

In the following, we instantiate a digital signature scheme which consists of four algorithms:

**SetupSS**\((1^k)\) which takes as input a security parameter \(1^k\) and outputs the scheme parameters \(p_{\text{sig}}\).

**KeyGen**\((p_{\text{sig}})\) which takes as input the signature scheme parameters and outputs a key pair \((sk, vk)\): \(sk\) is the signing key which only the signer is supposed to know and \(vk\) is the verification key which should be publicly known in order to verify the signed message.

**Sign**\((p_{\text{sig}}, m, sk)\) which takes as input the signature scheme parameters, the message \(m\) to sign, and the signing key \(sk\). It outputs a signature \(\text{sig}\).

**VerifySig**\((p_{\text{sig}}, \text{sig}, m, vk)\) which takes as input the signature scheme parameters, the signature \(\text{sig}\) to verify, the message \(m\), and the verification key \(vk\). It outputs 1 if \(\text{sig}\) is a valid signature on \(m\) verifiable with \(vk\), and 0 otherwise.

The instantiation of this scheme has the following form:

**SetupSS.** Let \(gk\) be the setup (generated by **SetupBM**). Additionally, **SetupSS** chooses public parameters \(pp = (F, K, T)\) where \(F, K, \text{ and } T\) are randomly drawn from \(G_1\). It outputs \(p_{\text{sig}} = (gk, pp)\).

**KeyGen.** On input \(p_{\text{sig}} = (gk, pp)\), **KeyGen** draws a random number \(x \in \mathbb{Z}_p\) and outputs the key-pair \((sk, vk)\) where \(sk = x\) and \(vk = (X, Y) = (xG, xH)\).

**Sign.** On input \(p_{\text{sig}} = (gk, pp)\), a message \(m\) which has the form \((M, N)\), and a secret signing key \(sk = x\), the signer computes a signature as follows: she chooses random numbers \(c, r \in \mathbb{Z}_p\) and outputs the tuple \(\text{sig} = (A, C, D, R, S)\) such that

\[
A := \frac{1}{x + c} \cdot (K + rT + M) \quad C := cF \quad D := cH \quad R := rG \quad S := rH.
\]

**VerifySig.** On input \(p_{\text{sig}} = (gk, pp)\), a signature \(\text{sig} = (A, C, D, R, S)\), a message \(m = (M, N)\), and a verification key \(vk = (X, Y)\), the verifier checks if the following pairing product equations hold:

\[
e(A, Y + D) = e(K + M, H) \cdot e(T, S)
\]

\[
e(C, H) = e(F, D) \quad e(R, H) = e(G, S).
\]

If all of them are true, the verifier outputs 1, otherwise it outputs 0.
Security aspects. Fuchsbauer [Fuc09] proves that this scheme is existentially unforgeable against chosen message attacks. This is the standard notion of security for digital signature schemes.

Observations. On the one hand, we can observe that the only part of the message that occurs in both, signature creation and verification, is $M$; so messages lie in $G_1$. On the other hand, the part of the verification key that is used to verify a signature $\text{sig}$ is $Y$, the $G_2$ part. This illustrates the simplicity of signing the verification key: we just take the $X$ part as message and the $Y$ part to verify the produced signature.

Consequently, this induces a disconnection of verification key parts since the two parts occur most often separately. An example for this separation is the evaluation step, that is, the statement (1.4) which has the following form before quantification:

$$\text{Prof says Reg}(\text{Stud}, \ell_{ec}) \land \text{Stud says Feedback}(\ell_{ec}, gr).$$

Its zero-knowledge statement, using two signatures $M_1$ and $M_2$ on the two atomic predicates, has the form

$$\text{ver}_{\text{sig}}(M_1, \text{vk}_\text{Prof}, \text{Reg}(\text{Stud}, \ell_{ec})) \land \text{ver}_{\text{sig}}(M_2, \text{vk}_\text{Stud}, \text{Feedback}(\ell_{ec}, gr)).$$

Here the student $\text{Stud}$ occurs with both parts as message in the first conjunct and as verification key in the second conjunct. However, the verifier has no way to establish the connection between the two parts. This is problematic since a verifier can never link two different values to the same key without any hint on the connection.

In order to overcome this problem we express the connection via an additional pairing product equation

$$e(X, H) = e(G, Y). \quad (2.11)$$

We can easily prove that this equation holds by exploiting the properties of the bilinear map:

$$e(X, H) = e(xG, H) = e(G, H)^x = e(G, xH) = e(G, Y).$$

Signatures on vectors of messages. We stated in Chapter 1 that an atomic predicate of the form $\text{Prof says Reg}(\text{Stud}, \ell_{ec})$ is usually implemented via a digital signature on the bit string encoding of the tuple $(\text{Reg}, \text{Stud}, \ell_{ec})$ signed with $\text{Prof}$’s signing key, that is, $\text{sign}(\text{Reg}, \text{Stud}, \ell_{ec})_{\text{sk}_\text{Prof}}$. This is so far not possible because we can only sign a single message and not a message vector. However, the above scheme is extendable to vectors of messages. We explain the construction and verification of such signatures in Appendix A.3.

2.4.6 Zero-Knowledge Proofs of Signatures

Signatures by themselves are not sufficient in order to provide the user with existential quantification on the logical level since they do not allow for hiding information. In order to realize existential quantification on the cryptographic level, we create for a
2.4 Cryptographic Implementation

given signature a zero-knowledge proof that proves the knowledge of that signature to a verifier. Therein the signature is hidden, but we can decide whether we want to reveal information about certain messages to the verifier.

In order to prove the knowledge of a signature we have to prove that the verification equations hold. More formally, we have to show that the three pairing product equations in (2.10) are valid. The Groth-Sahai proof scheme is well-suited to prove the knowledge of solutions to pairing product equations. Recall that pairing product equations are of the form

$$\prod_{i=1}^{n} e(A_i, Y_i) \cdot \prod_{i=1}^{m} e(X_i, B_i) = t_T.$$ 

We only need the last part of this equation, namely,

$$\prod_{i=1}^{m} \prod_{j=1}^{n} e(X_i, Y_j)^{\gamma_{ij}} = t_T. \quad (2.12)$$

We prove the following equations in the Groth-Sahai proof system:

$$e([C_A], [C_Y]) \cdot e([C_A], [C_D]) = e([C_X], [C_H]) \cdot e([C_M], [C_N]) \cdot e([C_T], [C_S])$$

$$e([C_C], [C_H]) = e([C_F], [C_D])$$

$$e([C_R], [C_H]) = e([C_B], [C_S]).$$

Rearranging the terms from the right to the left side, we get the corresponding pairing product equations

$$e([C_A], [C_Y]) \cdot e([C_A], [C_D]) \cdot e([C_X], [C_H])^{-1},$$

$$e([C_M], [C_N]) \cdot e([C_T], [C_S])^{-1} = O$$

$$e([C_C], [C_H]) \cdot e([C_F], [C_D])^{-1} = O$$

$$e([C_R], [C_H]) \cdot e([C_B], [C_S])^{-1} = O.$$

These equations have now the desired form where \( t_T = O \). This is necessary to achieve the zero-knowledge property of the computed proof (cf. Appendix A.1). Next, we exemplify how the second equation from above is instantiated correctly:

$$\hat{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} [C_C] \\ [C_F] \end{pmatrix} \quad \hat{Y} = \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} [C_H] \\ [C_D] \end{pmatrix} \quad \Gamma = \begin{pmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$
whose commitment is placed at position $i$ we remove the opening information in $O$ at position $i$.

We stated above that we have to connect verification key parts in case they occur with both parts, as message and as verification key. So we prove additionally (2.11) for each such connection:

$$e([C_X], [C_H]) \cdot e([C_G], [C_Y])^{-1} = O.$$ 

This completes the implementation of the proof and signature scheme and their combination which we need to give a full implementation of the PA-PCA framework.
Chapter 3

Service-Specific Pseudonyms

In this chapter we present a new cryptographic primitive called service-specific pseudonyms [MPR]. Extended with these pseudonyms, the pool of security properties that PA-PCA provides is enlarged by linearity constraints for sensitive actions. We proceed as follows: Section 3.1 gives an overview. In Section 3.2 we provide the deduction rules that serve as the extension of the PA-PCA framework. In Section 3.3 we show how to cryptographically implement the new primitive and carry out a security proof. Finally, Section 3.4 concludes this chapter by applying service-specific pseudonyms to our example.

3.1 Overview

PA-PCA yields a powerful tool to develop applications that grant access to sensitive resources only to authorized users while offering the possibility to preserve their privacy. However, it lacks the possibility to specify linearity constraints: they are present whenever an action is to be limited in its number of executions. PA-PCA does not have any means to do that since a principal can send a proof for the same action several times. The receiver of the proof will not notice that the arriving proofs originate from the same principals if this principal does not disclose her identity.

Mapping these thoughts to the lecture evaluation example, this can cause a biased evaluation since every student can hand in arbitrarily many feedbacks. This is certainly not the intended system behavior that the professor has in mind: every student should only be allowed to submit at most one feedback.

In order to solve this problem we introduce service-specific pseudonyms (SSP). Intuitively, given a service, a principal can create exactly one pseudonym for it and not more and no two principals can create the same pseudonym (uniqueness). In this way, principals are linkable to their pseudonyms within a given service without disclosing their identity. However, unlinkability is preserved across different services and hence anonymity. More formally, if \( psd \) is the service-specific pseudonym of \( A \) for a service \( s \), then we say that the predicate

\[
SSP(psd, \text{vk}_A, s)
\]  

(3.1)
Table 3.1 Cryptographic evidences for the extended system.

\[
\begin{align*}
ap & := \text{ver}_\text{sig}(u_s, u_A, F) \mid \text{ver}_\text{ssp}(u_{psd}, u_A, s) \mid E \\
S & := ap \mid S_1 \land S_2 \mid S_1 \lor S_2 \mid \exists x. S
\end{align*}
\]

(atomic predicates)

\[
[p\text{sd}] = \text{SSP}(psd, vk, s) \quad \text{if } \text{ver}_\text{ssp}(psd, vk, s)
\]

Table 3.2 Service-specific pseudonym extensions to privacy-aware E-DKAL and the PA-PCA zero-knowledge deduction system.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-SSP-Ver</td>
<td>( psd \in \Gamma ) \quad \text{ver}_\text{ssp}(psd, vk_A, s)</td>
<td>( \Gamma \vdash_{\text{ZK}} \text{ver}_\text{ssp}(psd, vk_A, s) )</td>
</tr>
<tr>
<td>Ver-SSP</td>
<td>( psd \in \Gamma ) \quad \text{ver}_\text{ssp}(psd, vk_A, s)</td>
<td>( \Gamma \vdash \text{SSP}(psd, vk_A, s) )</td>
</tr>
<tr>
<td>P-SSP</td>
<td>( A : \text{SSP}(psd, vk_A, s) ) \quad \text{ver}_\text{ssp}(psd, vk_A, s)</td>
<td>( A \text{ knows } psd )</td>
</tr>
</tbody>
</table>

holds. In order to enforce linearity constraints for a certain service, the service users create a pseudonym for it and send it to the service provider who can check if the pseudonym already accessed the service or not. This is easily achievable by just storing each SSP that accessed the service and check if the incoming SSP is already stored in this list. We can use the SSP predicate like an atomic predicate in the logic, that is, it fits nicely into the existing framework. We implement SSPs cryptographically via a powerful combination of digital signatures and zero-knowledge proofs (cf. Section 2.4.3 and Section 2.4.5).

3.2 Deduction Rules

We let \( psd \) range over pseudonyms and cryptographic evidences thereof, \( s \) over services, and we extend atomic predicates by a predicate \( \text{ver}_\text{ssp}(u_{psd}, u_A, s) \). This predicate is true if and only if \( psd \) is an SSP created by \( A \) for service \( s \). Additionally, the evaluation of \([ psd ]\) results in \( \text{SSP}(psd, vk_A, s) \) if and only if the pseudonym verification \( \text{ver}_\text{ssp}(psd, vk_A, s) \) holds. Table 3.1 shows the extension to the existing cryptographic evidence definitions. The former definitions of \([ \cdot ]\) and \([ \cdot ]_{\text{zk}}\) stay the same.

The deduction rules depicted in Table 3.2 fit smoothly into the existing system. There are three rules: P-SSP extends privacy-aware E-DKAL with a rule stating that if \( A \) assumes that the predicate \( \text{SSP}(psd, vk_A, s) \) holds and the pseudonym verification \( \text{ver}_\text{ssp}(psd, vk_A, s) \) evaluates to true, then \( A \) knows the SSP \( psd \); I-SSP-Ver extends the PA-PCA zero-knowledge deduction system with a rule stating that if the pseudonym verification \( \text{ver}_\text{ssp}(psd, vk_A, s) \) holds and \( psd \) is in the environment \( \Gamma \) then
\[ \Gamma \vdash_{\text{ZK}} \text{ver}_{\text{ssp}}(psd, vk_A, s). \] The statement is proven under the judgment \( \vdash_{\text{ZK}} \) since it is not possible to include SSPs into a disjunction and (as we will see in the next section) the implementation indeed also incorporates zero-knowledge proofs. \text{Ver-SSP} extends the verification rules. It is analogous to the corresponding rules \text{Ver-Sig} and \text{Ver-ZK}.

### 3.3 Cryptographic Implementation

The cryptographic implementation of service-specific pseudonyms comprises two phases: the interactive registration phase and the non-interactive generation phase. We make use of the Groth-Sahai proof scheme [GS08] and the digital signature scheme by Abe et al. [AFG+10]. This section is based on the corresponding section in [MPR].

#### Registration phase. If \( A \) wants to access a service \( s \) created by the service provider \( B \), then \( A \) first retrieves a signature on her verification key \( vk_A \) issued by \( B \) while revealing her identity. As \( A \) discloses her identity, \( B \) can verify whether or not to accept the registration; if \( B \) accepts, she issues the signature \( \text{sig} := \text{sig}(vk_A)_{sk_B} \) and sends it to \( A \), who uses \( \text{sig} \) in all upcoming interactions for authentication purposes.

#### Generation phase. In general, the service is represented by a publicly known string \( s \in \{0, 1\}^* \). This string is hashed directly into \( G_1 \) using a hash function \( h : \{0, 1\}^* \rightarrow G_1 \) (for instance, using the method of Icart [Ica09]), so that \( S = h(s) \). \( A \)'s verification key is of the form \( vk_A = xG \). This is all we need to construct the pseudonym: we let \( psd = xS \).

The authenticity of the pseudonym \( psd \) is established via the verification key \( vk_A \) and the corresponding signing key \( x \). However, these two values reveal \( A \)'s identity. Hence, rather than disclosing both values we let \( A \) compute a zero-knowledge proof of knowledge that shows (i) the knowledge of the signature obtained in the registration phase, (ii) the well-formedness of the service-specific pseudonym, that is, the pseudonym is computed correctly, (iii) links the pseudonym to the credential to authenticate the pseudonym, and (iv) hides all values that potentially compromise the anonymity guarantees of the pseudonym. The proof shows the following equations:

\[
\text{verify}_{\text{ssp}}(C_{\text{sig}}, C_{vk_A}, C_{vk_B}, C_x, C_G, C_S, C_{psd}) := \text{ver}_{\text{sig}}([C_{\text{sig}}], [C_{vk_A}], [C_{vk_B}]) \quad (3.2)
\wedge [C_x] \cdot [C_G] = [C_{vk_A}] \quad (3.3)
\wedge [C_x] \cdot [C_S] = [C_{psd}] \quad (3.4)
\]

The zero-knowledge proof hides the privacy-relevant values \( x \), \( vk_A \), and \( \text{sig} \), and opens all other commitments. However, it is also possible to open the commitment for \( vk_A \) and hide it not before it is really necessary. This makes the proof even more flexible. Equation (3.2) proves the knowledge of the signature \( \text{sig} \) issued by \( B \) on \( A \)'s verification key. Equation (3.3) proves that the proof creator is in possession of the secret signing key corresponding to the verification key authorized in (3.2). Equation (3.4) proves the well-formedness of the pseudonym \( psd \), that is, that it is correctly computed by a multiplication of the secret signing key \( x \) and the hash-value \( S \) of service \( s \).
3.3.1 Properties of Pseudonyms

In this section we prove two properties of service-specific pseudonyms: uniqueness and anonymity. Intuitively, a pseudonym offers uniqueness, if it is unique for a given user and a given service. It provides anonymity if, given the service, the pseudonym, and a list of candidate verification keys, it is impossible to identify the verification key involved in creating that pseudonym. As verification keys are strongly connected to their owner, also the principal corresponding to the verification key cannot be identified. In the following, we assume that the hash function \( h \) in our proofs is a random oracle\(^1\) that maps arbitrary string from \( \{0,1\}^* \) into \( \mathbb{G}_1 \) and we let \( n \) denote the security parameter.

