RUHR-UNIVERSITÄT BOCHUM

RUHR-UNIVERSITÄT BOCHUM

Security of Instant Messaging: Implementation of Asynchronous Ratcheted Key Exchange

Marco Smeets

Master's Thesis – May 6, 2019.
Chair for Network and Data Security.

Supervisor:	Prof. Dr. Jörg Schwenk
Advisor:	M. Sc. Robert Merget
Advisor:	M. Sc. Paul Rösler

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

Asynchronous Ratcheted Key Exchange (ARKE) is a further development of Ratcheted Key Exchange (RKE), a cryptographic technique used in instant messaging applications like Signal or WhatsApp. ARKE tries to resolve limitations of RKE by developing alternative security notions that not only consider *unidirectional* RKE, but also *bidirectional* RKE. So, the communication follows a human-like structure where participants only send messages if they feel like they want to contribute. The messages may also cross while transferring over the communication channel, thus, the communication is *asynchronous*. In this thesis we implement the Bidirectionally Ratcheted Key Exchange (BRKE) construction proposed by Poettering and Rösler (CRYPTO 2018) in Java. The construction uses generic primitives like KEMs, one-time signatures, and random oracles. Furthermore, the construction uses a modified version of a KEM, the key-updateable Key Encapsulation Mechanism (kuKEM), which is a Hierarchical Identity-Based Encryption (HIBE)-like component. We split the implementation into two parts: a generic BRKE implementation that does not rely on actual primitive implementations, and a BRKE instantiation in which we instantiate the generic BRKE construction with actual primitives. We further discuss possible choices for the different primitives and provide overviews for suitable algorithms for each primitive.
We implement the kuKEM by using the pairing-based Lewko-Waters HIBE. The HIBE is implemented in C++ and uses the Relic library, a state-of-the-art pairing library for C/C++, for pairing computations. We evaluate the BRKE implementation with four different pairing-friendly curves for two security levels. The evaluation shows that the kuKEM and, thus, the HIBE has the most influence on the performance of the BRKE implementation. We simulate four different communication sequences and show that the flow of communication strongly determines the performance of the BRKE implementation.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Date   author

Contents

1 Introduction

In this chapter we describe the topic and the structure of this thesis. In the first section the motivation for this thesis is explained. After that, we mention some related work and explain the contribution, which is achieved by our work.

1.1 Motivation

Nowadays more and more people are using Instant Messaging apps on their mobile devices. In 2017 the instant messenger app Whats App announced they have one billion daily active users with 55 billion messages being sent per day [2]. If we have a look at statistics that try to estimate the number of messaging app users for the coming years, we can easily see that the numbers are almost certainly rising every year [1]. With such a high number of users and messages sent per day, it becomes clear that security should not be neglected in an instant messaging context.
The Signal messaging protocol [79] is a security protocol for instant messaging which provides end-to-end encryption. It uses a technique called *ratcheting* to establish updated session keys with every message [64]. The Signal protocol or a modified version of it is used by several messenger applications like WhatsApp [89] or the Facebook Messenger [34]. Like a zip tie, a ratchet is constructed in a way that it only moves forward. A simple example of such a construction could be a "hash chain". We have an initial symmetric key, and every time the key is used a one-way function is applied to derive a new key from the old key. This ensures that a key never is used more than once and that an attacker cannot compute old session keys. The security of the Signal Messaging Protocol was recently analyzed by Cohn-Gordon et al. [29]. There is also work by Bellare et al. [16] that formally analyzes the security of *RKE*. They also provide syntax and security definitions for ratcheting. However, the security model established by Bellare et al. [16] is quite restricted because, for example, it only considers unidirectional communication. Unidirectional communication describes the communication in which only one user sends messages to another user, so for two users Alice and Bob which communicate by exchanging messages over a network (like the Internet), we only assume communication in the Alice-to-Bob direction. Furthermore, Bellare et al. [16] only assume that Alice's state can be exposed. Problematic with this assumption is the fact that it is sometimes impossible to defend against state exposure because the state information (which includes session keys, for example) kept in the memory might at one point

be swapped to disk and then can be stolen from there. For this reason, Poettering and Rösler [77] developed alternative security definitions to account for a more realistic communication environment. They consider bidirectional communication (so communication is possible in the Alice-to-Bob and the Bob-to-Alice direction) and consider asynchronous communication. Asynchronous communication describes a communication which represents a human-like conversation. So both Alice and Bob contribute to the conversation when they feel like it. Furthermore, messages may cross while transferring over the network.
So, Poettering and Rösler [77] introduce secure constructions for RKE in which two communication partners can establish updated session keys in a unidirectional case and a bidirectional case. While the construction for the unidirectional case called *Unidirectionally Ratcheted Key Exchange (URKE)* is similar to previous proposed RKE definitions, the construction of *BRKE* provides a primitive which is more suited for a typical instant messaging context. In BRKE both participants of a conversation can establish fresh session keys independently of each other. This means BRKE is applicable to be used for natural instant messaging conversations in which messages are sent asynchronously. The goal of this thesis is to develop a *theoretically* secure implementation of BRKE. *Theoretically* because the main goal is finding primitives that achieve the security requirements that are set by Poettering and Rösler [77] and then use these primitives to implement the BRKE construction. We do not analyze the implementation against side-channel attacks like *timing attacks*, for example. Furthermore, the performance of the individual primitives is initially no concern.
In this thesis we implement the BRKE construction in two parts: a generic BRKE implementation that does not rely on actual primitive implementations, and a BRKE instantiation in which we instantiate the generic BRKE construction with actual primitives. By using this approach the implementation is as flexible as possible, and primitives that become insecure can easily be interchanged. With this implementation, we can perform first performance tests to see how the BRKE construction performs compared to other ratcheting based schemes.

1.2 Related Work

Since the work by Poettering and Rösler [77] was released recently, we could not find much work related to the actual implementation of ARKE. However, we could find a GitHub repository, which includes a GO implementation of BRKE and several other ratcheting protocols. This implementation suits as a benchmark and is used to compare the performance aspects of different protocols. However, it seems that the primitives used in this implementation do not fulfill the security requirements set by Poettering and Rösler [77]. For example, the Gentry-Silverberg HIBE, which is used in this implementation, does not use the CCA transformation described by Gentry and Silverberg [37] and, thus, is only CPA secure. Durak and Vaudenay [32] compare the complexity of different ratcheting protocols. They include the BRKE construction of Poettering and Rösler [77], but unfortunately do not describe how they estimate the complexity.

1.3 Contribution

We implement the BRKE construction proposed by Poettering and Rösler [77] in Java. We split the implementation into two parts. First, we develop an implementation of the BRKE construction that only uses interfaces and is completely generic. If instantiated with working primitives, the BRKE protocol is executed and can establish session keys. We apply some formal changes to the construction so that we can deploy the BRKE implementation in a real-world instant messaging application. In the second part, we implement the primitives required for the BRKE construction. For this, we use primitives that achieve the security requirements that are set by Poettering and Rösler [77]. With these primitives we can execute the BRKE protocol and conclude a first performance analysis of the BRKE construction. We also provide an overview of currently recommended cryptographic primitives that achieve the specific security requirements. We can consult this overview if we want to compare different primitives in their performance. For the instantiation, we implement the kuKEM, a special kind of Key Encapsulation Mechanism (KEM) that was proposed by Poettering and Rösler [77]. The kuKEM internally uses a CCA secure HIBE which we implement in C++.

1.4 Organization of this Thesis

In Chapter 2 we give background information to all topics we address in this thesis. We start by giving some mathematical background that is required to describe and understand the different primitives. Then we describe the cryptographic primitives and security notions we require for the BRKE construction. In the end, we describe *ratcheting* and the constructions proposed by Poettering and Rösler [77]. In Chapter 3 we discuss the choices for the algorithms we use to implement the BRKE instantiation. We elaborate possible choices for every required primitive and then describe our choice. We always provide an overview of algorithms that are recommended and achieve the security requirements set by Poettering and Rösler [77]. In Chapter 4 we describe the generic part of the implementation. We explain our general approach in implementing the BRKE construction and describe the reasons for applying formal changes. After that we describe the instantiation the BRKE construction in Chapter 5. We describe the general ideas for the different algorithm implementations and decisions we make throughout the development. We evaluate the performance of the BRKE construction in Chapter 6. We consider two security levels, 100 bit and 128 bit, and evaluate the performance of the BRKE construction with the respective primitives. The reason for choosing two security levels is the performance influence of the elliptic curve that is used for the pairings in the HIBE. We conclude the thesis in Chapter 7.

2 Background

In this chapter, we describe the cryptographic primitives, which are used to implement the constructions proposed by Poettering and Rösler [77], and specify the notations used throughout this thesis. We mostly give only short summaries of the primitives, but we provide references that can be consulted by interested readers to deepen their knowledge further.

2.1 Notation

We let $S = \{a_0, ..., a_n\}$ for $n \in \mathbb{Z}$ denote a set of n elements. If $a \leqslant b$ we let $I = [\,a, .., b]$ denote the set $\{a, ..., b\}$. For this interval I, we write $I^{\vdash}$ for a and $I^{\dashv}$ for b. Furthermore, we let $a \circ b$ denote an operation on a and b that connects both values in a for now undefined way. We denote the boolean values *true* and *false* with 1 and 0, respectively.
For an (*deterministic* or *randomized*) algorithm $\mathcal{A}$ we let $\mathcal{A}(x)$ denote the invocation of $\mathcal{A}$ on input x. If we use the randomized algorithm, we write $y \Leftarrow \mathcal{A}(x)$ for the case that $\mathcal{A}$ returns y on invocation with x. For a deterministic algorithm, we write $y \leftarrow \mathcal{A}(x)$. If we define a variable or algorithm to have a specific meaning we use the operator ":=".
Security Games: When describing the cryptographic primitives, we define the security requirements for those primitives on some advantage for an adversary $\mathcal{A}$ which is bound to some probability. In a security game, we have an adversary $\mathcal{A}$ that has to accomplish a set goal. If the adversary achieves the goal, we say that the adversary wins the game. We treat the adversary as a black box. We give $\mathcal{A}$ a specific input and $\mathcal{A}$ returns a specific output. $\mathcal{A}$ sometimes has access to specific *oracles*. $\mathcal{A}$ can make an arbitrary number of queries to the oracle, and the oracle answers these queries. After $\mathcal{A}$ returned his output, thus, his solution to the game, the game outputs 1 if the adversary wins the game, and 0 otherwise.
In this thesis, we consider two types of games. *Indistinguishability* games where the adversary has to distinguish between two inputs and games where the adversary has to achieve a specific goal. In indistinguishability games, the adversary often has to return 0 or 1 depending on the input, so we always have to subtract $\frac{1}{2}$ of the winning probability of the adversary, because the adversary can guess. In the other games, we are concerned with the plain winning probability of the adversary. So, in other words: "What is the probability that the adversary can achieve the set goal?". To

keep the notions consistent with the notions used by Poettering and Rösler [77] we define $\mathcal{A}$'s success probability as an advantage. So in other words: "The advantage of an adversary $\mathcal{A}$ is the probability that $\mathcal{A}$ wins a specific game".
Typically, the game simulates only one instance of a given algorithm, but Poettering and Rösler [77] use a multi-instance version of the standard security notions. In this approach, the adversary is allowed to create new instances, which run independently and have their own uniformly random chosen secrets and expose them to learn those secrets.
We assume a scheme to be secure if for every adversary $\mathcal{A}$ carrying out an attack of some specific type, the probability that $\mathcal{A}$ succeeds in this attack is negligible [51]. Now it is left to describe how we prove that an adversary has a negligible probability of success. For this we use *reductions* [51]. When using reductions, the strategy is to assume some problem is hard to solve, and then prove that a given construction is secure given this assumption [51]. This is done by *reduction*. Generally, a reduction is showing "how to convert any efficient adversary $\mathcal{A}$ that succeeds in breaking the construction with non-negligible probability into an efficient algorithm $\mathcal{A}'$ that succeeds in solving the problem that was assumed to be hard" [51]. So, in other words, we conclude something like this: "Given a scheme S and a hard problem P. We assume S to be secure under the assumption that P is a hard problem because if there exists an adversary $\mathcal{A}$ that would break S, we could solve problem P."

2.2 Mathematical Background

In this section, we describe some mathematical backgrounds that we need for the cryptographic primitives and concepts we use and apply throughout this thesis. Just as in the other background sections we try to keep the explanations simple and reduced to the minimum while providing additional references for interested readers.

2.2.1 Finite Fields

Cryptography very often relies on or utilizes finite structures. One can find excellent explanations and descriptions in the literature [51, 52, 76], so we only describe the basic concepts we utilize in this thesis.
We start with the definition of a mathematical group [52, 76], the most basic algebraic concept.

Definition 2.1 (Group) *A (finite) group $(G, \circ)$ is a (finite) set of elements G with a group operation $\circ$ which combines two elements of G. The group operation $\circ$ satisfies the following conditions:*

Closure *For all $a, b \in G$, $a \circ b \in G$.*

Associativity *For all $a, b, c \in G$, $(a \circ b) \circ c = a \circ (b \circ c)$*

Neutral Element *There is an element e, such that $a \circ e = e \circ a = a$, $\forall a \in G$*

Inverse Element *For each $a \in G$ there exists an element $a^{-1} \in G$, such that $a \circ a^{-1} = a^{-1} \circ a = e$.*

A group is called abelian or commutative group, if $a \circ b = b \circ a$, $\forall a \in G$.

The next algebraic concept we describe is the (finite) Ring [52]:

Definition 2.2 (Ring) *A (finite) ring $(R, +, \times)$ is a (finite) set of elements R with two group operations denoted $+$ (addition) and $\times$ (multiplication). The group operations satisfy the following axioms:*

1. *$(R, +)$ is an abelian group where the neutral element is denoted as 0. And every element $a \in R$ has an inverse element denoted $-a$.*
2. *The operation $\times$ is associative. That means: $\forall a, b, c \in R$, $(a \times b) \times c = a \times (b \times c)$.*
3. *There is a multiplicative identity denoted 1.*
4. ***Distributivity** $\forall a, b, c \in R$, $a \times (b + c) = (a \times b) + (a \times c)$ and $(a + b) \times c = (a \times c) + (b \times c)$.*

The ring is called commutative if $a \times b = b \times c$, $\forall a, b \in R$.

Note that the notations $+$ and $\times$ not always describe the basic addition and multiplication we know from regular integers. That is especially important when working with pairings on elliptic curves, as we see in later sections.
Now the only basic arithmetic operation that is missing in the described structures is the division. For this reason, we now describe the (finite) field [52, 76]:

Definition 2.3 (Field) *A (finite) field $(F, +, \times)$ is a (finite) set of elements F with two group operations denoted $+$ (addition) and $\times$ (multiplication). The group operations satisfy the following three axioms:*

1. *$(F, +)$ is an abelian group where the neutral element is denoted as 0, and every element $a \in F$ has an inverse element denoted $-a$.*

2. *$(F \setminus \{0\}, \times)$ is an abelian group where the neutral element is denoted as 1, and every element $a \in F$ has an inverse element denoted a^{-1}.*

3. ***Distributivity** $\forall a, b, c \in F$, $a \times (b + c) = (a \times b) + (a \times c)$ and $(a + b) \times c = (a \times c) + (b \times c)$.*

In other words, we can say that a (finite) field is a (finite) commutative ring in which all non-zero elements have a multiplicative inverse [52]. The *order* of the group $|G|$ denotes the number of elements in the group. Furthermore, every element in a group has an order [52, 76]:

Definition 2.4 (Order of an element) *The order ord(a) of an element $a \in G$ is the smallest positive integer k such that*

$$a^k = \underbrace{a \circ a \circ ... \circ a}_{k\ times} = 1$$

, where 1 denotes the identity element of G.

If an element of a group has a maximum order ord(a)= $|G|$ then the group is called *cyclic*. Elements with maximum order are called *generators* [52, 76]. Note that these definitions also hold for fields, since they consist of groups (Def. 2.3).
The most important groups for cryptographic applications and the ones we are mostly interested in are the ones constructed by a prime number [76]. Let us have a look at a finite field of prime order p, for example, which is usually denoted by $GF(p) = (\mathbb{Z}_p, +, \times)$. This field consists of the elements $\{0, 1, ..., p-1\}$, and addition and multiplication are performed modulo p [52]. In this finite field we have two groups $(\mathbb{Z}_p, +)$ and $(\mathbb{Z}_p^*, \times)$. Since p is prime, $(\mathbb{Z}_p^*, \times)$ is an abelian finite cyclic group [76]. Note that $\mathbb{Z}_p^*$ has the same meaning as $\mathbb{Z}_p \setminus \{0\}$. These multiplicative groups of prime fields are prevalent to build discrete logarithm cryptosystems and have interesting properties for cryptographic applications [76].

2.2.2 Discrete Logarithm Problem

As mentioned in Section 2.1 formal security proofs often rely on specific assumptions. An example of those assumptions is the hardness to calculate a logarithm in a finite cyclic group. This assumption is based on the Discrete Logarithm Problem (DLP) [52, 68]:

Definition 2.5 (Discrete Logarithm Problem (DLP)) *Given a prime p, a generator α of $\mathbb{Z}_p$, and an element $\beta \in \mathbb{Z}_p^*$, find the integer x such that*

$$\alpha^x \equiv \beta (mod\ p)$$

.

Of course, this problem is not hard if the prime is not very large, but with well-chosen parameters computing a discrete logarithm modulo p is very hard. In contrast to that, exponentiation modulo p is computationally easier, so this problem forms a one-way function [76]. One-way functions help us build so-called chameleon hash functions, which are described in a later section.

2.2.3 Cryptographic Assumptions

In the last section, we described the Discrete Logarithm Problem (DLP). Since formal security proofs mostly rely on specific assumptions to hold, we now want to describe three assumptions we use in this thesis. The first assumption we describe is the *discrete logarithm* assumption [81] and is essentially only a reformulation of the DLP:

Definition 2.6 (Discrete Logarithm Assumption) *Given a group G of prime order p, a generator g of G, and g^x for some $x \in \mathbb{Z}_p$. The discrete logarithm assumption says it is hard to calculate x.*

So we could say that if the DLP problem is hard in a specific group G the *discrete logarithm* assumption holds for this group.
The next assumption we describe is the *Computational Diffie-Hellman (CDH)* assumption [81, 87]. As the name indicates the assumption is based around the Diffie-Hellmann protocol [31].

Definition 2.7 (Computational Diffie-Hellman (CDH) assumption) *Given a group G of prime order p, a generator g of G, g^x for some $x \in \mathbb{Z}_p$, and g^y for some $y \in \mathbb{Z}_p$. The CDH assumption says is is hard to calculate g^{xy}.*

The last assumption we use in this thesis is the *Decisional Diffie-Hellman (DDH)* assumption [81, 87]:

Definition 2.8 (Decisional Diffie-Hellman (DDH) assumption) *Given a group G of prime order p, a generator g of G, g^x for some $x \in \mathbb{Z}_p$, g^y for some $y \in \mathbb{Z}_p$, and g^z for some $z \in \mathbb{Z}_p$. The DDH assumption says is is hard to determine if $g^{xy} = g^z$.*

There are many more assumptions (and mathematical problems), which can be used to prove that a specific protocol or algorithm is secure. We need these three assumptions to construct a secure One-Time signature, but indirectly we rely on many more assumptions, which we do not describe in this thesis, but provide references.
Note that we can translate those assumptions into security games. Examples for game-based definitions of those assumptions are described by Katz [51].

2.2.4 Elliptic Curves

An elliptic curve is an abelian group, where the members of the set are defined by points (x, y) that are a solution to a special polynomial equation. For cryptographic uses, we are mostly interested in curves over a finite field. For this, the most popular choice is prime fields $GF(p)$, for a prime p [76].

Definition 2.9 (Elliptic Curve) *An elliptic curve E over $\mathbb{Z}_p$,$p > 3$ is defined by an equation of the form*

$$y^2 = x^2 + ax + b$$

where $a, b \in \mathbb{Z}_p$ satisfy the condition

$$4a^3 + 27b^2 \neq 0 \ mod \ p$$

The group operation in elliptic curves is addition, and the identity element, the point at infinity, is denoted $\mathcal{O}$.
In practice, elliptic curves are used as a primitive to build public key cryptosystems. One of the most well-known problems used in those cryptosystems is the *Elliptic Curve Discrete Logarithm Problem (ECDLP)* [76]:

Definition 2.10 (Elliptic Curve Discrete Logarithm Problem (ECDLP)) *Given an elliptic curve E, and two points $P, Q \in E$. The elliptic curve discrete logarithm problem is finding the smallest integer k, such that*

$$\underbrace{P + P + ... + P}_{k\ times} = k * P = T$$

This definition could easily be transformed into an assumption or game-based definition as described in Section 2.2.3, as well. We do not directly use this definition in this thesis, but we want to show an important characteristic when working with elliptic curves. Usually, DL-based cryptography is denoted in multiplicative notation, but since the group operation in elliptic curves is addition we cannot, e.g., multiply an elliptic curve point. However, as we can see in Definition 2.10, we can still use multiplicative notation to describe operations on elliptic curves. If we use pairings, operations on elliptic curves are often denoted in multiplicative notation despite describing additive operations. In Table 2.11 we can see a comparison between elliptic curve cryptography operations and the corresponding discrete logarithm-based operations. We use the notation as it is usual in the literature. So we use multiplicative notation, but use the corresponding additive operation on elliptic curves.

Elliptic Curve Cryptography	DL-Based Cryptography
Point Addition	Multiplication
Point Doubling	Squaring
Point Multiplication	Exponentiation
Point Subtraction	Division

Table 2.11: Comparison between ECC and DL-Based Cryptography [42]

2.2.5 Pairings

In this thesis we only describe the general view on bilinear pairings, so we omit the underlying mathematical foundations because they are beyond the scope of this thesis. A bilinear pairing can be defined as follows [63, 69]:

Definition 2.12 (Symmetric Pairing) *Let $\mathbb{G}$ and $\mathbb{G}_T$ be two cyclic groups of prime order p and let g be a generator of $\mathbb{G}$. A bilinear pairing or bilinear map e is an efficiently computable function*

$$e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$$

such that

1. *(Nondegeneracy)* $e(g, g) \neq 1$
2. *(Bilinearity)* $e(g^a, g^b) = e(g, g)^{ab}, \forall\ a, b \in \mathbb{Z}_p$

Typically, $\mathbb{G}$ is an additive group with identity $\mathcal{O}$ and $\mathbb{G}_T$ is a multiplicative group with identity 1 [69]. So $\mathbb{G}$ could be an elliptic curve and the target group $\mathbb{G}_T$ a finite field. If we only have one source group, namely $\mathbb{G}$, which is mapped to the target group we call the pairing *symmetric*.
There is also another kind of pairing called *asymmetric pairing* which is more practical, because it is easier to find families of elliptic curves that satisfy the requirements of the following definition [63]:

Definition 2.13 (Asymmetric Pairing) *Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be cyclic groups of prime order p. Let g_1 be a generator of $\mathbb{G}_1$ and g_2 be a generator of $\mathbb{G}_2$. A bilinear pairing or bilinear map e is an efficiently computable function*

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

such that

1. *(Nondegeneracy)* $e(g_1, g_2) \neq 1$
2. *(Bilinearity)* $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}, \forall\ a, b \in \mathbb{Z}_p$

Furthermore, if we use the pairings from Definition 2.13, we can set different requirements on the groups $\mathbb{G}_1$ and $\mathbb{G}_2$. For example, we could require that the CDH assumption holds in $\mathbb{G}_1$, but not in $\mathbb{G}_2$.
There are also pairings on elliptic curves with a composite group order [36, 63], but they tend to have worse performance over regular elliptic curves, so they are not that interesting for practical use, either [36].

2.2.6 Dual Pairing Vector Spaces (DPVS)

We now have a look at the concept of Dual Pairing Vector Spaces (DPVS) proposed by Okamoto and Takashima [74, 75]. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be groups of prime order p with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Instead of only referring to individual elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ we now also consider vectors of group elements. For a n-dimensional vector $\vec{v} = \{v_1, v_2, ..., v_n\} \in \mathbb{Z}_p^n$ and $g \in \mathbb{G}_1$ (respectively $\mathbb{G}_2$), we write $g^{\vec{v}}$ to denote a n-tuple of elements [58, 74, 75]:

$$g^{\vec{v}} := (g^{v_1}, g^{v_2}, ..., g^{v_n})$$

We can also perform scalar multiplication and vector addition in the exponent [58]. For any $a \in \mathbb{Z}_p$ and $\vec{v}, \vec{w} \in \mathbb{Z}_p^n$, we have:

$$g^{a\vec{v}} := (g^{av_1}, g^{av_2}, ..., g^{av_n}), g^{\vec{v}+\vec{w}} := (g^{v_1+w_1}, g^{v_2+w_2}, ..., g^{v_n+w_n})$$

And e_n is the product of the component-wise pairings [58]:

$$e_n(g^{\vec{v}}, g^{\vec{w}}) := \prod_{i=1}^{n} e(g^{v_i}, g^{w_i}) = e(g, g)^{\vec{v}*\vec{w}}$$

The dot product is taken modulo p.
Now let g_i be a generator of $\mathbb{G}_i$. We have a look at two vector spaces $\mathbb{V} := \overbrace{\mathbb{G}_1 \times ... \times \mathbb{G}_1}^{n}$ and $\mathbb{V}^* := \overbrace{\mathbb{G}_2 \times ... \times \mathbb{G}_2}^{n}$ whose elements consist of n-dimensional vectors $\boldsymbol{b} = g_1^{\vec{b}} = (g_1^{b_1}, g_1^{b_2}, ..., g_1^{b_n})$ and $\boldsymbol{b^*} = g_2^{\vec{b}^*} = (g_2^{b_1^*}, g_2^{b_2^*}, ..., g_2^{b_n^*})$, respectively. Let

$\mathbb{B} := (\overrightarrow{b}_1, \overrightarrow{b}_2, ..., \overrightarrow{b}_n)$ of $\mathbb{V}$ and $\mathbb{B}^* := (\overrightarrow{b}^*_1, \overrightarrow{b}^*_2, ..., \overrightarrow{b}^*_n)$ of $\mathbb{V}^*$ be two random bases up to the constraint that they are *dual orthonormal.* These bases form a $n \times n$ matrix consisting of elements from G_1 and G_2, respectively. If we now compute the pairing $e_n(b_i, b^*_j)$ the result is $e_n(g_1, g_2)^{b_i * b_j}$, which is $e(g_1, g_2)^1 \neq 1$ for $i = j$ and $e(g_1, g_2)^0 = 1$ for $i \neq j$ [74, 75]. We show how to generate those bases in Section 3.7.1.

