Puncturable Symmetric KEMs for Forward-Secret 0-RTT Key Exchange

MATILDA BACKENDAL

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Puncturable Symmetric KEMs for Forward-Secret 0-RTT Key Exchange

Matilda Backendal

Department of Electrical and Information Technology
Lund University

Supervisors: Felix Günther, Thomas Johansson

Examiner: Fredrik Rusek

June 2019

Printed in Sweden
Tryckeriet i E-huset, Lund

Till Peter,
som lärde mig att det inte finns några dumma frågor.
Juni 2019

Abstract

The latest version of the Transport Layer Security protocol (TLS 1.3) introduces a pre-shared key zero round-trip time (0-RTT) mode. This enables session resumption with no latency before the first application data can be sent, at the cost of losing forward secrecy and replay protection. There is high demand from Internet companies for this performance-enhancing feature, and some service providers have chosen to already enable it by default despite the security compromise currently associated to it. In this work we explore the possibility to achieve forward secrecy for resumed sessions in 0-RTT mode, mitigating the security risks presently adherent to it.

To abstract the key exchange in TLS 0-RTT mode, we introduce a new primitive which we call *symmetric-key key encapsulation mechanisms* (S-KEMs). Forward secrecy is attained through "puncturing" of the secret key, which we capture formally by *puncturable S-KEMs* (PS-KEMs). Furthermore, to enable optimizations that leverage ordering and to achieve the greatest possible generality of our model, we also introduce stateful versions of S-KEMs and PS-KEMs. We examine the relationship between these new primitives, give game-based functionality and security notions and show how pseudorandom functions (specifically based on the Goldreich-Goldwasser-Micali construction) can be used to build instantiations which meet the security goals.

Sammanfattning

I den senaste versionen av kommunikationsprotokollet Transport Layer Security[1] (TLS 1.3), introducerades en funktion som möjliggör att sessioner återupptas i "zero round-trip time"[2] (0-RTT) vilket innebär att snabbare uppkoppling är möjlig eftersom krypterad applikationsdata kan skickas redan med det första dataflödet från klient till server. För att uppnå detta används krypteringsnycklar som har förhandlats fram i tidigare sessioner, så kallade "pre-shared keys"[3]. Tyvärr innebär den nya funktionen att kommunikationen inte längre är säker mot framtida korruption av dessa långlivade nycklar. Detta kallas för avsaknad av "forward secrecy"[4].

Trots de lägre säkerhetsgarantierna är efterfrågan på 0-RTT-funktionen stor bland internetleverantörer eftersom den ger en markant hastighetsökning, framförallt för mobila nätverk där latensen är hög. I det här arbetet undersöker vi möjligheten att behålla den snabba återuppkoppling, men samtidigt uppnå samma säkerhet som vid en vanlig anslutning. Detta görs genom en abstraktion av nyckelutbytet i en TLS-session, vilket vi modellerar med ett nytt kryptografiskt objekt som vi har döpt till "symmetriska nyckelinkapslingsmekanismer" (S-KEMs). Forward secrecy uppnås genom införandet av "nyckelpunktering", vilket vi modellerar med s.k. *punkterbara* S-KEMs (PS-KEMs). Vi formaliserar funktionalitet- och säkerhetsmålen med hjälp av spelbaserade definitioner för S-KEMs och PS-KEMs. Vi ger även exempel på hur algoritmer kan konstrueras för att uppnå målen, samt utvidgar modellen till att även innefatta ordningsföljden av återanslutna sessioner för att möjliggöra optimeringar.

[1] Översatt: Transportlagersäkerhet

[2] Översatt: noll tur- och returtid

[3] Översatt: tidigare delade nycklar

[4] Översatt: framåtsäkerhet

Acknowledgments

First and foremost, I owe the idea behind this work to my advisor Felix Günther. Thank you for taking me on as your first master thesis student and for the generosity with which you have treated me, both to your time and advice. Your continuous support is what made this thesis possible.

A sincere thank you also to my supervisor Professor Thomas Johansson for enabling the arrangements of this project, for welcoming me in the crypto group at LTH and for helping with formalities and logistics.

Lastly, I would like to express my gratitude to several people who—through small but significant acts—supported me in this project and helped shape the outcome. Thank you Joseph Jaeger, for never saying no to a question and for helping me imagine symmetric-key KEMs before they existed. Thank you to my mentor Malin Jonsson, for sharing your experiences and insights on thesis-writing. Thank you Erik Sandström, for keeping me motivated and for showing me that the only true failure is not trying. Thank you also to Marie Backendal Lundin, for encouraging me to take a walk (you are the guardian of my physical and mental well-being), to Malin Lundin, for understanding me better than anyone else, and to Peter Lundin, for always reading my work, for your questions and feedback and for being my greatest supporter. I owe you my curiosity.