We start by defining the notion of a negligible function and negligible probability, and continue by stating basic properties about the distribution of hash values and secret signing keys.

**Definition 3.1 (Negligible function).** A function \( \nu \) is called negligible if for every positive integer \( c \) there exists an \( n_c \) such that we have for all \( k > n_c \) that

\[
\nu(k) < \frac{1}{k^c}.
\]

We say that a random variable \( X \) occurs with negligible (resp. overwhelming) probability if there exists a negligible function \( \nu \) such that for all \( k \) we have \( \Pr[X] < \nu(k) \) (resp. \( \Pr[X] > 1 - \nu(k) \)).

**Proposition 3.2.** The following probabilities are negligible in \( n \):

1. The output of the hash function \( h : \{0,1\}^* \rightarrow \mathbb{G}_1 \) is \( \mathcal{O} \), the neutral element of the group operation of \( \mathbb{G}_1 \).
2. The output of the hash function \( h : \{0,1\}^* \rightarrow \mathbb{G}_1 \) for polynomially in \( n \) many different inputs coincide.
3. A signing key \( x \) is 0.
4. Any two signing keys from a set of polynomially in \( n \) many different inputs coincide.

The proposition holds since the output of the random oracle implementing \( h \) and the signing key are chosen uniformly at random from an exponentially (in \( n \)) large set.

Before defining uniqueness for SSPs, we introduce an auxiliary function, which intuitively links a verification key to the corresponding pseudonym for a given service. It is important to notice that this function cannot be efficiently computed without knowing the secret signing key.

**Definition 3.3.** We define the pseudonym computation function \( Y : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_1 \) as \( Y(xG, S) = xS \).

\(^1\)A random oracle replies to a query with a truly random response and it answers consistently with previously answered queries.
As a next step we can define the uniqueness of a pseudonym and then we prove that our construction above fulfills this definition. To prove that our construction fulfills the definition, we note that scalar multiplication of numbers (for instance, signing keys $x$) in $\mathbb{Z}_p$ with a generator $G$ of a cyclic, prime-order group $G_1$ is bijective. Furthermore, we note that because $G_1$ is a prime-order group, all elements in $G_1$ except the neutral element $O$ are generators of $G_1$.

**Definition 3.4 (Uniqueness of Pseudonyms).** A pseudonym psd is unique if and only if, the following conditions hold with overwhelming probability:

1. for every service $S$ and two honestly generated verification keys $vk_1$ and $vk_2$ we have that $Y(vk_1, S) \neq Y(vk_2, S)$,
2. for every service $S$ and verification key $vk$, $Y(vk, S)$ is a unique value,
3. for every two different service descriptions $s_1$ and $s_2$ and a verification key $vk$ we have that $Y(vk, h(s_1)) \neq Y(vk, h(s_2))$.

**Theorem 3.5.** Under the random oracle assumption, a pseudonym psd constructed as above is unique.

*Proof.* Condition 1.: for any service $S$ and any two verification keys $vk_1$ and $vk_2$ we have that $Y(vk_1, S) = Y(vk_2, S)$ if and only if $S = O$, the neutral element and only non-generator of $G_1$, or $vk_1 = vk_2$; the two verification keys coincide if and only if the two corresponding, honestly-chosen signing keys coincide. These two events happen only with negligible probability by 1. and 4. of Proposition 3.2, respectively.

Condition 2.: follows immediately since $Y$ is a function.

Condition 3.: for any verification key $vk$ and two different service descriptions $s_1$ and $s_2$, we have $Y(vk, h(s_1)) = Y(vk, h(s_2))$ if and only if $h$ maps $s_1$ and $s_2$ to the same value, or the signing key is 0. These two events occur only with negligible probability by 2. and 3. of Proposition 3.2. 

Now we define the anonymity of service-specific pseudonyms and proceed with a proof showing that our construction of pseudonyms fulfills this definition.

**Definition 3.6 (Anonymity of Pseudonyms).** A pseudonym psd provides anonymity if and only if, given a set $S$ of verification keys, any polynomially (in $n$) bounded attacker can determine which verification key $vk \in S$ was used to compute psd with probability $1/|M| + \mu$ where $\mu$ is negligible in $n$.

The anonymity proof is a reduction to the decisional Diffie-Hellman (DDH) problem. For the sake of completeness, we give all the necessary definitions and assumptions followed by the main theorem.

**Definition 3.7 (DDH and DDH advantage).** Given the tuple $(G, xG, yG, c)$, where $G$ is a generator of $G_1$, and $x, y \in \mathbb{Z}_p$ are randomly chosen, the DDH problem is to decide whether $c = xyG$. 

The advantage of an DDH attacker is defined as

$$\text{Adv}^{\text{DDH}}(B) = |\Pr[1 \leftarrow B(1^n, G, xG, yG, xyG) | b = 1] - \Pr[1 \leftarrow B(1^n, G, xG, yG, z) | b = 0]|$$

where $z$ is a random value in $G_1$ and $b$ is chosen randomly from $\{0, 1\}$.

Intuitively, the advantage describes how much better than pure guessing an adversary can determine which verification key $\text{vk}$ was involved in computing a pseudonym $\text{psd}$.

**Assumption 3.8 (Hardness of DDH).** For all attackers $B$ that are polynomially bounded (in $n$), the advantage $\text{Adv}^{\text{DDH}}(B)$ is negligible in $n$.

The values $(G, \text{vk}, S, \text{psd})$ form a valid Diffie-Hellman tuple since $\text{vk} = xG$, $S = rG$ for some random $r$, and $\text{psd} = xS = xrG$. We now state and prove our main theorem.

**Theorem 3.9.** Under the random oracle and the DDH assumption, service-specific pseudonyms provide anonymity.

**Proof.** Figure 3.1 visualizes the steps of this reduction proof.

Let $\{\text{vk}_1, \ldots, \text{vk}_k\}$ be the values obtained by the service provider during the registration of $k$ principals. Suppose there is an attacker $A$ that, on input $\{\{\text{vk}_1, \ldots, \text{vk}_k\}, G, S, \text{psd}\}$, outputs $\ell$ such that $\text{vk}_\ell$ and $\text{psd}$ are associated with probability $1/k + \eta$ where $\eta$ is non-negligible. Based on this attacker, we construct an attacker $B$ that breaks a decisional Diffie-Hellman challenge with non-negligible probability.

The DDH challenger $C$ uniformly at random draws a bit $b \in \{0, 1\}$. Here and throughout the rest of the thesis, we use the notation $x \in X$ to denote that $x$ is drawn uniformly at random from the set $X$. If $b = 1$, $C$ generates a valid DDH tuple, if $b = 0$, $C$ generates a fake DDH tuple, that is, a tuple where $c = zG$ for a random $z \in Z_p$. This tuple is sent to attacker $B$.

Given that DDH challenge $(G, xG, yG, c)$, attacker $B$ must give a perfect simulation to attacker $A$, so that $A$ cannot differentiate between a normal challenge and a challenge constructed by $B$. We note that the value $S$ in our service-specific pseudonym
3.4 Application of Service-specific Pseudonyms

In order to illustrate the applicability of the new rules we show how to improve the feedback system in such a way that the professor can limit the number of feedbacks per student. Recall that each course participant Stud has a certificate Prof says Reg(Stud, lee), created by the professor stating that this student is registered for the course. Let fb be
the service for which students can create an SSP in the evaluation process. Every student that wants to evaluate the lecture creates a pseudonym \( psd \) such that \( SSP(psd, vk_{Stud}, fb) \) holds. The conjunction of this pseudonym predicate together with the former feedback statement and an existential quantification of the student's identity yield the proof of the final statement

\[
\exists x. \text{Prof says } Reg(x, lec) \land SSP(psd, vk_x, fb) \land x \text{ says } Feedback(\mathit{lec}, gr_x). \tag{3.7}
\]

The deduction of this proof is depicted in Table 3.3. The professor receives the proof and checks if the pseudonym \( psd \), which is directly connected to the existentially quantified student \( Stud \), already submitted a feedback; depending on this result she accepts the feedback or not. If it is accepted, the professor adds \( psd \) to her list of submitted feedbacks. Upon termination of the evaluation process, the professor publishes the results.
Chapter 4

Implementation

This chapter presents the implementation of the PA-PCA framework in Java. It comprises two application programming interfaces (API). The first one, presented in Section 4.1, is able to generate, modify, and verify proofs as given by the PA-PCA framework. We discuss the second API in Section 4.2. It provides functionalities to start and maintain a distributed network. Once started, it enables the user to send and receive data to known computers in this network. The user can decide whether she wants to send data anonymously or not. Finally, Section 4.3 illustrates the usage of both APIs by implementing the feedback system from Chapter 2.

4.1 Proof API

This section is dedicated to present and explain the proof API, that is, the implementation of the PA-PCA framework in Java. Section 4.1.1 overviews the implementation as well as the provided functionality and Section 4.1.2 points out the implementation details of the most important functions.

4.1.1 Overview

We provide the PA-PCA zero-knowledge deduction system via a Java implementation. Our API enables the user to construct zero-knowledge proofs as given by the deduction trees created with the deduction system. We also provide the programming language Asplada (cf. Appendix B) that makes the interaction with the underlying API much more convenient because it abstracts away all cryptographic details.

Building blocks. We need several building blocks in order to implement a useful API. The most important one is a cryptographic library to construct and verify signatures and zero-knowledge proofs. We implemented the signature scheme presented in Section 2.4.5 and the zero-knowledge proofs of signatures presented in Section 2.4.6 based on the jPBC\(^1\)

\(^1\)jPBC is a library to perform pairing based cryptography operations in Java [DC] which originates from the C library PBC.
library. The resulting library provides functions to create and verify signatures on single messages, on message pairs, and on message vectors. Each such function takes as input the message(s) to be signed and the signing key, and returns the corresponding signature. The verification function just takes as input the verification key and the signature and returns true if the verification succeeds or false otherwise.

In order to implement the zero-knowledge proofs of knowledge of signatures we need another library that enables us to create and verify proofs for all the different kinds of equations provable with the Groth-Sahai scheme. For this purpose we use a library which is currently under development. It implements all building blocks for the Groth-Sahai proof scheme like commitments, proof constructions, and proof verifications for the different types of equations. Using this library, we create zero-knowledge proofs of signatures for a single message, for message pairs, and for message vectors. We can easily verify the resulting proofs since they come with a verification function and we can easily hide certain information (messages or identities) by just removing the captured opening information. This opening information is present from the beginning and is only removed if we want to hide identities or messages.

Given these building blocks, we can now almost start describing our API. However, we should mention that the cryptographic library as it is currently implemented is not aware of identities or string messages. It deals only with elliptic curve elements. In the design of the API and the implementation thereof we have to take care of this. We provide the connection between numbers and meaningful strings in a very easy to understand manner, that is, we connect each signature or zero-knowledge proof to an instance of our Statement class which represents the atomic predicates not in a cryptographic way but in the logical way given by the PA-PCA framework, so that they are understandable.

**Functionalities.** We implemented a method for each of the inference rules depicted in Table 2.2. The class providing all these methods is called ProofObject. It represents a finite conjunction of signatures or zero-knowledge proofs. We say that a ProofObject is on signature level or it is in signature representation if it is composed of signatures; we say that it is on zero-knowledge level or it is in zero-knowledge representation if it is composed of zero-knowledge proofs. Note that a mixture of signatures and zero-knowledge proofs does not exist. In order to instantiate a ProofObject there are two possibilities: we can either use the constructor corresponding to I-S-Ver or the constructor corresponding to I-ZK-Ver. Both of these constructors take a signature (respectively a zero-knowledge proof of a signature) and an atomic predicate as input, and initializes the ProofObject. ProofObject objects instantiated in this way can be modified corresponding to all the inference rules: transformation of signatures to zero-knowledge proofs, conjunction on the signature and on the zero-knowledge proof level, existential quantification, and also conjunction splitting. We also implemented the re-randomization of existing proofs although this is not included in the logic. Furthermore, we implemented existential instantiation of hidden principals or messages which is nothing else than recomputing the commitment followed by a check if the recomputed and the existing commitment correspond. If this is the case, the opening information is attached to the proof.
Figure 4.1 The interaction of the different building blocks. From top to bottom the level of abstraction decreases. The uppermost layer is Asplada interacting with the ProofObject API. ProofObject uses the cryptographic library below to implement the deduction rules. The cryptographic library uses the lowermost library under development that provides the mathematical zero-knowledge proof calculations.

In addition to construction and modification of zero-knowledge proofs, our API provides a verification method. This method checks if a given proof is correct, that is, it is constructed validly using the deduction system and not modified in a way that makes it invalid. ProofObject can also transform an instance into a byte array to distribute proofs on the network. The same holds for the other direction, that is, reading a ProofObject from a byte array.

Figure 4.1 visualizes the architecture and the dependency among the components. The programmer has two options: she can either use Asplada to have a very high-level, cryptographically free interface of specifying proofs; or she can work directly on ProofObject to create and verify proofs. This, however, requires a deeper understanding of the API design.
4.1.2 Details

Now we detail the different functions. We discuss key requirements and means of establishing the connection between identities.

**Construction.** We store the signature (respectively zero-knowledge proof\(^{\text{2}}\)) and the atomic predicate. Additionally, we build up our occurrence map for principals, that is, we store all occurrences of each principal in the zero-knowledge statement specified by the user. Consider our example from before, \(\text{Prof says Reg(Stud, \ell ec)}\). For this statement we produce three triples \((\text{Prof}, 1, 0)\), \((\text{Stud}, 1, 1)\), and \((\ell ec, 1, 2)\). The first component is the name or identity and the second component indicates in which proof in the conjunction\(^{\text{3}}\) the name is placed (in this case there is only one proof). The third component indicates where the name or identity is placed in the second component’s proof: if the index is 0, we refer to the identity before the says; if the index is different from 0, we refer to the specified position after the says. This map causes conveniences for later book-keeping since we can easily browse through all proofs where a certain principal occurs.

**Transformation of signature into zero-knowledge proof.** This transformation requires a few steps. At first, every signature must be converted into a zero-knowledge proof thereof. In these proofs, we have to be careful considering the commitments of equal identities. As we want to preserve their linkability in the proof, the commitments have to be the same among equal identities. After creating these proofs we have to build the connections described in (2.11) for the separated parts of a principal’s verification key. We sometimes call such a connection a connector proof. These were not required for the signature level since there, no commitments on verification keys are involved.

**Conjunction.** Here we must be very careful. There are several cases which we have to consider: we can only build the conjunction of proofs that are on the same level, that is, either both are in signature representation\(^{\text{4}}\) or both are in zero-knowledge proof representation.

**Signature representation.** We only merge the occurrence maps. The rest is just a concatenation of signatures and atomic predicates.

**ZKP representation (no quantification).** In addition to the above step we have to merge the connections, that is, we have to connect principals occurring both as verification key and as message. For example, if we connect a zero-knowledge proof for \(A\) says Good\((B)\) and one for \(B\) says Good\((A, C, B)\) then we have to add a connection between \(A\) in the first statement as verification key and in the second statement as message and a connection between \(B\)

---

\(^{\text{2}}\)This case never happens whenever we use Asplada.

\(^{\text{3}}\)We do not offer disjunctions. For an explanation see the paragraph “Not implemented so far” on page 41.

\(^{\text{4}}\)Representation means that the finite conjunction is composed either of signatures or of zero-knowledge proofs, but not both.
in the first statement as message, and in the second statement as verification key. Furthermore, we have to take care of the fact that commitments must be made equal in order to preserve linkability, that is, in the above case we would have to make equal the commitments of $B$ in the first and the second statement for the occurrence as an argument of the predicate. We realize this via re-randomization as already explained in Section 2.4.4.