2.3 Cryptographic Primitives

In this section, we describe the cryptographic primitives required to implement the constructions proposed by Poettering and Rösler [77]. For this, we first describe the general functionality of the primitive and then describe security notions that are important in the context of ARKE. To keep the security notions consistent with the notions used by Poettering and Rösler [77] we use the same definitions.

2.3.1 Message Authentication Code

Message Authentication Codes (MACs) are used to authenticate a message. This means it confirms that the message came from the stated sender (Authenticity) and that its content has not been changed (Integrity). A MAC is a tuple of algorithms MAC = (tag, vrfy). The algorithm $\tau \Leftarrow \text{tag}_k(m)$ takes as input a symmetric key k and a message m, and outputs a tag τ. The algorithm $\text{vrfy}_k(m, \tau)$ takes as input a symmetric key k, a message m, and a tag τ and outputs 1 if τ is a valid tag for m and 0 otherwise. The most commonly used designs to implement MACs are based on block ciphers, hash functions, or universal hash functions [4].

Security

The security requirement for MACs used by Poettering and Rösler [77] is a multi-instance version of (strong) unforgeability. In this security game, the adversary has to produce a fresh and valid tag for a message of its choosing. For this the adversary is supported by tag generation and verification oracles, meaning the adversary can generate tags and verify tags. Since Poettering and Rösler [77] use a multi-instance version of the standard notion for unforgeability the adversary is also allowed to create new instances or learn their keys. So for a Message Authentication Code MAC = (tag, vrfy), an adversary $\mathcal{A}$ has a strong unforgeability advantage $\text{Adv}^{\text{suf}}_{\text{MAC}}(\mathcal{A}) := \Pr[\text{SUF}(\mathcal{A}) \Rightarrow 1]$. If this advantage is negligible for all practical adversaries, we define the MAC as secure. For the full description of the security game SUF, we refer to [77].

2.3.2 Digital Signature

Just as MACs, digital signatures are used to authenticate a message [52]. However, in addition to providing Authenticity and Integrity, digital signatures also provide Non-Repudiation which means that the sender of a message can not deny having sent the message. A digital signature is a tuple of algorithms Sig = (gen, sgn, vrfy). The algorithm $(sgk, vfk) \Leftarrow \text{gen}$ generates a random signer key *sgk* and verifier key *vfk*. The algorithm $\sigma \Leftarrow \text{sgn}(sgk, m)$ takes as input a signer key *sgk* and a message m and outputs a signature σ. The algorithm $\text{vrfy}(vfk, m, \sigma)$ takes as input a verifier key *vfk*, a message m, and a signature σ, and outputs 1 if σ is a valid signature for m signed with the matching signer key of *vfk*, and 0 otherwise. Digital Signatures are realized by the use of asymmetric cryptography. For this reason, the signer key and verifier key are an asymmetric key pair and represent the private key (*sgk*) and the public key (*vfk*) explaining why the vrfy algorithm only outputs 1 if we use the correct verifier key, matching the signer key used for the signature creation. Poettering and Rösler [77] require a special kind of signature for their constructions, namely, one-time Signatures. One-time Signatures are digital signatures that can only be used to sign one message. Otherwise, a signature can be forged by an adversary [52].

Security

The security requirement for Signatures used by Poettering and Rösler [77] is similar to the security requirements for MACs. The security notion is (strong) unforgeability for one-time signatures. In this security game, the adversary has to produce a message-signature pair that is accepted by the verifier and was not processed by the signer. So there is a signer oracle which the adversary can use to sign messages and a verifier oracle which can be used to verify signatures. The signer oracle records all message-signature pairs given to the adversary, and if the adversary makes the verifier accept a message-signature pair, which is not stored by the signer, the adversary wins the game. For a one-time Signature Sig = (gen, sgn, vrfy), an adversary $\mathcal{A}$ has a strong unforgeability advantage $\text{Adv}^{\text{suf}}_{\text{Sig}}(\mathcal{A}) := \Pr[\text{SUF}(\mathcal{A}) \Rightarrow 1]$. If this advantage is negligible for all practical adversaries, we define the Signature as secure. For the full description of the security game SUF, we refer to [77]. In the literature, the game SUF is often referred to as SUF-CMA [48, 72]. We need to keep that in mind when we choose a signature scheme in Chapter 3.

2.3.3 Hash functions

Hash functions are functions that can efficiently compute a mapping from an arbitrary length input to a fixed output. They are also sometimes called one-way functions. In cryptography, they have many different applications [52]. For example, hash functions can be used in digital signatures to reduce the size of the message to be signed.

The constructions proposed by Poettering and Rösler [77] do not explicitly require hash functions but can be used to realize the random oracle used in the constructions. Before continuing with the description of hash functions, we give a short introduction to random oracles.

Random Oracle

A random oracle $RO(\cdot)$ is a black box that outputs truly random values [14]. Calling a random oracle with a specific input of arbitrary length nets a specific random output of fixed length. Since we assume a random oracle only outputs truly random values, random oracles cannot be realized in the "real world". However, a random oracle can be used to prove a scheme on the assumption that a hash function only returns random values. By definition, a random oracle provides Collision-Resistance meaning we cannot find two inputs m, m' such that $RO(m) = RO(m')$. Furthermore, a random oracle is a One-Way Function meaning given an output r we cannot efficiently find an input such that $RO(m) = r$. Random oracles are used in the Random Oracle Model (ROM) [14, 27]. If a scheme uses a hash function the security proof of the scheme can be carried out in the ROM. In these security proofs calls to the hash function are replaced by calls to a random oracle, so in general, we assume that the hash function behaves like a real random function.

Now that we know what the hash function in the constructions proposed by Poettering and Rösler [77] is used for we define hash functions and describe the security properties which a hash function should achieve. A hash function $H(\cdot)$ is a function that takes an input $m \in \{0,1\}^*$ of arbitrary size and computes an output $h \in \{0,1\}^n$ of fixed length $n \in \mathbb{N}$. Since the security proof of the constructions uses the ROM, the hash function should, in a best-case scenario, behave like a random function.
Security
There are mainly three security properties a hash function should achieve to be considered cryptographically secure [3, 52]:

Collision Resistance A Hash function $H(\cdot)$ is collision resistant if we cannot efficiently find two inputs m, m' such that $H(m) = H(m')$.

Second Preimage Resistance A Hash function $H(\cdot)$ is second preimage resistant if given an input m we cannot efficiently find a second input m' such that $H(m) = H(m')$.

One-Way Function A Hash function $H(\cdot)$ is a one-way function if given a value h we cannot efficiently find an input such that $H(m) = h$.

2.3.4 Chameleon Hash Functions

A chameleon hash function is a special kind of hash function which utilizes a one-way function (or trapdoor) to compute a specific output for a given input [48, 72].

A chameleon hash function is a tuple of algorithms $\mathrm{CH} = (\mathrm{gen}, H, H^{-1})$ [72]. The algorithm $(\mathit{ek,td}) \Leftarrow \mathit{gen}()$ generates a pair of random keys named *evaluation key* and *trapdoor key*. The algorithm $h(ek, m, r) \leftarrow H(\mathit{ek,m,r})$ takes the evaluation key ek, a message m, and randomness r and outputs a hash value $h(ek, m, r)$. The algorithm $r' \leftarrow H^{-1}(td, m, r, m')$ takes as input the trapdoor key td, messages m, m', and randomness r and outputs r' such that $h(ek, m, r) = h(ek, m', r')$. We utilize these chameleon hash functions to build strong-unforgeable One-Time Signatures as described by Mohassel [72] or Jager [48].
Security
The only security requirement we require a chameleon hash function to fulfill is *collision resistance* [48, 72]. So an adversary that does not know the trapdoor key td should not be able to find a collision for a given H [48]. For a chameleon hash function $\mathrm{CH} = (\mathrm{gen}, H, H^{-1})$ an adversary $\mathcal{A}$ has the advantage [72]

$$\begin{aligned}\mathrm{Adv}^{\mathrm{colres}}_{\mathrm{CH}}(\mathcal{A}) := \Pr[(m, r) \neq (m', r') \wedge h(ek, m, r) = h(ek, m', r') :\\ (\mathit{ek,td}) \leftarrow \mathit{gen}();\\ (m, r, m', r') \leftarrow \mathcal{A}(ek, H)]\end{aligned}$$

If this advantage is negligible for all practical adversaries, we define the chameleon hash function as secure.

2.3.5 Key Encapsulation Mechanism

A KEM is an asymmetric protocol, which can be used to exchange a symmetric key over an unsecure channel. For this the KEM generates a random symmetric key and encrypts it using the recipient's public key. So in general, a KEM works like a public key encryption scheme, but the encryption, which is also called *encapsulation*, takes only the recipient's public key as an input [84]. A KEM is a tuple of algorithms KEM = (gen, enc, dec). The algorithm $(\mathit{sk, pk}) \Leftarrow \mathrm{gen}$ generates a random secret key sk and public key pk. The algorithm $(k, c) \Leftarrow \mathrm{enc}(pk)$ takes as input a public key pk and outputs a random symmetric key k, and ciphertext c. The algorithm $k \leftarrow \mathrm{dec}(sk, c)$ takes as input a secret key sk and a ciphertext c, and outputs a symmetric key k, if sk is the matching secret key to the public key used for the encapsulation.
Security
The security requirement for KEMs used by Poettering and Rösler [77] is a multi-user version of one-way security. In this security game the adversary obtains challenge ciphertexts and has to recover any of the encapsulated symmetric keys. For this the adversary is supported by a key-checking oracle. For a provided pair of ciphertext and (candidate) symmetric key, the oracle tells whether the ciphertext decapsulates to the symmetric key. Since Poettering and Rösler [77] use a multi-user version of the standard notion for one-way security the adversary is also allowed to create new receivers or learn their keys. So for a Key Encapsulation Mechanism KEM = (gen, enc, dec), an adversary $\mathcal{A}$ has a one-way advantage

$\mathrm{Adv}^{\mathrm{ow}}_{\mathrm{KEM}}(\mathcal{A}) := \Pr[\mathrm{OW}(\mathcal{A}) \Rightarrow 1]$. If this advantage is negligible for all practical adversaries, we define the KEM as secure. For the full description of the security game OW, we refer to [77].

2.3.6 Key-Updateable Key Encapsulation Mechanism

A kuKEM is a special type of KEM introduced by Poettering and Rösler [77]. Like a KEM a kuKEM generates random symmetric keys and is able to encapsulate and decapsulate the keys. Furthermore, the kuKEM provides a key update algorithm, which derives new ('updated') keys from old ones. Using an auxiliary input called the *associated data* a secret key is updated to a new secret key, and a public key is updated to a new public key, respectively. If a secret key and public key pair was updated compatibly, meaning with matching *associated data*, the key pair remains functional, so a symmetric key encapsulated under a public key can be recovered by decapsulating to the matching secret key [77]. A kuKEM is a tuple of algorithms kuKEM = (gen, enc, dec, up). The algorithms gen, enc, dec are the same as for regular KEMs. There are two kinds of the key-update algorithm up. The first update algorithm $pk' \leftarrow \mathrm{up}(pk, ad)$ takes as input a public key pk and an associated data ad, and outputs an updated public key pk'. The second update algorithm $sk' \leftarrow \mathrm{up}(sk, ad)$ takes as input a secret key sk and an associated data ad, and outputs an updated secret key sk'. Using a HIBE we can directly construct a kuKEM [77].
Security
The security requirement for kuKEMs is similar to the security requirement of KEMs. It is also a multi-user version of one-way security, but also reflects forward security in a case of secret key updates [77]. This characteristic is realized by allowing the adversary to update public keys and secret keys held by the respective user. So for a key-updateable Key Encapsulation Mechanism kuKEM = (gen, enc, dec, up), an adversary $\mathcal{A}$ has a one-way advantage $\mathrm{Adv}^{\mathrm{kuow}}_{\mathrm{kuKEM}}(\mathcal{A}) := \Pr[\mathrm{KUOW}(\mathcal{A}) \Rightarrow 1]$. If this advantage is negligible for all practical adversaries, we define the kuKEM as secure. For the full description of the security game KUOW, we refer to [77].

2.3.7 Hierachical Identity Based Encryption

A Hierarchical Identity-Based Encryption (HIBE) is a special kind of public key encryption scheme. HIBE are based on Identity-Based Encryption (IBE), for this reason we briefly describe IBE to obtain a better idea of the motivation and purposes of IBE and HIBE. The main motivation for IBE was to simplify certificate management in a closed group, e.g., companies, especially for the e-mail systems of such groups [83]. Instead of generating random public-secret key pairs and distributing the public key via certificates, the public key can be derived from the name or some identifying information like an e-mail address. So if Alice wants to send an encrypted message to Bob, she does not need to look up Bob's public key, which would require a secure channel to a public key directory, for example, but can directly use Bob's e-mail address as a public key [19, 83]. If Bob receives this message, he contacts a trusted third party, which is called the Private Key Generator (PKG), to obtain his private key. IBE has a few disadvantages, which are simultaneously the motivation for HIBE [19]. For Bob to obtain his private key, he has to authenticate himself at the PKG and establish a secure channel for communication. Furthermore, Bob's PKG has to publish public parameters, which need to be obtained by Alice before she is able to encrypt messages for Bob. And lastly, in IBE key escrow is inherent because the PKG knows the secret keys of all users. Looking at the disadvantages, we can easily see that the PKG is the bottleneck of an IBE scheme. Secret key generation is computationally expensive and the PKG has to establish secure authenticated channels with every user to transmit secret keys [37].

HIBE tries to solve those problems by distributing the workload of a root PKG to lower-level PKGs [37, 44]. This means in a HIBE we have different levels of PKG. We have a root-level PKG, which is able to extract keys for its domain-level PKGs. These domain level PKGs are able to extract keys for their domains in the next level. So an advantage of a HIBE scheme is that Bob only needs to communicate with his *root* PKG. Equally, Alice only needs to obtain the public parameters of Bob's *root* PKG [37]. Because of this structure, the inherent problem with key escrow is less hazardous. If a domain-level PKG is disclosed, higher level PKGs are usually not affected [37].

A Hierarchical Identity-Based Encryption (HIBE) is a tuple of algorithms HIBE = (gen, enc, dec, extract). The algorithm $(sk, pk) \Leftarrow$ gen generates a random secret key sk and public key pk. Note that the secret key sk is the secret key of the root-level PKG and the public key contains the public parameters needed for encryption. The algorithm $c \Leftarrow \text{enc}(pk, m, id)$ takes as input the public key (public parameters) pk, a message m, and an identity id and outputs a ciphertext c. The algorithm $m \leftarrow \text{dec}(sk, c)$ takes as input a secret key sk, and a ciphertext c and outputs the message m, if the secret key belongs to the identity used for encryption. The algorithm $sk' \leftarrow \text{extract}(sk, id)$ takes as input a secret key sk, and a identity id and outputs a secret key sk' for the identity id. Note that a PKG can only extract a secret key for an identity that lies in a lower-level than itself. Furthermore, sometimes in the literature there is a difference between the root PKG that generates a secret key for a domain-level PKG and a domain-level PKG that generates a key for a lower-level PKG. The former algorithm is then called *extract* or *keygen* and the latter *delegate* [58]. Throughout this thesis, we make the same distinction, because this makes the descriptions more clear. In this thesis we focus on HIBE that are realized through bilinear pairing groups, but there are HIBEs that are based on other primitives, e.g., lattices [5, 28], as well.

Security

In the constructions proposed by Poettering and Rösler [77] we need a HIBE to realize the kuKEM. So the security requirements for the HIBE are directly correlated

to the security requirements for the kuKEM. There are mainly two security definitions for HIBEs that are interesting in our context. The first security definition is *indistinguishability against hierarchical ID-based adaptive chosen ciphertext attacks* (IND-HID-CCA) proposed by Gentry and Silverberg [37]. A slightly different name for the same security definition, which is also often used in literature, is *indistinguishability against ID-based adaptive chosen ciphertext attacks* (IND-ID-CCA) proposed by Boneh et al. [20]. In the second security definition a HIBE is defined as an one-way encryption (OWE) [37]. This leads to the name *hierarchical ID-based one-way encryption* (HID-OWE). A slightly different way to name this definition is *one-way identity-based chosen ciphertext attack* (OW-ID-CCA). In this thesis, we use the names IND-ID-CCA and OW-ID-CCA, since they are used more often in the literature.
In the security game IND-ID-CCA, an adversary chooses an identity *ID* and two messages m_0, m_1 from which one is randomly selected and encrypted under the public key of the selected *ID*. The adversary than receives the challenge ciphertext c and has to guess the message that got encrypted. For this, the adversary is supported by three oracles. A public key oracle, which the adversary can query with an *ID* to obtain the public key of *ID*. An extraction oracle, which the adversary can query with an *ID* to obtain the secret key of *ID*. And a decryption oracle, which the adversary can query with an *ID* and a ciphertext c to obtain the plaintext m. Any of those three oracles can be queried before or after the adversary has chosen the two messages. To exclude trivial wins for the adversary, the adversary is not allowed to query the extraction oracle with the selected *ID* or any of its ancestors. Furthermore, he is not allowed to call the decryption oracle with *ID* or any of this ancestors with the challenge ciphertext c. The game OW-ID-CCA is very similar, with the only difference being that the message used to produce the challenge ciphertext c is generated randomly. The *ID* used to encrypt the challenge ciphertext is still picked by the adversary [37]. Apart from that, the adversary has access to the same oracles under the same conditions.
So for a Hierarchical Identity-Based Encryption HIBE = (gen, enc, dec, extract) an IND-ID-CCA (resp. OW-ID-CCA) adversary $\mathcal{A}$ has an advantage

$$\mathrm{Adv}_{\mathrm{HIBE}}^{\text{IND-ID-CCA}}(\mathcal{A}) := |\Pr[\text{IND-ID-CCA}(\mathcal{A}) \Rightarrow 1] - \frac{1}{2}|$$

(resp. $\mathrm{Adv}_{\mathrm{HIBE}}^{\text{OW-ID-CCA}}(\mathcal{A}) := \Pr[\text{OW-ID-CCA}(\mathcal{A}) \Rightarrow 1]$). If this advantage is negligible for all practical adversaries, we define the HIBE as secure. For the full description of the security games IND-ID-CCA and OW-ID-CCA, we refer to [20, 37].

2.4 Asynchronous Ratcheted Key Exchange (ARKE)

In this section, we describe the constructions proposed by Poettering and Rösler [77] and describe ratcheting as it is used today. We only give brief summaries of URKE and Sesquidirectionally Ratcheted Key Exchange (SRKE), because the main focus of this thesis is the BRKE construction.

2.4.1 Ratcheting

Ratcheting is a cryptographic technique often used in an instant messaging context [29, 64]. A ratchet is constructed in a way that it only moves forward. This means we cannot go back to old states. Similar to a zip tie the ratchet always moves one step at a time. As described in the introduction a simple example of such a construction could be a "hash chain" in which we have an initial symmetric key, and every time the key is used a one-way function is applied to derive a new key from the old key [77]. Ratcheting aims to provide security against two types of attack: compromise of long-term secrets or session states. Security against the compromise of long-term secrets is also referred to as *forward secrecy*. So if an adversary is able to obtain a copy of the long-term secrets of a user the adversary is still not able to reveal later communication contents of that user. Security against the compromise of session states is a security goal of modern chat protocols [77] and it means that users can recover from session state leakages. The recovery is desired because messaging sessions are often kept alive for a long time. Furthermore, it is sometimes impossible to defend against state exposure because the state information (which includes session keys, for example) kept in the memory might at one point be swapped to disk and then can be stolen from there.
In the Double Ratchet Algorithm [64], for example, the initial key is evolved by using a symmetric-key ratchet and a Diffie-Hellman (DH) ratchet. The symmetric-key ratchet consists of a so-called Key Derivation Function (KDF)-chain and is used so a key is never used twice. The DH-ratchet is a DH key pair which is updated whenever possible and is used to recover from state exposure. Bellare et al. [16] formally analyze the security of RKE. They also provide syntax and security definitions for ratcheting. The security model established by Bellare et al. [16] is quite restricted because the model assumes only unidirectional communication (in the Alice-to-Bob direction) and also assumes that only Alice's state can be exposed. Unfortunately, this is a counter-intuitive assumption, since usually instant messaging is a communication protocol which is performed bidirectionally and asynchronous. This means for two users A and B which communicate over a network (like the Internet) we consider communication in both directions. From Alice-to-Bob and Bob-to-Alice (bidirectional). Furthermore, we assume that the conversation is human-like which means that each of the users contributes to the conversation when they feel like it. The messages can even cross while transferring over the network (asynchronous).
For this reason, Poettering and Rösler [77] developed alternative security definitions to account for a more realistic communication environment. This lead to the three constructions URKE, SRKE, and BRKE where BRKE ultimately considers full asynchronous and bidirectional communication between two parties.

2.4.2 Unidirectionally Ratcheted Key Exchange (URKE)

Unidirectionally Ratcheted Key Exchange is the first of three proposed constructions by Poettering and Rösler [77]. Similar to Bellare et al. [16] this construction also only considers unidirectional communication, but also consider state exposure attacks on Bob. This approach leads to a security model, which better represents the real world and thus to a stronger security notion. Poettering and Rösler [77] also provide a construction only using standard cryptographic primitives like MAC, KEM, and Hashes. There are two involved parties in the URKE construction, A and B, which communicate over an insecure network. Every message from A to B establishes a new session key, which can safely be transmitted to B via ciphertexts. One specialty of the construction is that the sending and receiving algorithms process an associated data string, which has to match on both A and B. We do not go into more detail in this thesis, but note that all three constructions (URKE, SRKE, BRKE) are somewhat based on each other and every construction takes more attack vectors into account to better represent a real-world application.

2.4.3 Sesquidirectionally Ratcheted Key Exchange (SRKE)

Sesquidirectionally Ratcheted Key Exchange (SRKE) is the second of the proposed constructions by Poettering and Rösler [77]. SRKE provides the same functionality as the URKE construction, but also lets B generate and send ciphertexts to A. These ciphertexts are not used to establish new session keys, but to let B recover from state exposure attacks. For this, the construction uses a MAC, a one-time signature, a hash function, and a new type of primitive called the kuKEM. A kuKEM is a special type of KEM which allows key updates. We explain the details of the kuKEM in Section 2.3.6.

2.4.4 Bidirectionally Ratcheted Key Exchange (BRKE)

Bidirectionally Ratcheted Key Exchange (BRKE) is the third of the proposed constructions by Poettering and Rösler [77]. BRKE extends the functionality of SRKE by now establishing new session keys with every message that is sent by either A or B. This construction now provides security in a real-world instant messaging communication scenario. For this, the BRKE construction uses a one-time signature, a hash function, and the kuKEM. Poettering and Rösler [77] propose two constructions of BRKE. One is constructed by using two SRKE schemes linked with a one-time signature, and the other one is an ad-hoc construction directly adopting the SRKE construction. The ad-hoc construction is shown in Figure 4.2. The goal of this thesis is to implement the ad-hoc construction of the BRKE protocol. We analyze the functionality of the BRKE ad-hoc construction in Section 4.3.

3 Algorithm Choices

In this section, we explain the choices for the algorithms we chose for the realization of the BRKE construction. We first describe how those algorithms are used in the context of BRKE, discuss the reasoning of our choices, and show possible alternatives. Before we start describing the algorithm choices, we explain our approach to choosing secure and state-of-the-art algorithms.

3.1 Choosing Algorithms

Instead of just randomly looking for algorithms that fulfill the requirements we set in Section 2.3, we want to have a guideline with recommendations, which we can consult when choosing an algorithm and corresponding parameters. The starting point for this is the website *keylength.com* [17], which summarizes several governmental and non-governmental recommendation papers on cryptographic algorithms and key sizes. The website is mostly concerned with choosing secure key lengths for several primitives, but also provides a small overview for recommended algorithms. From this website, we choose the two most recent publications and use them for the choice of our algorithms and parameters. These two publications are the ECRYPT-CSA report on "Algorithms, Key Size and Protocols" [4], and the report "Cryptographic Techniques: Recommendations and Key Lengths" [3] from the German Federal Office for Information Security (BSI), both from 2018. Since these reports are the most conservative from the reports summarized on *keylength.com* [17], we think that choice is well suited for our purpose.
The next step in finding algorithms that fulfill our purpose is checking if the recommended algorithms provide the security level we need. For example, we have to find proof that the KEM is OW-CCA secure. For this, we have a look at the proposals and security proofs of the specific algorithms. After that, we check if we can find an implementation of the algorithm. Preferably this implementation is provided by *bouncy castle* [86] for Java, since we choose to implement the construction and the instantiation in Java (see Chapters 4 and Chapter 5). So the general steps are:

1. Consult recommendations from ECRYPT-CSA [4] and BSI [3].

2. Check if the algorithm meets the security requirements we set in Section 2.3.

3. Check if there is an implementation of the algorithm in Java (preferably in bouncy castle [86]).

Since initially we are not concerned with performance, we only check if the security requirements are met and not which algorithm is the most efficient. If two algorithms are equally suited, but one is said to have better performance, we may directly choose the latter algorithm.