**ZKP representation (with quantification).** In addition to the previous step we have to take care of the fact that after conjunction there does not exist any connection between a quantified message or principal and any message or principal (quantified or not) in the other conjunct. For instance, assume that we have the statements $\exists x. \text{say } Good(B)$ and $\exists y. \text{say } Good(C)$. Let the commitments for $x$ in the first statement and $y$ in the second statement be equal (this can have different reasons, for example, that the variables refer to the same identity). When we compose the two statements in conjunctive form there should not be a link between $x$ and $y$ because the two variables can correspond to any identity and do not necessarily refer to the same. We use re-randomization to achieve unlinkability.

**Conjunction elimination.** Here we just remove the specified conjunct from the conjunction. Additionally, we have to remove the connections to every identity occurring in the removed conjunct.

**Existential quantification.** There are two methods to existentially quantify messages and identities. We can either quantify all occurrences of a message or identity or only a selection, that is, a subset of all occurrences. The first case is easier since we only have to change the statement representing the proof, that is, remove the identity and replace it with a meaningless string, and to remove the opening information for the corresponding commitment in the proof itself. However, selectively quantifying means that connections must be removed because we do not want to link quantified variables to unquantified identities. Removing the connections is of no use if we do not also destroy the connection in the commitments. Note that even though we removed the opening information for the selected positions, the commitments are still equal. This does not destroy the connection. Hence, we have to re-randomize the selected positions with a new common randomness followed by building new connections.

**Re-randomization.** We have to take special care for this operation. It takes as input a selection of positions that should be re-randomized. For each such position we have to check if there are connections and delete them after re-randomization. We must also re-connect broken connections in case two positions with the same identity are re-randomized. In practice, the situation is complicated by the fact that we have to treat existentially quantified variables different from identities. If we choose to re-randomize an occurrence of an identity then we have to re-randomize all occurrences since otherwise linkability would break. However, variables can be arbitrarily re-randomized.
Whenever we re-randomize an identity or variable which has connections to other identities or variables, we have to re-randomize all of these connections. This seems not necessary at first sight. However, the connection verification uses the commitment that is stored in the proof; this commitment has changed because we re-randomized it. As the connection is a zero-knowledge proof we also have to adapt it to the new commitment. So if we do not also re-randomize the connector proof it will not verify anymore. For instance, suppose that we have a proof for the statement $A$ says Good($B$) $\land$ $B$ says Good($C$). Assume that $B$ is represented by a commitment $c_1$ in the first conjunct and by a commitment $c_2$ in the second conjunct.

The commitments are different since they are elements of different groups. But there is also a connector proof combining $c_1$ and $c_2$. The commitments are stored in the ProofObject and they are shared by the verification functions of the proof of the statement and the connector proof. So we have to adapt both proofs in order to succeed in the verification step.

**Existential instantiation.** This function requires to know the opening information of the message or identity we want to instantiate since it is inverse to existential quantification. Given the opening information, this method tries to open the commitments at every specified position. There are several things we should note at this position; we can retrieve the opening information of every not yet quantified identity or message. After existential quantification we can instantiate the identity or message using the former retrieved opening information as long as nobody else modified the commitment of the message or identity we want to instantiate. This is due to the fact that changing the commitment also changes the opening information. Like in the former cases we have to take care of the connections. If not all commitments are opened, connections break immediately. In order to fix these disconnections we apply the same idea as for existential quantification.

This functionality has different applications. For instance, suppose that we combine the lecture evaluation system with a game in which the winner has to show that she is the creator of the feedback in order to receive a prize. The professor has all the proofs for statements of the form $\exists x. \text{Prof says Reg}(x, \text{lec}) \land x \text{ says Feedback}(\text{lec}, \text{gr})$ at her disposal. She chooses the winner by picking one of these feedbacks and distributes it among all participating students. If the winning student wants to show that she is the originator of the feedback she can do so by instantiating her identity back into the proof and sending the modified proof to the professor. Although this example is not useful in practice, it shows the typical application scenario for this functionality. We refer to Appendix B.2 for a better, but also more complex example.

**Verification.** This method comprises several independent checks which all have to succeed in order to verify a proof. In the case that the ProofObject is on signature level, we simply have to check that each signature in the conjunction verifies correctly with the given verification key. If the ProofObject is on zero-knowledge level, we have to take care of several more dependencies:
1. Every zero-knowledge proof in the conjunction has to verify.
2. We recompute for each non-hidden identity or message its commitment and check if the recomputed one is equal to the stored one. This is of course only possible if the identity or message to check is not hidden since otherwise the opening information to recompute the commitment would be missing.
3. We recompute all connections of identities occurring both as verification key and as message and compare the recomputed map with the one captured in the `ProofObject`.
4. Given the connections which are based on the statement representation and not on the proofs themselves, we check if equal identities have equal commitments which is crucial to achieve linkability between equal identities within the proof. This implies that we also have to check that identities, messages, or variables that are not equal also have different commitments in order to not link parts of the proof which should not be linked.
5. We did not mention predicate names so far. Nevertheless, they are important since they give meaning to the atomic predicates. For instance, \textit{A says Good}(B) has a different meaning than \textit{A says Bad}(B). In order to check that predicate names have not changed, we take the name and compute its commitment. We check if the recomputed one is equal to the stored one.
6. We verify each connector proof. In order to do that we need the commitments from the zero-knowledge proofs. At this point we understand why it is crucial to also re-randomize the connector proofs when we re-randomize the other proofs: the connector proofs are directly connected to the zero-knowledge proofs since they share the commitments. Verification succeeds only if the proofs share the same commitment.

\textbf{Not implemented so far.} So far, we did not implement the logical disjunction because this is highly non-trivial. Proofs that include logical disjunction grow in complexity much faster than those without logical disjunction. We did also not yet incorporate the new protocol SSP into our proof API. We have a version of the SSP implementation that works well but it cannot be used in combination with the current \texttt{Asplada} version. However, when using by hand we can combine all pieces together which we did in order to get our evaluation results in Chapter 5.

\subsection*{4.1.3 Optimization}
We implemented the complete API in Java. However, we used a wrapper to go from jPBC to classical PBC which makes our resulting programs faster up to 70\% because PBC is more machine-oriented.

We also included simple multi-threading: whenever it is possible to compute commitments, proofs, or verifications in parallel, we started multiple threads. For instance, we start a new thread for each of the three pairing product equation proofs for the proof of knowledge of a standard automorphic signature. We apply the same idea for the two
levels above, that is, for each of the four standard signature proofs needed to compute a
proof of a pair signature (cf. Appendix A.3.1), we start a thread doing the job, and we
start a new thread for each of the \( n + 1 \) proofs of pair signatures needed for a proof of a
vector signature (cf. Appendix A.3.2).

There is a more advanced method called batch-verification \([BFI^{+}10]\) for verifying
proofs; the potential speed up for the verification is up to 90\%. We did, however, not
implement this method so far due to complexity.

4.2 Sender API

This section is dedicated to present the sender API. With this API users can build a
distributed network with which they are able to send and receive messages (byte arrays)
either by revealing their identity or anonymously. The API needs two building blocks
which we describe next. On the one hand we use JXTA \([JXS]\) to ensure reliable communi-
cation which we discuss in Section 4.2.1 and on the other hand we use Tor \([DMS04]\) – an
onion routing network – to ensure anonymous communication (cf. Section 4.2.2). In
Section 4.2.3 we get around to the functionalities of our API and explain how to use it.

4.2.1 JXTA

JXTA is a project originated by Sun Microsystems in 2001. It comprises several abstract
protocol specifications which are independent of any programming language. JXTA is
a virtual overlay network that strongly resembles a distributed hash table like Tapestry
\([ZHS^{+}04]\). It allows its nodes to interact with each other using the protocols it defines.
We use JXSE for our implementation, the Java implementation of JXTA. There are
several books about JXTA and JXSE \([Wil02, Gra02, OTG02]\) that are not up to date
anymore. We refer to \([Ver10]\), the most recent one, for further reading.

Notions and Protocols. Here we only talk about the protocols and the notions which
we need for this thesis. JXTA is built up in three main layers: on the bottom we have the
core layer that comprises two protocols, the Peer Resolver Protocol and the Endpoint
Routing Protocol. The middle layer is called service layer and comprises four protocols,
the Peer Discovery Protocol, the Peer Information Protocol, the Pipe Binding Protocol,
and the Rendezvous Protocol. Applications are developed using the top layer which is
called application layer. It can use the underlying layers as APIs. Before we detail all
these protocols we discuss several notions JXTA defines.

Peer. A peer in JXTA is nothing else but a node in the network, that is, any comput-
ing device (computer, PDA, smartphone, etc.) that implements the core JXTA
protocols. It has access to at least the core protocols and can interact with other
nodes in the network if there is a connection. There are several kinds of peers. We
refer to Figure 4.2 for an illustration of how a JXTA network can be setup and
which peers can connect to which peers. In order to operate the network we need
rendezvous peers and relay peers. Rendezvous peers can coalesce with several other
peers. However, they are not aware of talking to peers located behind a firewall or NAT. Relay peers solve these issues (in this thesis it does not matter how this is done). Basic peers can have at most one direct connection. This connection must be to a rendezvous or relay peer. We can imagine the rendezvous peers as routers in the network whereas basic peers are just clients that need the routers to talk to other clients. A mixture of rendezvous and relay peer is also available.

**Peer Group.** JXTA uses this notion to talk about networks or compositions of peers. There are at least two peer groups in JXTA, that is, the *world peer group* and the *net peer group* to one of which any peer is connected. The world peer group is conceptually located above the net peer group. Peers organize themselves in peer groups to form groups of interest. So we can think of, for example, a group for professors, a group for students, or different groups for different lectures. Peer groups should provide at least the core services to facilitate group communication. We explain the notion of service below.

**Advertisement.** In JXTA every object is described by an advertisement. Advertisements are XML files that comprise information about a particular object. For instance, a peer advertisement includes a unique peer ID, the peer group ID to
which it belongs to, a name, a description and an address to find it on the network. These advertisements are distributed in order to publish information.

Pipe. A pipe is a direct connection between two peers. There are different kinds of pipes: unidirectional pipes where only one peer can send and the other can only receive, bidirectional pipes where communication can go in both directions, and multidirectional pipes where communication can go from one to many other peers.

Service. JXTA defines core and standard services. These are used to implement the protocols. There are peer services which run on each peer and peer group services that are provided within the peer group. We describe some services which we need for our API.

Discovery Service. This standard service finds peers in the network. More formally, it does not find the peer itself but its representation, that is, the peer advertisement.

Endpoint Service. This core service provides the functionality of sending data from one peer to another using the specified network transportation protocol, in general either TCP or HTTP.

Peer Info Service. This standard service retrieves the status information of other peers in the network.

Pipe Service. This standard service establishes the connection from one peer to another via a pipe.

Rendezvous Service. Rendezvous peers operate this standard service in order to manage their job more efficiently, that is, forwarding and processing queries about other peers or services in their peer group.

Resolver Service. This core service manages the routing of queries between peers.

All necessary notions explained, we can now continue by describing the different protocols which JXTA defines.

Endpoint Routing Protocol. This protocol tries to establish a route from a peer $A$ to a peer $B$. In order to do that it maintains a cache of known routes. Given a routing request, it first takes this cache into consideration to find a route. If it does not find a route in the cache it queries neighboring peers for routes to $B$. After receiving the responses it takes the optimal route among all known. If a route is found it uses the endpoint service to send the data from $A$ to $B$.

Rendezvous Protocol. In JXSE the rendezvous service as described above is the implementation of this protocol. It is built on top of the endpoint routing protocol and allows to broadcast messages (to send them to all peers in the network) to peers that do not support multicasting. As multicasting is not available it uses the underlying network transportation layer, either HTTP or TCP.

Pipe Binding Protocol. This protocol is responsible for establishing the pipe connection between at least two peers. It uses the endpoint routing protocol to send initial
messages until a connection is established. The responsibility ends at this point. It is worthwhile mentioning that this protocol always creates a unidirectional pipe where traffic flows only in one direction. Hence, whenever we want to communicate in both directions we have to execute this protocol twice.

**Peer Resolver Protocol.** This protocol is used to query and answer questions between peers. For example, to answer the question if a certain service is available on that peer or not.

**Peer Information Protocol.** This optional protocol is based on the peer resolver protocol. It is used to retrieve information about other peers on the network. JXTA does not describe it more specifically.

**Peer Discovery Protocol.** This protocol is used to retrieve all kinds of resources which are available on the network. By resources we mean advertisements describing those resources. Those can be peers, peer groups, pipes, and many more. We also use it to publish advertisements on the network. We, as a peer, can do this either locally such that other peers have to visit us to fetch advertisements, or remotely such that other peers receive our advertisements.

The implementation of all these protocols can use any programming language. As our implementation of the sender API uses Java we take JXSE, the Java implementation of JXTA. Due to its high level of abstraction, JXTA is relatively easy to use. However, it takes some time to get used to it, that is, the documentation is not exhaustive and not always helpful. We had to browse through a high amount of forums in order to eliminate certain problems. Occurring failures are often not documented at all and since JXTA is running out known problems are partly unresolved. As our implementation does not need all of the features that JXTA provides we touch only a small part of what is possible in JXTA.

### 4.2.2 Tor

JXTA is sufficient for sending data in a distributed network. However, JXTA does not support sending this data anonymously. In order to provide anonymity in our API, which is crucial to protect the privacy in case a principal hides her identity in a proof, we use Tor [DMS04], the state-of-the-art onion routing network. We do not detail how Tor works; this goes beyond the scope of this thesis since we use it only as a building block. Instead, we give a rough overview. The Tor network comprises many nodes. Some of these nodes are entry or exit nodes of the network and some of them are only internal nodes. If peer $A$ wants to send a message $m$ to peer $B$ over the Tor network the following happens: at first, a route is selected through the Tor network. Let the selected route be $(A, C_1, C_2, \ldots, C_n, B)$. $A$ tries to obtain a symmetric key with each node $C_1, \ldots, C_n$ using handshakes. Let these keys be $K_1, K_2, \ldots, K_n$. Assume that every node $C_1, \ldots, C_n$
knows its successor. Then $A$ creates the following encrypted message where we denote by $\{m\}_K$ the symmetric encryption of message $m$ using the symmetric encryption key $K$:

$$E = \{\ldots \{m\}_{K_n} \ldots \}_{K_2}_{K_1}.$$ 

$A$ sends $E$ to $C_1$ which decrypts the message using the symmetric key that it shares with $A$. As $C_1$ knows that the next node is $C_2$ she forwards the message, which is again an encryption, to $C_2$. This process is repeated until $C_n$ is reached. $C_n$ decrypts the plain message and sends it to the intended destination $B$. Notice that the message $m$ can also be an encryption; specifying the system to send messages in plain is just more flexible. As the global route is not known to the internal nodes, Tor guarantees anonymity. The illustration below shows a route from $A$ to $B$ via $C_1$ and $C_2$.

$$A \xrightarrow{\{m\}_{K_1}} C_1 \xrightarrow{\{m\}_{K_2}} C_2 \xrightarrow{m} B$$ 

In order to get a connection to a Tor entry node, the user of the API has to use any available Tor client. We recommend to use Vidalia, a Tor client that forwards every packet via a local proxy to the Tor entry node which forwards it again until it reaches its destination. So in order to send data anonymously a user just has to download this client and configure it accordingly. Our API does the rest.

### 4.2.3 Functionalities

Figure 4.3 illustrates the interaction of the sender API with the network. We detail the interaction as well as the usage of the API in the following. The class JxtaSender implements a sender that uses the JXTA network and selectively the Tor network. It implements all necessary methods and manages connections to the JXTA network. We discuss in the following the most important methods to use and on which actions to pay attention to.