3.2 One-Time Signature

As described in Section 2.3.2 a one-time signature is a special kind of signature, which is only used to sign one message. In the BRKE construction we only need a one-time signature instead of a regular digital signature, because the keys are changed after every signing or verification operation. In fact, we could use a regular digital signature, which fulfills the same security requirement as our one-time signature. This means we could also use a signature for that the SUF advantage for an adversary is negligible. Intuitively, such a signature scheme implies a SUF secure one-time signature scheme. This is important, because if we have a look at the recommended signatures by ECRYPT-CSA [4] and BSI [3] we see that neither considers one-time signatures.
In Table 3.1 we can see recommended signature schemes and their security assumption. Furthermore, we provided a reference for the security proof. As we can see

Scheme	Security	Reference
RSA-PSS	EUF-CMA	Jonsson [50]
ISO-9796-2 RSA-DS2	EUF-CMA	Jonsson [50]
PV Signatures	EUF-CMA	Pointcheval and Vaudenay [78]
(EC)Schnorr	SUF-CMA	Kiltz et al. [54]
(EC)KDSA	EUF-CMA	Brickell et al. [24]
XMSS	EUF-CMA	Buchmann et al. [26]

Table 3.1: Recommended signature schemes and their security assumption.

from Table 3.1 there is only one recommended signature scheme that fits our security requirement of SUF-CMA security. Unfortunately, as of writing this thesis, we found no well-documented and referenced implementation of the Schnorr signature in Java, so we have a look at other possibilities to realize a SUF-CMA secure (one-time) signature.
As a first alternative, we consider signatures which use pairings as the underlying primitive. BLS and BBS signatures [82] for example are both EUF-CMA secure and unique, which implies that they are SUF-CMA secure [82]. However, as of writing this thesis, there are no suitable pairing libraries for Java (see Section 5.1), so we do not consider them for now.

Now we have a look at "real" one-time signatures. One way to build SUF-CMA secure one-time signatures is through Lamport constructions [53, 57]. One-time signatures build via the Lamport construction are based on hash functions. Examples for such SUF-CMA one-time signatures are W-OTS [25] and W-OTS+ [46]. However, again we could not find any implementation of those algorithms. Though the implementation of XMSS in bouncy-castle [86] internally uses W-OTS+, the implementation is not public, and we therefore cannot use it. There are more proposals for SUF-CMA (one-time) signatures and even transformations from EUF-CMA secure signatures to SUF-CMA signatures [22, 45], but ultimately we did not find any implementation of these schemes.
Another option to build one-time signatures are via chameleon hash functions [48, 72]. With a generic transformation via signatures [48] or hash functions [72] the resulting one-way signatures can be shown to be SUF-CMA secure. As of writing this thesis we cannot find implementations of these schemes either, but since they seem to be the most straight forward constructions and performance is no concern, we choose to implement a SUF-CMA secure one-time signature based on a chameleon hash function.
We choose to implement the one-time signature with a chameleon hash function based on the DLP problem. In particular, we implement the construction proposed by Mohassel [72], since it only requires a hash function for the transformation to SUF-CMA security instead of an additional signature scheme as described by Jager [48]. Now we have to find parameters for a group in which the DLP problem is hard. We propose to not generate parameters from scratch, but use recommended and, thus, well-studied parameters instead. So we propose to use a group that is recommended for the use in the TLS Diffie-Hellman Ephemeral key exchange mode [38]. The Diffie-Hellman key exchange is secure under the assumption that the DDH assumptions holds in the used group [51]. It easy to show that the DDH assumption can be reduced to the CDH assumption and the CDH assumption can be reduced to the DLP problem [67, 87]. For this reason, we can assume that if the DDH assumption in the recommended groups hold, the DLP problem is hard. The last parameter we have to discuss is the size of the prime of the finite field. ECRYPT-CSA recommends at least a prime size of 3072 Bit [4], and the BSI recommends a prime size of at least 2000 Bit [3] for the finite field DLP. If we have a look at the group "ffdhe3072" [38] it has a prime size of 3072 Bit, and an estimated symmetric-equivalent strength of 125 bits [38]. We decide to use this group to construct the DLP-based chameleon hash one-time signature.
DLP-Based Chameleon Hash One-Time Signature Since we have to implement the one-time signature ourselves, we will shortly introduce and describe the construction proposed by Mohassel [72]. To construct the signature, we need a collision-resistant hash function and the chameleon hash function based on the DLP problem. Note that Mohassel [72] actually requires only a target collision hash function, which is a weaker notion, but since we already require collision resistant hash functions (see Section 2.3.3), we directly use a collision-resistant hash function. For the description of the chameleon hash function we refer to Mohas-

sel [72] or Jager [48], we directly show the resulting one-time signature scheme [72]:

Key Generation:

1. Let g_1 be a generator for a group G of size p. Let T be a collision resistant hash function that maps elements of G to a subset of $\mathbb{Z}_p$.

2. Generate random $x, x' \in \mathbb{Z}_p$ and compute $g_2 = g_1^x$ and $g_3 = g_1^{x'}$.

3. Compute $z_0 = T(g_1 g_2^r)$ and $z_1 = T(g_1 g_3^{r'})$ for random $r, r' \in \mathbb{Z}_p$

4. The verification key is $vk := (g_1, g_2, g_3, z_0)$ and the signing key is $sk := (y := x^{-1}, y' := x'^{-1}, r, r', z_1)$

Signing: For a message m, compute and return signature $\sigma = (\sigma_0, \sigma_1)$ with $\sigma_0 = y'(1 - m) + r'$ and $\sigma_1 = y(1 - z_1) + r$.

Verification: For a message m and signature $\sigma = (\sigma_0, \sigma_1)$, accept if $T(g_1^{T(g_1^m g_3^{\sigma_0})} g_2^{\sigma_1}) = z_0$ and reject otherwise.

3.2.1 Summary

Table 3.2 shows possible candidates for the required SUF secure signature we discussed in this section. Note that there are more signature schemes which are SUF secure and, thus, could be used in the BRKE construction. We choose to implement

Scheme	Primitive
(EC)Schnorr	Elliptic Curve, Finite Field
BLS	Bilinear Pairings
BBS	Bilinear Pairings
W-OTS	One-Way Function
W-OTS+	One-Way Function
Chameleon Hash Based	DLP, RSA Problem
Any + transformation [22, 45]	-

Table 3.2: Possible candidates for the SUF secure signature discussed in this section.

a DLP-Based Chameleon Hash One-Time Signature with the group “ffdhe3072” from RFC7919 [38] which has a prime size of 3072 Bit and an estimated symmetric-equivalent strength of 125 bits [38].

3.3 Hash Functions

Several times throughout the implementation we need hash functions. As discussed in Section 2.3.3 we want to use cryptographically secure hash functions. Sometimes it would be sufficient to use a hash function that only achieves a weaker security notion, e.g., target collision hash function, but in those cases, we still use a cryptographically secure hash function. Recommended hash functions are shown in Table 3.3. Bouncy-

Primitive	Output Length
SHA-2	256, 384, 512, 512/256
SHA-3	256, 384, 512
SHA-3	SHAKE128, SHAKE256
Whirlpool	512
BLAKE	256, 384, 512

Table 3.3: Recommended hash functions that are considered cryptographically secure [3, 4].

castle [86] implements all of those hash functions so that we can use any of those. Most of the time we use SHA256, but we if we have to choose a hash function we explain our choice in the specific section.

3.3.1 Summary

We only use hash functions that are considered cryptographically secure throughout this thesis, even if a weaker hash function is sufficient. All in Table 3.3 shown hash functions can be used. Most of the time we use SHA256, but we address the choice of the hash function if necessary in the specific sections.

3.4 Random Oracle

As described in Section 2.3.3 the random oracle is a theoretical primitive that is used to assume a function has random output. Furthermore, we described that a random oracle is often realized by a hash function in real-world applications. In the BRKE construction the random oracle is used to produce key material (Line 28/57, BRKE construction, Figure 4.2). Thus, we do not directly use a hash function to realize the random oracle, but a KDF that we instantiate with a hash function. A KDF is used to derive cryptographic keys from a source of keying material [4]. Table 3.4 shows all recommended (hash-based) KDFs. We could also use none hash-based KDF, since KDFs are often modelled as a random oracle [4], whether instantiated with a hash function or not. However, since we defined to realize the random oracle with a hash function, we limit our choice to the KDFs shown in Table 3.4. It is left

Scheme	Building Block
X9.63-KDF	Hash
NIST-800-56-KDF-A/B	Hash
HKDF	HMac
IKE-v2-KDF	HMac
TLS-v1.2-KDF	HMac
KDFs described in ISO 18033-2	Hash

Table 3.4: Recommended (hash-based) KDFs schemes.

to find secure hash functions to instantiate the KDF. For this, we can use any of the hash functions described in Section 3.3.
Of these KDFs HKDF and the ISO 18033-2 KDFs are implemented in bouncy-castle [86]. We choose to use HKDF [56] with SHA256.

3.4.1 Summary

Instead of directly using a hash function to realize the random oracle, we choose to use a KDF. The reason for this is that the random oracle in the BRKE construction is used to generate keys. Recommended (hash-based) KDFs are shown in Table 3.4. These can be instantiated with any hash function discussed in Section 3.3. We choose to use HKDF [56] with SHA256.

3.5 Key Encapsulation Mechanism

The security requirement for the KEM is OW security. A different notion for this is OW-CCA security (*one-way chosen ciphertext attack*). Furthermore, we can use a IND-CCA [51] (*indistinguishability against chosen ciphertext attack*) secure KEM, because intuitively IND-CCA implies OW-CCA security. In Table 3.5 we can see recommended KEM schemes and their security assumption. All three schemes are

Scheme	Security	Reference
RSA-KEM	IND-CCA	Shoup [47, 85]
PSEC-KEM	IND-CCA	Shoup [47, 85]
ECIES-KEM	IND-CCA	Shoup [47, 85]

Table 3.5: Recommended KEM schemes and their security assumption.

IND-CCA secure under the assumption that the underlying KDF is a random oracle. For this reason, we also have to choose a secure KDF. Which means the KDF should ideally behave like a random function. Recommended KDFs are shown in Table 3.6. All in all, we can take any combination of a KEM shown in Table 3.5 and a KDF

Scheme	Building Block
NIST-800-108-KDF(all modes)	PRF
X9.63-KDF	Hash
NIST-800-56-KDF-A/B	Hash
NIST-800-56-KDF-C	Mac
HKDF	HMac
IKE-v2-KDF	HMac
TLS-v1.2-KDF	HMac
KDFs described in ISO 18033-2	Hash

Table 3.6: Recommended KDFs schemes and the underlying building block.

shown in Table 3.6. Furthermore, the KDF has to be instantiated with a secure primitive [4]. If we use a hash-based KDF we can take any of the hash functions discussed in Section 3.3.
Of the three KEMs two are implemented in bouncy-castle [86] namely the ECIES-KEM and the RSA-KEM. We choose to use the ECIES-KEM. This means we have to find suitable parameters for the elliptic curve used in the ECIES-KEM. Again we will use a recommended elliptic curve and do not generate a curve from scratch. The BSI recommends using one of the following elliptic curves for the use with an ECIES scheme:

- brainpoolP256r1 [61]
- brainpoolP320r1 [61]
- brainpoolP384r1 [61]
- brainpoolP512r1 [61]

ECRYPT-CSA recommends at least 256 Bit for the size of the group order [4] but does not recommend any specific curves. We choose to use the curve "brainpoolP256r1" since it is recommended by the BSI and complies to the recommendations of ECRYPT-CSA. KDFs that are supported by the ECIES-KEM implementation in bouncy-castle are KDF1 and KDF2 described in ISO 18033-2 [47]. Since both KDFs are among the recommended KDFs we can choose either of those instantiated with a hash function discussed in Section 3.3. We use KDF2 with SHA256 in our implementation.

3.5.1 Summary

Possible candidates for the OW secure KEM are shown in Table 3.5. All schemes are secure with the assumption that the underlying KDF is a random oracle. Possible candidates for the KDF are shown in Table 3.6. These have to be instantiated with a secure primitive [4]. We choose to use the ECIES-KEM with KDF2 instantiated with SHA256. As the elliptic curve we use “brainpoolP256r1” [61].

3.6 Key-Updateable Key Encapsulation Mechanism

As we described in Section 2.3.6 the kuKEM can be constructed by a HIBE. Before discussing the choices for the HIBE we want to analyze the use the of HIBE by the kuKEM. This helps explaining and understanding the choices for the HIBE. First, we analyze the functionality of the kuKEM and what this means for the HIBE.
The kuKEM has four algorithms that we need to analyze and compare to the HIBE functionality. The kuKEM algorithms gen_K, enc_K, and dec_K have mostly the same functionality as the HIBE algorithms gen_H, enc_H, and dec_H with the difference being that the kuKEM encapsulates a symmetric key and the HIBE encrypts a message. But this difference is trivial, since we can simply generate the key inside the kuKEM and use it as a message, or even generate the key inside the HIBE and directly encrypt it. The algorithm which sets the requirements on the HIBE is the *update* algorithm of the kuKEM keys. Before we have a look at the two update algorithms, we analyze the use of the associated data *ad* which the kuKEM uses for key updates. It turns out that *ad* is interpreted as the identity for the HIBE. And as described in Section 2.4.4 in the BRKE construction the kuKEM receives *ad* and a ciphertext as parameters for the key update algorithms. For this reason we call combination of the parameters the kuKEM receives for key updates “identity information”. So no matter if we refer to *ad* or $ad|ciphertext$ we think of it as the identity information that is used in the HIBE. Now we have a look at the two kinds of key updates: public key updates and secret key updates.
Public key updates: The public key update algorithm takes a public key *pk* and associated data *ad*, and outputs an updated public key pk'. As described by Poettering and Rösler [77] the public key for the kuKEM consists of the public key (public parameters) of the HIBE and an identity. Thus, updating a public key is simply appending *ad* to the identity of the public key. This sets no specific requirements on the HIBE, but we have to ensure that we can process *ad* (and ultimately the ciphertext) as an identity for the HIBE.
Secret key updates: The secret key update algorithm takes a secret key *sk* and associated data *ad*, and outputs an updated secret key sk'. As described by Poettering and Rösler [77] the secret key is updated by extracting a secret key for the “identity” *ad* (This helps to understand why we also call *ad* identity information). In the context of the HIBE a domain-level PKG generates a key for a lower-level PKG. So for two associated data ad_0 and ad_1, and a secret key sk_0 (generated for ad_0) we update the secret key $sk_1 = up(sk_0, ad_1)$ by delegating a HIBE secret key from ad_0 to ad_1.
This means for every key update we informally add one level to the HIBE. This sets an important requirement on the HIBE choice, since as we discuss in Section 3.7

most HIBE are bounded in their depth. Furthermore, we do not need every function of a HIBE. When we update a key, we always update it for the next lower level, thus, we do not need to extract a secret key with the root level secret key more than once. It is also unnecessary to be able to generate two keys for one hierarchy level, which is possible in a HIBE. So we might be able to optimize an existing HIBE by omitting specific functions.

3.6.1 Summary

The functionality of the kuKEM sets some requirements on the HIBE and also does not need some (usually supported) features of the HIBE. The input parameters for the update algorithm of the kuKEM are used as the identity information in the HIBE. Thus we refer to them under that name. The main aspects are:

- Identity information has to be usable by the HIBE.
- Every key update adds one level to the depth of the HIBE.
- The kuKEM does only require to be able to delegate keys to the next lower hierarchy level.
- The kuKEM does not require to be able to generate more than one key for each hierarchy level.

The first two aspects set some hard requirements on the implementation of the HIBE, and the last two aspects may enable us to optimize the implementation.

3.7 Hierarchical Identity Based Encryption

The HIBE is used to realize the kuKEM. As the security requirement, we set OW-ID-CCA security. Again we can use IND-ID-CCA secure HIBE, as well, because IND-ID-CCA implies OW-ID-CCA security. Unlike the other schemes and primitives HIBE is no “standard” algorithm used in many applications and protocols. For this reason, there are no recommendations from either ECRYPT-CSA [4] or the BSI [3]. This means we have to research manually to find suitable candidates for the choice of the HIBE and try to obtain a broad overview so that we can make an elaborate choice. Besides the requirement that the HIBE is OW-ID-CCA secure, the kuKEM sets some requirements as well. These are discussed in Section 3.6.
Table 3.7 shows a overview over seven HIBE that are based on pairings. We also specify the security assumption, and the indication if a HIBE is *unbounded*. If a HIBE is specified as *unbounded* we do not need to set the maximum hierarchy depth when generating the HIBE. As described in Section 3.6 we set the requirement that the HIBE should be unbounded so that we can update the kuKEM keys unlimited

Scheme	Security	Unbounded
Gentry-Silverberg [37]	IND-ID-CPA	✓
Boneh-Boyen-Goh [20]	IND-sID-CPA	-
Li-Zhang-Wang [60]	IND-sID-CPA	-
Waters [88]	IND-sID-CPA	-
Boneh-Boyen [18]	IND-sID-CPA	-
Lewko-Waters [59]	IND-ID-CPA	✓
Ryu-Lee-Park-Lee [80]	IND-sID-CPA	✓

Table 3.7: Comparison of different HIBE

times. We evaluate the effects of this decision on performance and key sizes in Chapter 6.

As we can see from Table 3.7 there are only three HIBE that fulfill the requirement of being unbounded. The HIBE proposed by Ryu et al. [80] is only selective-ID (sID) CPA secure, so we do not consider it for our purpose. Zhang et al. [91] compare the security of twelve HIBE and only consider two unbounded HIBE, as well. One of those two HIBE is proposed by Zhang et al. [91] themselves, but we do not consider this HIBE in this thesis, too, because it provides anonymity, which we do not require for our HIBE. This means there are only two HIBE to choose from. Furthermore, both unbounded HIBE that are left are only IND-ID-CPA secure, whereby we require the HIBE to be IND-ID-CCA secure. This means we have to use a transformation from IND-ID-CPA to IND-ID-CCA security like described by Boneh et al. [21]. Gentry and Silverberg [37] also provide a transformation from CPA to CCA security.

To our knowledge there are no implementations of the Gentry-Silverberg [37] HIBE as of writing this thesis. The Lewko-Waters [59] HIBE is implemented in the jPBC [30] (in Java) and the Charm crypto library [6] (in Python). More precisely, the implementation in the Charm crypto library implements the prime order translation of the Lewko-Waters [58] HIBE. Both implementations use the PBC library by Lynn [62] internally. Unfortunately, this library is outdated and does not provide implementations for secure pairings based on current research [8]. For this reason, we decide to implement the Lewko-Waters HIBE with the CPA to CCA transformation described by Boneh et al. [21] ourselves. Like the Charm crypto library, we implement the prime-order translation of the HIBE described by Lewko [58]. Firstly because the recommended pairings are not of a composite order [8], and secondly because as shown by Guillevic [41] the prime-order translation is way more efficient, e.g., 192 times more efficient for key delegation. Now we need to find a suitable pairing group for the implementation.

To find suitable pairing groups we use a different approach than we use for the other primitives. So instead of primarily consulting the recommendations by ECRYPT-CSA and the BSI, we search for pairing groups, which are proven to have a specific security level. The reason for that are recently found attacks against the security of pairing groups [8, 10, 70]. This means even if a pairing fulfills the recommendation set by the ECRYPT-CSA, e.g., the order of both elliptic curves have to be greater than 256 Bit [4], the pairing still might be insecure because of other parameters. We estimate the security of a pairing according to the estimated symmetric-equivalent strength. Table 3.8 shows four pairing-friendly curves and their corresponding estimated symmetric-equivalent strength. The reason that the table shows two BLS12

Curve	Estimated Symmetric-Equivalent Strength
BN-P256	100 [10, 90]
B12-381	127 [23, 70, 90]
B12-461	132 [10]
KSS16-340	128 [10]

Table 3.8: Comparison of Pairing Friendly Curves and Their Estimated Symmetric-Equivalent Strength

curves is that Barbulescu and Duquesne [10] state that the B12-381 curve has only an estimated symmetric-equivalent strength of about 110 bits, because of a specific attack, the extended tower number field sieve (exTNFS) [55]. However, there is currently no efficient method which achieves the computational time described by Barbulescu and Duquesne [10], so often the B12-381 curve is denoted as the optimistic choice for the 128-bit security level, and the B12-461 is denoted as the conservative choice for the 128-bit security level [8, 90]. The BSI recommends a symmetric key length of 100 Bits up to 2022, and 128 bits after that. ECRYPT-CSA recommends 128 bit for all applications. This means that according to the recommendations of the BSI we could use any of the curves shown in Table 3.8 and according to the ECRYPT-CSA we could not use the BN-P256 curve. We do not decide to use a specific curve, but we compare the different curves in the implementation because the choice of the curve has significant impacts on the performance of the construction.

3.7.1 Lewko-Waters Unbounded HIBE (Prime Order Translation)

Now we describe the prime order translation of the Lewko-Waters HIBE [58] and the CCA transformation described by Boneh et al. [21]. Lewko [58] describes the scheme with a symmetric pairing. As described in Section 2.2.5, a symmetric pairing is a pairing in which there is only one source group G. We use an asymmetric pairing, thus, a pairing in which there are two source groups G_1 and G_2. Most of the time elements from G_1 have around half the size of elements from G_2. If we use an asymmetric pairing in the Lewko-Waters HIBE we can let one group represent a specific part of the HIBE, e.g., public keys, and the other group represent another part. This way we can influence the size of the resulting HIBE parts. We can use two possible strategies: Set the ciphertexts in G_1 and the secret keys in G_2. This

shortens the ciphertexts but leads to larger secret keys. Alternatively, we can set the secret keys in G_1 and the ciphertexts in G_2. This makes the secret keys smaller but leads to larger ciphertexts. We choose the first strategy because we are more interested in shorter ciphertexts than smaller secret keys. The last thing we have to decide before describing the scheme is the size of the dimensions for the Dual Pairing Vector Spaces (DPVS). Lewko [58] uses a dimension of 10. Similar to Guillevic [41] we set the dimension to 6. This is possible because Lewko [58] uses dimensions 7-10 only for the semi-functional space and the ephemeral semi-functional space. These semi-functional objects are only used in the security proof and Lewko [58] stresses ‘that these algorithms are used for definitional purposes, and are not part of the real system.’ [58] For this reason, the semi-functional and the ephemeral semi-functional objects are not part of the actual HIBE algorithms which further emphasizes that they are not needed.
The algorithms of the HIBE are performed as follows [41, 58]:
PP,MSK $\Leftarrow$ Setup(): Let G_1, G_2, G_T be groups of prime order p with a bilinear map $e : G_1 \times G_2 \to G_T$. Let g_i be a generator of G_i. We set the dimension $n = 6$. The algorithm samples random dual orthonormal bases $(\mathbb{D}, \mathbb{D}^*) \leftarrow \mathit{Dual}(\mathbb{Z}_p^6)$. Let $\vec{d}_1, ..., \vec{d}_6$ denote the elements of $\mathbb{D}$, and $\vec{d}_1^*, ..., \vec{d}_1^*$ the elements of $\mathbb{D}^*$. It also chooses random exponents $\alpha_1, \alpha_2, \theta, \sigma, \gamma, \xi \in \mathbb{Z}_p$. The public parameters are

$$\mathrm{PP} := \{G_1, G_2, G_T, p, e(g_1, g_2)^{\alpha_1 \vec{d}_1 \vec{d}_1^*}, e(g_1, g_2)^{\alpha_2 \vec{d}_2 \vec{d}_2^*}, g_1^{\vec{d}_1} ..., g_1^{\vec{d}_6}\}$$

and the master secret key is

$$\mathrm{MSK} := \{\alpha_1, \alpha_2, g_2^{\vec{d}_1^*}, g_2^{\vec{d}_2^*}, g_2^{\vec{d}_1^*\gamma}, g_2^{\vec{d}_2^*\xi}, g_2^{\vec{d}_3^*\theta}, g_2^{\vec{d}_4^*\theta}, g_2^{\vec{d}_5^*\sigma}, g_2^{\vec{d}_6^*\sigma}\}.$$

SK$_{\mathbf{ID}}$ $\Leftarrow$ KeyGen(MSK,(ID$_1$, ...ID$_j$)): The key generation algorithm chooses random values $r_1^i, r_2^i \in \mathbb{Z}_p$ for each i from 1 to j. It also chooses random values $y_1, ..., y_j \in \mathbb{Z}_p$ and $w_1, ..., w_j \in \mathbb{Z}_p$ up to the constraints that $y_1+y_2+...+y_j = \alpha_1$ and $w_1+w_2+...+w_j = \alpha_2$. For each i from 1 to j, it computes:

$$K_i := g_2^{y_i \vec{d}_1^* + w_i \vec{d}_2^* + r_1^i \mathrm{ID}_i \theta \vec{d}_3^* - r_1^i \theta \vec{d}_4^* + r_2^i \mathrm{ID}_i \sigma \vec{d}_5^* - r_2^i \theta \vec{d}_6^*}.$$

The secret key is formed as:

$$\mathrm{SK_{ID}} := \{g_2^{\vec{d}_1^*\gamma}, g_2^{\vec{d}_2^*\xi}, g_2^{\vec{d}_3^*\theta}, g_2^{\vec{d}_4^*\theta}, g_2^{\vec{d}_5^*\sigma}, g_2^{\vec{d}_6^*\sigma}, K_1, ..., K_j\}.$$

SK$_{\mathbf{ID}|\mathbf{ID}_{j+1}}$ $\leftarrow$ Delegate(SK$_{\mathbf{ID}}$, ID$_{j+1}$): The delegation algorithm chooses random values $\omega_1^i, \omega_2^i \in \mathbb{Z}_p$ for each i from 1 to $j+1$. It also chooses random values $y_1', ..., y_{j+1}' \in \mathbb{Z}_p$ and $w_1', ..., w_{j+1}' \in \mathbb{Z}_p$ subject to the constraints that $y_1', ..., y_{j+1}' =$

$0 = w'_1, ..., w'_{j+1}$. With the elements from $\mathrm{SK}_{\mathrm{ID}}$ $\mathrm{SK}_{\mathrm{ID}|\mathrm{ID}_{j+1}}$ is formed as:

$$\begin{aligned}\mathrm{SK}_{\mathrm{ID}|\mathrm{ID}_{j+1}} := \{ & g_2^{\vec{d}_1^*\gamma}, g_2^{\vec{d}_2^*\xi}, g_2^{\vec{d}_3^*\theta}, g_2^{\vec{d}_4^*\theta}, g_2^{\vec{d}_5^*\sigma}, g_2^{\vec{d}_6^*\sigma} \\ & K_1 * g_2^{y'_1\vec{d}_1^* + w'_1\vec{d}_2^* + \omega_1^1 \mathrm{ID}_1\theta\vec{d}_3^* - \omega_1^1\theta\vec{d}_4^* + \omega_2^1\mathrm{ID}_1\sigma\vec{d}_5^* - \omega_2^1\theta\vec{d}_6^*}, \\ & ..., K_j * g_2^{y'_j\vec{d}_1^* + w'_j\vec{d}_2^* + \omega_1^j \mathrm{ID}_j\theta\vec{d}_3^* - \omega_1^j\theta\vec{d}_4^* + \omega_2^j\mathrm{ID}_j\sigma\vec{d}_5^* - \omega_2^j\theta\vec{d}_6^*}, \\ & g_2^{y'_{j+1}\vec{d}_1^* + w'_{j+1}\vec{d}_2^* + \omega_1^{j+1} \mathrm{ID}_{j+1}\theta\vec{d}_3^* - \omega_1^{j+1}\theta\vec{d}_4^* + \omega_2^{j+1}\mathrm{ID}_{j+1}\sigma\vec{d}_5^* - \omega_2^{j+1}\theta\vec{d}_6^*} \}.\end{aligned}$$

CT $\Leftarrow$ Encrypt(PP,M,(ID$_1$, ...ID$_j$)): The encryption algorithm chooses random values $s_1, s_2 \in \mathbb{Z}_p$, as well as random values t_1^i, t_2^i for each i from 1 to j. It computes:

$$C_0 := Me(g_1, g_2)^{\alpha_1 s_1 \vec{d}_1 \vec{d}_1^*} e(g_1, g_2)^{\alpha_2 s_2 \vec{d}_2 \vec{d}_2^*},$$

as well as

$$C_i := g_1^{s_1\vec{d}_1 + s_2\vec{d}_2 + t_1^i * \vec{d}_3 + \mathrm{ID}_i t_1^i \vec{d}_4 + t_2^i \vec{d}_5 + \mathrm{ID}_i t_2^i \vec{d}_6}$$

for each i from 1 to j. The ciphertext is $\mathrm{CT} := \{C_0, C_1, ..., C_j\}$.
M $\leftarrow$ Decrypt(CT, SK$_{\mathbf{ID}}$): The decryption algorithm computes

$$B := \prod_{i=1}^{j} e_6(C_i, K_i)$$

and computes the message as

$$M := C_0/B.$$

Now we describe the algorithm to generate random dual orthonormal bases as described by Zhang [92] and Okamoto and Takashima [74, 75]:
$(\mathbb{D}, \mathbb{D}^*) \leftarrow$ **Dual**$(\mathbb{Z}_p^n)$**:** Let G_1, G_2, G_T be groups of prime order p with a bilinear map $e : G_1 \times G_2 \to G_T$. Let g_i be a generator of G_i. To generate random dual orthonormal bases $\mathbb{D}$ and $\mathbb{D}^*$ perform:

1. Create canonical bases $\mathbb{A}$ and $\mathbb{A}^*$:
   $\mathbb{A} := \vec{a}_1, ..., \vec{a}_n$ where $\vec{a}_1 := (g_1, 1, ..., 1)$, $\vec{a}_2 := (1, g_1, 1, ..., 1)$,..., $\vec{a}_n := (1, ..., 1, g_1)$. $\mathbb{A}^* := \vec{a}_1^*, ..., \vec{a}_n^*$ where $\vec{a}_1^* := (g_2, 1, ..., 1)$, $\vec{a}_2^* := (1, g_2, 1, ..., 1)$,..., $\vec{a}_n^* := (1, ..., 1, g_2)$.

2. Generate randomly chosen linear transformation:
   Choose $X := (\mathcal{X}_{i,j}) \xleftarrow{\$} GL(n, \mathbb{Z}_p)$ and compute $\vartheta_{i,j} := (X^T)^{-1}$. Note that $GL(n, \mathbb{Z}_p)$ denotes the general linear group of dimension n over $\mathbb{Z}_p$ [9]. In other words X is a $n \times n$ matrix filled with random elements in $\mathbb{Z}_p$ and ϑ is the inverse of the transposed matrix X.

3. Perform a base change on $\mathbb{A}$ and $\mathbb{A}^*$:
   Canonical base $\mathbb{A}$ is changed to $\mathbb{D}$ by computing $\vec{d}_i := \sum_{i=1}^n \mathcal{X}_{i,j} a_j$ and canonical base $\mathbb{A}^*$ is changed to $\mathbb{D}^*$ by computing $\vec{d}_i^* := \sum_{i=1}^n \vartheta_{i,j} a_j^*$.
   Output : $(\mathbb{D}, \mathbb{D}^*)$

Note that if we generate the dual orthonormal bases this way $\vec{d}_1 = g_1^{\vec{d}_1}, ..., \vec{d}_6 = g_1^{\vec{d}_6}$ and $\vec{d}_1^* = g_2^{\vec{d}_1^*}, ..., \vec{d}_6^* = g_2^{\vec{d}_6^*}$. So the public parameters for the Lewko-Waters HIBE look like this:

$$\begin{aligned} \mathrm{PP} := \{&G_1, G_2, G_T, p, \\ e(\vec{d}_1, \vec{d}_1^*)^{\alpha_1} &= e(g_1, g_2)^{\alpha_1 \vec{d}_1 \vec{d}_1^*}, \\ e(\vec{d}_2, \vec{d}_2^*)^{\alpha_2} &= e(g_1, g_2)^{\alpha_2 \vec{d}_2 \vec{d}_2^*}, \\ &\vec{d}_1 ..., \vec{d}_6\} \end{aligned}$$

This applies to all parameters.

3.7.2 CPA to CCA Transformation

Now it is left to describe the CPA to CCA transformation described by Boneh et al. [21]. We directly describe the more efficient transformation using a MAC and an encapsulation scheme [21]. Note that there is also a transformation using a one-time signature. Since this transformation is only required for our specific choice of the HIBE, we do not discuss the choices for the algorithms used in the transformation, but simply list the requirements on the algorithms and our choice of the algorithm. Note that there might be several (more efficient) alternatives for the choices we make.
For the transformation we need a strong one-time message authentication code and an encapsulation scheme that is *binding* and *hiding* [21]. As for the other algorithms we can also choose a general strong MAC instead of an one-time MAC. We choose to use HMAC which is proven to be a Pseudo Random Function (PRF) if it is instantiated with a hash function that has a (weak) collision resistance [13] and thus is SUF secure [15]. Since all hash functions described in Section 3.3 are considered cryptographically secure, we can choose any of those hash functions. We choose SHA256. The encapsulation scheme is based on any universal one-way hash function (UOWHF) family $\{H_s : \{0,1\}^{k_1} \to \{0,1\}^k\}$ (where $k_1 > 3k$ is a function of the parameter k). It is described as follows [21]:

Init(k) chooses a hash function h from a family of pairwise-independent hash functions mapping k_1 bit strings to k bit strings, and also chooses a random key s defining H_s. It outputs $pub := (h, s)$.

Encapsulation S(pub) chooses a random $x \in \{0,1\}^{k_1}$ and then outputs $(r := h(x), \mathrm{com} := H_s(x), \mathrm{dec} := x)$

Recovery R(pub,com,dec) outputs $h(dec)$ if $H_s(\mathrm{dec}) = \mathrm{com}$, and $\perp$ otherwise.

This scheme is binding and hiding under the assumption that H_s is a UOWHF family. As described by Boneh et al. [21] and, Naor and Yung [73] we can use a second-preimage resistant hash function to construct a UOWHF family. This means we

can use any hash function from Section 3.3 (with modifications to support the input length) to construct a encapsulation scheme. Now we describe the transformation to obtain a CCA secure HIBE from any CPA secure HIBE. Note that Boneh et al. [21] only show this transformation for an IBE, we apply this transformation to the HIBE:
Let H=(Setup, KeyGen, Delegate, Enc, Dec) be a CPA secure HIBE, M=(Mac, Vrfy) be a strong MAC, and E=(Init,S,R) be a binding and hiding encapsulation scheme. For an identity $\overrightarrow{\mathrm{ID}} := (id_1, ...id_L) \in (\{0,1\}^n)^L$ in H, let the procedure encode($\overrightarrow{\mathrm{ID}}$) denote:

$$\mathrm{encode}(\overrightarrow{\mathrm{ID}}) := \{0\mathrm{id}_0, ..., 0\mathrm{id}_L\} \in (\{0,1\}^{n+1})^L$$

The construction for a CCA secure HIBE H' is shown in Figure 3.9.

```
Setup:
  (PP,MSK)⇐H.setup()
  pub ⇐E.Init()
  return (MSK, PP'=(PP,pub))

KeyGen(MSK, $\mathrm{ID}_1, ..., \mathrm{ID}_j$)
  for i = 0 to j
    $\mathrm{ID}'_i \leftarrow Encode(\mathrm{ID}_i)$
  return H.KeyGen(MSK,$\mathrm{ID}'_1, ..., \mathrm{ID}'_j$)

Delegate($\mathrm{SK}_{ID}, \mathrm{ID}_{j+1}$)
  $\mathrm{ID}'_{j+1} \leftarrow Encode(\mathrm{ID}_{j+1})$
  return H.Delegate($\mathrm{SK}_{ID}$,$\mathrm{ID}'_{j+1}$)

Encrypt(PP,M,($\mathrm{ID}_1, ...\mathrm{ID}_j$)):
  (PP',pub) ⇐ PP
  (r,com,dec) ⇐ E.S(pub)
  for i = 0 to j
    $\mathrm{ID}'_i \leftarrow Encode(\mathrm{ID}_i)$
  $\mathrm{ID}'_{j+1}$ := '1com'
  C ⇐ H.Encrypt(PP', M ∘ dec, $\mathrm{ID}_1, ...\mathrm{ID}_{j+1}$)
  tag ⇐ M.Mac(r, C)
  ret CT = (C, com, tag)

 Decrypt(CT, $\mathrm{SK}_{ID}$)
   (C, com, tag) ← CT
   ID := '1com'
   ID' ← Encode(ID)
   $\mathrm{SK}_{ID'}$ ⇐ H.delegate($\mathrm{SK}_{ID}$, ID')
   M ∘ dec ← H.decrypt(CT, $\mathrm{SK}_{ID'}$)
   r ← E.R(pub,com,dec)
   if r ≠ 0 and M.Vrfy(C,tag,r)=1
     ret M
```

Figure 3.9: CCA secure HIBE H' from CPA secure HIBE H, MAC M, and encapsulation scheme E

3.7.3 Summary

To our knowledge there are no directly usable OW-CCA secure HIBE. Both unbounded HIBE, namely the Gentry-Silverberg HIBE [37] and the Lewko-Waters HIBE [59], we discussed in this section are only CPA secure. However, we can use transformations [21, 37] to make them CCA secure. We choose to implement the prime-order translation of the Lewko-Waters HIBE [58] (described in Section 3.7.1) and use the transformation described by Boneh et al. [21] (described in Section

3.7.2).
Recommended pairing friendly curves and their corresponding estimated symmetric-equivalent strength are shown in Table 3.8. We compare their efficiency and their impact on the performance in Chapter 6.

3.8 Conclusion

In this chapter, we discussed the possible choices for the algorithms which we use to instantiate the BRKE construction. Except for the one-time signature and the HIBE we rely on external implementations that provide the algorithms. Unfortunately, we have to implement the signature and the HIBE ourselves because to our knowledge there are no implementations of those schemes suitable for our purpose. Table 3.10 shows the final choice of all schemes. There exists at least one alternative to

Primitive	Chosen Algorithm
One-Time Signature	DLP-Based Chameleon Hash OT Signature
Key Encapsulation Mechanism	ECIES-KEM
Random Oracle	HKDF
HIBE	Lewko-Waters HIBE + CCA Transformation
Hash	SHA256, SHA512

Table 3.10: Set of algorithms we chose in this chapter

every algorithm. This might be interesting if we want to compare the efficiency and performance between different algorithm sets.

4 Generic Implementation

In this chapter, we explain the implementation of the generic BRKE construction. The motivation for implementing BRKE in two parts is modularity and flexibility of the implementation. It should be possible to replace any of the used primitives without affecting the functionality of the construction. This approach enables us to replace an algorithm if it becomes insecure, or change algorithm parameters without affecting other parts of the construction. We start by providing the specifications of the used programs to compile and run the implemented code. After that, we explain the general approach to the implementation. We then explain the reasoning for some formal changes to the BRKE construction and describe those modifications.

4.1 Specifications

The implementation is configured to run on Linux and to be buildable by Maven [71], a software tool for Java project management and build automation. The goal for the generic BRKE implementation is that it does not rely on external libraries so that it is as flexible as possible. The only external library we use in the generic part of the implementation is the JUnit library to perform unit tests. The implementation is tested on the operating system Ubuntu 18.04.2 with Maven version 3.5.2 and Java version 1.8.0_201. All code is uploaded to a Github repository.

4.2 General Approach

Figure 4.1 shows the general idea behind the generic implementation. The idea is to provide the BRKE construction with an *algorithm set* which holds all primitives. More precisely the algorithm set holds a factory for each primitive which creates an object of the corresponding primitive. This way we can easily replace any of the primitives. In the generic implementation, all of those objects are implemented as interfaces to ensure that the actual instantiation provides the desired functionality. Now we start by explaining changes we applied to the BRKE construction to represent an actual real-world application.

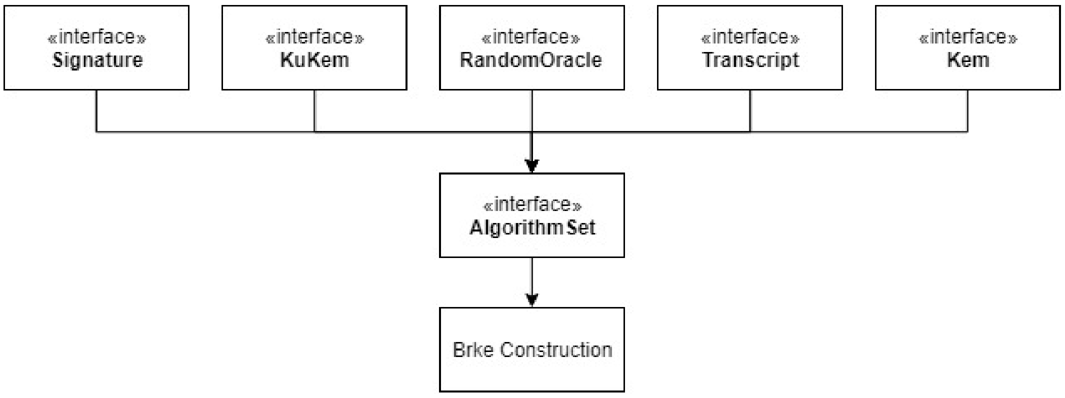


Figure 4.1: Rough structure for the generic BRKE implementation.

4.3 The BRKE Ad-Hoc Construction

To better understand the BRKE construction we analyze its functionality in this section (for readability we assume communication in the A-to-B direction, but of course everything can be applied to the B-to-A direction, as well):

States: Each participant has a sending state S and a receiving state R. When both users are in sync (have received all messages) the sending state S_A (of user A) reflects the receiving state R_B (of user B) and vice versa. This means, for example, that the public keys that are saved in S_A match the secret keys in R_B. Each state consists of the following elements: kuKEM keys (public keys in S, secret keys in R), the epoch variable (each consisting of first supported epoch $E^{\vdash}$, and last supported epoch $E^{\dashv}$), message counter (sent messages s, and received messages r), state variables L (which contain past transcript fragments), a signature key (verification key in S, signing key in R), chaining key K (which is used in the random oracle), and a transcript (sending transcript t_S, and receiving transcript t_R).

Epochs: One of the main difficulties in an asynchronous communication setting is that A cannot be sure which messages are received by B and how many messages B sent after B's last sent message. For this reason, A and B each have an epoch variable E (E_S in the sending state, and E_R in the receiving state) that keeps track of the supported epochs. Roughly speaking the epochs keep track of 'floating' ciphertexts, so ciphertexts for which A cannot be sure that B has received them, and 'active' kuKEM keys. A and B include the information of their last supported epoch (Fig. 4.2, L.20) and the number of the total received messages r (Fig. 4.2, L.18) in the ciphertext. So every time A receives a message, A has the information what messages were received by B and can update the state accordingly. So the purpose of the epochs is to keep track of sent and received messages so that the states can be updated in

sync with the communication partner.
Every time a message is sent an epoch starts in E_R (so for each saved kuKEM secret key there is an epoch), and this epoch is active as long as A did not get the information that B has received the ciphertext. An epoch in E_R ends if B used the corresponding kuKEM key. Moreover, every time a message is received an epoch starts in E_S (so for each kuKEM public key there is an epoch), and this epoch is active as long as A does not send any messages. As soon as A sends a message and uses the kuKEM public key the epoch ends.

Key Updates: Key updates are only performed in the receiving algorithm of the BRKE construction (Fig. 4.2, L.43/60). Every kuKEM public key A receives in a ciphertext is updated from the state variable L_S which contains ciphertexts sent by A (concatenated with the corresponding associated data). The public key is updated one time for each message B did not receive yet. If B is in sync the public key is not updated. Secret keys are updated with the current ciphertext received by A (concatenated with the corresponding associated data). For every message B did not receive yet A updates a secret key. All secret keys for which A can be sure that B received the messages (ensured by the epochs) are discarded. So in general, for every 'floating' ciphertext there exists a kuKEM key pair and every time a message is received, A updates its keys for every additional 'floating' ciphertext. As it turns out every kuKEM key that is the first of an epoch can be a KEM key [77].

Transcripts: Both user maintain a sending transcript t_S and a receiving transcript t_R. t_S represents t_R of the communication partner and vice versa. For this reason, Poettering and Rösler [77] introduced labels $\triangleleft$ and $\triangleright$ to keep track of the direction of the ciphertext. These transcripts are included in the input for the random oracle to ensure further that both users only can generate the correct keys if they update their state correctly.

State Variables L: The state variables L_S and L_R store information that is needed at a later point in the BRKE construction. The variable L_S stores ciphertext associated data pairs for public key updates (Fig. 4.2, L.31/43) and the variable L_R stores transcript fragments which are used to update the receiving transcript t_r (Fig. 4.2, L.25/49). So the state variables are used to store information until A can be sure that the B has received the corresponding ciphertexts.

This concludes the description of the key points of the BRKE construction. Every other operation in the BRKE construction in Figure 4.2 should be straightforward.

```
**Proc** init
00   For $u \in \{A, B\}$:
01      $(sgk_u, vfk_u) \leftarrow_\$ \mathrm{gen}_\mathsf{S}$
02      $(sk_u, pk_u) \leftarrow_\$ \mathrm{gen}_\mathsf{K}$
03      $K_u \leftarrow_\$ \mathcal{K}; t \leftarrow \epsilon$
04      $E^{\vdash} \leftarrow 0; E^{\dashv} \leftarrow 0$
05      $s \leftarrow 0; r \leftarrow 0$
06      $PK_u[\cdot] \leftarrow \bot; PK_u[0] \leftarrow pk$
07      $SK_u[\cdot] \leftarrow \bot; SK_u[0] \leftarrow sk$
08      $L_S[\cdot] \leftarrow \bot; L_R[\cdot] \leftarrow \bot; L_S[0] \leftarrow \diamond$
09      $S_u \leftarrow (PK_{\bar{u}}, E, s, L_S, vfk_{\bar{u}}, K_{\bar{u}}, t)$
10      $R_u \leftarrow (SK_u, E, r, L_R, sgk_u, K_u, t)$
11      $ST_u \leftarrow (R_u, S_u)$
12   Return $(ST_A, ST_B)$

**Proc** snd$(ST, ad)$
13   $(R, S) \leftarrow ST$
14   $(SK, E_R, r, L_R, sgk, K_R, t_R) \leftarrow R$
15   $(sk^*, pk^*) \leftarrow_\$ \mathrm{gen}_\mathsf{K}$
16   $(sgk^*, vfk^*) \leftarrow_\$ \mathrm{gen}_\mathsf{S}$
17   $E_R^{\dashv} \leftarrow E_R^{\dashv} + 1; SK[E_R^{\dashv}] \leftarrow sk^*$
18   $C \leftarrow r \,\|\, pk^* \,\|\, vfk^*$

19   $(PK, E_S, s, L_S, vfk, K_S, t_S) \leftarrow S$
20 · $k^* \leftarrow \epsilon; C \overset{\shortparallel}{\leftarrow} E_S^{\dashv}$
21   For $e' \leftarrow E_S^{\vdash}$ to $E_S^{\dashv}$:
22      $(k, c) \leftarrow_\$ \mathrm{enc}(PK[e'])$
23      $k^* \overset{\shortparallel}{\leftarrow} k; C \overset{\shortparallel}{\leftarrow} c$

24   $\sigma \leftarrow_\$ \mathrm{sgn}(sgk, ad \,\|\, C)$
25 · $C \overset{\shortparallel}{\leftarrow} \sigma; L_R[E_R^{\dashv}] \leftarrow \triangleright_u \,\|\, ad \,\|\, C$
26   $R \leftarrow (SK, E_R, r, L_R, sgk^*, K_R, t_R)$

27   $t_S \overset{\shortparallel}{\leftarrow} \triangleright_u \,\|\, ad \,\|\, C$
28   $k.o \,\|\, K_S \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(K_S, k^*, t_S)$
29   $pk \leftarrow \mathrm{gen}_\mathsf{K}(sk)$
30   $PK[..., (E_S^{\dashv} - 1)] \leftarrow \bot; PK[E_S^{\dashv}] \leftarrow pk$
31   $E_S^{\vdash} \leftarrow E_S^{\dashv}; s \leftarrow s + 1; L_S[s] \leftarrow ad \,\|\, C$
32   $S \leftarrow (PK, E_S, s, L_S, vfk, K_S, t_S)$
33   $ST \leftarrow (R, S)$
34   Return $(ST, k.o, C)$

**Proc** rcv$(ST, ad, C)$
35   $(R, S) \leftarrow ST$
36   $(PK, E_S, s, L_S, vfk, K_S, t_S) \leftarrow S$
37 · $t^* \leftarrow ad \,\|\, C; t_S \overset{\shortparallel}{\leftarrow} \triangleright_{\bar{u}} \,\|\, t^*; C \,\|\, \sigma \leftarrow C$
38   Require $\mathrm{vfy}_\mathsf{S}(vfk, ad \,\|\, C, \sigma)$
39 · $r \,\|\, pk^* \,\|\, vfk \,\|\, C \leftarrow C$
40   Require $L_S[r] \neq \bot$
41   $L_S[..., (r - 1)] \leftarrow \bot; L_S[r] \leftarrow \diamond$
42   For $s' \leftarrow r + 1$ to $s$:
43      $pk^* \leftarrow \mathrm{up}(pk^*, L_S[s'])$
44   $E_S^{\dashv} \leftarrow E_S^{\dashv} + 1; PK[E_S^{\dashv}] \leftarrow pk^*$
45   $S \leftarrow (PK, E_S, s, L_S, vfk, K_S, t_S)$

46   $(SK, E_R, r, L_R, sgk, K_R, t_R) \leftarrow R$
47   $k^* \leftarrow \epsilon; e \,\|\, C \leftarrow C$
48   Require $E_R^{\vdash} \leq e \leq E_R^{\dashv}$
49   $t_R \overset{\shortparallel}{\leftarrow} L_R[E_R^{\vdash} + 1] \,\|\, \ldots \,\|\, L_R[e]$
50   $L_R[..., e] \leftarrow \bot$
51   For $e' \leftarrow E_R^{\vdash}$ to $e$:
52      $c \,\|\, C \leftarrow C$
53      $k \leftarrow \mathrm{dec}(SK[e'], c)$
54      Require $k \neq \bot$
55      $k^* \overset{\shortparallel}{\leftarrow} k$
56 · $t_R \overset{\shortparallel}{\leftarrow} \triangleright_{\bar{u}} \,\|\, t^*$
57   $k.o \,\|\, K_R \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(K_R, k^*, t_R)$
58   $SK[..., (e - 1)] \leftarrow \bot; SK[e] \leftarrow sk$
59   For $e' \leftarrow e + 1$ to $E_R^{\dashv}$:
60      $SK[e'] \leftarrow \mathrm{up}(SK[e'], t^*)$
61   $E_R^{\vdash} \leftarrow e; r \leftarrow r + 1$
62   $R \leftarrow (SK, E_R, r, L_R, sgk, K_R, t_R)$
63   $ST \leftarrow (R, S)$
64   Return $(ST, k.o)$
```

Figure 4.2: Description of the ad hoc BRKE construction [77].

4.4 Protocol Changes

In Section 4.3 we described the BRKE construction as it is proposed by Poettering and Rösler [77]. We apply some formal changes to the construction to represent a real-world application for the protocol:

General: Instead of using BRKE as an algorithm as described by Poettering and Rösler [77] we use the BRKE construction as an object. This better represents the

principle of object-oriented programming in Java. Furthermore, this enables us to better structure the program. The general idea is to keep the information where it is used. An example of this is the user state. In our implementation the user state is saved inside the BRKE object instead of having a user state class which holds all variables.
Another big change is the general approach to updates and storing past communication fragments. It would be unpleasant to have endlessly growing arrays and transcripts because memory is limited. For this reason, we have to change how we handle transcripts and the general approach to the epochs (which are always growing in the ad-hoc BRKE construction).
States: As already mentioned we omitted the user states described by Poettering and Rösler [77] and replaced them with a BRKE state.

```
1  private SignatureManager signatureAlgorithm;
2  private KeyedRandomOracle randomOracleAlgorithm;
3  private QueuedKuKem queuedKuKemAlgorithm;
4  private int numberOfUnsynchronizedSentMesssages;
5  private int numberOfUnsynchronizedReceivedMessages;
6  private Transcript receivingTranscript;
7  private Transcript sendingTranscript;
8  private boolean initiator;
```

Listing 4.3: New BRKE 'User State'

Listing 4.3 shows the new state of a BRKE object. We shortly address some changes but elaborate them further in their respective paragraph. All keys are now managed by the algorithm which uses the keys (Lines 1-3). So the classes SignatureManager, KeyedRandomOracle, and QueuedKuKem contain keys of the user itself and the corresponding keys of the communication partner. Furthermore, these classes are responsible for updating the keys according to the BRKE description. We explain this approach further in Section 4.5.
The variables in Lines 4 and 5 combine the epoch variables E_R and E_S, and the variables r and s from the BRKE construction (Fig. 4.2).
Every user still has two transcripts T_R and T_S, but their use is changed a little bit (Lines 6-7). Lastly, every user has an *initiator* variable (Line 8) which stores the information if a user is the initiator of a conversation.
Epochs: In the ad-hoc BRKE construction the values in the epochs are always growing. Since we want to avoid this, so we do not have endlessly growing arrays, we have to rework the approach to epochs a little bit. For this reason we introduce the variables *numberOfUnsynchronizedSentMesssages* and *numberOfUnsynchronizedReceivedMessages* (Lines 4,5). As mentioned in Section 4.3 one function of the epochs is to keep track of the 'floating' ciphertexts. The two variables do the same thing. The other function, keeping track of kuKEM keys, is now managed by the queued kuKEM so we do not need this functionality outside of it.
Key Updates: Key updates are now directly managed by the queued kuKEM algorithm which we explain in Section 4.5. So the BRKE construction does not need

to handle these.
State Variables L: As described in Section 4.3 the state variables L_S and L_R are used to store information that is needed at a later time. These functions are also taken care of by the use of queued algorithms. Instead of storing the information in a separate class and obtaining it if we need it, we directly store the information where it is needed and trigger the update from a queue.