JxtaSender($S1, S2, S3, I1, I2, L1, L2$). The constructor for this sender takes several arguments. There is a String $S1$ for the peer ID which can just be left null since then a new identity is created. String $S2$ and $S3$ contain a name and a description. The user should specify two ports to use for outgoing and incoming tcp connections, one for non-anonymous usage ($I1$) and one for anonymous usage ($I2$). JXTA recommends to use ports located between 9700 and 9799. $L1$ and $L2$ are lists containing ip addresses for rendezvous and relay peers located on the internet. These addresses have the following form: tcp://<ip-address>:<port> or tcp://<domain>:<port>. This constructor generates a new peer with the given specification and starts it. Starting means that it connects the peer to one of the specified rendezvous or relay peers. Together with that it starts several observers which take care of incoming advertisements and messages. Once connected, the sender starts to publish information about itself to inform other participants of its presence.
4.2 Sender API

Figure 4.3 An overview of the sender API. The programmer uses a JxtaSender to send messages selectively over JXTA or Tor. She registers a SenderListener with the JxtaSender in order to receive messages from both networks.

class User implements SenderListener {
    JxtaSender sender;
    public User() {
        //initialization stuff
        sender.registerListener(this);
    }
    public void anonymousMessageArrived(byte[] m) {
        //some action
    }
    public void messageArrived(byte[] m, URI a) {
        //some action
    }
    public void send(URI a, byte[] m, boolean anonymous) {
        if (anonymous) {
            //send m via Tor to a
        } else {
            //send m via JXTA to a
        }
    }
}

stop(). This method stops all actions the sender is performing and disconnects it from the JXTA network, more specifically, from the rendezvous or relay peer to which it is connected.

getConnectedAddresses...(), getConnectedAnonymousAddresses...(). These methods provide information about all known peers in the network. They return maps that contain names and their corresponding addresses on the network. These information are needed to send messages to other peers. Either the name or the address are necessary. As names are not necessarily unique we recommend to always use the unique address to send messages. If this is not desired we recommend to use unique names like e-mail addresses.

send(URI addr, byte[] m, boolean anonymous). The user calls this method whenever she wants to send a message m to a known connected peer with address addr. This peer does not need to be a direct connection; there can be chains of peers in between. The message is a byte array. This is the most general format that we can imagine and it perfectly fits the ProofObject format since we can transform it
into such arrays. The truth value anonymous decides whether the message should be sent over JXTA or anonymously over Tor using Vidalia.

**Receive messages.** In order to receive messages from the JxtaSender, an object using the sender should implement the interface SenderListener and register this object with the sender via the method registerListener(SenderListener). The interface SenderListener defines two methods:

messageReceived(byte[] message, URI sender). This method is called by the sender whenever a message arrives over the JXTA network. As it is sent via JXTA the sender is known immediately.

anonymousMessageReceived(byte[] message). This method is called whenever the sender receives a message over the Tor network. Due to its anonymity property, the sender cannot be determined.

**Where to find the rendezvous and relay peers?** This question is easy to answer: there does not exist any. Everyone who wants to program a system has to take care of this issue by herself. Nevertheless, it is easy to provide these peers. Depending on the size of the system an increasing number of rendezvous and relay peers should be used. Note that relay peers are only necessary when some peers are located behind firewalls or NATs. The code snippet in Listing 4.1 shows the simple implementation of such a peer. In order to get a real network these peers should also be connected among each other. It does not help to have hundreds of these peers not knowing each other.

**Shortcomings.** Our API is convenient to use since the functionality is mostly limited to the basic sending and receiving operations. However, we cannot provide a library function for every JXTA feature, that is, we use JXTA in such a way that we obtain the information that we need in order to provide our functionalities, and not more. We do not implement any service and somehow abuse JXTA’s features. Nevertheless, getting it work is a difficult task. Especially if it comes to connecting to peers via a socket. There are many failures occurring and exceptions thrown. This made us doubting if those problems could be solved at all. Browsing the web for answers to the question where these failures originated, we concluded that just ignoring them and trying repeatedly is the best way to cope with them. For example, we sometimes get connection timeouts although the corresponding time threshold is set to infinity. We did not find anybody who solved this problem. Our solution is just to try to connect as often as necessary until the connection is established. However, we think that this is a very poor solution. Consequently, it can take a while until a connection is established but it needs not. We did not manage to find out the reasons for this problem: sometimes it occurred when many peers tried to get a connection to one specific peer which is somehow reasonable, but it also sometimes occurred if only two peers tried to interact with each other which is not reasonable.

6 A socket is another kind of pipe that offers a bit more functionality. It can be seen as a bidirectional pipe.
Listing 4.1: A simple implementation of a rendezvous-relay peer.

```java
public class RendezvousPeer {
    public static String name = "Rendezvous 1";
    public static int tcpPort = 9701;
    public static List<URI> rendezvousSeeds, relaySeeds;
    // these should be filled with some valid addresses of other
    // rendezvous and relay peers

    public static void main(String... args) throws IOException,
            PeerGroupException {
        // create a new configuration file that stores necessary
        // information
        File file = new File(new File(".cache"), name);
        NetworkManager.RecursiveDelete(file);
        NetworkManager manager = new NetworkManager(NetworkManager.
                ConfigMode.RENDZVOUS_RELAY, name, file.toURI());
        NetworkConfigurator configurator = manager.getConfigurator();

        configurator.setTcpInterfaceAddress(InetAddress.getLocalHost().
                getHostAddress());

        // configure Tcp traffic appropriately
        configurator.setTcpEnabled(true);
        configurator.setTcpIncoming(true);
        configurator.setTcpOutgoing(true);
        configurator.setTcpPort(tcpPort);

        configurator.setUseMulticast(true);

        // drive the network manager to the rendezvous and relay peers
        for (URI address : rendezvousSeeds) {
            configurator.addSeedRendezvous(address);
        }

        for (URI address : relaySeeds) {
            configurator.addSeedRelay(address);
        }

        // start the network and assign the peerGroup and the peerID
        PeerGroup netPeerGroup = manager.startNetwork();

        PeerID peerID = IDFactory.newPeerID(netPeerGroup.getPeerGroupID()
                , name.getBytes());

        configurator.setPeerID(peerID);
    }
}
```
4.3 Implementation of the Feedback System

In order to use our APIs we just include them as libraries and use the classes and functions they provide. We provide the interaction between the proof API and the sender API via transformation of ProofObjects to byte arrays and sending these arrays via the sender to the receiver.

We demonstrate the usage of both APIs by implementing our example from Chapter 2 in Asplada to conveniently interact with the proof API. However, since service-specific pseudonyms are not yet included in the current proof API we implement the example in the state before fixing the problem with multiple feedbacks. We restrict it a bit further: we only show the fundamental creation of proofs and sending these proofs. In order to simplify the sending we need to define several auxiliary methods: the methods defined in Listing 4.2 create a new message that comprises a byte that indicates which kind of message it is and the object itself. The corresponding messageReceived and anonymousMessageReceived methods have to take this into account and process the incoming messages accordingly.

Given these auxiliary methods, we can now have a deeper look at how we create proofs and how we send them. Recall that the professor has an authorization policy (a slightly modified version of (1.3) that allows for quantifying the student’s identity) stating the fact that only registered students are permitted to give their feedbacks:

\[
\forall y, z. (\exists x. \text{Prof says Reg}(x, y) \land x \text{ says Feedback}(y, z)) \rightarrow \text{Rate}(y, z).
\]

Listing 4.3 depicts the code for the registration process of the student. She creates a request of the form

\textit{Stud says Register}(\ell ec)

and sends it to the professor who processes the request and is asked to let the student be registered or not. In case the professor agrees she sends a confirmation of the form

\textit{Prof says Reg}(\textit{Stud}, \ell ec)

which the student verifies to store it for later usage.

After the student has received the confirmation she can enter the feedback process depicted in Listing 4.4. The student uses the confirmation to create a proof for her feedback of the form

\[
\exists x. \text{Prof says Reg}(x, \ell ec) \land x \text{ says Feedback}(\ell ec, gr)
\]

and sends it to the professor. The professor receives this proof and verifies it in order to check if the incoming message passes the authorization policy. If this is the case the professor puts the feedback on the rating board. After the feedback process is finished by every student, the professor can broadcast all feedbacks to the students in order to let them know how the overall quality of the course was.

This example usage shows how convenient it is to use our APIs in applications. In less than 20 lines of Asplada we specify a system whose classical development could take
4.3 Implementation of the Feedback System

Listing 4.2: The auxiliary methods simplifying sending operations for professor and students.

```java
// Professor's auxiliary methods
public void sendConfirmation(ProofObject P, URI address);

public void sendFeedBacks(String feedbacks);

// Student's auxiliary methods
public void sendFeedback(ProofObject text);

public void register(ProofObject registration);
```

ages without using any general programming interface. We admit that in order to get a working program we have to invest a little bit more but everything besides the above code snippets is nothing else but book-keeping and management of user interfaces and storage. Our interface makes such developments very fast. Even without Asplada the API to create proofs and modify them is very general and powerful but of course much more technical.
Listing 4.3: The registration process between the student and the professor only described in Asplada and sender operations.

```plaintext
// If the student presses the register button this action is performed
/*$ */
let S = Stud says Register(Lec)
$*/
studSender.register(S);

// If the professor agrees in registering the student A then a request
// answer is created which is sent back to A (sender)
/*$ */
verify @A says Register(Lec) with P
let RequestAnswer = Prof says Reg(A,Lec)
$*/
profSender.sendConfirmation(RequestAnswer, sender);

// The student verifies that P, the confirmation arrived, is correct and
// stores it for further usage
/*$ */
verify @Prof says Reg(Stud,Lec) with P
$*/
Confirmation = P;
```
4.3 Implementation of the Feedback System

Listing 4.4: The feedback process in Asplada and sender operations.

```plaintext
// The student uses the registration confirmation provided by the
// professor to create her feedback ExAns which she sends via the
// sender to the professor.

String Feed = "my feedback";
/*
   let Q = Stud says Feedback(Lec, Feedback)
   let Ans = Confirmation and Q
   let ExAns = exists Stud. Ans
*/
studSender.sendFeedBack(ExAns);

// Upon receiving this feedback as Q the professor verifies it and puts
// it on the rating board. After all feedbacks arrived she publishes
// the feedbacks for the students.

$/
verify exists X. Prof says Reg(X,Lec) \ X says Feedback(Lec,M) with Q
$/
putOnRatingBoard(M.getName());
publishFeedbacks();
```
Chapter 5

Performance Evaluation

We carry out a performance evaluation for several test cases. This evaluation only affects the proof API and not the sender API since we cannot influence network properties or network qualities. We implemented several case studies to test our proof API for PA-PCA and to test the service-specific pseudonyms presented in Chapter 3. This chapter proceeds as follows: we present a case study on social networks in Section 5.1, one for a health insurance certificate in Section 5.2, and one for an anonymous Like button in Section 5.3. Finally, we present the results of the evaluation in Section 5.4.

5.1 Social Network

We implemented some of the proofs used in the social network framework provided by Backes et al. [BMP11]. In the following, we explain the functionalities of these proofs together with the statements representing them which we have to create. We assume that if two principals $A$ and $B$ are in a friendship, $A$ has given a certificate to $B$ describing this relation:

$A$ says $\text{Relation} (\mathcal{R}, B)$

where $\mathcal{R}$ describes the exact relation between $A$ and $B$.

Resource access. There are several functions that allow for accessing resources such as posting on a friend’s wall or viewing a friend’s pictures. Additionally, a principal can specify whether or not a friend of hers is allowed for hiding her identity or not. Therefore we have two\footnote{In general there are three authentication mechanisms: in addition to the relation and the anonymous authentication there is the pseudonymous authentication. However, our system does not support pseudonyms as given by [BMP11].} authentication protocols that establish the connection to access resources.

Relation authentication. This protocol reveals all information except $B$. Concerning the social network framework, relation authentication reveals in particular the relation tag $\mathcal{R}$. So if $B$ wants to execute this protocol he has to provide a proof for
the following statement:

\[ \exists x. \ A \text{ says Relation}(R, x) \]

where \( x \) was \( B \) before quantification. \( B \) sends this certificate together with a request for the desired resource to \( A \).

**Anonymous authentication.** This protocol hides the relation tag and the identity which makes \( B \) anonymous for \( A \). Using relation authentication, \( A \) could potentially deduce by the relation tag that the quantified \( x \) was \( B \). For instance, when \( B \) is the only friend with relation \( R \). Here, \( B \) creates a proof for the following statement together with a proof for the desired resource:

\[ \exists x, y. \ A \text{ says Relation}(x, y). \]

The rest of the protocol is the same as above.

**Friend-of-friend.** This protocol is very advanced. Suppose that \( A \) is a friend of \( B \) and \( B \) is also a friend of \( C \), but \( A \) and \( C \) are not in a friendship so far. This is expressed by the following two statements:

\[
\begin{align*}
A & \text{ says Relation}(R, B) \quad (5.1) \\
B & \text{ says Relation}(R', C). \quad (5.2)
\end{align*}
\]

Now \( C \) wants to be in a friendship with \( A \). The friend-of-friend protocol is able to establish this friendship without the interaction of \( A \). The protocol is based on a communication between \( B \) and \( C \) while \( C \) stays anonymous. At first, \( C \) formulates a request using the signature in (5.2) and sends it to \( B \). This request has the following form:

\[ \exists x, y. \ B \text{ says Relation}(y, x) \land x \text{ says Request}(A). \]

\( B \) receives this request, verifies it, and creates a proof for the following statement:

\[ \exists x, y, z. \ z \text{ says Relation}(y, x) \land A \text{ says Relation}(R, z) \]

using the signature in (5.1). In order to achieve unlinkability, \( B \) has to re-randomize his identity after quantification. Upon receiving this statement from \( B \), \( C \) re-instantiates her identity into the statement. She obtains the friendship relation between her and \( A \). Note that \( C \) does not necessarily know who exactly sent the response since it could be that she requested the friendship from several friends. The final friendship certificate is

\[ \exists x, y. \ A \text{ says Relation}(R, x) \land x \text{ says Relation}(y, C). \]

This statement looks and indeed is a transitive connection from \( A \) to \( C \). The construction of the proof ensures that the person in between is a common friend of \( A \) and \( C \).
5.2 Health Insurance Certificates

If an employee is ill she has to undergo a medical examination. After this medical examination the doctor creates a health insurance certificate. The patient has to forward this certificate to both the employer and the health insurance so that it is okay that she stays away from office. However, the diagnosis should not be revealed to the employer but to the health insurance. Suppose, for instance, that the patient is pregnant. In case the employer knows this fact she could not want to renew the patient’s contract. There are many more cases that cause disadvantages for the patient.

PA-PCA is well-suited to solve these problems. Any licensed doctor has a certificate of the form

\[ \text{Hosp says IsDoctor}(B). \]

This certificates stands for the approval of a hospital Hosp for the doctor B. A has to meet the following policy of her employer if she is ill and wants to stay at home from office:

\[ \forall p, x. \exists d, y. \text{Hosp says IsDoctor}(d) \land d \text{ says Visit}(p, x, y) \rightarrow \text{OkOff}(p). \]

This policy says that if there is some approved doctor d who attests the patient p a visit at date x with result y then it is okay for p to stay at home. When A visits the doctor B in case she is ill, B tells her the results res of the examination at date date via a certificate of the form

\[ \text{Hosp says IsDoctor}(B) \land B \text{ says Visit}(A, \text{date}, \text{res}). \]

Then A existentially quantifies the results and the doctor, and forwards the so obtained certificate

\[ \exists d, r. \text{Hosp says IsDoctor}(d) \land d \text{ says Visit}(A, \text{date}, r) \]  \hspace{1cm} (5.3)

to her employer. The received proof meets the policy and it is okay for A to stay home. In order to illustrate the beauty of open-endedness once more we can use the doctor’s certificate to get medicine from the pharmacy. So only the results and the patient is necessary but both the doctor and the date are not needed in this scenario. A changes the doctor’s certificate into the following one:

\[ \exists d, t. \text{Hosp says IsDoctor}(d) \land d \text{ says Visit}(A, t, \text{res}). \]

Then the pharmacy can provide the appropriate medicine for A.

5.3 Anonymous Like Button

When we browse the web today, we will find social plugins on most websites. Social plugins are objects integrated into websites such as the Like button from Facebook [Fb] or Google+1, a very similar button in Google+ [GPO]. These social plugins provide several functionalities. For instance, Google+1 informs our friends of pages, videos, or
any kind of objects equipped with such a button that we like. Additionally, Google somehow includes the results of these button click statistics in its page ranking engine. These plugins are very conveniently to use since one click is sufficient to provide our friends with information about our preferences. However, in order to use these plugins one has to be logged into the system that offers the button. If we want to use Google+1 we have to be logged into Google+. However, whenever we are logged in, all information we provide our friends is also provided to the operator of the system. This entails many privacy issues, for instance, Nic Cubrilovic [Cub] encountered an issue when users log out of Facebook: several cookies, including the one which includes the account number, are kept and hence forwarded every time a user loads a website with Facebook integrated, particularly if there is a Like button on that website. By doing this, Facebook could track users even when they are logged out. This issue seemed to be removed by the end of 2011 but it shows the vulnerabilities of these plugins.