4.5 Queue-Based Algorithms

In this section, we explain the idea behind the queue based algorithms and the queued kuKEM. As already introduced in Section 4.4, we use the queued algorithms to avoid the epochs and lists of the ad-hoc brke construction (Fig. 4.2). The epochs and the two state variables L_S and L_R handle the state updates. For this reason, we have to be able to ensure correct state updates without using either of those. We have a look at three examples for the use of the epochs:

- Fig. 4.2, Line 17: We add the generated kuKEM secret key to the kuKEM secret key list as the last key of the supported epoch E_R. So this means we add a secret key to the secret key list.

- Fig. 4.2, Line 20/21: We put the last supported epoch $E_S^{\dashv}$ in the ciphertext and then use the kuKEM keys of the supported epoch E_S for the encapsulation. In other words, we save the number of used keys in the ciphertext and then use all currently stored keys for the encapsulation.

- Fig. 4.2, Line 25: We add the currently generated transcript fragment in the state variable L_R as the last transcript fragment of the supported epoch E_R. Similar to the secret key we add the transcript fragment to the state variable L_R.

So, in general, we use the epochs E_S and E_R to determine which position of the different lists we have to access. This also means the lists only contain elements from $E^{\vdash}$ to $E^{\dashv}$. Also, except for the public key updates, all accesses to lists start at $E^{\vdash}$ and the content is deleted after the access. As result queues are an ideal data structure to manage state updates.
For this reason, we replace all lists except L_S by queues. L_S is used for public key updates, and we do not delete the elements of L_S after we accessed them, so we keep it a regular list. We also move the state variables L_S and L_R inside the objects where they are required. So we move L_R inside the transcript object and L_S inside the kuKEM object. We also introduce a special kind of kuKEM for the use inside the BRKE object, the *queued kuKEM*.

4.5.1 Queued kuKEM

The queued kuKEM is used to manage queued key updates, encapsulations, and decapsulations. The state of a queued kuKEM is shown in Listing 4.4.

```java
private KeyUpdateableKem kuKemAlgorithm;
private KeyEncapsulationMechanism kemAlgorithm;
private KuKemAssociatedDataFactory associatedDataFactory;
private KemSecretKey kemSecretKey;
private KemPublicKey communicationPartnerKemPublicKey;
private Queue<KuKemSecretKey> secretKeys;
private Queue<KuKemPublicKey> communicationPartnerPublicKeys;
private LinkedList<KuKemAssociatedData> publicKeyUpdateInformationList;
```

Listing 4.4: Queued kuKEM state

Internally it uses a KEM algorithm and a kuKEM algorithm. As described in Section 4.3 the first key of an epoch can be a KEM key. The *publicKeyUpdateInformationList* represents the state variable L_S and stores *KuKemAssociatedData*. We use *KuKemAssociatedData* instead of a ciphertext associated data pair so that the kuKEM implementation can be used outside of the BRKE project.
Listing 4.5 shows the encapsulation procedure of the queued kuKEM.

```java
1     KemOutput kemOutput = kemAlgorithm.encapsulate(communicationPartnerKemPublicKey)
2     communicationPartnerKemPublicKey = null;
3     int numberOfEncapsulations = 1;
4     SymmetricKey generatedKey = kemOutput.getKey();
5     if (!communicationPartnerPublicKeys.isEmpty()) {
6       Queue<KuKemCiphertext> ciphertext = new LinkedList<KuKemCiphertext>();
7       while (!communicationPartnerPublicKeys.isEmpty()) {
8         KuKemOutput kuKemOutput = kuKemAlgorithm.encapsulate(communicationPartnerPublicKeys.
              poll());
9         numberOfEncapsulations++;
10        generatedKey.mixToKey(kuKemOutput.getKey());
11        ciphertext.add(kuKemOutput.getCiphertext());
12      }
```

Listing 4.5: Queued kuKEM encapsulation (shortened)

Lines 1-3: Encapsulate a key using the regular KEM. Delete the KEM public key afterward. Also initialize the *numberOfEncapsulations* variable with 1.

Line 5: If there are saved kuKEM keys this means we have floating ciphertexts and have to encapsulate more than one time.

Lines 6-12: Encapsulate a key with every saved kuKEM public key. Since we *poll* the keys from the queue, they are deleted afterward. The symmetric key is constructed by mixing all keys (Line 9). Ciphertexts are saved in a queue (Lines 5,10).

The final ciphertext always consists of at least a KEM ciphertext and the variable *numberOfEncapsulations*. If *numberOfEncapsulations*>1 then the ciphertext also contains a kuKEM ciphertext queue of size *numberOfEncapsulations*-1. This procedure implements Lines 20-23 and Line 30 of the ad hoc BRKE construction.

```
1   if (numberOfUsedKeysForEncapsulation == 1) {
2     SymmetricKey generatedKey = kemAlgorithm.decapsulate(kemSecretKey, ciphertext.
          getKemCiphertext());
3     kemSecretKey = null;
4     return generatedKey;
5   } else {
6     Queue<KuKemCiphertext> ciphertexts = new LinkedList<KuKemCiphertext>(ciphertext.
          getKuKemCiphertexts());
7     SymmetricKey generatedKey = kemAlgorithm.decapsulate(kemSecretKey, ciphertext.
          getKemCiphertext());
8     kemSecretKey = null;
9     for (int i = 0; i < numberOfUsedKeysForEncapsulation - 1; i++) {
10      generatedKey.mixToKey(kuKemAlgorithm.decapsulate(secretKeys.remove(), ciphertexts.
            remove()));
11    }
12    return generatedKey;
```

Listing 4.6: Queued kuKEM decapsulation (shortened)

The decapsulation is shown in Listing 4.6. If the *numberOfUsedKeysForEncapsulation*=1 we only need to decapsulate using the regular KEM (Lines 2-4). Otherwise we decapsulate *numberOfUsedKeysForEncapsulation*-1 times. This procedure implements Lines 51-55 and Line 58 of the ad hoc BRKE construction.
Listing 4.7 shows the update public key procedure. The procedure receives three inputs: the public key to add and update, the variable *messagesReceivedByPartner*, and the variable *numberOfUpdates*.

```
while (messagesReceivedByPartner > 0) {
  publicKeyUpdateInformationList.remove();
  messagesReceivedByPartner--;
}
for (int i = 0; i < numberOfUpdates; i++) {
  publicKey = kuKemAlgorithm.updatePublicKey(publicKey, publicKeyUpdateInformationList.get
      (i));
}
communicationPartnerPublicKeys.add(publicKey);
```

Listing 4.7: Queued kuKEM public key update (shortened)

The variable *messagesReceivedByPartner* is used to determine how many elements from the public key update information list we can delete. If the communication partner has a received a message we can safely delete the corresponding update information, because it is not needed anymore. The variable *numberOfUpdates* is used to determine how many floating ciphertexts there are left because we have to update the public key for every floating ciphertext. After the key is updated, it is saved in the public key queue. This procedure implements Lines 40-44 of the ad hoc BRKE construction.

```
    kemSecretKey = kemAlgorithm.gen(seed).getSecretKey();
    Queue<KuKemSecretKey> updatedSecretKeys = new LinkedList<KuKemSecretKey>();
    KuKemAssociatedData kuKemAssociatedData = associatedDataFactory.createAssociatedData(ad,
         ciphertext);
    while (!secretKeys.isEmpty()) {
      updatedSecretKeys.add(kuKemAlgorithm.updateSecretKey(secretKeys.remove(),
           kuKemAssociatedData));
    }
```

Listing 4.8: Queued kuKEM secret key update (shortened)

Listing 4.8 shows the update secret key procedure. It receives a secret key seed, and a ciphertext associated data pair. The procedure then generates a regular KEM secret key and updates all secret keys in the kuKEM secret key queue using the ciphertext associated data pair. This procedure implements Lines 58-61 of the ad hoc BRKE construction.

4.6 The Modified BRKE Protocol

Now that we described the queued algorithms and reasons for our modifications we describe the modified BRKE protocol. When initializing a BRKE object the user needs to provide a source of randomness and the information if the user is the initiator of the conversation or not. The source of randomness has to match on the user's and its communication partner's side. This way we can ensure that the user and the communication partner generate matching keys. The source of randomness can be seeded with keying material from an initial key agreement protocol, for example. Listing 4.9 shows the send procedure of the BRKE object.

```
  public BrkeSendOutput send(SecureRandom randomness, AssociatedData ad) {
    KuKemPublicKey kuKemPublicKey = queuedKuKemAlgorithm.gen(randomness);
    SignatureVerificationKey signatureVerificationKey = signatureAlgorithm.gen(randomness);
    int numberOfUsedKeysForEncapsulation = queuedKuKemAlgorithm.getNumberOfSavedPublicKeys();
    QueuedKuKemOutput kuKemOutput = queuedKuKemAlgorithm.encapsulate();
    BrkeCiphertext ciphertext = new BrkeCiphertext(numberOfUnsynchronizedReceivedMessages,
         kuKemPublicKey,
        signatureVerificationKey, numberOfUsedKeysForEncapsulation, kuKemOutput.getCiphertext
             ());
    ciphertext.computeSignature(signatureAlgorithm, ad);
    receivingTranscript.addToTranscriptQueue(initiator, ad, ciphertext);
    sendingTranscript.updateTranscript(initiator, ad, ciphertext);
    KeyedRandomOracleOutput randomOracleOutput = randomOracleAlgorithm
        .querySendRandomOracle(kuKemOutput.getGeneratedKey(), sendingTranscript);
    queuedKuKemAlgorithm.addMatchingPublicKey(randomOracleOutput.getSecretKeySeed());
    queuedKuKemAlgorithm.addToPublicKeyUpdateInformationQueue(ad, ciphertext);
```

```
15    numberOfUnsynchronizedSentMesssages++;
16    numberOfUnsynchronizedReceivedMessages = 0;
17    return new BrkeSendOutput(randomOracleOutput.getSessionKey(), ciphertext);
```

Listing 4.9: Send Operation of the modified BRKE protocol

In the following we describe the modified send procedure. We always provide the corresponding lines in the ad hoc BRKE construction shown in Figure 4.2.

Lines 2-3 Generate a kuKEM public key, and a verification key. The corresponding secret keys are directly saved in the algorithm objects. (Fig. 4.2, Lines 15-17)

Line 4 Obtain the number of currently saved public keys in the queued kuKEM. (Fig. 4.2, Line 20)

Line 5 Encapsulate a key using the queued kuKEM. (Fig. 4.2, Lines 21-23)

Lines 6-8 Create the ciphertext and sign it. The signature object internally uses the correct signing key and deletes it afterwards. (Fig. 4.2, Lines 18,20,24,25)

Lines 9-10 Update the sending transcript and add the current communication fragment to the receiving transcript update queue. (Fig. 4.2, Lines 25,27)

Line 11 Query the random oracle. (Fig. 4.2, Line 28)

Line 13-14 Generate a new public key in the queued kuKEM and add the current communication fragment to the public key update list. (Fig. 4.2, Lines 29-31)

Lines 15-16 Update *numberOfUnsynchronizedSentMesssages* and *numberOfUnsynchronizedReceivedMessages* to keep track of the current communication state. (Fig. 4.2, Line 31)

Similar to the modified send procedure we describe the modified receive procedure in the following. The modified receive procedure is shown in Listing 4.10.

```
1   public SymmetricKey receive(AssociatedData ad, BrkeCiphertext ciphertext) {
2     if (!signatureAlgorithm.verify(ad, ciphertext)) {
3       return null;
4     }
5     sendingTranscript.updateTranscript(!initiator, ad, ciphertext);
6     int communicationPartnerReceivedMessages = ciphertext.getNumberOfReceivedMessages();
7     KuKemPublicKey ciphertextKuKemPublicKey = ciphertext.getPublicKey();
8     SignatureVerificationKey ciphertextSignatureVerificationKey = ciphertext.
          getVerificationKey();
9     numberOfUnsynchronizedSentMesssages -= communicationPartnerReceivedMessages;
10    if (numberOfUnsynchronizedSentMesssages < 0) {
11      return null;
12    }
13    queuedKuKemAlgorithm.addUpdatedPublicKey(ciphertextKuKemPublicKey,
          communicationPartnerReceivedMessages, numberOfUnsynchronizedSentMesssages);
14    signatureAlgorithm.setVerificationKey(ciphertextSignatureVerificationKey);
15    receivingTranscript.updateTranscriptfromQueue(ciphertext.getNumberOfUsedKeys());
16    SymmetricKey generatedKey = queuedKuKemAlgorithm.decapsulate(ciphertext.
          getNumberOfUsedKeys(), ciphertext.getCiphertext());
```

```
17      receivingTranscript.updateTranscript(!initiator, ad, ciphertext);
18      KeyedRandomOracleOutput randomOracleOutput = randomOracleAlgorithm.
            queryReceiveRandomOracle(generatedKey, receivingTranscript);
19      queuedKuKemAlgorithm.updateSecretKeys(randomOracleOutput.getSecretKeySeed(), ad,
            ciphertext);
20      numberOfUnsynchronizedReceivedMessages++;
21      return randomOracleOutput.getSessionKey();
```

Listing 4.10: Send Operation of the modified BRKE protocol

Lines 2-4 Check the signature inside the ciphertext, return *null* if the signature is invalid. In contrast to the ad hoc BRKE construction, we move the signature verification to the beginning of the procedure. This way a faulty ciphertext can not invalidate a user's state. In the ad hoc BRKE construction the transcript is updated before the signature verification. (Fig. 4.2, Line 38)

Line 5 Update the sending transcript. (Fig. 4.2, Line 37)

Line 6-8 Unpack the ciphertext. (Fig. 4.2, Line 39)

Lines 9-12 Update *numberOfUnsynchronizedSentMesssages*. We increase this variable for every sent message by one. The ciphertext contains the variable *communicationPartnerReceivedMessages* which we then subtract from *numberOfUnsynchronizedSentMesssages*. This way we can keep track of floating ciphertexts. So in a honest BRKE run the variable cannot become < 0, because the communication partner cannot receive more messages than the user has sent. For this reason we abort if *numberOfUnsynchronizedSentMesssages* < 0, because this means the user is no longer part of a honest protocol run.

Lines 13-14 Add and update the kuKEM public key and add the verification key.(Fig. 4.2, Lines 40-45)

Line 15 Update the receiving transcript from the queue. (Fig. 4.2, Lines 49-50)

Line 16 Decapsulate a symmetric key using the queued kuKEM. (Fig. 4.2, Lines 51-55)

Line 17 Update the receiving transcript. (Fig. 4.2, Line 56)

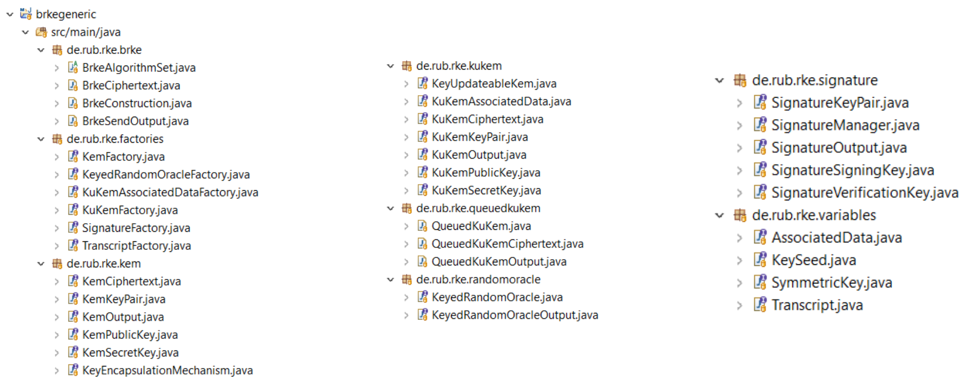
Lines 18-19 Query the random oracle and update the secret keys of the queued kuKEM. (Fig. 4.2, Lines 57-61)

Lines 20-21 Increase *numberOfUnsynchronizedReceivedMessages* by one and return the generated symmetric session key. (Fig. 4.2, Lines 61,64)

We can see that the modified BRKE procedures rely on the algorithms itself to perform correct state updates. The BRKE object only keeps track of unsynchronized sent and received messages to keep track of floating ciphertexts and trigger correct state updates.

4.7 Project Structure

The structure of the generic BRKE project is shown in Figure 4.1. In this section, we explain the purpose of the different packages. To keep the implementation as generic as possible, we often use interfaces that specify how a class should behave. Furthermore, we create interfaces for all parts of an algorithm. So for a, e.g., KEM, we have an interface for the key pair, the individual keys, the output, the ciphertext, and the algorithm itself. This way the implementation is as flexible as possible.



```
brkegeneric
  src/main/java
    de.rub.rke.brke
      BrkeAlgorithmSet.java
      BrkeCiphertext.java
      BrkeConstruction.java
      BrkeSendOutput.java
    de.rub.rke.factories
      KemFactory.java
      KeyedRandomOracleFactory.java
      KuKemAssociatedDataFactory.java
      KuKemFactory.java
      SignatureFactory.java
      TranscriptFactory.java
    de.rub.rke.kem
      KemCiphertext.java
      KemKeyPair.java
      KemOutput.java
      KemPublicKey.java
      KemSecretKey.java
      KeyEncapsulationMechanism.java
    de.rub.rke.kukem
      KeyUpdateableKem.java
      KuKemAssociatedData.java
      KuKemCiphertext.java
      KuKemKeyPair.java
      KuKemOutput.java
      KuKemPublicKey.java
      KuKemSecretKey.java
    de.rub.rke.queuedkukem
      QueuedKuKem.java
      QueuedKuKemCiphertext.java
      QueuedKuKemOutput.java
    de.rub.rke.randomoracle
      KeyedRandomOracle.java
      KeyedRandomOracleOutput.java
    de.rub.rke.signature
      SignatureKeyPair.java
      SignatureManager.java
      SignatureOutput.java
      SignatureSigningKey.java
      SignatureVerificationKey.java
    de.rub.rke.variables
      AssociatedData.java
      KeySeed.java
      SymmetricKey.java
      Transcript.java
```

Figure 4.11: Project structure of the generic BRKE construction.

The purpose of the different packages is:

brke The *brke* package contains the classes that are specific to the BRKE construction.

factories The *factories* package contains interfaces for the factories for each of the algorithms. Factories are used to create an instance of a specific algorithm.

kem The *kem* package contains interfaces for all parts of a KEM implementation. The KEM implementation should provide the functionality of a regular KEM. It does not need to handle keys itself.

kukem The *kukem* package contains interfaces for all parts of a kuKEM implementation. Similar to the KEM the implementation of the kuKEM does not need to handle keys itself. The queued kuKEM is responsible for this.

queuedkukem The *queuedkukem* package contains the implementation of the queued kuKEM, the ciphertext produced by the queued kuKEM, and the output produced by the queued kuKEM.

randomoracle The *randomoracle* package contains interfaces for all parts of a random oracle implementation. The random oracle needs to provide a *querySendRandomOracle* and a *queryReceiveRandomOracle* function. In each function, it needs to use the corresponding chaining key K_S or K_R.

signature The *signature* package contains interfaces for all parts of a signature implementation. The signature needs to provide the functionality to sign and verify a ciphertext associated data pair. For this, the signature needs to store its own signing keys and the verification keys for the communication partner.

variables The *variables* package contains interfaces for associated data, symmetric keys, key seeds, and the transcript. We use interfaces for these classes to enable the use of different kinds of symmetric keys, associated data, and so on.

In General, the interfaces are used to set requirements on the actual implementation and ensure that the BRKE object can use these classes to perform the protocol. As an example for test implementations of those interfaces, we can consult the mock implementations for the Unit tests.

4.8 Unit Tests

To test the generic BRKE implementation, we have to use actual implementations of the several interfaces. Since we avoid using actual implementations of the algorithms, we implement the algorithms as mock objects [66]. In this section, we explain the approach and the idea of the mock implementations.

- Instead of implementing the functionality of the specific algorithms we instead check if inputs were matching. So for a, e.g., KEM, we implement encapsulation by generating a key and then saving it with the inputs to the procedure as a ciphertext. If we decapsulate the ciphertext, we check if the inputs to the decapsulation are matching the inputs saved in the ciphertext. This way we can ensure that a real KEM would decapsulate the key.
- Keys are identified by an integer. If we generate a key, we generate a random integer and save it in the respective object. This way we can simulate symmetric keys and asymmetric keys without any modification. One integer generates two matching symmetric keys, and an asymmetric key pair is also generated by the same integer.
- We test each mock algorithm individually to ensure that the algorithms provide the required functionality.

With the mock classes, we can test the generic BRKE construction and ensure that it works if provided with actual algorithms.
To simulate asynchronous communication, we randomize if a user receives a message or not. We let each user send 20 messages, but randomize if the other user receives a message before sending another message. Basic pseudocode for this approach is shown in Listing 4.12. If all 40 generated keys $k_A^i = k_B^i$ and $l_A^i = l_B^i \ \forall\ 0 \leqslant i < 20$, then we assume the BRKE object works as intended.

```
for(int i=0; i<20; i++)
  $(c_A^i, k_A^i) \Leftarrow$ brkeUserA.send(ad)
  if(rng.nextBoolean)
    $k_B^i \leftarrow$ brkeUserB.receive( next unreceived $c_A^i$)
  $(c_B^i, l_B^i) \Leftarrow$ brkeUserB.send(ad)
  if(rng.nextBoolean)
    $l_A^i \leftarrow$ brkeUserA.receive( next unreceived $c_B^i$)
```

Listing 4.12: Pseudocode for the BRKE unit test

4.9 Conclusion

In this Chapter, we described the generic BRKE implementation. The motivation for splitting the implementation into two parts is to have a BRKE construction that functions independently from any real cryptographic primitives. If the BRKE construction is provided with actual working primitives, it achieves its purpose and can be used to generate secure session keys for two users. This way we can easily interchange different primitives to compare the performance or remove insecure primitives. To better represent the BRKE construction in a real-world application, we apply some formal changes to the construction described by Poettering and Rösler [77]. We replace the epochs and lists with queued algorithms. This enables us two omit the endlessly growing lists and arrays of the ad hoc BRKE construction. We also introduce a special kind of kuKEM, the queued kuKEM, which handles all key updates, encapsulations, and decapsulations. We test the generic BRKE implementation with Unit tests. For the test, we use mock implementations of the various interfaces.

5 Instantiation

In this chapter, we describe the implementation of the BRKE instantiation. We start by discussing the choices of the external libraries. After that, we describe the general approach to the implementation of the instantiation. Then we describe the key points of the different algorithm implementations.

5.1 Libraries

Our goal is to use already existing and established libraries to implement the classes for the BRKE instantiation. The first library we use is the bouncy castle library [86]. The bouncy castle API provides many implementations of various cryptographic algorithms. Unfortunately, bouncy castle does not provide an implementation of pairing-based cryptography. For this reason, we have to use an additional library for the pairings to implement the HIBE. The most known library for Java is the jPBC [30] library, which is a Java implementation of the PBC Library by Lynn [62]. However, this library is outdated and does not support (at least not efficiently) current state of the art pairing-friendly curves. As it turns out there are no pairing libraries for Java that fulfill our purpose. For this reason, we choose to use the Relic library [7], which is the current state of the art pairing implementation library [8, 39]. Furthermore, Relic is still in development, so it receives regular patches and Relic supports some of the pairing-friendly curves we described in Section 3.7. Unfortunately, Relic is a C/C++ library. For this reason, we have to use Java Native Interface (JNI) [40] to be able to use the code in Java.

5.2 Specifications

Similar to the generic BRKE implementation we use Maven to manage to build the project. We use bouncy castle version 1.61, and the relic library commit $b984e90$. The implementation is tested on the operating system Ubuntu 18.04.2 with Maven version 3.5.2 and Java version 1.8.0_201. The C++ compiler version is *gcc 7.3.0*. We use *CMake* [65] version 3.10.2 to build the relic library. All code is uploaded to a Github repository.

5.3 General Approach

Before we start describing specifics we give some general information for our implementation of the BRKE instantiation. Although, we chose specific algorithms in Chapter 3 we try to keep the actual implementations as generic as possible. This way we can easily change parameters without affecting the functionality of the algorithms. So we try to avoid hard coding parameters whenever possible. We provide the classes with the actual parameters, e.g., elliptic curves, when creating the object, thus, calling the constructor. This is the purpose of the factory classes. They create objects and provide the actual parameters. With this approach, we can easily test other parameters, because we can add another factory procedure which creates an object with different parameters.
When initializing the BRKE object, we have to make sure that two users can generate the corresponding keys of their partner. For this reason we most of the time generate the keys for both users and then discard the keys the algorithms do not need. We give an example of this approach in Section 5.4 in which we describe the One-Time Signature implementation. Since we use algorithms provided by bouncy castle some algorithm objects are straightforward, so we do not go into very much detail. For those algorithms we provide a short description in the following:

5.3.1 Random Oracle

As described in Section 3.4 we decide to implement the random oracle by the use of HKDF. The implementation is independent of the size of the internal keys and the generated keys, and the choice of the internal hash function. We use *Digest* class from bouncy castle and set the hash function as well as the key sizes when creating a random oracle object. The internal key size specifies the size of the chaining keys K_S and K_R. The generated key size specifies the size of the session key and key seed which are generated by the random oracle. Since we use the *HKDFBytesGenerator* class from bouncy castle the implementation of the random oracle is straightforward. If the random oracle is queried, we use the input to the random oracle and the corresponding chaining key (K_S for the query in the sending procedure, K_R for the query in the receiving procedure) as input to the HKDF and then generate a session key and a key seed. Furthermore, we update the chaining key.