To overcome these privacy issues we use service-specific pseudonyms. Note that using PA-PCA is not possible since we want users to like an object only once. Hence SSPs are well-suited for this purpose. The operator of the system provides a certificate proving a statement of the form

\[
\text{Operator says Permitted} (\text{System, A})
\]

which says that the operator Operator of System approves A to be a participant of it. Whenever Operator places a Like button on a website she creates a new service s. The service is in this case the object for which the button is placed. When A wants to like s she creates a service-specific pseudonym psd and a proof for the statement

\[
\exists x. \text{Operator says Permitted} (\text{System, x}) \land \text{SSP} (\text{psd, x, s}) \land x \text{ says Like} (\text{comment}).
\]

She sends the proof to Operator who verifies it and processes the information. It is guaranteed by the uniqueness of the pseudonym psd that every participant can like one particular object only once. Additionally, the SSP does not reveal any information about A except psd which cannot be linked to A.

5.4 Results

We use a computer with a quad core processor\(^2\) and 4 GB of RAM for our experiments. We measured the proof generation and proof verification time as well as the size of the proofs in kilobytes for all our experiments described above.

Security Parameter. In order to get a prediction for different application scenarios we run our experiments with different security parameters. These parameters are directly connected to the size of the elliptic curves which we use. These curves are such that the security parameter is half the size of the curve [NIS11]: for example, if our curve has size 224 bits then the security parameter has size 112 bits. Since 2010 a security

\(^2\)Intel Core i7-2760QM CPU @ 2.40GHz
5.4 Results

A parameter of 112 bits is recommended. However, from about 2030 experts recommend to have a security parameter higher than that. Therefore, we also measured the times for a security parameter of 128 bits. To have a glance even further into the future we also evaluated times and sizes for a parameter of 256 bits.

**Discussion.** The results of the case studies described above together with our evaluation system example are depicted in Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, and Figure 5.5. Overall, we observe that the graphs are slightly growing and that the shape is in all cases similar. Additionally, the times for a security parameter of 512 bits are only of theoretical interest since they go beyond the scope of today’s computing power and are highly impractical. We just show the results without detailing them. It is just interesting to see what is possible today for requirements which we need to fulfill in perhaps twenty years.

The results for the complete evaluation process depicted in Figure 5.1 are promising. Note that we have three proof verifications. These three verifications take not more than about 25 seconds for a security parameter of 112 bits. Generation times are even faster with about 10 seconds. The complete proof size is about 230 KB which is reasonable given the continuous spread of broad-band internet connections. Even for a higher security parameter of 128 bits the overall verification time for three verifications lies under 38 seconds and the generation time at about 14 seconds. The proof size does not grow too much and is reasonable with under 275 KB. These times seem to be almost practical in a sense that if we assume that the system does not need to provide real-time evaluations, it is acceptable that a proof takes 10 seconds to be verified. In a lecture with one hundred students this results in a complete verification time of under twenty minutes.

The results for the authentication process in social networks depicted in Figure 5.2 are very convincing. Generation times of about 4 seconds and verification times of about 7 seconds for a 112 bits security are practical for real systems. Also for the even higher 128 parameter generation times of under 6 and verification times of 10 seconds are not that much higher. Proof sizes of about 133 (resp. 154) KB for 112 (resp. 128) bits security are acceptable.

The results for the friend-of-friend authentication in Figure 5.3 are not completely convincing. As generation times lie at 10 (resp. 14) seconds and verification times at 23 (resp. 33) seconds for 112 (resp. 128) bits security, this protocol is not suited for chatting. However, posting messages on a wall does not require real-time evaluations. Therefore it depends on the application. Proof sizes which are even below the one for the evaluation system are acceptable.

The results for the health insurance certificate shown in Figure 5.4 are again auspicious. Proof generation times of 9 (resp. 12) seconds and verification times of 15 (resp. 21) seconds are more practical than the one for the friend-of-friend protocol, particularly because we do not need real-time evaluations in this protocol. Proof sizes of 264 (resp. 306) KB are not disturbing.

Finally, we do not deliver a judgment concerning the results for the anonymous Like button depicted in Figure 5.5. The proof generation times of about 10 (resp. 13) seconds and the verification times of about 15 (resp. 22) seconds are relatively high.
Figure 5.1 The results for the lecture evaluation system, that is, the complete process including registration and feedback submission. In total we have three verifications. The code for this system is described and discussed in Section 4.3.

(a) Proof and verification times.

(b) Proof size.

Figure 5.2 The results for the authentication process for social networks from Section 5.1. The proof generation and verification times do not differ for both processes, that is, relation authentication and anonymous authentication. However, since the anonymous authentication hides an identity the sizes differ. We do not show both size graphs since the difference is not visible.

(a) Proof and verification times.

(b) Proof size.

Proof size rel. auth.
5.4 Results

**Figure 5.3** The results for social networks friend-of-friend authentication as presented in Section 5.1.

(a) Proof and verification times.

(b) Proof size.

**Figure 5.4** The results for the health insurance certificates in (5.3) as presented in Section 5.2.

(a) Proof and verification times.

(b) Proof size.

**Figure 5.5** The results for the anonymous Like button as described in Section 5.3.

(a) Proof and verification times.

(b) Proof size.
Chapter 6

Conclusion

We have presented a framework to automatically generate executable, security sensitive distributed systems. With this framework developers are enabled to synthesize distributed systems that are capable of managing authorization, protect user privacy and the privacy of data, and enforce linearity constraints on certain actions. The base for our framework is the privacy-aware proof-carrying authorization (PA-PCA) framework [BMP12, MP11] which builds on the proof-carrying authorization (PCA) framework [AF99] and E-DKAL [ABN11, GN08]. This tool provides us already with authorization and privacy. Only linearity is a missing property.

In this thesis, we have introduced service-specific pseudonyms. We advocate the usage of service-specific pseudonyms in applications which restrict the execution of certain actions to a limited number of times. Due to their implementation, service-specific pseudonyms are unique if the service is fixed; unique means in this context that every principal can generate exactly one pseudonym per service and that no two different principals can generate the same pseudonym for the same service. It turns out that service-specific pseudonyms fit smoothly into the existing PA-PCA system. For this reason, we have extended PA-PCA with service-specific pseudonyms and have ended up with a framework that can synthesize exactly the desired distributed systems.

This framework is conveniently usable and proven to generate the intended systems. In order to transfer this into practice, we have implemented this framework in Java. We provide the implementation in form of two independently usable libraries (APIs). Taken together, the two libraries provide the functionality of the PA-PCA framework. The first API allows for generating, modifying, and verifying proofs and signatures as given by the PA-PCA framework. Together with this API we deliver the programming language Asplada that establishes a convenient method to interact with the proof API and can be integrated and translated into Java code. The translated Java code consists of calls to the proof API. So the user does not have to be an expert in the API and even more important, she does not have to be an expert in cryptography. The second API establishes sending and receiving operations on a distributed network; developers can choose between non-anonymous sending and anonymous sending.
We have demonstrated the expressiveness of our extension to the PA-PCA framework and of our implementation by using our APIs to implement the evaluation system which we have presented in this thesis. Additional to the complete implementation of the evaluation system, we have carried out several other case studies and measured proof generation and verification times as well as proof sizes for these case studies. The results are promising.

**Future Work.** As we have already mentioned in Section 4.1.3 there is a more sophisticated method to implement verification of Groth-Sahai proofs called batch verification technique. This could potentially result in a speed-up of up to 90%.

Another approach could be to implement the APIs in C. As C is more machine-oriented than Java which runs in its own virtual machine we could potentially gain additional speed-ups. However, the expected improvements from an implementation in C are rather small. Further, C is not platform independent like Java. We cannot just give a jar-file that provides the corresponding library functions. Even the integration of PBC instead of jPBC creates problems concerning platform independency. In order to work the system we need special libraries which need to be compiled specifically for the computer on which the API runs.

The extension of PA-PCA with SSPs is well-suited to model distributed systems that manage authorization, are aware of privacy of data and user privacy, and can enforce linearity constraints. However, in order to enforce the linearity in the systems, the developer has to take care of this fact in the implementation, whereas authorization and privacy are modeled within the logic of PA-PCA. The logic underlying PA-PCA does not support linearity constraints since it is not linear. A follow-up work could be to change the logic to a linear or at least affine one. This is not trivial since not all parts of the system can be treated linear, like, for instance, authorization policies. A possible starting point is the PhD thesis of Limin Jia [Jia08], which introduces a logic that incorporates both linear and non-linear components.
Bibliography


Appendix A

Cryptographic Implementation

A.1 Groth-Sahai Zero-Knowledge Proofs

This section details the Groth-Sahai [GS08]. We start with a few preliminary notions (cf. Section A.1.1), explain how the commitments are computed, and show which components the common reference string comprises (cf. Section A.1.2). Then we present the proof construction (cf. Section A.1.3) followed by the proof verification (cf. Section A.1.4) and the correctness proofs (cf. Section A.1.5).

A.1.1 Preliminaries

Definition A.1 (R-module). Let \((R, +, \cdot)\) be a commutative ring. \((A, +)\) is an \(R\)-module if \((A, +)\) is an abelian group and for all \(r, s \in R\) and \(x, y \in A\)

\[
(r + s)x = rx + sx \quad r(x + y) = rx + ry \quad r(sx) = (rs)x \quad 1x = x
\]

hold.

The abbreviation which we introduce next is generally persistent for any set \(R\) and any \(R\)-module \(A\). But since we are only using \(\mathbb{Z}_p\)-modules we concentrate from now on the special case of \(\mathbb{Z}_p\) modules.

In order to make the equations in Table 2.8 more readable, we introduce an abbreviation. Let \(A_1, A_2, A_T\) be \(\mathbb{Z}_p\)-modules and \(f : A_1 \times A_2 \rightarrow A_T\) be a bilinear map on those modules. Let \(\vec{x} \in A_1\) and \(\vec{y} \in A_2\). We define \(\cdot : A_1 \times A_2 \rightarrow A_T\) as

\[
\vec{x} \cdot \vec{y} := \sum_{i=1}^{n} f(x_i, y_i).
\] (A.1)

Lemma A.2. Let \(\vec{x} \in A_1^n\), \(\vec{y} \in A_2^n\), and \(\Gamma \in \mathbb{Z}_p^{n \times m}\). Then we have

\[
\vec{x} \cdot \Gamma \vec{y} = \Gamma^T \vec{x} \cdot \vec{y}.
\]
Proof. Let $\vec{x} \in A_1^n$, $\vec{y} \in A_2^m$ and $\Gamma \in \mathbb{Z}_p^{n \times m}$. Then we have
\[
\vec{x} \cdot \Gamma \vec{y} = \sum_{i=1}^{n} f \left( \sum_{j=1}^{m} \gamma_{ij} y_j \right) = \sum_{i=1}^{n} \sum_{j=1}^{m} f(x_i, \gamma_{ij} y_j) = \sum_{j=1}^{m} \sum_{i=1}^{n} f(\gamma_{ij} x_i, y_j) = \sum_{j=1}^{m} \Gamma^\top \vec{x} \cdot \vec{y}.
\]
This chain of equations holds by exploiting the properties of the bilinear map. □

With equation (A.1) and Lemma A.2 we can transform the equations in Table 2.8 into those in Table A.1 where all the $A_i, B_i, X_i, Y_i, a_i, b_i, x_i, y_i$ are collected in vectors $\vec{A}, \vec{B}, \vec{X}, \vec{Y}, \vec{a}, \vec{b}, \vec{x}, \vec{y}$ and the $\gamma_{ij}$ are collected in a matrix $\Gamma$. For the pairing product equation we define $f$ to be the bilinear map $e$, for the multi-scalarmultiplication equations we define $f$ as $f(x, y) = y X$, and for the quadratic equations we define $f$ as $f(x, y) = xy \mod n$.

Table A.1 The equations in abbreviated form from Table 2.8.

<table>
<thead>
<tr>
<th>Equation Type</th>
<th>Abbreviation</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE with $f(x, y) = e(x, y)$</td>
<td>$\vec{A} \cdot \vec{Y}$</td>
<td>$(\vec{A} \cdot \vec{Y})(\vec{A} \cdot \vec{B})(\vec{A} \cdot \vec{X} \cdot \vec{Y}) = t_T$</td>
</tr>
<tr>
<td>MSM1 with $f(X, Y) = y X$</td>
<td>$\vec{A} \cdot \vec{Y}$</td>
<td>$\vec{A} \cdot \vec{Y} + \vec{x} \cdot \vec{b} + \vec{y} \cdot \vec{X} = \tau_1$</td>
</tr>
<tr>
<td>MSM2 with $f(X, Y) = y X$</td>
<td>$\vec{A} \cdot \vec{Y}$</td>
<td>$\vec{A} \cdot \vec{Y} + \vec{x} \cdot \vec{b} + \vec{y} \cdot \vec{X} = \tau_2$</td>
</tr>
<tr>
<td>QE with $f(x, y) = xy \mod n$</td>
<td>$\vec{A} \cdot \vec{Y}$</td>
<td>$\vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{y} \cdot \vec{X} = \tau_3$</td>
</tr>
</tbody>
</table>

### A.1.2 Commitment Computation and the Common Reference String

The commitment scheme is based on the idea that we have $\mathbb{Z}_p$-modules $A_1, A_2$, and $A_T$ with a bilinear map $f$; we transform the values of these modules in other $\mathbb{Z}_p$-modules $B_1, B_2$, and $B_T$ with a bilinear map $F$; we commit in these modules. For the transformation there are the mappings $\iota_1 : A_1 \rightarrow B_1, \iota_2 : A_2 \rightarrow B_2$, and $\iota_T : A_T \rightarrow B_T$ with the corresponding inverse mappings $p_1, p_2$ and $p_T$. The following graph illustrates the possible transformations quickly:

\[
\begin{align*}
A_1 & \times A_2 \xrightarrow{f} A_T \\
\iota_1 \uparrow p_1 & \quad \iota_2 \uparrow p_2 \\
B_1 & \times B_2 \xrightarrow{F} B_T \\
\iota_T \uparrow p_T
\end{align*}
\]

The next abbreviation is similar to that in equation (A.1). However, with this one we address multiplications of already transformed values. Let $\vec{u} \in B_1^n$, $\vec{v} \in B_2^n$, and $F$ be a bilinear map. Then we define $\bullet : B_1 \times B_2 \rightarrow B_T$ as
\[
\vec{u} \bullet \vec{v} = \sum_{i=1}^{n} F(u_i, v_i)
\]
(A.2)
The \( \bullet \) operation has the same property as the \( \cdot \) operation with respect to matrix multiplication, hence:

**Lemma A.3.** Let \( \vec{u} \in \mathbb{B}_1^n \), \( \vec{v} \in \mathbb{B}_2^m \) and \( \Gamma \in \mathbb{Z}_p^{n \times m} \). Then we have

\[
\vec{u} \bullet \Gamma \vec{v} = \Gamma^T \vec{u} \bullet \vec{v}.
\]

**Proof.** The proof is analogous to the proof of Lemma A.2 but replacing \( f \) with \( F \), \( x \) with \( u \) and \( y \) with \( v \). \( \square \)

Recall that the algorithm \texttt{SetupBM} produced a setup \( gk \) containing the group order, the respective groups \( G_1 \), \( G_2 \), and \( G_T \), the bilinear map \( e \), and the group generators \( G \) and \( H \). Let \( O \) be the neutral element for \( G_1 \), \( G_2 \), and \( G_T \). Note that \( O \) may be pairwise different between the different groups, but for convenience we just take one notation for all three possibly different neutral elements. Given \( gk = (p, G_1, G_2, G_T, e, G, H) \), our proof scheme instantiation sets

\[
A_1 = G_1 \quad A_2 = G_2 \quad A_T = G_T
\]

\[
B_1 = G_1^2 \quad B_2 = G_2^2 \quad B_T = G_T^4
\]

In order to transform the values from \( A_i \) to \( B_i \) and from \( \mathbb{Z}_p \) to \( B_i \), respectively, we define the mappings

\[
i_1 : A_1 \to B_1 \quad p_1 : B_1 \to A_1 \quad i_2 : A_2 \to B_2 \quad p_2 : B_2 \to A_2
\]

\[
i_T : A_T \to B_T \quad p_T : B_T \to A_T \quad i'_1 : \mathbb{Z}_p \to B_1 \quad p'_1 : B_1 \to \mathbb{Z}_p
\]

\[
i'_2 : \mathbb{Z}_p \to B_2 \quad p'_2 : B_2 \to \mathbb{Z}_p \quad i'_T : \mathbb{Z}_p \to B_T \quad p'_T : B_T \to \mathbb{Z}_p
\]

In order to define these functions, we need (i) the common reference string (CRS), which has the form \((u_1,u_2,v_1,v_2)\) and (ii) the definition of the bilinear map \( F \) for the transformed sets. We start by defining the CRS components.