5.3.2 Transcript

The transcript keeps track of received and sent messages and is used as one input of the random oracle. In the ad hoc BRKE description the transcript is a concatenation of all past ciphertext associated data pairs. As described in Chapter 4 we avoid using endlessly growing lists, so we have to establish a way to store all ciphertext associated data pairs without actually storing them individually. The most intuitive way to achieve this is by using a hash function. By using a cryptographic hash function, we can 'link' the ciphertext associated data pairs. For this reason, we implement the transcript by using SHA256. The receiving transcript has a queue for transcript updates, the list L_R of the ad hoc BRKE construction (Figure 4.2), so as described 4.5 we have to include an update queue in the transcript class. In order that we do not have to store all individual ciphertext associated data pairs, we hash the individual pairs before updating the transcript or storing the hash in the queue. Listing 5.1 shows the update procedure of the transcript.

```
1   if (state != null) {
2     hashFunction.reset(state);
3   }
4   byte[] hashedInput = CiphertextEncoder.hashAdCiphertext(ad, ciphertext);
5   hashFunction.update(sender ? (byte) 1 : (byte) 0);
6   hashFunction.update(hashedInput, 0, hashedInput.length);
7   state = hashFunction.copy();
```

Listing 5.1: Updating the transcript

First, we check if we have an active state. This is always the case after the first transcript update. Then we let the *CiphertextEncoder* class hash the ciphertext associated data pair. After that, we update the transcript with one byte. This byte is 1 if the user is the sender of the message, and 0 otherwise. This byte represents the labels $\triangleleft$ and $\triangleright$. In the end, we update the transcript with the hashed ciphertext associated data pair and save the state. If we add a ciphertext associated data pair to the update queue, we perform the steps from Lines 4-6, but save the hash in a queue instead of updating the transcript state. If the update from the queue is triggered we update the transcript using the hashes of the ciphertext associated data pairs, this way we do not have to store the individual pairs.

5.3.3 Utility Classes

We introduce three utility classes for our BRKE instantiation. The *CiphertextEncoder*, *SecureRandomBuilder*, and *SymmetricKeyCombiner*:

CiphertextEncoder: The *CiphertextEncoder* provides functions to encode and decode a ciphertext generated with our proposed algorithm set. It has a function to hash a ciphertext associated data pair for the *signing* and *verifying* procedure of the signature. Additionally, it provides a function to hash a ciphertext associated data pair including the signature. Lastly, it provides a function to convert a ciphertext to Base64 and back. For this function, we utilize the Jackson API [35] which is a JSON parser for Java. The functions convert the ciphertext to JSON and encode it in Base64 and vice versa. We implement this function so that ciphertexts can be sent between different users.

SecureRandomBuilder: The *SecureRandomBuilder* is used to create SecureRandom objects that are seedable. Sometimes we need seedable SecureRandom objects to ensure that both users generate matching keys. Usually, Java picks the SecureRandom instance which is not seedable on some architectures. So we have to create seedable SecureRandom objects reliably.

SymmetricKeyCombiner The *SymmetricKeyCombiner* is used to mix two symmetric keys as needed by the queued kuKEM which requires the ability to mix several keys. To mix two keys, we use a KDF, namely HKDF.

5.3.4 Variables

The implementations of interfaces of the variables package are straightforward. We explained the transcript implementation in a previous Section, thus, the only missing classes of the variables package are: *BrkeAssociatedData*, *BrkeKeySeed*, and *BrkeSymmetricKey*. All three classes have an internal byte array which represents their content. This way we can process their content where needed. Furthermore, we can use the *BrkeSymmetricKey* to create a *CipherParameters* object which is used in bouncy castle as a key for symmetric ciphers.

5.3.5 Key Encapsulation Mechanism

We decided to implement the ECIES-KEM. The implementation is independent of the choice of the elliptic curve, and the internal KDF. The only requirement on the KDF is the functionality inside the ECIES-KEM implementation of bouncy castle. As described in Section 3.5 we found that only KDF1 and KDF2 described in ISO 18033-2 [47] are working inside the ECIES-KEM implementation. Furthermore, we can set the generated key size when creating a KEM object. Since the KEM is used inside the queued kuKEM and thus does not store any keys internally the implementation is straight forward. There are no changes to the functionality of the KEM. All classes are implemented as an interface between the ECIES-KEM implementation of bouncy castle and the queued kuKEM.

5.4 One-Time Signature

As a quick recap: We decided to implement the DLP-Based Chameleon Hash One-Time Signature described by Mohassel [72]. In general, we directly implement the algorithms we describe in Section 3.2 so we only describe some specific decisions. The actual implementation is independent of the choice of the group and hash function, so we easily can change those later. For this we use the *DHParameters* and *Digest* classes from bouncy castle.

Key Generation: For the key generation, we make use of the *DHKeyPairGenerator* class from bouncy castle. We use the *DHKeyPairGenerator* to generate x_1 and x_2. The reason for that is that the *DHKeyPairGenerator* not only picks a random value but makes sure that the value fulfills some requirements. For example, the *DHKeyPairGenerator* ensures the generated x has a high weight (contains many 1 bits) and is at least of a specific bit size. Those requirements are set in the group we provide the signature algorithm with.

```
if (initiator) {
  signingKeys.add(signingKey[0]);
  communicationPartnerVerificationKey = verificationKey[1];
} else {
  signingKeys.add(signingKey[1]);
  communicationPartnerVerificationKey = verificationKey[0];
}
```

Listing 5.2: Signature Key Generation - Storing the Keys

Listing 5.2 shows how the signature object saves the keys when initializing the object. At this point we generated two keypairs saved in *signingKey[0]* and *verificationKey[0]*, and *signingKey[1]* and *verificationKey[1]*. So depending on the initiator variable, we save the own signing key and the communication partners verification key. We use this approach in all of the algorithm objects.
Sign and Verify: We perform the sign and verify algorithm as described in Section 3.2. The signature algorithm is used to sign a ciphertext associated data pair, so we have to make sure to be able to map this pair to an element in $\mathbb{Z}_p$. For this reason, we hash the input to the signature before computing the signature. We use the *CiphertextEncoder* class (one of the utility classes we describe in Section 5.3) which provides the procedure *hashAdCiphertextPartsForSigning* that takes a ciphertext and associated data and produces a hash. We then use this hash to construct an element we can use for signing.

5.5 Hierarchical Identity-Based Encryption

In this section, we describe the implementation of the prime order version of the Lewko-Waters HIBE [58] as described in Section 3.7.1. We start by describing the C++ code. Then we describe how the Java and C++ parts are interacting. Lastly, we describe the implementation of the CCA transformation. The HIBE is also implemented so that the underlying curves can be exchanged.

5.5.1 The C++ Part

Since this is the most complex part of the implementation and there are very few implementations of the Lewko-Waters HIBE, we explain the implementation of the

HIBE a bit more thorough. We use the Relic library to compute pairings, so we try to adapt the general structure of a scheme written with Relic. Now we have a look at every algorithm of the HIBE and explain the key points starting with the setup algorithm as described in Section 3.7.1:
Setup: The setup algorithm consists of two steps: Generating random dual orthonormal bases and generating the keys. We first describe generating random dual orthonormal bases. Listing 5.3 shows the first step of the generation process.

```
g1_get_ord(modulus);
for(int i = 0; i<dimension; i++){
  for(int j = 0;j<dimension; j++){
    bn_rand_mod(linearTrans[i][j], modulus);
    bn_copy(tempArray[j][i], linearTrans[i][j]);
  }
}
```

Listing 5.3: Generating random dual orthonormal bases - Choosing random matrix X

We obtain the order of the elliptic curve and generate random values which we save in the array *linearTrans*, which is the matrix X. At the same time, we create the transposed matrix and save it in *tempArray*.

```
for(int i=0; i<dimension; i++){
  bn_set_dig(tempArray[i][i+dimension],1);
}
for(int i=0; i<dimension; i++){
  bn_gcd_ext_basic(elementB, elementA, elementC, tempArray[i][i], modulus);
  for(int j = 0; j<dimension*2; j++){
    bn_mul_basic(tempArray[i][j], tempArray[i][j], elementA);
    bn_mod_basic(tempArray[i][j], tempArray[i][j], modulus);
  }
  for(int k=0; k<dimension; k++){
    if((k-i)!=0){
      bn_copy(elementB, tempArray[k][i]);
      for(int j=0; j<dimension*2;j++){
        bn_mul_basic(elementC, elementB, tempArray[i][j]);
        bn_mod_basic(elementC, elementC, modulus);
        bn_sub(tempArray[k][j], tempArray[k][j], elementC);
        bn_mod_basic(tempArray[k][j], tempArray[k][j], modulus);
      }
    }
  }
}
```

Listing 5.4: Generating random dual orthonormal bases - Inverting X

The next step is to invert the array X. This is shown in Listing 5.4. We prepare the *tempArray* for the inversion and then use the gauss jordan elimination [43] in $\mathbb{Z}_p$ to invert the array. Now we create the canonical bases A and A^* as shown in Listing 5.5. Note that instead of 0 we use the point at infinity.

```
for(int i=0 ; i<dimension; i++){
  for(int j=0; j<dimension; j++){
    if(i==j){
      g1_get_gen(basisA1[i][i]);
      g2_get_gen(basisA2[i][i]);
    } else {
      g1_set_infty(basisA1[i][j]);
      g2_set_infty(basisA2[i][j]);
    }
  }
}
```

Listing 5.5: Generating random dual orthonormal bases - Creating canonical bases

Now it is left to create the dual orthonormal bases. We show how to compute basis B in Listing 5.6. The corresponding orthonormal base B^* is computed the same, but uses A^* instead of A.

```
for(int i=0; i<dimension; i++){
  g1_mul(basisB[0][i], basisA1[0][0], linearTrans[i][0]);
  g1_mul(basisB[1][i], basisA1[1][0], linearTrans[i][0]);
  g1_mul(basisB[2][i], basisA1[2][0], linearTrans[i][0]);
  g1_mul(basisB[3][i], basisA1[3][0], linearTrans[i][0]);
  g1_mul(basisB[4][i], basisA1[4][0], linearTrans[i][0]);
  g1_mul(basisB[5][i], basisA1[5][0], linearTrans[i][0]);
  for(int j=1; j<dimension; j++){
    g1_mul(intermediateResult1[0], basisA1[0][j], linearTrans[i][j]);
    g1_mul(intermediateResult1[1], basisA1[1][j], linearTrans[i][j]);
    g1_mul(intermediateResult1[2], basisA1[2][j], linearTrans[i][j]);
    g1_mul(intermediateResult1[3], basisA1[3][j], linearTrans[i][j]);
    g1_mul(intermediateResult1[4], basisA1[4][j], linearTrans[i][j]);
    g1_mul(intermediateResult1[5], basisA1[5][j], linearTrans[i][j]);
    g1_add(basisB[0][i], basisB[0][i],intermediateResult1[0]);
    g1_add(basisB[1][i], basisB[1][i],intermediateResult1[1]);
    g1_add(basisB[2][i], basisB[2][i],intermediateResult1[2]);
    g1_add(basisB[3][i], basisB[3][i],intermediateResult1[3]);
    g1_add(basisB[4][i], basisB[4][i],intermediateResult1[4]);
    g1_add(basisB[5][i], basisB[5][i],intermediateResult1[5]);
  }
}
```

Listing 5.6: Generating random dual orthonormal bases - Computing dual orthonormal bases

Now the columns of B and $BStar$ consist of the elements $\overrightarrow{d}_1, ..., \overrightarrow{d}_6, \overrightarrow{d^*}_1, ..., \overrightarrow{d^*}_6$. These can be directly used to generate the keys of the HIBE. The actual key generation is straightforward and exactly performs as described in Section 3.7.1.

KeyGen: The key generation algorithm takes as input: a master secret key, an identity, the identity length, and the level. We handle the identities as a byte array. That lets us easily map any data to an identity. The identity length denotes the length of one identity, and the level denotes the depth of the identity. The identity byte array has to be $identitylength * level$ large. So for each level of the HIBE we add one identity to the identity array. Using this approach we are flexible in the choice of the actual identity information.

```
uint8_t identity[idLength];
for(int i=0; i < idLength; i++){
  identity[i] = id[i];
}
bn_read_bin(encodedId, identity, idLength);
bn_mod_basic(encodedId, encodedId, modulus);
```

Listing 5.7: Mapping an identity to a field element.

Listing 5.7 shows how we map an identity to an element of $\mathbb{Z}_p$. We read *idLength* bytes from the *id* array and then use it as a field element. The remaining algorithm is performed as described in Section 3.7.1.
Key Delegation: The key delegation algorithm works very similarly to the key generation algorithm. It receives a secret key, an identity, the identity length, and the level as an input and outputs a secret key. Otherwise, it performs the key delegation as described in Section 3.7.1.
Encryption: The encryption algorithm takes as input: public parameters, a message, an identity, the identity length, and the level. The identity is handled as in the other algorithms. The message has to be an element of G_T, which sets some requirements on the data that is encrypted. The remainder of the algorithm is performed as described in Section 3.7.1.
Decryption: The decryption algorithm takes as input a ciphertext and a secret key and outputs the message. Like the rest of the algorithms, we perform all computations as described in Section 3.7.1.

5.5.2 Java to C++ and Vice Versa

We implement a wrapper class in C++ that acts as an interface between the Java code and the HIBE implementation. For this, we use the functionality of JNI. All data that is transferred between Java and C++ is handled as byte arrays. So if we call an HIBE function from Java, we provide all input as bytes. For this reason, we implement encoding and decoding procedures for the parts of the HIBE like keys, parameters, and ciphertexts. These procedures encode, e.g., a secret key to a byte array and vice versa. This way we can transfer the structs of the C++ code via Java. Furthermore, we provide functions that return the size of a specific element, so that we can easily calculate the size of the parameters inside Java.
The wrapper class only provides the functions we need for the kuKEM described in Section 3.6. For this reason, we are only able to delegate a key to the next hierarchy level, and we directly generate a secret key for an initial identity when calling the *setup* procedure. The *master secret key* is directly discarded. Since we need to be able to seed the HIBE so that we can generate matching keys for both user, we have to disable the seeding of the Relic library. For this reason, we

always have to provide a secure seed to the procedures of the wrapper class. Listing 5.8 shows how we initialize the Relic library at every function call to the wrapper class.

```
if (core_init() != STS_OK) {
  core_clean();
  return NULL;
}
if (ep_param_set_any_pairf() == STS_ERR) {
  THROW(ERR_NO_CURVE);
  core_clean();
  return NULL;
}
```

Listing 5.8: Initialization of the Relic library.

The procedure *core_init* initializes the Relic library and enables us to use its functions. The procedure *ep_param_set_any_pairf()* sets the pairing curve according to the prime size that is specified when compiling the library. So the function always picks the curve specific to the prime size used when compiling the library. For this reason, we indirectly specify the curve at compile time.

5.5.3 The Java Part

The Java part of the HIBE implementation is used as a KEM. So we generate a symmetric key in the HIBE class and then encrypt it using the C++ implementation. The reason for that is that the message that is encrypted in the HIBE has to be an element of G_T. So we obtain a random element from G_T and use this element to derive a key. The CCA transformation is directly applied in the HIBE class itself. We do not implement a further encapsulation class. Listing 5.9 shows the native functions we use in the Java code. These are the functions that call the C++ wrapper class.

```
private static native int getSizeOfBnModZp();
private static native int getSizeOfG1();
private static native int getSizeOfG2();
private static native int getSizeOfGT();
private static native int getSizeOfuncompressedGT();
private static native byte[] getRandomGtElement(byte[] seed);
private static native byte[] setup(byte[] identity, int identityLength, byte[] seed);
private static native byte[] encrypt(byte[] publicParameter, byte[] message, byte[]
    identity, int identityLength, int numberOfIdentities, byte[] seed);
public static native byte[] decrypt(byte[] secretKey, byte[] ciphertext, int
    numberOfIdentities);
private static native byte[] delegate(byte[] delegatorSecretKey, byte[] identity, int
    identityLength, int numberOfIdentities, byte[] seed);
```

Listing 5.9: Native functions for the HIBE implementation.

There are functions that obtain the size of different elements of the different groups of the pairing (Lines 1-5). We use these functions to calculate the size of the different structures, e.g., keys. Using this approach we can ensure that the implementation still works if we change the underlying curve for the pairings. Furthermore, there is a function to obtain a random element of G_T (Line 6). As already mentioned this element is used as the message for encryption. Lastly, there are the HIBE functions setup, encrypt, decrypt, and delegate (Lines 7-10). Now we describe the different procedures of the Java HIBE part. Since the CCA transformation is directly applied, we explain the code in more detail.
Setup: Listing 5.10 shows the first part of the Java HIBE implementation.

```java
public HibeKeyPair setup(byte[] identity, SecureRandom randomness) {
  byte[] seed = new byte[sizeOfSeed];
  randomness.nextBytes(seed);
```

Listing 5.10: Java HIBE Setup procedure - Part 1.

As already explained, we always have to provide the C++ part with a seed to ensure random values. For this reason, we always create a seed from the randomness that is provided when calling a function. This is done in every procedure of the HIBE. Listing 5.11 shows the next part of the setup procedure.

```java
  byte[] encodedIdentity = new byte[sizeOfCCAIdentityData];
  encodedIdentity[0] = 0;
  System.arraycopy(identity, 0, encodedIdentity, 1, identity.length);
  byte[] encodedKeys = setup(encodedIdentity, sizeOfCCAIdentityData, seed);
```

Listing 5.11: Java HIBE Setup procedure - Part 2.

As described in Section 3.7.2 we have to encode the identity we use in the HIBE. This is done before calling the C++ code. The variable *encodedKeys* contains the public parameters and the secret key for the encoded identity.

```java
  byte[] encapsulationKey = new byte[generatedKeyLength];
  randomness.nextBytes(encapsulationKey);
```

Listing 5.12: Java HIBE Setup procedure - Part 3.

After that we generate a key we use for the encapsulation, which is also required for the CCA transformation (Listing 5.12).

```java
  int sizeOfEncodedPublicKey = sizeOfcompressedGT * 2 + sizeOfG1 * (dpvsDimension *
      dpvsDimension);
  int sizeOfEncodedSecretKey = sizeOfG2 * (dpvsDimension * dpvsDimension) + sizeOfG2 *
      dpvsDimension;
  byte[] publicParameter = new byte[sizeOfEncodedPublicKey];
  byte[] secretKey = new byte[sizeOfEncodedSecretKey];
```

```
5     System.arraycopy(encodedKeys, 0, publicParameter, 0, sizeOfEncodedPublicKey);
6     System.arraycopy(encodedKeys, sizeOfEncodedPublicKey, secretKey, 0, sizeOfEncodedSecretKey
          );
```

Listing 5.13: Java HIBE Setup procedure - Part 4.

Listing 5.13 shows how we are able to keep the code independent from the choice of the underlying curve for the pairings. We first calculate the size of the public parameters and the secret key (Lines 1-2). Then we create byte arrays and split the *encodedKeys* array (which is the output of the *setup* algorithm (Listing 5.11, Line 4)) into the respective keys. Similarly, the size of the other parameters can be calculated, as well.

Encapsulate: Now we describe the encapsulation procedure. We omit code parts that are similar to the parts explained in the *setup* procedure.

```
1     byte[] randomElement = getRandomGtElement(seed);
2     byte[] com = new byte[k];
3     byte[] r = new byte[k];
4     byte[] dec = Arrays.copyOf(randomElement, k1);
5     keyedHash.init(new HKDFParameters(inputForBytesGeneration, null, null));
6     keyedHash.generateBytes(com, 0, k);
7     encapsulationHash = new SHA256Digest();
8     encapsulationHash.update(dec, 0, k1);
9     encapsulationHash.doFinal(r, 0);
```

Listing 5.14: Java HIBE Encapsulate procedure - Part 1.

Listing 5.14 shows the first key part of the encapsulation procedure. The listing shows how we generate (r, *com*, *dec*) for the CCA transformation, so this part implements the algorithm S(pub) of the encapsulation scheme. We generate a random element from G_T which we can encrypt with the HIBE (Line 1). We then obtain the first $k1$ bytes of the random element and use it as *dec* for the encapsulation. The variables k and $k1$ are static variables that are set to 32 and 96 respectively. So they meet the requirement of $k1 \geqslant 3k$. We then use HKDF instantiated with SHA256 to produce a hash of *com* (Line 5). The variable *inputForBytesGeneration* which is used as a parameter for the HKDF (Line 4) contains the encapsulation key and *com*. After that we use regular SHA256 to hash *com* to produce r (Line 9). Listing 5.15 shows the encryption an the generation of the MAC tag.

```
    byte[] ciphertext = encrypt(publicParameter.getEncodedHibePublicParameter(), randomElement
        , encodedIdentities, sizeOfCCAIdentityData, level + 1, seed);
    hmacAlgorithm.init(new KeyParameter(r));
    hmacAlgorithm.update(ciphertext, 0, ciphertext.length);
    byte[] mactag = new byte[hmacAlgorithm.getMacSize()];
    hmacAlgorithm.doFinal(mactag, 0);
```

Listing 5.15: Java HIBE Encapsulate procedure - Part 2.

We encrypt the *randomElement* for the identity *encodedIdentities*, which contain the encoded identites as described in the CCA transformation (Line 1). After that we generate the MAC tag using r as the key (Lines 2-5). We then use the remaining bytes of the *randomElement* variable and use those to generate a symmetric key using another HKDF instance. By using this approach, we combined the encapsulation scheme for the CCA transformation with the encapsulate procedure of the HIBE.
After explaining these two procedures of the Java part of the HIBE, the decapsulate procedure, and the delegate procedure are straightforward. We always encode the identities as described in the CCA transformation and then use the ‘CCA identities’ in the HIBE. When performing the decapsulate procedure, we first delegate a key to the new identity and then decrypt the ciphertext. After that we can compute r and check the MAC tag.

5.5.4 Summary

In this section, we explained the implementation of the HIBE. We implemented the HIBE in three parts. The actual Lewko-Waters HIBE in C++, a wrapper class that uses JNI in C++ that acts as an interface between Java and C++, and the Java part that acts as a KEM and directly includes the CCA transformation. We implemented the HIBE such that the underlying curve for the pairing can easily be changed. Furthermore, classes that interact with the HIBE do not notice any change to the identity data or any of the changes caused by the CCA transformation.

5.6 Key-Updateable Key Encapsulation Mechanism

With the implementation of the HIBE we can implement the kuKEM. As for all classes we keep the implementation of the kuKEM independent of the underlying HIBE so we can replace the HIBE at a later point in time. In this Section, we describe the implementation of the kuKEM. As explained in Section 3.6 the associated data is used as the identity information in the HIBE. Specifically in BRKE the identity information consists of a ciphertext associated data pair. For this reason, we represent the identity information as a bytes array. This way we can use it in the HIBE. Furthermore, the kuKEM has a own class *KuKemAssociatedData* which enables us to keep the kuKEM implementation independent from the BRKE construction.
Gen: When generating a kuKEM key pair, we construct an initial identity in which we set all bytes to 1. This identity is used to generate a HIBE key pair. Instead of setting the identity information to 0 as proposed by Poettering and Rösler [77] we set the identity information to 1, because the identity might be used as an exponent in the HIBE (this is the case in the Lewko-Waters HIBE) and thus cancels out specific parts of the key. This does not directly influence the functionality of the HIBE, but

we want to avoid potential influences on the security when using an 'empty' identity. A kuKEM public key consists of the HIBE public key (or public parameters), the identity, and the level of the identity. The kuKEM secret key consists of the HIBE secret key for an identity, the identity, and the level of the identity.
Encapsulate: The encapsulation procedure calls the encapsulate procedure of the HIBE using the information stored in the kuKEM public key.
Decapsulate: Similar to the *encapsulation* procedure, we simply call the decapsulation procedure of the HIBE with the information stored in the kuKEM secret key.
Update Public Key: When updating a kuKEM public key we take the *KuKemAssociatedData* and append it to the identity stored in the kuKEM public key.
Update Secret Key: When updating a kuKEM secret key we delegate a secret key for the *KuKemAssociatedData*. We then save the generated secret key, the identity (which is the identity of the initial secret key concatenated with the *KuKemAssociatedData*), and the new level as a kuKEM secret key.
KuKemAssociatedData: The *KuKemAssociatedData* we use in our implementation of the kuKEM is generated by a ciphertext associated data pair. To generate *KuKemAssociatedData* we hash the ciphertext associated data pair to 32 byte and use it as the identity information. This way we can directly use it in the HIBE.
Since the kuKEM directly used the HIBE to perform its algorithms the actual implementation is straightforward. We have to ensure the kuKEM correctly acts as an interface between the queued kuKEM and the HIBE. The kuKEM keys contain the identity information and the respective level of the identity information so that the kuKEM can directly make calls to the HIBE.

5.7 Conclusion

In this Chapter, we describe the implementation of the BRKE instantiation. We implement classes for all interfaces required by the generic BRKE implementation. We describe important design decisions and utility classes for use in a real-world application. We describe the implementation of the HIBE and thus, indirectly the kuKEM a bit more thorough. The reason for that is that all other algorithms are mostly implemented in bouncy castle and we only have to make sure the algorithms provide the functionality the BRKE construction requires. With the implementations we describe in this Chapter, we can perform the BRKE protocol and establish symmetric session keys. We test the implementation with similar Unit tests as the generic BRKE implementation to make sure two users generate matching symmetric keys even when communicating asynchronously.

6 Evaluation

In this chapter, we evaluate the instantiation of the BRKE construction. We analyze the size of the states and the ciphertext for different communication sequences. Additionally, we observe the performance of the different communication sequences. We start by giving an overview of the complete BRKE project. Then we discuss the approach to the evaluation and, lastly, show our results.