We define \( u_1 \) and \( v_1 \) as

\[
u_1 := (G, \alpha G) \quad v_1 := (H, \beta H)
\]

for some \( \alpha, \beta \in \mathbb{Z}_p \) where \( \in \mathbb{Z}_p \) means that the elements on the left-hand-side are chosen uniformly at random.

Depending on the commitment scheme we define \( u_2 \) and \( v_2 \):

\[
u_2 := \begin{cases} t_1 u_1 & \text{if binding} \\ t_1 u_1 - (O, G) & \text{if hiding} \end{cases}
\]

\[
v_2 := \begin{cases} t_2 v_1 & \text{if binding} \\ t_2 v_1 - (O, H) & \text{if hiding} \end{cases}
\]

If we choose a hiding commitment scheme we will achieve witness-indistinguishability; otherwise the commitment is an El-Gamal encryption. Hence, it is binding since encryptions can only be opened with the corresponding decryption key. Our implementation
takes a binding commitment key in order to get proofs of knowledge. Note that in order to encrypt a message, one has to know it in advance; this is an intuitive explanation of proofs of knowledge.

Let $u := u_2 - (\mathcal{O}, \mathcal{G})$ and $v := v_2 - (\mathcal{O}, \mathcal{H})$. We define $F : \mathbb{G}_1^2 \times \mathbb{G}_2^2 \rightarrow \mathbb{G}_T^4$ as follows:

$$F \left( \left( \mathcal{X}_1, \mathcal{Y}_1 \right), \left( \mathcal{X}_2, \mathcal{Y}_2 \right) \right) = \left( e(\mathcal{X}_1, \mathcal{Y}_1) e(\mathcal{X}_1, \mathcal{Y}_2), e(\mathcal{X}_2, \mathcal{Y}_1) e(\mathcal{X}_2, \mathcal{Y}_2) \right).$$

Now we are ready to define the above functions. We begin with those needed to compute the commitments.

$$\iota_1(Z) := (\mathcal{O}, Z) \quad p_1(Z_1, Z_2) := Z_2 - \alpha Z_1$$
$$\iota_2(Z) := (\mathcal{O}, Z) \quad p_2(Z_1, Z_2) := Z_2 - \beta Z_1$$
$$\iota'_1(z) := zu \quad p'_1(z_1 \mathcal{G}, z_2 \mathcal{G}) := z_2 - \alpha z_1$$
$$\iota'_2(z) := zv \quad p'_2(z_1 \mathcal{H}, z_2 \mathcal{H}) := z_2 - \beta z_1$$

We need the other mappings to generate fitting values for the proof construction. The definitions are below.

$$\nu_T(z) := \left( \begin{array}{cc} 1 & 1 \\ 1 & z \end{array} \right) \quad p_T(\left( \begin{array}{c} z_{11} \\ z_{21} \\ z_{12} \\ z_{22} \end{array} \right)) := z_{22} z_{21}^{\alpha} (z_{21} z_{11}^{\alpha})^{-\beta}$$
$$\nu'_T(z) := F(\iota'_1(z), \iota'_2(z)) = F(u, v)^z \quad p'_T(z) := \log_{\psi(\mathcal{G}, \mathcal{H})} (p_T(z))$$
$$\nu_T(Z) := F(\iota_1(Z), \iota_2(Z)) = F(\mathcal{O}, Z) \quad p_T(z) := c^{-1}(p_T(z))$$
$$\nu'_T(Z) := F(\iota'_1(Z), \iota'_2(1)) = F(\mathcal{O}, Z, v) \quad p'_T(z) := c^{-1}(p_T(z))$$

Given these functions, we can now define how we commit elements from the different groups into the corresponding other groups.

**Commitment to** $\mathcal{X} \in \mathbb{G}_1$. In order to compute the commitment $c$ we pick randomesses $r_1$ and $r_2$ and compute

$$c := \iota_1(\mathcal{X}) + r_1 u_1 + r_2 u_2.$$

**Commitment to** $\mathcal{Y} \in \mathbb{G}_2$. In order to compute the commitment $d$ we pick randomesses $s_1$ and $s_2$ and compute

$$d := \iota_2(\mathcal{Y}) + s_1 v_1 + s_2 v_2.$$

**Commitment to** $x \in \mathbb{Z}_p$. In order to compute a commitment of an exponent $x$ into one of the groups $\mathbb{G}_1$ or $\mathbb{G}_2$ we pick randomness $r_1$ or $s_1$, depending on which group we choose and compute either

$$c' := \iota'_1(x) + r_1 u_1 \quad \text{or} \quad d' := \iota'_2(x) + s_1 v_1.$$
A.1.3 Proof Construction

Given the commitment construction functions and the common reference string, we can now proceed by explaining how we construct the proofs for the different types of equations.

Let $\vec{X} \in G^n_1$, $\vec{Y} \in G^n_2$, $\vec{x} \in \mathbb{Z}^{n'}_p$, and $\vec{y} \in \mathbb{Z}^{m'}_p$ be the witnesses of the proof. Then, at first, we compute the commitments to the different witnesses. So we define for each of the different $\iota$ functions the extension to vectors for some $n$-ary vector $\vec{z}$ as

$$
\iota(\vec{z}) := \begin{pmatrix}
\iota(z_1) \\
\vdots \\
\iota(z_n)
\end{pmatrix},
$$

Furthermore, let $\vec{u} = (u_1, u_2)\top$ and $\vec{v} = (v_1, v_2)\top$. We randomly choose $R \in \mathbb{Z}^{n \times 2}_p$, $S \in \mathbb{Z}^{m \times 2}_p$, and compute

$$
\bar{c} := \iota_1(\vec{X}) + Ru \\
\bar{d} := \iota_2(\vec{Y}) + Sv.
$$

Now we can continue with the proof constructions for the different equation types.

**Pairing Product Equations.** For equations of the form $(\vec{A} \cdot \vec{Y})(\vec{X} \cdot \vec{B})(\vec{X} \cdot \Gamma \vec{Y}) = t_T$ where $\vec{A} \in G^n_1$, $\vec{B} \in G^n_2$, $\Gamma \in \mathbb{Z}^{n \times m}_p$, and $t_T \in G_T$, choose a random matrix $T \in \mathbb{Z}^{2 \times 2}_p$ and compute the pair of vectors $(\vec{\pi}, \vec{\theta}) \in G^{2 \times 2}_2 \times G^{2 \times 2}_1$ as

$$
\vec{\pi} := R^\top \iota_2(\vec{B}) + R^\top \Gamma \iota_2(\vec{Y}) + (R^\top \Gamma S - T^\top) \vec{u} \\
\vec{\theta} := S^\top \iota_1(\vec{A}) + S^\top \Gamma^\top \iota_1(\vec{X}) + T \vec{u}.
$$

(A.3) (A.4)

If either of both, $\vec{X}$ or $\vec{Y}$ is not present, that is, we have either an equation of the form $\vec{A} \cdot \vec{Y} = t_T$ or $\vec{X} \cdot \vec{B} = t_T$ then we compute either

$$
\vec{\pi} := R^\top \iota_2(\vec{B}) \text{ or } \vec{\theta} := S^\top \iota_1(\vec{A}).
$$

**Multi-Scalar Multiplication Equations in $G_1$.** For equations of the form $\vec{A} \cdot \vec{y} + \vec{X} \cdot \vec{B} + \vec{X} \cdot \Gamma \vec{y} = T_1$ where $\vec{A} \in G^n_1$, $\vec{B} \in \mathbb{Z}^{n'}_p$, $\vec{x} \in \mathbb{Z}^{n \times m'}_p$, and $T_1 \in G_1$, choose a random matrix $T \in \mathbb{Z}^{1 \times 2}_p$ and compute the vector-value pair $(\vec{\pi}, \vec{\theta}) \in G^{2 \times 2}_2 \times G^{2 \times 2}_1$ as

$$
\vec{\pi} := R^\top \iota_2'(\vec{B}) + R^\top \Gamma \iota_2'(\vec{y}) + (R^\top \Gamma S - T^\top) \vec{v}_1 \\
\vec{\theta} := S^\top \iota_1(\vec{A}) + S^\top \Gamma^\top \iota_1(\vec{X}) + T \vec{u}.
$$

(A.5) (A.6)
Cryptographic Implementation

If either of both, $\vec{X}$ or $\vec{Y}$ is not present, that is, we have either an equation of the form $\vec{A} \cdot \vec{Y} = \mathcal{T}_1$ or $\vec{X} \cdot \vec{b} = \mathcal{T}_1$ then we compute either

$$\vec{\pi} := R\top \iota_2(\vec{b}) \quad \text{or} \quad \theta := \vec{s}\top \iota_1(\vec{A}).$$

**Multi-Scalar Multiplication Equations in $\mathbb{G}_2$.** For equations of the form $\vec{a} \cdot \vec{Y} + \vec{X} \cdot \vec{b} = \mathcal{T}_2$ where $\vec{a} \in \mathbb{Z}_p^m$, $\vec{b} \in \mathbb{G}_2^m$, $\Gamma \in \mathbb{Z}_p^{m' \times m}$, and $\mathcal{T}_2 \in \mathbb{G}_2$, choose a random matrix $\mathcal{T} \in \mathbb{Z}_p^{2 \times 1}$ and compute the value-vector pair $(\vec{\pi}, \vec{\theta}) \in \mathbb{G}_2^2 \times \mathbb{G}_1^2$ as

$$\vec{\pi} := r\top \iota_2(\vec{b}) + r\top \Gamma \iota_2(\vec{Y}) + (r\top \Gamma S - \mathcal{T}\top)\vec{v} \quad \text{(A.7)}$$
$$\vec{\theta} := S\top \iota_1(\vec{a}) + S\top \Gamma \iota_1(\vec{x}) + Tu_1. \quad \text{(A.8)}$$

If either of both, $\vec{x}$ or $\vec{Y}$ is not present, that is, we have either an equation of the form $\vec{a} \cdot \vec{Y} = \mathcal{T}_2$ or $\vec{X} \cdot \vec{b} = \mathcal{T}_2$ then we compute either

$$\vec{\pi} := r\top \iota_2(\vec{b}) \quad \text{or} \quad \vec{\theta} := S\top \iota_1(\vec{a}).$$

**Quadratic Equations.** For equations of the form $\vec{a} \cdot \vec{Y} + \vec{x} \cdot \vec{b} + \vec{x} \cdot \Gamma \vec{Y} = t$ where $\vec{a} \in \mathbb{Z}_p^m$, $\vec{b} \in \mathbb{Z}_p^m$, $\Gamma \in \mathbb{Z}_p^{m' \times m}$, and $t \in \mathbb{Z}_p$, choose a random value $\mathcal{T} \in \mathbb{Z}_p$ and compute the value pair $(\vec{\pi}, \vec{\theta}) \in \mathbb{G}_2^2 \times \mathbb{G}_1^2$ as

$$\vec{\pi} := r\top \iota_2(\vec{b}) + r\top \Gamma \iota_2(\vec{y}) + (r\top \Gamma S - \mathcal{T}\top)v_1 \quad \text{(A.9)}$$
$$\vec{\theta} := s\top \iota_1(\vec{a}) + s\top \Gamma \iota_1(\vec{x}) + Tu_1. \quad \text{(A.10)}$$

If either of both, $\vec{x}$ or $\vec{Y}$ is not present, that is, we have either an equation of the form $\vec{a} \cdot \vec{Y} = t$ or $\vec{x} \cdot \vec{b} = t$ then we compute either

$$\vec{\pi} := r\top \iota_2(\vec{b}) \quad \text{or} \quad \vec{\theta} := s\top \iota_1(\vec{a}).$$

### A.1.4 Proof Verification

Given a constructed proof tuple we verify them according to the equations they are supposed to prove by checking equality for the following equations.

**Pairing Product Equations.** For equations of the form $(\vec{A} \cdot \vec{Y})(\vec{X} \cdot \vec{B})(\vec{X} \cdot \Gamma \vec{Y}) = \mathcal{T}_T$ and a given proof tuple $(\vec{\pi}, \vec{\theta})$ we check that

$$\iota_1(\vec{A}) \bullet \vec{d} + \vec{c} \bullet \iota_2(\vec{B}) + \vec{c} \bullet \Gamma \vec{d} = \mathcal{T}_T(\mathcal{T}_T) + \vec{u} \bullet \vec{\pi} + \vec{\theta} \bullet \vec{v}. \quad \text{(A.11)}$$

If the left-hand-side is equal to the right-hand-side we say that the proof verifies and otherwise not.
Multi-Scalar Multiplication Equations in $\mathbb{G}_1$. For equations of the form $\vec{A} \cdot \vec{y} + \vec{X} \cdot \vec{b} + \vec{X} \cdot \Gamma \vec{y} = T_1$ and a proof tuple $(\vec{a}, \vec{b})$ we check that

$$\nu_1(\vec{A}) \cdot \vec{a} + \vec{c} \cdot \nu_2(\vec{b}) + \vec{c} \cdot \Gamma \vec{d} = \nu_T(T_1) + \vec{u} + F(\theta, \nu_1). \quad (A.12)$$

Multi-Scalar Multiplication Equations in $\mathbb{G}_2$. For equations of the form $\vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{x} \cdot \Gamma \vec{y} = T_2$ and a proof tuple $(\vec{a}, \vec{b})$ we check that

$$\nu_1'(\vec{a}) \cdot \vec{a} + \vec{c} \cdot \nu_2'(\vec{b}) + \vec{c} \cdot \Gamma \vec{d} = \nu_T'(T_2) + F(u_1, \nu) + \vec{v}. \quad (A.13)$$

Quadratic Equations. For equations of the form $\vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{x} \cdot \Gamma \vec{y} = t$ and a proof tuple $(\vec{a}, \vec{b})$ we check that

$$\nu_1'(\vec{a}) \cdot \vec{a} + \vec{c} \cdot \nu_2'(\vec{b}) + \vec{c} \cdot \Gamma \vec{d} = \nu_T'(t) + F(u_1, \nu) + F(\theta, \nu_1). \quad (A.14)$$

A.1.5 Correctness Proofs

In this section we prove that the verification step succeeds if and only if, the proof to verify was computed correctly.

**Lemma A.4.** Let $\vec{X} \in \mathbb{G}_1^m, \vec{Y} \in \mathbb{G}_2^m, R \in \mathbb{Z}_p^{n \times 2}$ and $S \in \mathbb{Z}_p^m$ such that they satisfy

$$\vec{c} = \nu_1(\vec{X}) + R\vec{u} \quad \vec{d} = \nu_2(\vec{Y}) + S\vec{v} \quad (\vec{A} \cdot \vec{Y})(\vec{X} \cdot \vec{B})(\vec{X} \cdot \vec{Y}) = tT.$$

Then we have for all $T \in \mathbb{Z}_p^{2 \times 2}$, $\vec{a}$ as constructed in (A.3) and $\vec{b}$ as constructed in (A.4) that the verification equation (A.11) holds.

**Proof.** By the commutative property of the linear and bilinear maps we have for the above equation

$$(\nu_1(\vec{A}) \cdot \nu_2(\vec{Y}))((\nu_1(\vec{X}) \cdot \nu_2(\vec{B}))(\nu_1(\vec{X}) \cdot \Gamma \nu_2(\vec{Y}))) = \nu_T(tT).$$

Let $T \in \mathbb{Z}_p^{2 \times 2}$ be a random matrix. Then we have

$$\nu_1(\vec{A}) \cdot \vec{a} + \vec{c} \cdot \nu_2(\vec{B}) + \vec{c} \cdot \Gamma \vec{d}
= \nu_1(\vec{A}) \cdot ((\nu_2(\vec{Y}) + S\vec{v}) + (\nu_1(\vec{X}) + R\vec{u}) \cdot \nu_2(\vec{B}) + (\nu_1(\vec{X}) + R\vec{u}) \cdot \Gamma (\nu_2(\vec{Y}) + S\vec{v})
= (\nu_1(\vec{A}) \cdot \nu_2(\vec{Y}))(\nu_1(\vec{X}) \cdot \nu_2(\vec{B}))(\nu_1(\vec{X}) \cdot \Gamma \nu_2(\vec{Y}))) + \vec{u} \cdot T \vec{v} - T\vec{u} \cdot \vec{v}
= \nu_T(tT) + \vec{u} \cdot (( \Gamma \nu_2(\vec{B}))) + \Gamma \nu_2(\vec{Y}) + R\vec{u} \cdot \Gamma S\vec{v} + \nu_1(\vec{X}) \cdot \Gamma S\vec{v}
= \nu_T(tT) + \vec{u} \cdot \left[ (R^T \nu_2(\vec{B}) + R^T \Gamma \nu_2(\vec{Y}) + R\vec{u} \cdot \Gamma S\vec{v} + \nu_1(\vec{X}) \cdot \Gamma S\vec{v}) \right] - \vec{v}
= \nu_T(tT) + \vec{u} \cdot \vec{v} + \vec{a} \cdot \vec{b}.$$
Lemma A.5. Let $\vec{X} \in \mathbb{G}_m^1, \vec{y} \in \mathbb{Z}_p^{m'}, R \in \mathbb{Z}_p^{n \times 2}$ and $\vec{s} \in \mathbb{Z}_p^{m'}$ such that they satisfy
\[
\vec{c} = \iota_1(\vec{X}) + R\vec{a} \quad \vec{d}' = \iota'_2(\vec{y}) + \vec{s}\vec{v}_1 \quad \vec{A} \cdot \vec{y} + \vec{b} + \vec{X} \cdot \vec{y} = T_1.
\]
Then we have for all $T \in \mathbb{Z}_p^{1 \times 2}$, $\vec{\pi}$ as constructed in (A.5) and $\vec{\theta}$ as constructed in (A.6) that the verification equation (A.12) holds.

Proof. The proof is analogous to that of Lemma A.4. \qed

Lemma A.6. Let $\vec{x} \in \mathbb{Z}_p^{m'}, \vec{Y} \in \mathbb{G}_m^2, \vec{r} \in \mathbb{Z}_p^{n \times 2}$ and $S \in \mathbb{Z}_p^{m \times 2}$ such that they satisfy
\[
\vec{c}' = \iota'_1(\vec{x}) + \vec{r}\vec{u}_1 \quad \vec{d}' = \iota_2(\vec{Y}) + S\vec{v} \quad \vec{a} \cdot \vec{Y} + \vec{x} \cdot \vec{b} + \vec{x} \cdot \vec{y} = T_2.
\]
Then we have for all $T \in \mathbb{Z}_p^{2 \times 1}$, $\vec{\pi}$ as constructed in (A.7) and $\vec{\theta}$ as constructed in (A.8) that the verification equation (A.13) holds.

Proof. The proof is analogous to that of Lemma A.4. \qed

Lemma A.7. Let $\vec{x}, \vec{r} \in \mathbb{Z}_p^{m'}, \vec{y}, \vec{s} \in \mathbb{Z}_p^{m'}$ such that they satisfy
\[
\vec{c}' = \iota'_1(\vec{x}) + \vec{r}\vec{u}_1 \quad \vec{d}' = \iota'_2(\vec{y}) + \vec{s}\vec{v}_1 \quad \vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{x} \cdot \vec{y} = t.
\]
Then we have for all $T \in \mathbb{Z}_p^{r \times 1}$, $\vec{\pi}$ as constructed in (A.9) and $\vec{\theta}$ as constructed in (A.10) that the verification equation (A.14) holds.

Proof. The proof is analogous to that of Lemma A.4. \qed

Corollary A.8. The composition of the Lemmas A.4, A.5, A.6, and A.7 gives the desired correctness result for the Groth-Sahai proof scheme.

A.2 Re-randomization of Zero-Knowledge Proofs

The construction in this section is inspired by [BCC+09] whereas they use multiplicative notation where we use additive notation. Additionally, we show how to re-randomize all kinds of equations. We proceed as before: we start with the modification of commitments (cf. Section A.2.1), continue with the proof modification (cf. Section A.2.2) followed by the verification of modified proofs (cf. Section A.2.3), and finally, we prove our construction correct (cf. Section A.2.4).

A.2.1 Commitment Modification

Given witnesses $\vec{X}$, $\vec{Y}$, $\vec{x}$, and $\vec{y}$, and the corresponding commitments $\vec{c}$, $\vec{d}$, $\vec{c}'$, and $\vec{d}'$, which satisfy for specific $R$, $S$, $\vec{r}$, and $\vec{s}$
\[
\vec{c} = \iota_1(\vec{X}) + R\vec{a} \quad \vec{d} = \iota_2(\vec{Y}) + S\vec{v} \quad \vec{c}' = \iota'_1(\vec{x}) + \vec{r}\vec{u}_1 \quad \vec{d}' = \iota'_2(\vec{y}) + \vec{s}\vec{v}_1
\]
we re-randomize the commitments in the following way. We pick new randomnesses $R'$, $S'$, $\vec{r}'$, and $\vec{s}'$, and compute the adapted commitments
\[
\vec{c} := \vec{c} + R'\vec{u} \quad \vec{d} := \vec{d} + S'\vec{v} \quad \vec{c}' := \vec{c}' + \vec{r}'\vec{u}_1 \quad \vec{d}' := \vec{d}' + \vec{s}'\vec{v}_1.
\]
With these new commitments we can now proceed by adapting the proofs accordingly.
A.2 Re-randomization of Zero-Knowledge Proofs

A.2.2 Proof Modification

Let \( \vec{c}, \vec{d}, \tilde{c}, \) and \( \tilde{d} \) be the adapted commitments as constructed above. Then we re-randomize the proofs constructed in Section A.1.3 as follows.

**Pairing Product Equations.** We choose a random matrix \( T' \in \mathbb{Z}_p^{2 \times 2} \) and compute the adapted proof pair \((\vec{\pi}', \vec{\theta}')\) as

\[
\vec{\pi}' := \vec{\pi} + R'\top \left[ \iota_2(\vec{b}) + \Gamma \vec{d} + \Gamma S' \vec{v} \right] - T'\top \vec{v},
\]

\[
\vec{\theta}' := \vec{\theta} + S'\top \left[ \iota_1(\vec{A}) + \Gamma \top \vec{c} \right] + T' \vec{u}.
\]

**Multi-Scalar Multiplication Equations in \( G_1 \).** We choose a random matrix \( T' \in \mathbb{Z}_p^{1 \times 2} \) and compute the adapted proof pair \((\vec{\pi}', \vec{\theta}')\) as

\[
\vec{\pi}' := \vec{\pi} + R'\top \left[ \iota_2(\vec{b}) + \Gamma \vec{d} + \Gamma S' \vec{v}_1 \right] - T'\top \vec{v}_1,
\]

\[
\vec{\theta}' := \vec{\theta} + S'\top \left[ \iota_1(\vec{A}) + \Gamma \top \vec{c} \right] + T' \vec{u}_1.
\]

**Multi-Scalar Multiplication Equations in \( G_2 \).** We choose a random matrix \( T' \in \mathbb{Z}_p^{2 \times 1} \) and compute the adapted proof pair \((\pi', \theta')\) as

\[
\pi' := \pi + \vec{r}' \top \left[ \iota_2(\vec{b}) + \Gamma \vec{d} + \Gamma S' \vec{v} \right] - T'\top \vec{v},
\]

\[
\theta' := \theta + S'\top \left[ \iota_1(\vec{a}) + \Gamma \top \vec{c} \right] + T' \vec{u}_1.
\]

**Quadratic Equations.** We choose a random value \( T' \in \mathbb{Z}_p \) and compute the adapted proof pair \((\pi', \theta')\) as

\[
\pi' := \pi + \vec{r}' \top \left[ \iota_2(\vec{b}) + \Gamma \vec{d} + \Gamma S' \vec{v}_1 \right] - T' \vec{v}_1,
\]

\[
\theta' := \theta + S'\top \left[ \iota_1(\vec{a}) + \Gamma \top \vec{c} \right] + T' \vec{u}_1.
\]

A.2.3 Verification

In order to verify that a given proof actually shows the knowledge of a solution to an equation, we apply the Equations (A.11), (A.12), (A.13), and (A.14). We apply them to the adapted commitments and proofs, that is, \( \vec{c}, \vec{d}, \tilde{c}, \) and \( \tilde{d}, \) and \( \vec{\pi}', \pi', \vec{\theta}', \) or \( \theta' \).

A.2.4 Correctness

In this section, we prove the correctness of the above construction for pairing product equations. The proofs for all other equations are analogous.
Lemma A.9. Let $\vec{X}, \vec{Y}, R, S, R'$, and $S'$ such that they satisfy
\[ \vec{c} = \iota_1(\vec{X}) + R\vec{u} + R'\vec{u} \quad \vec{d} = \iota_2(\vec{Y}) + S\vec{v} + S'\vec{v} \quad (\vec{A} \cdot \vec{Y})(\vec{X} \cdot \vec{E})(\vec{X} \cdot \vec{F}) = t_T. \]
Then, for all $T, \vec{\pi}^T$ as constructed in (A.15), and $\vec{\theta}^T$ as constructed in (A.16), the verification equation (A.11) is fulfilled.

Proof. By the commutative property of the linear and bilinear maps we have for the above equation
\[ (\iota_1(\vec{A}) \cdot \iota_2(\vec{Y}))(\iota_1(\vec{X}) \cdot \iota_2(\vec{B}))(\iota_1(\vec{X}) \cdot \Gamma \iota_2(\vec{Y})) = \nu_T(t_T). \]
Let $T' \in \mathbb{Z}_{p^2}^{2 \times 2}$ be a random matrix. Then we have
\[
\begin{align*}
\nu_1(\vec{A}) \cdot \vec{d} + \vec{c} \cdot \nu_2(\vec{B}) + \vec{c} \cdot \Gamma \vec{d} & = \nu_1(\vec{A}) \cdot \vec{d} + S\vec{v} + (\vec{c} + R'\vec{u}) \cdot \nu_2(\vec{B}) + (\vec{c} + R'\vec{u}) \cdot \Gamma (\vec{d} + S'\vec{v}) \\
& = \nu_1(\vec{A}) \cdot \vec{d} + \vec{c} \cdot \nu_2(\vec{B}) + \vec{c} \cdot \Gamma \vec{d} + T'\vec{u} \cdot \vec{v} - \vec{u} \cdot T'^\top \vec{v} \\
& + \nu_1(\vec{A}) \cdot S'\vec{v} + R'\vec{u} \cdot \nu_2(\vec{B}) + R'\vec{u} \cdot \Gamma \vec{d} + \vec{c} \cdot \Gamma S'\vec{v} + R'\vec{u} \cdot \Gamma S'\vec{v} \\
& = \nu_T(t_T) + \vec{u} \cdot \vec{\pi} + \vec{\theta} \cdot \vec{v} + T'\vec{u} \cdot \vec{v} - \vec{u} \cdot T'^\top \vec{v} \\
& + \nu_1(\vec{A}) \cdot S'\vec{v} + R'\vec{u} \cdot \nu_2(\vec{B}) + R'\vec{u} \cdot \Gamma \vec{d} + \vec{c} \cdot \Gamma S'\vec{v} + R'\vec{u} \cdot \Gamma S'\vec{v} \\
& = \nu_T(t_T) + \vec{u} \cdot \left( \vec{\pi} + R'^\top \left[ \iota_2(\vec{B}) + \Gamma \vec{d} + \Gamma S'\vec{v} \right] - T'^\top \vec{v} \right) \quad \substack{\text{=} \vec{\pi}^T \vec{v} \\ \text{=} \vec{\theta}^T \vec{v}} \\
& + \left( \vec{\theta} + S'^\top \left[ \iota_1(\vec{A}) + \Gamma \vec{c} \right] + T'^\top \vec{u} \right) \cdot \vec{v} \quad \substack{\text{=} \vec{\pi}^T \vec{v} \\ \text{=} \vec{\theta}^T \vec{v}} \\
& = \nu_T(t_T) + \vec{u} \cdot \vec{\pi}^T + \vec{\theta} \cdot \vec{v}. \quad \Box
\end{align*}
\]

A.3 Automorphic Signatures for Message Vectors

We show how to sign message vectors. This requires two steps. Firstly, in Section A.3.1 we explain how to sign message pairs. Based on that, in Section A.3.2, we show how to sign message vectors.

A.3.1 Automorphic Tuple Signatures

The idea behind signing message pairs is the following: firstly, we generate an internal key-pair; secondly, we use this key-pair to sign and verify three different combinations of the two messages. More formally, the instantiation of the signature scheme for message pairs is given by the following two algorithms (SetupSS and KeyGen are the same as in Section 2.4.5):
PairSign\((p_{\text{sig}}, m_1, m_2, sk)\). On input \(p_{\text{sig}}\), two messages \(m_1\) and \(m_2\), and a secret signing key \(sk\), this algorithm performs the following steps:

- It uses KeyGen to generate a fresh key-pair \((sk_0, vk_0)\) that is used in the internal signing process.
- It creates the following four signatures using the algorithm Sign:

\[
\begin{align*}
\sigma_1 &= \text{Sign}(p_{\text{sig}}, vk_0, sk) \quad \text{(A.23)} \\
\sigma_2 &= \text{Sign}(p_{\text{sig}}, m_1, sk_0) \quad \text{(A.24)} \\
\sigma_3 &= \text{Sign}(p_{\text{sig}}, m_1 \cdot m_2, sk_0) \quad \text{(A.25)} \\
\sigma_4 &= \text{Sign}(p_{\text{sig}}, m_1 \cdot m_3^2, sk_0) \quad \text{(A.26)}
\end{align*}
\]

- It outputs the tuple \(\sigma = (vk_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4)\).

Equation (A.23) represents a signature issued by the creator of the overall signature \(sk\) on the freshly generated verification key \(vk_0\). Equation (A.24) shows a signature on the first message \(m_1\) using the signing key \(sk_0\) corresponding to the just authenticated verification key \(vk_0\). The same key is used to generate the signatures in Equations (A.25) and (A.26) on two message combinations \(m_1 \cdot m_2\) and \(m_1 \cdot m_3^2\), respectively. Fuchsbauer [Fuc09] proves that these three signatures on linear combinations of \(m_1\) and \(m_2\) are indeed necessary to protect the scheme against attacks. For instance, the exponents of \(m_1\) and \(m_2\), respectively, may neither be linearly dependent nor linearly independent if only two such combinations are signed. Also, finding fitting exponents for three such message combinations is not trivial.

PairVerify\((p_{\text{sig}}, \sigma, m_1, m_2, vk)\). On input \(p_{\text{sig}}\), a signature tuple \(\sigma = (vk_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4)\), two messages \(m_1\) and \(m_2\), and a verification key \(vk\) this algorithm returns 1 if all of the following signature verifications succeed and 0 otherwise

\[
\begin{align*}
\text{VerifySig}(p_{\text{sig}}, \sigma_1, vk_0, vk) & \quad \text{VerifySig}(p_{\text{sig}}, \sigma_2, m_1, vk_0) \\
\text{VerifySig}(p_{\text{sig}}, \sigma_3, m_1 \cdot m_2, vk_0) & \quad \text{VerifySig}(p_{\text{sig}}, \sigma_4, m_1 \cdot m_3^2, vk_0)
\end{align*}
\]

Fuchsbauer [Fuc09] proves that the scheme \((\text{SetupSS}, \text{KeyGen}, \text{PairSign}, \text{PairVerify})\) is secure against chosen-message attacks if the scheme \((\text{SetupSS}, \text{KeyGen}, \text{Sign}, \text{VerifySig})\) is secure. In fact, this is no surprise since the pair scheme depends only on the underlying signature scheme for one message.