6.1 The Complete BRKE project

As described in the respective Chapters we upload all code to a Github repository. We use Maven to build the complete project in one go. For this, we use Maven modules. We have five modules:

brkegeneric contains the generic BRKE implementation described in Chapter 4.

brkeinstantiation contains the implementation of the algorithms we use to instantiate BRKE. These are described in Chapter 5.

hibe-native contains the HIBE C++ code described in Chapter 5. The code is compiled with *gcc* which is also triggered by Maven.

relic-lib contains the Relic library. The library also is built by Maven.

brkeevaluation contains the evaluation module which we explain in this chapter.

As long as the project is built on Linux, the complete project is built and installed without any additional prerequisites. As described in Section 5.5 we set the curve for the pairing by compiling the Relic library with a specific prime size. For this reason, we only have to recompile the Relic library with a different prime size to test different curves. Since we implemented all of our algorithms independent from the specific primitives, we can test different curves without much effort.

6.2 Approach

As already mentioned we want to test the BRKE implementation with four different pairing-friendly elliptic curves. The first reason for that is that we assume the HIBE to have the most influence on the performance of the implementation. If we change the underlying elliptic curve for the pairings, we assume that the send and receive algorithms of the BRKE implementation take longer and that the size of the state and the ciphertexts change. This is because elliptic curves that have a higher security level tend to have larger parameters and, thus, have worse performance over elliptic curves with a lower security level [8]. The second reason is that pairings are a highly active research topic and that choosing a specific curve for the pairings may not be straightforward. We addressed this in Section 3.7 and discussed four different pairing-friendly elliptic curves. Unfortunately, the Relic library does not directly provide the parameters for all curves described in Section 3.7. So we have to use different elliptic curves in the evaluation.

Choices for pairing-friendly curves: The Relic library provides the curve parameters for BN-P256 and B12-P381, so we do not have to choose alternatives for those two curves. However, the Relic library does not provide parameters for KSS16-340 and BLS12-461. For this reason, we choose the curves BN-P382 and B12-P455 as alternatives. Like BN-P256, BN-P382 is a Baretto-Naehrig curve [11] but uses a larger prime field (382 Bit instead of 256 Bit). It is said to have 128 Bit security, but there is no research result on the security evaluation [90]. The B12-P455 curve uses a slightly smaller prime field than the BLS12-461 curve (455 Bit instead of 461), but is also said to fulfill the requirements set by Barbulescu-Duquesne [10] and, thus, have 128-Bit estimated symmetric-equivalent strength [7, 33]. So we choose two types of curves, Baretto-Naehrig (BN) [11] and Baretto-Lynn-Scott (B12) [12], each with two different prime field sizes. Computations on the BN-P256 curve are highly optimized because it was the state-of-the-art pairing-friendly curve for the 128-Bit security level until the research by Barbulescu-Duquesne [10]. This is the reason why we decide to consider this curve in the evaluation, despite having only 100 Bit estimated symmetric-equivalent strength. So we can estimate the performance if the HIBE uses an optimized pairing-friendly curve. Computations on the curves for today's 128-Bit security level are not optimized yet, and performance may still change. However, theoretically, the Baretto-Lynn-Scott curves should perform better than the BN-P382 curve [8, 10, 33].

How we test: We implement an evaluation module in Java that simulates four different communication sequences between two users. Every communication sequence is performed 50 times, and we measure the time that each send and receive algorithm takes, the size difference of the user states before and after every send and receive algorithm and the size of the kuKem ciphertexts for every iteration. Furthermore, we measure the size of the base ciphertext (without the kuKem ciphertext) and the size of the state of a user after the initialization. The averages of the measurements of the 50 iterations are printed to the Java console, and every individual

measurement is printed to a CSV file so that it can be opened and evaluated with a spreadsheet. We perform the evaluation program with each of the prior discussed elliptic curves. When the evaluation is run with the *BN-P256* curve, we also use a different Diffie-Hellman group for the chameleon hash function, namely *ffdhe2048* from RFC7919 [38] which has an estimated symmetric-equivalent strength of 103 bits. Similar to the choice of the pairing-friendly elliptic curve this group complies to the requirements on finite fields set by the BSI, but not to the requirements set by ECRYPT-CSA.
Specifications: We perform the evaluation program on a 3.4 GHz Intel Core i5-8250U 64-bit PC with 8 GB RAM and Ubuntu 18.04.2. We use Java version 1.8.0_201. The evaluation code is included in the BRKE project and uploaded to a Github repository as a branch. Like the rest of the implementation, the evaluation module is built by Maven. We use Maven version 3.5.2. To estimate the size of the states and ciphertexts we use Java Object Layout (JOL) [49]. Measuring the size of an object in Java is not straightforward, so the measured sizes by JOL might not be hundred percent accurate, but they give us a good indication of the size which suits our purpose.

6.2.1 Communication Sequences

The four communication sequences we use in the evaluation are lockstep communication, asynchronous communication without crossing messages, asynchronous communication with crossing messages, and 'worst-case' communication. We describe these communication sequences in this section. Generally, we always loop the sequences, so if a communication sequence has, e.g., six steps we always jump from step 6 to step 1. We do not reset the user states after every run, because in a real-world application users would not be reset either. This means that the first run of a communication sequence is different from the remaining runs because the user states after a communication run differ from the freshly initialized user states. For this reason, we do not consider the first run of a communication sequence when computing the average. However, the first run is always written to the CSV file so that we could analyze it separately. In the following descriptions of the communication sequences we are always concerned with communication between two BRKE users A and B using associated data ad.
Lockstep Communication: Listing 6.1 shows the lockstep communication sequence. In this sequence, we simulate a conversation between two users in which the users alternately send messages and directly receive the message of its communication partner.

```
0 $(k_{A,0}, c_{A,0}) \Leftarrow \text{send}_A(ad)$
1 $k_{A,0} \leftarrow \text{receive}_B(c_{A,0})$
2 $(k_{B,0}, c_{B,0}) \Leftarrow \text{send}_B(ad)$
```

```
$k_{B,0} \leftarrow \text{receive}_A(c_{B,0})$
```

Listing 6.1: Lockstep communication sequence

Asynchronous Communication Without Crossing Messages: Listing 6.2 shows the asynchronous communication sequence without crossing messages. This sequence simulates a conversation between two users who communicate asynchronously without crossing messages. This means every message is directly received. In this sequence, A sends three messages successively. After that B responds with two messages.

```
$(k_{A,0}, c_{A,0}) \Leftarrow \text{send}_A(ad)$
$k_{A,0} \leftarrow \text{receive}_B(c_{A,0})$
$(k_{A,1}, c_{A,1}) \Leftarrow \text{send}_A(ad)$
$k_{A,1} \leftarrow \text{receive}_B(c_{A,1})$
$(k_{A,2}, c_{A,2}) \Leftarrow \text{send}_A(ad)$
$k_{A,2} \leftarrow \text{receive}_B(c_{A,2})$
$(k_{B,0}, c_{B,0}) \Leftarrow \text{send}_B(ad)$
$k_{B,0} \leftarrow \text{receive}_A(c_{B,0})$
$(k_{B,1}, c_{B,1}) \Leftarrow \text{send}_B(ad)$
$k_{B,1} \leftarrow \text{receive}_A(c_{B,1})$
```

Listing 6.2: Asynchronous communication sequence without crossing messages

Asynchronous Communication With Crossing Messages: The asynchronous communication sequence with crossing messages is shown in Listing 6.3. This communication sequence is similar to the asynchronous communication sequence without crossing messages, but in this sequence, messages are not directly received. Thus, they cross.

```
$(k_{A,0}, c_{A,0}) \Leftarrow \text{send}_A(ad)$
$(k_{A,1}, c_{A,1}) \Leftarrow \text{send}_A(ad)$
$(k_{B,0}, c_{B,0}) \Leftarrow \text{send}_B(ad)$
$(k_{B,1}, c_{B,1}) \Leftarrow \text{send}_B(ad)$
$k_{B,0} \leftarrow \text{receive}_A(c_{B,0})$
$(k_{A,2}, c_{A,2}) \Leftarrow \text{send}_A(ad)$
$k_{A,0} \leftarrow \text{receive}_B(c_{A,0})$
$k_{A,1} \leftarrow \text{receive}_B(c_{A,1})$
$k_{A,2} \leftarrow \text{receive}_B(c_{A,2})$
$k_{B,1} \leftarrow \text{receive}_A(c_{B,1})$
```

Listing 6.3: Asynchronous communication sequence with crossing messages

Worst case Communication: Listing 6.4 shows the 'worst-case' communication sequence. This communication sequence simulates a communication in which we try to create many active epochs which need to be updated. For this we let A send five messages of which B receives none. After that, B sends two messages which A receives. Afterward, A sends another message, and B receives all messages.

```
$(k_{A,0}, c_{A,0}) \Leftarrow \text{send}_A(ad)$
$(k_{A,1}, c_{A,1}) \Leftarrow \text{send}_A(ad)$
$(k_{A,2}, c_{A,2}) \Leftarrow \text{send}_A(ad)$
$(k_{A,3}, c_{A,3}) \Leftarrow \text{send}_A(ad)$
$(k_{A,4}, c_{A,4}) \Leftarrow \text{send}_A(ad)$
$(k_{B,0}, c_{B,0}) \Leftarrow \text{send}_B(ad)$
$k_{B,0} \leftarrow \text{receive}_A(c_{B,0})$
$(k_{B,1}, c_{B,1}) \Leftarrow \text{send}_B(ad)$
$k_{B,1} \leftarrow \text{receive}_A(c_{B,1})$
$(k_{A,5}, c_{A,5}) \Leftarrow \text{send}_A(ad)$
$k_{A,0} \leftarrow \text{receive}_B(c_{A,0})$
$k_{A,1} \leftarrow \text{receive}_B(c_{A,1})$
$k_{A,2} \leftarrow \text{receive}_B(c_{A,2})$
$k_{A,3} \leftarrow \text{receive}_B(c_{A,3})$
$k_{A,4} \leftarrow \text{receive}_B(c_{A,4})$
$k_{A,5} \leftarrow \text{receive}_B(c_{A,5})$
```

Listing 6.4: Worst case communication sequence

We choose the first two communication sequences because we want to simulate a typical conversation between two users. In these sequences there are no crossing messages, which means the communication partner directly receives every message. For this reason, there are no key updates in those two communication sequences. The latter two communication sequences simulate a conversation when there are crossing messages. Now there are several active epochs and, thus, we require key updates. While the third communication sequence simulates a somewhat more probable communication sequence, because the messages are not that delayed, the third communication sequence is constructed to lead to several active epochs that need to be updated.

6.3 Evaluation Results

In this section, we show and discuss the results of the evaluation. We first have a look at the overall state size and the ciphertext size, and then we evaluate the four communication sequences. Table 6.5 shows the size of the user state after the initialization for each of the four elliptic curves. We see that the user states for every

State in Bytes	**BN-P256**	**B12-P381**	**BN-P382**	**B12-P455**
	73776	74800	74800	74800

Table 6.5: User state sizes after initialization.

curve except BN-P256 is even. The reason for this is that we do not use the kuKEM in the initialization, so we do not use the elliptic curve. As explained in Section 4.3 the first key of an epoch can always be a regular KEM key. For this reason, the state contains no kuKEM keys and, thus, the curves do not influence the size of the freshly initialized state. The difference between the BN-P256 curve is caused by using a smaller Diffie-Hellman group for the one-time signature (since we use a 100-Bit security level). Table 6.6 shows the size of the base ciphertext (without the kuKEM ciphertext) for every elliptic curve. These parts are the constant part of the ciphertext. We can see that the choice of the curve for the HIBE influences the

Base Ciphertext Size in Bytes	**BN-P256**	**B12-P381**	**BN-P382**	**B12-P455**
kuKEM Public Key	1864	2696	2696	3160
OT-Signature Verification Key	768	1024	1024	1024
KEM Ciphertext	104	104	104	104
Signature	648	904	904	904
Integer	64	64	64	64
Total	**3448**	**4792**	**4792**	**5256**

Table 6.6: Base Ciphertext Size (without kuKEM ciphertext).

size of the ciphertext. The kuKEM public key grows depending on the choice of the curve. The kuKEM public keys and, thus, the ciphertexts, of B12-P381 and BN-P382 are of the same size, because of the size of the elements of the curves. The size of the elements of B12-P381 and BN-P382 are shown in Table 6.7. We can see that G_1, G_2, and G_T elements have the same size across both curves. However, elements in Z_p have a different size. This leads to kuKEM public keys, kuKEM secret keys, and kuKEM ciphertexts having the same size. However, as we see in the evaluation of the communication sequence, this leads to different computational times. The reason for that is that the kuKEM keys and ciphertexts only consist of elements in G_1, G_2, and G_T, but computations include multiplications with elements from Z_p.

Element Size in Bytes	**B12-P381**	**BN-P382**
Z_p	32	48
G_1	49	49
G_2	97	97
G_T	567	567

Table 6.7: Element Sizes of B12-P381 and BN-P382.

We can already see that even without the kuKEM ciphertext the kuKEM has the most influence on the size of the ciphertext. The kuKEM public key makes up for more than 50% of the ciphertext (60% in the case of B12-P455).

6.3.1 Lockstep Communication

Table 6.8 shows the results of the lockstep communication sequence. The table consists of three parts. The first part shows the average time each of the communication steps takes in milliseconds. The second part shows the average size difference in Bytes of the user state before and after a communication step. And the third part shows the size of the kuKEM ciphertext in Bytes. Of course, there are only ciphertexts for send operations.
From the timings, we can see that the BN-P256 curve is the most performant. This

Time in ms	**BN-P256**	**B12-P381**	**BN-P382**	**B12-P455**
A sends (Step 0)	179	363	470	571
B receives (Step 1)	122	250	316	395
B sends (Step 2)	179	363	469	570
A receives (Step 3)	122	250	316	395
State in Bytes				
A sends (Step 0)	6014	6526	6526	6814
B receives (Step 1)	-6019	-6531	-6531	-6819
B sends (Step 2)	6016	6528	6528	6816
A receives (Step 3)	-6016	-6528	-6528	-6816
KuKemCT in Bytes				
A sends (Step 0)	992	1376	1376	1592
B receives (Step 1)	-	-	-	-
B sends (Step 2)	992	1376	1376	1592
A receives (Step 3)	-	-	-	-

Table 6.8: Results of the Lockstep Communication Sequence

is expected, since as already described this curve is highly optimized. The BRKE algorithms take around double the time if the HIBE is instantiated with the B12-P381 curve. If we use the BN-P382 curve, the algorithms take around 2.5 times the time of the instantiation with BN-P256. And lastly, the instantiation with B12-P455 is 3 times less efficient than the instantiation with BN-P256.
The choice of the curve also affects the size of the user state. Between BN-P256 and B12-P455 there is only a difference of 800 Bytes. In the instantiation with B12-P381 and BN-P382, the size is 500 Bytes larger than in the instantiation with BN-P256. So we can see that it does affect the size of the state, but since the base state is already around 74.800 Bytes, the difference has a minimal effect on the total size. However, from the results, we can see that in the lockstep communication a send operation increases the size of the state, and the receive operation decreases the size of the state. We would assume that send and receive would balance out the size of the state, and if we have a look at the full measurements for the lockstep communication, it turns out that most of the time the receive algorithm decreases the size of the state by the amount the send algorithm increases the size. There are a few measurements that are ± 100 Bytes off that cause the average, which is shown in the table, to indicate that the algorithms do not balance out the state, but in reality, they do. We assume that those size differences are either caused by JOL, since we cannot be sure that the size estimation is a hundred percent accurate, or by some Java-specific objects being allocated or freed.
Table 6.8 also shows that every ciphertext in the lockstep communication sequence contains a kuKEM ciphertext. Naturally, the size of the kuKEM ciphertext is dependent on the choice of the elliptic curve in the HIBE. If we once more pick the BN-P256 curve as a reference, the kuKEM ciphertext grows for around 40% if the used curve is B12-P381 or BN-P382, and for around 60% if we use B12-P455.
All in all, we can say that the lockstep communication sequence is pretty balanced meaning both users have the same computational times for the send and receive algorithms, and all ciphertexts are of the same size. However, we can already see

that the choice of the curve has a high impact on the time the respective BRKE algorithms take.

6.3.2 Asynchronous Communication without Crossing Messages

Table 6.9 shows the results of the asynchronous communication sequence without crossing messages. Again the BN-P256 is the most performant curve. The perfor-

Time in ms	BN-P256	B12-P381	BN-P382	B12-P455
A sends (Step 0)	213	426	549	678
B receives (Step 1)	234	468	600	758
A sends (Step 2)	147	299	387	464
B receives (Step 3)	12	31	31	31
A sends (Step 4)	148	299	387	463
B receives (Step 5)	11	31	31	31
B sends (Step 6)	246	489	631	783
A receives (Step 7)	345	687	885	1121
B sends (Step 8)	148	299	388	463
A receives (Step 9)	11	31	31	31
State in Bytes				
A sends (Step 0)	4128	3808	3808	3632
B receives (Step 1)	-9136	-10992	-10992	-12032
A sends (Step 2)	3120	4464	4464	5216
B receives (Step 3)	1888	2720	2720	3184
A sends (Step 4)	3120	4464	4464	5216
B receives (Step 5)	1888	2720	2720	3184
B sends (Step 6)	2240	1088	1088	448
A receives (Step 7)	-12256	-15456	-15456	-17248
B sends (Step 8)	3120	4464	4464	5216
A receives (Step 9)	1888	2720	2720	3184
KuKemCT in Bytes				
A sends (Step 0)	1952	2720	2720	3152
B receives (Step 1)	-	-	-	-
A sends (Step 2)	0	0	0	0
B receives (Step 3)	-	-	-	-
A sends (Step 4)	0	0	0	0
B receives (Step 5)	-	-	-	-
B sends (Step 6)	2912	4064	4064	4712
A receives (Step 7)	-	-	-	-
B sends (Step 8)	0	0	0	0
A receives (Step 9)	-	-	-	-

Table 6.9: Results of the Asynchronous Communication Sequence without Crossing Messages

mance impact has the same ratio as for the lockstep communication sequence. So the BRKE algorithms take around 2 times longer with B12-P381, around 2.5 times longer with BN-P382, and roughly 3 times longer with B12-P455. From the timings, we can see some interesting properties of the BRKE algorithms. Every consecutively sent message (Step 2, Step 4, and Step 8) always takes the same amount of time. Furthermore, every consecutively received message takes the same amount of time (Step 3, Step 5, and Step 9). The time for the receive algorithm is thereby the same across the curves for the 128-Bit security level. The reason for that is that only the

KEM is used in these situations and, thus, we do not use the HIBE. The difference between BN-P256 and the other curves is caused by choice of the group for the one-time signature. We can see that choosing a smaller group speeds up the verification by 20 ms. When sending messages, we always have to generate a kuKEM key pair. For this reason, the send algorithms have no constant time across the curves.
Now we have a look at the first call of the send algorithm (Step 0 and Step 6), after receiving messages. If we call the send algorithm after receiving a message the queued kuKEM uses the kuKEM keys, which we received in the ciphertexts. For this reason, the send algorithm takes longer for every additional received message, because for each received message we have another kuKEM encapsulation. A receives two messages before its call to the send algorithm in Step 0 (note that the communication sequence is looped) and, thus, the send algorithm takes 213 ms with the BN-P256 curve. B receives three messages before its call to the send algorithm in Step 6 and, thus, the send algorithm takes 246 ms with the BN-P256 curve. If we further take the send algorithm from the lockstep communication (179 ms, Table 6.8) into account, we can compute that every previously received message increases the time the send algorithm takes by roughly 33 ms when the HIBE is instantiated with BN-P256. For this reason, we can roughly calculate the required time for the send algorithm for each curve if there are no crossing messages. The reason for that is that for every received message the send algorithm performs another kuKEM encapsulation. We now see that the same applies to the receive algorithm. Again, we only consider BN-P256, for now, but the same idea applies to the other curves. B's receive algorithm in Step 1 takes 234 ms. Since A uses two kuKEM encapsulations in Step 0, B has to use two kuKEM decapsulations. Similar A uses three kuKEM decapsulations when receiving B's message in Step 7, which results in a time of 345 ms. Again, if we take the receive algorithm from the lockstep communication (122 ms, Table 6.8) into account, we can compute that every previously received message increases the time the send algorithm takes by roughly 111 ms when the HIBE is instantiated with BN-P256.
This shows that if we have no crossing messages, we can roughly estimate the times each of the BRKE algorithms take for a specific curve.
The user state size differences behave similarly, which means consecutively sent or received messages have the same effect on the state (Step 2,3,4,5,8,9), but opposite our observation from the lockstep communication sequence consecutively received message increase the size of the user state. The reason for that is that the kuKEM public key that is contained in the ciphertext is added to the public key queue in the queued kuKEM, and unlike other receives the algorithm does not delete kuKEM secret keys. In the receives in Step 1 and Step 7, we can see that the size of the user state is significantly decreased. The reason for that is that the respective user receives the information that sent messages are received and, thus, can delete all now unnecessary stored data, e.g., kuKEM secret keys. We can see that the send algorithm in Step 0 and Step 6 increase the state, but that the size increase is lower for the higher security curves than for BN-P256. This occurs only for the first send after receiving messages. The first send uses all prior received kuKEM keys and,

thus, they are deleted from the user state. The keys for the kuKEM keys for the higher security curves are larger which means we free more space when deleting the keys. We assume this is the reason for the size difference.
The ciphertext does not contain a kuKEM ciphertext when a user sends more than one message consecutively. The reason for that is that the user only uses the KEM to encapsulate a key. Otherwise the kuKEM ciphertext grows linearly for each used kuKEM encapsulation. Again if we consider BN-P256 as the elliptic curve for the HIBE and take the kuKEM ciphertext size of the lockstep communication sequence (992 Bytes, Table 6.8) into account, we can see that for every kuKEM encapsulation the kuKEM ciphertext grows by roughly 960 Bytes.
So with this communication sequence we can see that as long that there are no crossing messages, we can roughly estimate the times that each BRKE algorithm takes, and the size of the ciphertexts depending on the choice of the elliptic curve. However, until now the communication sequences required no key updates, since all messages were directly received by the respective users. This will change in the next two communication sequences.

6.3.3 Asynchronous Communication with Crossing Messages

Table 6.10 shows the results of the asynchronous communication sequence with crossing messages. In this communication sequence, there are crossing messages which lead to key updates in the kuKEM. So we focus on the communication steps which are affected by key updates and do not describe every step. So a key update is necessary if a user receives a message and receives the information that the communication partner did not receive a sent message yet. This situation is simulated in the asynchronous communication sequence with crossing messages. Unfortunately, we can not directly say that only the key updates are causing the BRKE algorithms to take longer than in the other communication sequences. This communication sequence leads to multiple kuKEM en-/decapsulations in one algorithm call mixed with key updates. If we have a look at Step 4, for example. A receives the ciphertext sent by B in Step 2. Since we loop the communication sequence, B includes the information that it received three messages (Step 6-8) and, thus, it uses the three kuKEM public keys contained in the three received ciphertexts. For this reason, A not only has to perform three kuKEM decapsulations, but also has to update the keys that are generated in Step 1 and Step 2. So we see, in this communication sequence many different parts affect the performance of the BRKE algorithms, which makes it less straight forward to analyze as the first two communication sequences. For this reason, we focus more on the results shown in Table 6.10 and do not try to analyze the reason for, e.g., long algorithm times. So we see that this communication sequence leads to long computational times in the send and receive algorithms. Even with the optimized BN-P256 curve Step 4 takes 734 ms. If we use the B12-P455 curve, it takes 2,3 s to generate the key. This shows that crossing messages affect the performance of the BRKE construction. However, this is expected, since

Time in ms	BN-P256	B12-P381	BN-P382	B12-P455
A sends (Step 0)	211	424	555	677
A sends (Step 1)	146	299	387	464
B sends (Step 2)	295	592	774	961
B sends (Step 3)	146	299	387	463
A receives (Step 4)	734	1441	1924	2391
A sends (Step 5)	200	404	526	642
B receives (Step 6)	414	816	1100	1349
B receives (Step 7)	223	441	621	730
B receives (Step 8)	360	708	959	1169
A receives (Step 9)	295	584	819	971
State in Bytes				
A sends (Step 0)	5920	6432	6432	6720
A sends (Step 1)	3120	4464	4464	5216
B sends (Step 2)	2080	928	928	288
B sends (Step 3)	3120	4464	4464	5216
A receives (Step 4)	-13456	-17232	-17232	-19360
A sends (Step 5)	5952	6464	6464	6752
B receives (Step 6)	-6384	-7088	-7088	-7480
B receives (Step 7)	2800	4016	4016	4704
B receives (Step 8)	-1616	-2320	-2320	-2728
A receives (Step 9)	-1536	-128	-128	672
KuKemCT in Bytes				
A sends (Step 0)	1584	2256	2256	2632
A sends (Step 1)	0	0	0	0
B sends (Step 2)	3896	5528	5528	6448
B sends (Step 3)	0	0	0	0
A receives (Step 4)	-	-	-	-
A sends (Step 5)	1384	1960	1960	2288
B receives (Step 6)	-	-	-	-
B receives (Step 7)	-	-	-	-
B receives (Step 8)	-	-	-	-
A receives (Step 9)	-	-	-	-

Table 6.10: Results of the Asynchronous Communication Sequence with Crossing Messages

crossing messages require all functionality of the kuKEM and, thus, the HIBE.
This communication sequence also has a high impact on the state of the users. As we discussed in the lockstep communication section the send and receive algorithms balance out the size of the state. If we have a look at Step 4 in Table 6.10 we see that the state is decreased by 13 to 19 kB, which implies that the state at least grows by the same amount. If we take a base state size of 74800 Bytes or 74 kB as the reference an increase of 19 kB means that the state grows by around 25%. The size of the kuKEM ciphertext is strongly affected by this communication sequence, as well. If we take the ciphertext produced in Step 2 the kuKEM ciphertext is larger than the actual base ciphertext. If we now compute the influence of the kuKEM on the BRKE ciphertext we see that the kuKEM makes up 80% of the ciphertext for the BN-P256 curve (with total ciphertext size: 7344 Bytes), 80% of the ciphertext for the B12-P381 and BN-P382 curve (with total ciphertext size: 10320 Bytes), and 83% for the B12-P455 curve (with total ciphertext size: 11704 Bytes).
All in all, we can see that crossing messages have a significant impact on the overall

behavior of the BRKE algorithms. With the B12-P455 curve, the key generation in Step 4 takes 2,1 s. Furthermore, the impact on the size of the state and ciphertext is much stronger than in the other communication sequences. It is open to question how likely such a communication sequence occurs.