### A.3.2 Automorphic Vector Signatures

Based on the transformation above we can now extend our message space to message vectors of length \(n\) lying in the message space \(DH^n\). Again, SetupSS and KeyGen are the same as in Section 2.4.5. We just instantiate two new algorithms for message vectors which use the pair algorithms. Let \(I : \{1, \ldots, |DH|\} \to DH\) be an efficiently computable injection.
VectorSign\((p_{\text{sig}}, \mathbf{m}, \mathbf{sk})\). On input \(p_{\text{sig}}\), messages \(\mathbf{m}\), and a secret signing key \(\mathbf{sk}\) this algorithm performs the following steps:

- It uses KeyGen to generate a fresh key-pair \((\mathbf{sk}_0, \mathbf{vk}_0)\) that is used in the internal signing process.
- It returns the following vector using the algorithm PairSign:

\[
\mathbf{sig} = (\mathbf{vk}_0, \text{PairSign}(p_{\text{sig}}, \mathbf{vk}_0, I(n), \mathbf{sk}), \text{PairSign}(p_{\text{sig}}, m_1, I(1), \mathbf{sk}_0), \ldots, \text{PairSign}(p_{\text{sig}}, m_n, I(n), \mathbf{sk}_0)).
\]

Just like in the pair signature scheme, this algorithm first chooses a fresh key-pair and authenticates the internal verification key \(\mathbf{vk}_0\) together with a value indicating the length of the vector. This length parameter is indispensable to make the scheme secure. Otherwise a forgery compromising the length of the message vector would be easily possible. Due to this fact the length parameter \(I(j)\) is also included in every message signature for \(j\) from 1 to \(n\). This protects against forgeries that shuffle the messages.

VectorVerify\((p_{\text{sig}}, \mathbf{sig}, \mathbf{vk})\). On input \(p_{\text{sig}}\), a vector signature \(\mathbf{sig} = (\mathbf{vk}_0, \mathbf{sig}_0, \mathbf{sig}_1, \ldots, \mathbf{sig}_n)\), and a verification key \(\mathbf{vk}\) this algorithm returns 1 if all of the following verifications succeed and 0 otherwise:

\[
\text{PairVerify}(p_{\text{sig}}, \mathbf{sig}_0, \mathbf{vk}_0, I(n), \mathbf{vk})
\]
\[
\text{PairVerify}(p_{\text{sig}}, \mathbf{sig}_1, m_1, I(1), \mathbf{vk}_0) \quad \cdots \quad \text{PairVerify}(p_{\text{sig}}, \mathbf{sig}_n, m_n, I(n), \mathbf{vk}_0).
\]

Again, Fuchsbauer [Fuc09] proves that this scheme is secure if the underlying pair signature scheme is secure. By transitivity, this scheme is secure if the signature scheme \((\text{SetupSS}, \text{KeyGen}, \text{Sign}, \text{VerifySig})\) is secure.

The ability to sign message vectors is necessary to provide the user with an implementation of the PA-PKA framework that allows for arbitrary predicates. The number of signatures that we have to create for a predicate of length \(n\) is given by the following formula:

\[
\#_{\text{sig}}(n) = 4 \cdot (n + 1). \tag{A.27}
\]

This comes from the fact that we have to create \(n + 1\) pair signatures for \(n\) messages incurring 4 standard signatures each. For instance, the predicate \textit{Prof says Reg(Stud, \ell ec)} requires to sign the vector \((\text{Reg, Stud, \ell ec})\), that is, 3 arguments, resulting in 16 standard signature creations.

Concerning zero-knowledge proofs of signatures, the number of pairing product equations that we have to prove is important rather than the number of signatures. As each proof of a standard signature requires the solution of three pairing product equations, the number of pairing product equations for a vector signature is given by the following formula:

\[
\#_{\text{Equations}}(n) = 3 \cdot 4 \cdot (n + 1) = 12 \cdot (n + 1). \tag{A.28}
\]
Appendix B

Asplada – Language Specification

In order to abstract away cryptography as much as possible, we invented a language to create and manipulate identities and proofs. We call it “language for automated synthesis of privacy-, linearity- and authorization-aware distributed applications” (Asplada). In Section B.1, we explain the syntax and semantics of Asplada, that is, how the syntactic primitives are translated into Java code, and in Section B.2, we present an example to give an impression on its usage convenience.

B.1 Language Description

The complete syntax for this language is depicted in Table B.1. A general integration of Asplada into Java is easy. We start with /*$, continue with the Asplada block, and finish with $*/. An illustration is depicted in Listing B.1. This way we make the mixed Java-Asplada code again valid Java code since the Asplada block is written like a comment in Java.

In Asplada, there are primitives to construct and verify proofs as described in Chapter 2. The idea is to go from a logical description of the proof, that is, the deduction tree produced in the PA-PCA zero-knowledge deduction system in step-wise form from top to bottom. The following paragraphs describe the primitives in more detail.

Predicate. If we read an atomic predicate or a zero-knowledge statement, for instance, exists X. A says P(X), we create the corresponding AtomicPredicate object for this

```
1 //java code
2 /*$
3 meta language code
4 $*/
5 //java code
6 //...
```
Table B.1 The grammar for Asplada.

| Name ::= (A | ... | Z) (a | ... | z | A | ... | Z | 0 | ... | 9 | ')* |
| Newline ::= (\r? \n)+ |
| Predicate ::= Exists? Sentence (/\ Sentence)* |
| Exists ::= exists Name (, Name)* . |
| Sentence ::= (? | @)? Name says Name ( Arguments ) |
| Arguments ::= (? | @)? Name, Arguments |
| Program ::= Statement+ |
| Statement ::= (initfields Newline)? |
| | ((Create | Verify | Split | And | Quantify |
| | fillfields | Keep | Randomize | Instantiate |
| | NewName | NewId | Clear | StoreKey | LoadKey)|Newline)+ |
| Create ::= let Name = Predicate |
| Verify ::= verify Predicate with Name |
| Split ::= split Name as ( Name , Name ) |
| And ::= let Name = Name and Name |
| Quantify ::= let Name = exists Name . Name |
| | let Name = exists Name . Predicate from Name |
| Keep ::= let Name = openinfo for Name in Name |
| Randomize ::= let Name = randomize Name as Predicate |
| Instantiate ::= let Name = instantiate Name with Name as Predicate |
| NewName ::= newName Name | newName Name = Name |
| | init newName Name | fill newName Name with Name |
| NewId ::= newId Name | newId Name = Name |
| | init newId Name | fill newId Name with Name |
| Clear ::= clear Name |
| StoreKey ::= store Name at Name |
| LoadKey ::= load Name from Name |
| | load Name from Name as Name |

predicate with the issuer A, the predicate name P, and the argument X; A is of type IdKey and X is of type Variable.

**NewName and NewId.** These two primitives create new names and identities, respectively. The user can decide whether she wants to fill the content directly or later on. This is very helpful to program efficiently in object oriented programming languages like Java. There are four types of usage which we detail below:
newName A [= B] / newId A [= B]. We create a new instance of class Name or IdKey, respectively. This means that we take the string representation of A, which is essentially 'A', and forward this as an argument to the constructor. In case of an identity, we additionally provide the PKI with a new key pair for A and the constructor uses this key pair. If, additionally, the optional = B is written, we use for the name the content of the String variable B rather than the string representation of A. This flexibilizes instantiating class instances not only with fixed values but also with, for instance, inputs from the network.

init newName A / init newId A. This primitive defines a new name or identity but not already instantiates it. Such a statement is typically written at those positions where global variables are placed.

fill newName A with B / fill newId A with B. This is the followup of the above initialization primitive. It is typically written in the constructor to fill the previously created name or identity with the String stored in B.

Clear. Our implementation of the compiler uses a list of arguments to keep track of already known names and identities. For instance, if A is already defined, there is no need to create it twice. Indeed, there is most often not the intention to duplicate it at all. This works fine as far as there are no function definitions or separate code blocks. Consequently, if there are such blocks, a programmer is willing to define the same name several times without meaning the same name in a global context. The Clear statement helps to manage the recreation of already used name bindings. For instance, the statement clear A removes A from the global argument list and A can be reused in another context.

StoreKey and LoadKey. These primitives provide features to load keys from disk and to store them on disk. store A at B stores the keys of the identity A on disk at the path given in the String variable B.

load A from B [as C] offers the possibility to load formerly stored keys for principals from disk. After execution, the variable A is the identity instantiated with the keys (signing and verification key) stored at B. If the optional as C is also present, the name of A is the string stored in the variable C.

These functions are useful to keep the state of a program. For instance, most systems are not made for permanent login. Hence, when a user logs out, her keys are stored on disk (or on a secure device).

Create. With this primitive the user can create atomic predicates. For instance, a statement such as let P = A says Good(B) creates the corresponding atomic predicate A says Good(B) and stores it in P. More specifically, the statement is translated into code that generates a new instance of ProofObject incorporating a zero-knowledge proof of knowledge of the signature sign(Good(B))_{sk,A}. The signature must indeed be created before we can prove the knowledge thereof. Note that only the principal in possession of
Asplada — Language Specification

Asplada, which is in general only A itself, can construct the signature. P further refers to the constructed proof.

Verify. Given a statement and a ProofObject, this primitive checks that they match each other and that the ProofObject indeed verifies. For example, the compiled version of the statement verify A says Good(B) with P first checks if P verifies. If this verification step succeeds, it proceeds by checking if the predicate to check, in our example A says Good(B), is the one proven by P: to this end we check if both, the given statement and the statement incorporated in P, are equal up to alpha-renaming of quantified variables. If this is the case the execution proceeds. In any case of failure we throw an exception which directly terminates the execution.

Split. This statement separates a conjunction apart. An instance of this statement has the form split P as (Q1, Q2). At first, note that P has to be a ProofObject instance that proves a conjunction of the form \( \bigwedge_{i=1}^{n} \text{pred}_i \). P is then split into ProofObject instances Q1 and Q2 such that Q1 proves the conjunction \( \bigwedge_{i=1}^{n-1} \text{pred}_i \) and Q2 proves the predicate \( \text{pred}_n \).

And. This statement creates a conjunction of two given ProofObjects. A statement instance that we expect is of the form let P = Q1 and Q2. The compiled code performs a conjunction of both (possibly also conjunctions) proofs, that is, if Q1 proves a conjunction \( \bigwedge_{i=1}^{n} \text{pred}_i \) and Q2 proves a conjunction \( \bigwedge_{i=1}^{m} \text{pred}_i \) then P proves the conjunction \( \bigwedge_{i=1}^{n} \text{pred}_i \land \bigwedge_{i=1}^{m} \text{pred}_i \) in this order. As P refers to an instance of ProofObject it can be used as such.

Quantify. There are two ways to quantify names or identities in a proof. The first one quantifies all occurrences of the specified name or identity in the statement. The statement is of the form let P = exists A. Q. This quantifies all occurrences of A in Q and copies the result into P. As usual, P can be used further as an instance of ProofObject.

The second way to quantify is selective quantification. In this version we do not specify an identity to quantify but we specify directly the variable which afterwards points to the positions to quantify. We expect statements of the following form: let P = exists X. X says Good(A) from Q. For instance, assume that Q has the form A says Good(A) which does not make sense in real applications, but for the sake of example. The primitive then compares the given statement with the statement proven by Q and quantifies the found variable at all specified positions.

We will throw an exception in the case that the input is not a well-formed statement. For instance, if the statement proven by Q is A says Good(A,B) and the given statement is exists X. A says Good(X,X). This is not well-formed since two identities that are not linked so far would be linked afterwards.

Keep. This statement stores the opening information of a given identity for later usage. We expect a statement of the form let OI = openinfo for A in P. Keep is strongly
connected to existential instantiation, that is, in order to perform an existential instantiation one has to provide the opening information fetched by \texttt{Keep}. In the above example, it stores the opening information for the identity \texttt{A} from the proof \texttt{P} in the variable \texttt{OI}. This opening information contains the identity’s witness and randomness. Note that in order to keep opening information one needs a proof in which the identity from which we want to keep the information is not quantified.

\textbf{Instantiate.} This statement re-instantiates an identity that is quantified in the given proof, provided the corresponding opening information. We expect a statement of the form - by considering the example from above - \texttt{let P = instantiate Q with OI as A says Good(A,B)} where \texttt{Q} proves the predicate \texttt{exists X. A says Good(X,B)}. Upon execution \texttt{P} contains the proof for the predicate where \texttt{A} is instantiated back with the use of \texttt{OI}, the opening information formerly stored using \texttt{Keep}. In case the given statement does not match the statement proven in \texttt{Q}, up to existential quantification and alpha-renaming of variables, an exception is thrown. However, if the opening information \texttt{OI} does not fit the identity that we want to instantiate then nothing happens since the instantiation does not succeed.

\textbf{Randomize.} This statement re-randomizes a selection of equal variables in a given proof. We expect a statement of the form, again taking the above example, \texttt{let P = randomize Q as exists X'. A says Good(X',B)} Here both, \texttt{P} and \texttt{Q}, are instances of \texttt{ProofObject} and \texttt{Q} should prove the statement \texttt{exists X. A says Good(X,B)}. We select the variables which we intend to re-randomize by marking them with an apostrophe. One can also re-randomize not quantified identities but that does not make sense since nothing will be hidden to observers: the commitment and the opening information will change, but both is revealed anyway. Note that only the apostrophized variables are really re-randomized, so marking them correctly is crucial. One should also keep in mind that a variable, once re-randomized, can never be instantiated back again. This is because the randomness of the kept opening information does not correspond to the one which is used in the re-randomized version. There is also no way to extract the changed randomness from the proof since it is a secret.

\section*{B.2 Examples}

We illustrate the convenience of \texttt{Asplada} by an example. We implement the friend-of-friend protocol presented in Section 5.1. We choose this example since it uses almost every feature that \texttt{Asplada} provides. We simplify the example to not use the relation predicate but an easier friend predicate that directly represents the relation.

\textbf{Example B.1.} We briefly recall the protocol specified in PA-PCA. Suppose \texttt{Bob} is a friend of \texttt{Alice} and \texttt{Alice} is also a friend of \texttt{Charlie}. This means in PA-PCA that we have two statements; one at \texttt{Bob}'s disposal and one at \texttt{Alice}'s disposal stating the friendship, namely, \texttt{Alice says Friend(Bob)} and \texttt{Charlie says Friend(Alice)}. Now \texttt{Bob} wants to be also
Figure B.1 The protocol to establish an indirect friendship without the need of interaction for Charlie.  

Charlie says Friend(Alice)  
Alice says Friend(Bob)  
∃x. Alice says Friend(x) ∧ x says Request(Charlie)  
∃x, y. Charlie says Friend(x) ∧ x says Friend(y)  
∃x. Charlie says Friend(x) ∧ x says Friend(Bob)  

a friend of with Charlie and the protocol is able to establish this friendship without the interaction of Charlie. The protocol is illustrated in Figure B.1.

We show the statements that we need to construct in order to implement the protocol in our API. These statements are the following:

Charlie says Friend(Alice)  
Alice says Friend(Bob)  
∃x. Alice says Friend(x) ∧ x says Request(Charlie)  
∃x, y. Charlie says Friend(x) ∧ x says Friend(y)  
∃x. Charlie says Friend(x) ∧ x says Friend(Bob)

We implement these statements in Asplada with the code in Listing B.2. There, we only show the construction of the statements abstracting from the process itself. The statements are created in the same order as above. In order to indicate when a step in the process is finished, we put one line of space after each such step.
Listing B.2: The implementation of the friend-of-friend protocol.

```plaintext
newId Alice
newId Bob
newId Charlie

let Fca = Charlie says Friend(Alice)
let Fab = Alice says Friend(Bob)

let ReqFirst = Bob says Request(Charlie)
let ReqSecond = Fab and ReqFirst
let OI = openinfo for Bob in ReqFirst
let Request = exists Bob. ReqSecond

verify exists X. Alice says Friend(X) \ X says Request(Charlie) with Request
split Request as (Fr,Re)
let AnswerFirst = Fca and Fr
let AnswerTwo = exists Alice. AnswerFirst
let Answer = randomize AnswerTwo as exists X',Y. Charlie says Friend(X') \ X' says Friend(Y)

verify exists X,Y. Charlie says Friend(X) \ X says Friend(Y) with Answer
let Fcb = instantiate Answer with OI as exists X. Charlie says Friend(X) \ X says Friend(Bob)

$*/
```