6.3.4 Worst-Case Communication

Since the worst-case communication sequence contains 16 communication steps, we split the result table into three parts and evaluate the parts individually. Table 6.11 shows the results for the time measurements of the worst-case communication sequence. We can see that this communication sequence greatly increases the time

Time in ms	BN-P256	B12-P381	BN-P382	B12-P455
A sends (Step 0)	147	300	387	464
A sends (Step 1)	147	299	386	462
A sends (Step 2)	147	299	386	463
A sends (Step 3)	146	299	387	463
A sends (Step 4)	147	298	387	464
B sends (Step 5)	444	886	1161	1459
A receives(Step 6)	1527	2992	4009	4984
B sends (Step 7)	146	299	387	463
A receives (Step 8)	541	1059	1495	1776
A sends (Step 9)	317	633	836	1033
B receives (Step 10)	156	314	430	510
B receives (Step 11)	222	442	618	729
B receives (Step 12)	289	570	804	946
B receives (Step 13)	357	700	993	1167
B receives (Step 14)	424	826	1181	1386
B receives (Step 15)	726	1414	1878	2337

Table 6.11: Time Results of the Worst-Case Communication Sequence

required for the BRKE algorithms. Step 0 to Step 4 are consecutively sent messages, so they always have the same performance (note that we loop the sequence). In Step 5 B takes all keys it receives in Step 10-15 for the kuKEM encapsulation, so the time required for the send algorithm is increased. In Step 6 A receives the ciphertext produced by B in Step 5. B did not receive any of the ciphertexts from Step 0-4, so A has to perform five key updates and has to perform six kuKEM decapsulations. This leads to a very high computational time. With the B12-P455 the key generation takes almost five seconds. When A receives the next message (Step 8) it only has to update the keys but does not need to perform a kuKEM decapsulation. This leads to the lower required time for the receive algorithm in Step 8 than the receive algorithm in Step 6. In Step 10-15 B receives all messages sent by A. The required time for the receive algorithm always increases, because the amount of key updates increases with every received message. Furthermore, the last received message (Step 15) requires kuKEM decapsulations, which further increases the required time for the receive algorithm. The other receives only require KEM decapsulations.

Table 6.13 shows the results for the state size difference of the worst-case communication sequence. Similar to the last section we only look at the entire state and not

State in Bytes	BN-P256	B12-P381	BN-P382	B12-P455
A sends (Step 0)	3120	4464	4464	5216
A sends (Step 1)	3120	4464	4464	5216
A sends (Step 2)	3120	4464	4464	5216
A sends (Step 3)	3120	4464	4464	5216
A sends (Step 4)	3120	4464	4464	5216
B sends (Step 5)	-3744	-7392	-7392	-9424
B sends (Step 6)	-23576	-31768	-31768	-36376
A receives (Step 7)	3120	4464	4464	5216
A receives (Step 8)	4168	5960	5960	6984
A sends (Step 9)	3808	3488	3488	3312
B receives (Step 10)	-2000	-784	-784	-96
B receives (Step 11)	2800	4016	4016	4704
B receives (Step 12)	2800	4016	4016	4688
B receives (Step 13)	2800	4016	4016	4688
B receives (Step 14)	2784	4000	4000	4688
B receives (Step 15)	-8560	-12336	-12336	-14464

Table 6.12: State Difference Results of the Worst-Case Communication Sequence

analyze every state difference independently. So we see that the state size decreases by at most 36 kB. If we use the same argumentation as for the other communication sequences, we assume that the state has to grow by at least the same amount. For this reason, we can assume that the state grows up to almost 50% of its original size if the HIBE is instantiated with the B12-P455 curve. If we use BN-P256, the state grows by around 30%. This again shows the impact of the conversation sequence on the overall size of the user state.
Table 6.11 shows the results for the kuKEM ciphertext size of the worst-case communication sequence. As we can see only two ciphertexts in the communication

KuKemCT in Bytes	BN-P256	B12-P381	BN-P382	B12-P455
A sends (Step 0)	0	0	0	0
A sends (Step 1)	0	0	0	0
A sends (Step 2)	0	0	0	0
A sends (Step 3)	0	0	0	0
A sends (Step 4)	0	0	0	0
B sends (Step 5)	7752	11016	11016	12872
B sends (Step 6)	-	-	-	-
A receives (Step 7)	0	0	0	0
A receives (Step 8)	-	-	-	-
A sends (Step 9)	3936	5664	5664	6624
B receives (Step 10)	-	-	-	-
B receives (Step 11)	-	-	-	-
B receives (Step 12)	-	-	-	-
B receives (Step 13)	-	-	-	-
B receives (Step 14)	-	-	-	-
B receives (Step 15)	-	-	-	-

Table 6.13: KuKEM Ciphertext Results of the Worst-Case Communication Sequence

sequence contain kuKEM ciphertexts. The kuKEM ciphertext produced in Step 9

has around the same size as the largest ciphertext in the third communication sequence, but the kuKEM ciphertext produced in Step 5 more than triples the size of the base BRKE ciphertext. So in Step 5 for the BN-P256 curve, the BRKE ciphertext is 11200 Bytes, for B12-P381/BN-P382 the BRKE ciphertext is 10456 Bytes, and for B12-P455 the ciphertext is 18128 Bytes large.
We constructed this communication sequence to see the impact of the kuKEM for different curves if we have to perform multiple key updates and have several active epochs. We can see that the performance suffers significantly if a user has to keep several active epochs. Moreover, with an increasing amount of active epochs the kuKEM has an increasing impact on the performance of BRKE.

6.4 Conclusion

In this chapter we evaluate the BRKE instantiation. We choose four pairing-friendly elliptic curves to test the performance impact of the elliptic curve choice on the performance and behavior of the BRKE instantiation. We choose BN-P256, which provides 100-Bit security, and B12-P381, BN-P382, and B12-P445, which provide 128-Bit security. We choose the first curve because it is a highly optimized curve and this helps us to obtain an idea what the performance could look like if the HIBE is instantiated with an efficient elliptic curve. We discussed B12-P381 and B12-P455 (more precisely a slightly different version) in Chapter 3. We described that the former curve is the optimistic choice for the 128-Bit security level and the latter curve is the conservative choice for the 128-Bit security level. We consider BN-P382 as a comparison to the other pairing-friendly curves for the 128-Bit security level. Furthermore, we change the Diffie-Hellman group for the one-time signature if we use BN-P256 for the pairings. So we use a 100-Bit security level algorithm set and a 128-Bit security level algorithm set.
We construct four communication sequences and test the BRKE instantiation with all four curves. Two of the communication sequences simulate a regular conversation between two users without crossing messages. The other two simulate communication sequences in which messages cross while transferring over the network. We measure the average required time of the BRKE algorithms, the size difference of the state before and after an algorithm call, and the size of the kuKEM ciphertext. The results show that the required time of the BRKE algorithms increases linearly if there are no crossing messages. Furthermore, the kuKEM ciphertext grows linearly depending on the conversation flow. If there are crossing messages the amount of active epochs considerably impacts the performance of BRKE. The more active epochs, the more influence the kuKEM has on the required time for the algorithms, the size of the state, and the size of the ciphertext.

7 Conclusion

The goal of this thesis was to implement the Bidirectionally Ratcheted Key Exchange (BRKE) construction proposed by Poettering and Rösler [77] in Java. We split the implementation into two parts: a generic part and an instantiation of the generic part. The goal for the generic part was that the BRKE implementation can be instantiated with any actual implementations of the required algorithms and does not depend on any cryptographic libraries. The instantiation should then use actual algorithms which achieve the security requirements set by Poettering and Rösler [77].

After describing the different required cryptographic primitives, security requirements, and the required mathematical backgrounds, we discuss possible choices for each of the required primitives. We use recommendations by the BSI and ECRYPT-CSA to find suitable algorithms and then research if those algorithms fulfill the security requirements set by Poettering and Rösler [77]. Furthermore, we check if there are already implemented versions of those algorithms in Java. The performance of the respective algorithms is no primary concern for the implementation. For each of the primitives, we provide an overview that shows suitable algorithms for the use in the BRKE construction. Except for the one-time signature and the HIBE we use implementations provided by bouncy-castle [86]. We choose to implement a one-time signature based on a chameleon hash function, and the HIBE ourselves because we could not find implementations of those two primitives that fulfill our requirements. Furthermore, we discuss several pairing-friendly elliptic curves for the use in the HIBE. It turns out that pairings are a currently highly active research topic and choosing a suitable pairing-friendly elliptic curve is not straightforward, due to the recently found attacks on those curves. After discussing the algorithms, we describe the generic BRKE implementation. To better represent a real-world application we proposed several changes to the BRKE ad-hoc construction as described by Poettering and Rösler [77]. These changes are necessary to avoid endlessly growing arrays and so that the implementation complies with the Java standard of object-oriented programming. We furthermore propose the queued kuKEM, which is a queue-based modification of the kuKEM, which directly performs key updates, and queued en-/decapsulations. The generic BRKE implementation does not depend on any specific algorithms. It only uses interfaces which specify the functionality of the respective algorithms. This enables us to interchange different algorithms without affecting the BRKE functionality easily.

For the instantiation, we use the bouncy-castle library [86] and the Relic library [7]. We use the Relic-library for the pairings which we need in the HIBE. Unfortu-

nately, Relic is only available in C/C++. For this reason, we implement the HIBE in C++ and use JNI to use the implementation in Java. Since our choice of the HIBE, namely the Lewko-Waters HIBE [58], does not fully comply with the security requirements needed for the kuKEM we also apply a transformation to the HIBE. With this implementation of the HIBE we can implement the kuKEM as described by Poettering and Rösler [77]. By using the implementations of the algorithms we construct a BRKE algorithm set, which can be used to instantiate the generic BRKE construction and test its functionality. Even the implementation of the specific algorithms is kept as generic as possible, so that we can exchange the underlying primitives, e.g., we can exchange the elliptic curve used for the KEM.
In the end, we evaluate the BRKE implementation with our algorithm set. We perform the evaluation with four different pairing-friendly elliptic curves for the HIBE to analyze what effect the choice of the curve, and what effect the HIBE and the kuKEM have on the BRKE construction. The results show that the effect of the kuKEM depends on the flow of communication. If there are no crossing messages between two communicating users, the performance of BRKE is linear and can be roughly estimated. However, if there are crossing messages the kuKEM and, thus, the elliptic curve used for the pairings have a significant impact on the performance and the behavior of the BRKE construction. This shows that the kuKEM or more precisely the HIBE has the most impact on the performance of BRKE.

Further Work

There are many interesting aspects which can continue the work on the implementation of BRKE. Firstly, it can be interesting to compare the performance impact on different algorithms on BRKE. We already established that the HIBE has the most influence on the BRKE implementation, but it still might be interesting to test other alternatives for the one-time signature, KEM and the other algorithms. For this, we can use the algorithm overviews provided in Chapter 3. However, since the HIBE has the highest impact on the performance, the HIBE probably is the most interesting algorithm which we can change to optimize the performance of BRKE. We could think about changing the HIBE to a bounded HIBE so that the HIBE has a more consistent performance. However, at the same time, we would have to think about a suitable depth of the HIBE. Furthermore, we have to think about how the BRKE implementation should handle if the maximal depth is reached. As pointed out by Poettering and Rösler [77] the maximum depth is bounded by the number of ciphertexts sent by a user during one round-trip time (RTT) on the network between the user and its communication partner. So an idea would be dynamically setting the maximum depth of the HIBE. Another aspect which affects the performance of the HIBE is the choice of the elliptic curve. As we discussed, BN-P256 is a highly optimized curve, but only provides 100-Bit security. If computations on the other

pairing-friendly curves are further optimized, performance for the 128-Bit security level might achieve a similar performance level. The Relic library is still under development, so the performance might still improve.
Before employing the implementation in a real-world application, we would also have to perform a security analysis on the implementation. Our implementation is theoretically secure which means we use algorithms which individually fulfill the security requirements set by Poettering and Rösler [77], but we do not analyze the security of the implementation against side-channel attacks like timing attacks, for example. Furthermore, we would have to compute the actual security level of our algorithm sets by using the security proofs of Poettering and Rösler [77] with our algorithm and parameter choices.
Independently from the optimization we can apply to the BRKE implementation, we would have to think about reasonable specifications for a real-world application. So we would have to determine what is an acceptable time a key generation with BRKE can take before it is user unfriendly. In this context it also might be interesting to compare the performance of BRKE with other ratcheting-based protocols.

Bibliography

[1] Statista: Number of Mobile Phone Messaging App Users Worldwide From 2016 to 2021 (in Billions). `https://www.statista.com/statistics/483255/number-of-mobile-messaging-users-worldwide/`, 2017.

[2] WhatsApp Blog: Connecting One Billion Users Every Day. `https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day`, 2017.

[3] Kryptographische Verfahren: Empfehlungen und Schluessellaengen [Cryptographic Techniques: Recommendations and Keylengths]. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2018.

[4] Algorithms, Key Size and Protocols Report (2018). Technical report, ECRYPT-CSA, 2018.

[5] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient Lattice (H)IBE in the Standard Model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 553–572. Springer, 2010.

[6] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013. ISSN 2190-8508. doi: 10.1007/s13389-013-0057-3. URL `http://dx.doi.org/10.1007/s13389-013-0057-3`.

[7] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[8] Diego F. Aranha. Pairings Are Not Dead, Just Resting. *ECC 2017*, 2018.

[9] Emil Artin. *Geometric Algebra*. Courier Dover Publications, 2016.

[10] Razvan Barbulescu and Sylvain Duquesne. Updating Key Size Estimations for Pairings. *Journal of Cryptology*, pages 1–39, 2018.

[11] Paulo SLM Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[12] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *International Conference on Security in Communication Networks*, pages 257–267. Springer, 2002.

[13] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In *Annual International Cryptology Conference*, pages 602–619. Springer, 2006.

[14] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[15] Mihir Bellare, Oded Goldreich, and Anton Mityagin. The Power of Verification Queries in Message Authentication and Authenticated Encryption. *IACR Cryptology ePrint Archive*, 2004:309, 2004.

[16] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted Encryption and Key Exchange: The Security of Messaging. In *Annual International Cryptology Conference*, pages 619–650. Springer, 2017.

[17] BlueKrypt. *Cryptographic Key Length Recommendation*, 2018 (accessed April, 2019). URL `https://www.keylength.com/`.

[18] Dan Boneh and Xavier Boyen. Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 2004.

[19] Dan Boneh and Matt Franklin. Identity-Based Encryption From the Weil Pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[20] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical Identity Based Encryption with Constant Size Ciphertext. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 440–456. Springer, 2005.

[21] Dan Boneh, Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-Ciphertext Security From Identity-Based Encryption. *SIAM Journal on Computing*, 36(5): 1301–1328, 2006.

[22] Dan Boneh, Emily Shen, and Brent Waters. Strongly Unforgeable Signatures Based on Computational Diffie-Hellman. In *International Workshop on Public Key Cryptography*, pages 229–240. Springer, 2006.

[23] Sean Bowe. BLS12-381: New zk-SNARK Elliptic Curve Construction. *zCash Blog*, 2017. URL `https://z.cash/blog/new-snark-curve/`.

[24] Ernest Brickell, David Pointcheval, Serge Vaudenay, and Moti Yung. Design Validations for Discrete Logarithm Based Signature Schemes. In *International Workshop on Public Key Cryptography*, pages 276–292. Springer, 2000.

[25] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the Security of the Winternitz One-Time Signature Scheme. In *International Conference on Cryptology in Africa*, pages 363–378. Springer, 2011.

[26] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS-a Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011.

[27] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM (JACM)*, 51(4):557–594, 2004.

[28] David Cash, Dennis Hofheinz, Eike Kiltz, and Chris Peikert. Bonsai Trees, or How to Delegate a Lattice Basis. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–552. Springer, 2010.

[29] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. `https://eprint.iacr.org/2016/1013`.

[30] Angelo De Caro and Vincenzo Iovino. jPBC: Java Pairing Based Cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, pages 850–855, Kerkyra, Corfu, Greece, June 28 - July 1, 2011. IEEE. URL `http://gas.dia.unisa.it/projects/jpbc/`.

[31] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[32] F Betül Durak and Serge Vaudenay. Bidirectional Asynchronous Ratcheted Key Agreement without Key-Update Primitives. Technical report, Cryptology ePrint Archive, Report 2018/889, 2018.

[33] Keita Emura and Takuya Hayashi. A Revocable Group Signature Scheme with Scalability from Simple Assumptions and Its Implementation. In *International Conference on Information Security*, pages 442–460. Springer, 2018.

[34] Facebook. Messenger Secret Conversations: Technical White Paper. `https://fbnewsroomus.files.wordpress.com/2016/07/secret_conversations_whitepaper-1.pdf`, 2016. Accessed: 2018-10-22.

[35] FasterXML. Jackson API. `https://github.com/FasterXML/jackson`.

[36] David Freeman, Michael Scott, and Edlyn Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of cryptology*, 23(2):224–280, 2010.

[37] Craig Gentry and Alice Silverberg. Hierarchical ID-Based Cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 548–566. Springer, 2002.

[38] Daniel Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, RFC Editor, August 2006. URL `https://tools.ietf.org/html/rfc7919`.

[39] Github:artjomb. List of Pairing Libraries. *Github*, 2019. URL `https://gist.github.com/artjomb/f2d720010506569d3a39`.

[40] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice-Hall, Inc., 1998.

[41] Aurore Guillevic. Comparing the Pairing Efficiency Over Composite-Order and Prime-Order Elliptic Curves. In *International Conference on Applied Cryptography and Network Security*, pages 357–372. Springer, 2013.

[42] Max Hoffmann. Lecture: Software Implementation of Cryptographic Schemes, May 2018.

[43] Leslie Hogben. *Handbook of Linear Algebra (Discrete Mathematics and Its Applications 81).* Chapman and Hall/CRC, 2016. ISBN 9781466507296.

[44] Jeremy Horwitz and Ben Lynn. Toward Hierarchical Identity-Based Encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 466–481. Springer, 2002.

[45] Qiong Huang, Duncan S Wong, and Yiming Zhao. Generic Transformation to Strongly Unforgeable Signatures. In *International Conference on Applied Cryptography and Network Security*, pages 1–17. Springer, 2007.

[46] Andreas Hülsing. W-OTS+–Shorter Signatures for Hash-Based Signature Schemes. In *International Conference on Cryptology in Africa*, pages 173–188. Springer, 2013.

[47] ISO/IEC 18033-2. Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric Ciphers. Standard, International Organization for Standardization, January 2004.

[48] Tibor Jager. Script: Digitale Signaturen [Digital Signatures], September 2018.

[49] Oracle Corporation Open JDK. JOL (Java Object Layout). `https://openjdk.java.net/projects/code-tools/jol/`.

[50] Jakob Jonsson. Security Proofs for the RSA-PSS Signature Schemes and its Variants. In *SECOND OPEN NESSIE WORKSHOP*. Citeseer, 2001.

[51] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* CRC press, 2007.

[52] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography.* CRC press, 1996.

[53] Eike Kiltz. Script: Cryptographic Protocols, July 2018.

[54] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal Security Proofs for Signatures from Identification Schemes. In *Annual International Cryptology Conference*, pages 33–61. Springer, 2016.

[55] Taechan Kim and Razvan Barbulescu. Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case. In *Annual International Cryptology Conference*, pages 543–571. Springer, 2016.

[56] Hugo Krawczyk and Pasi Eronen. Hmac-Based Extract-and-Expand Key Derivation Function (HKDF). 2010.

[57] Leslie Lamport. Constructing Digital Signatures From a One-Way Function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[58] Allison Lewko. Tools for Simulating Features of Composite Order Bilinear Groups in the Prime Order Setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 318–335. Springer, 2012.

[59] Allison Lewko and Brent Waters. Unbounded HIBE and Attribute-Based Encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 547–567. Springer, 2011.

[60] Jin Li, Fangguo Zhang, and Yanming Wang. A New Hierarchical ID-Based Cryptosystem and CCA-Secure PKE. In *International Conference on Embedded and Ubiquitous Computing*, pages 362–371. Springer, 2006.

[61] Manfred Lochter and Johannes Merkle. Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation. Technical report, 2010.

[62] Ben Lynn. PBC Library. *Online: http://crypto. stanford. edu/pbc*, 59:76–99, 2006.

[63] Ben Lynn. *On the Implementation of Pairing-Based Cryptosystems.* PhD thesis, Stanford University Stanford, California, 2007.

[64] Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm. `https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf`, 2016. Accessed: 2018-10-22.

[65] Ken Martin and Bill Hoffman. *Mastering CMake: a Cross-Platform Build System.* Kitware, 2010.

[66] Vincent Massol and Ted Husted. *JUnit in Action.* Manning Publications, 2003. ISBN 1930110995.

[67] Ueli M Maurer. Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms. In *Annual International Cryptology Conference*, pages 271–281. Springer, 1994.

[68] Kevin S McCurley. The Discrete Logarithm Problem. In *AMS Proc. Symp. Appl. Math*, volume 42, pages 49–74, 1990.

[69] Alfred Menezes. An Introduction to Pairing-Based Cryptography. *Recent trends in cryptography*, 477:47–65, 2009.

[70] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges With Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography. In *International Conference on Cryptology in Malaysia*, pages 83–108. Springer, 2016.

[71] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Apache Maven.* Alpha Press, 2010. ISBN 6130652194, 9786130652197.

[72] Payman Mohassel. One-Time Signatures and Chameleon Hash Functions. In *International Workshop on Selected Areas in Cryptography*, pages 302–319. Springer, 2010.

[73] Moni Naor and Moti Yung. Universal One-Way Hash Functions and Their Cryptographic Applications. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 33–43. ACM, 1989.

[74] Tatsuaki Okamoto and Katsuyuki Takashima. Homomorphic Encryption and Signatures from Vector Decomposition. In *International Conference on Pairing-Based Cryptography*, pages 57–74. Springer, 2008.

[75] Tatsuaki Okamoto and Katsuyuki Takashima. Hierarchical Predicate Encryption for Inner-Products. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 214–231. Springer, 2009.

[76] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners.* Springer Science Business Media, Berlin Heidelberg, 2009. ISBN 978-3-642-04101-3.

[77] Bertram Poettering and Paul Rösler. Towards Bidirectional Ratcheted Key Exchange. In *Annual International Cryptology Conference*, pages 3–32. Springer, 2018.

[78] David Pointcheval and Serge Vaudenay. On Provable Security for Digital Signature Algorithms. 1996.

[79] GitHub: Signal Messenger Protocol. libsignal-protocol-java. `https://github.com/signalapp/libsignal-protocol-java`, 2016.

[80] Geumsook Ryu, Kwangsu Lee, Seunghwan Park, and Dong Hoon Lee. Unbounded Hierarchical Identity-Based Encryption With Efficient Revocation. In *International Workshop on Information Security Applications*, pages 122–133. Springer, 2015.

[81] Jörg Schwenk. Script: Authenticated Key Exchange: An Introduction to its Formal Analysis, June 2018.

[82] Hovav Shacham. *New Paradigms in Signature Schemes*. PhD thesis, Stanford University, December 2005.

[83] Adi Shamir. Identity-Based Cryptosystems and Signature Schemes. In *Workshop on the theory and application of cryptographic techniques*, pages 47–53. Springer, 1984.

[84] Victor Shoup. Using Hash Functions as a Hedge Against Chosen Ciphertext Attack. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 275–288. Springer, 2000.

[85] Victor Shoup. A Proposal for an ISO Standard for Public Key Encryption (Version 2.1). *IACR e-Print Archive*, 112, 2001.

[86] The Legion of the Bouncy Castle Inc. Bouncy Castle API. URL `https://www.bouncycastle.org/java.html`.

[87] Frederik Vercauteren. Discrete Logarithms in Cryptography. *ESAT/COSICKU Leuven ECRYPT Summer*, 2008.

[88] Brent Waters. Efficient Identity-Based Encryption Without Random Oracles. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 114–127. Springer, 2005.

[89] WhatsApp. Whatsapp Encryption - Overview: Technical White Paper. `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`, 2017. Accessed: 2018-10-22.

[90] Chikara Yonezawa, Lepidum. IETF Draft: Pairing-Friendly Curves (WIP). *IETF Draft*, 2019. URL `https://tools.ietf.org/html/draft-yonezawa-pairing-friendly-curves-01`.

[91] Mingwu Zhang, Bo Yang, Chunzhi Wang, and Tsuyoshi Takagi. Unbounded Anonymous Hierarchical IBE With Continual-Key-Leakage Tolerance. *Security and Communication Networks*, 7(11):1974–1987, 2014.

[92] Shiwei Zhang. Dual Pairing Vector Space, October 2014. URL `https://www.uow.edu.au/~fuchun/seminars/031014.pdf`.

Acronyms

ARKE Asynchronous Ratcheted Key Exchange.

BRKE Bidirectionally Ratcheted Key Exchange.

BSI German Federal Office for Information Security.

CDH Computational Diffie-Hellman.

DDH Decisional Diffie-Hellman.

DLP Discrete Logarithm Problem.

DPVS Dual Pairing Vector Spaces.

ECDLP Elliptic Curve Discrete Logarithm Problem.

HIBE Hierarchical Identity-Based Encryption.

IBE Identity-Based Encryption.

JNI Java Native Interface.

JOL Java Object Layout.

KDF Key Derivation Function.

KEM Key Encapsulation Mechanism.

kuKEM key-updateable Key Encapsulation Mechanism.

MAC Message Authentication Code.

PKG Private Key Generator.

PRF Pseudo Random Function.

RKE Ratcheted Key Exchange.

ROM Random Oracle Model.

SRKE Sesquidirectionally Ratcheted Key Exchange.

URKE Unidirectionally Ratcheted Key Exchange.