Acronyms

0-RTT	Zero Round-Trip Time
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
OS-KEM	Ordered Symmetric-key Key Encapsulation Mechanism
PS-KEM	Puncturable Symmetric-key Key Encapsulation Mechanism
PRF	Pseudorandom Function
PRG	Pseudorandom Generator
PSK	Pre-Shared Key
POS-KEM	Puncturable Ordered Symmetric-key Key Encapsulation Mechanism
RFC	Request For Comments
S-KEM	Symmetric-key Key Encapsulation Mechanism
TLS	Transport Layer Security
URL	Uniform Resource Locator

Table of Contents

Chapter 1

Introduction and Overview

1.1 Introduction

The original goal of cryptography is to provide two parties, let's call them Alice and Bob, with tools that allow them to communicate openly, but securely, in the face of some adversary, Eve. By *openly*, we mean that the actual communication is not hidden from Eve. Rather, it occurs across a public and insecure channel and the adversary is both aware of the communication and assumed to have means to intercept and interact with it. The definition of *securely* is less straight-forward, but in classic cryptography the goal was mainly to achieve *privacy*, i.e. that the information exchanged between Alice and Bob could not be learned by Eve. The art of hiding vulnerable information from the curious eyes of enemies has been used for thousands of years in everything from warfare to love letters. Today, in the digital era, cryptography has become indispensable to the general public as the Internet has risen to be our main platform for communication.

Cryptography has likewise risen to the challenge and provides a wealth of tools for diverse tasks such as identification, authentication and private communication. Most people today use cryptography without even noticing it, for example each time they access a web page through `https`, sign a contract digitally or use a messaging app. Users of online services unconsciously assume that their data is protected, but still expect high performance from applications. Almost everyone becomes frustrated when a web page loads slowly, but few are those who would accept that their Internet bank publicly broadcast their account details to speed up the connection. The combined goal of security and performance is the main working area of modern cryptography, and while they may not seem incompatible they often end up going head-to-head.

One example of where performance and privacy end up competing for importance is in secure browsing. Whenever a user accesses a URL starting with `https`, a cryptographic protocol called TLS (Transport Layer Security [42]) is used to ensure that the communication is private. In order to do so, cryptographic material (such as secret keys) must first be negotiated between the communicating parties. The parties in the applications we are concerned with will generally be a user in a browser, called the *client*, and a service provider (e.g. the host of a web page) called the *server*. In order to establish the shared cryptographic parameters and keys, an initial phase of TLS called a *handshake* is run. This handshake does not contribute to the actual data exchange between client and server, rather it is added before the real communication begins to ensure that subsequent messages are secure. This means that a client accessing a web page via `https` will experience a slow-down compared to when browsing insecurely via for example `http` (notice the ending *s* of `https`, which stands for *secure*). The slow-down comes from *transmission latency* – the time it takes for a message to travel from sender to receiver across the network. The time of one message being sent, received and responded to is called one *round-trip*, and each round-trip in the handshake phase adds latency to the communication. More latency means slower responding applications and more frustrated users, so shaving off round-trips is a major goal in current development of Internet protocols.

Therefore, in the latest version of the Transport Layer Security protocol, TLS 1.3 [42], a new feature called zero round-trip time (0-RTT) resumption has been added. The novelty consists of allowing communication sessions to be resumed (i.e., restarted for a client and server who have been communicating previously) with zero round-trips in the TLS handshake phase. This means that the actual payload data is sent right away, and no time is spent on initially negotiating cryptographic parameters and exchanging keys before the real communication can begin. This leads to a significant speed up in resumed sessions, which make up around 40 % of the total connections [49]. 0-RTT resumption is possible thanks to the fact that the communicating parties have been connected previously, and already share some secret from their previous session(s). This pre-shared secret can be used again to derive keying material to encrypt the first message in the resumed session. However, the performance increase comes at a price.

When users of the current version of TLS 1.3 re-use the pre-shared secret across sessions, they lose a security property called *forward secrecy* [23, 31]. Briefly explained, forward secrecy means that if at some future point one of the communicating parties is compromised and the secret key they hold at that point is revealed, then all past communication sessions are still secure. Usually this is achieved by using keys derived from ephemeral key material, e.g. via a Diffie-Hellman key exchange [22], so that if the secret of one session is compromised, messages in all other sessions are still undecipherable to the attacker. In 0-RTT resumption, the pre-shared key is long-lived and kept during several sessions (up to 7 days in TLS 1.3 [42, p. 74]), and if it is compromised all session keys derived from it are too. Hence messages encrypted under keys derived from the pre-shared key are not forward-secret.

In this work, we explore the possibility to achieve forward secrecy whilst keeping the pre-payload communication at zero round-trips. This is done by introducing a new cryptographic object which we call *puncturable symmetric key encapsulation mechanisms*. Key encapsulation mechanisms (KEMs) are well-studied objects in public-key cryptography [15, 17, 18, 19, 46], and here we bring the idea over to the symmetric-key setting to match the context of session resumption. Puncturing is added to give the desired forward secrecy. Essentially it is an operation on the secret key which "punctures" the key, making it impossible to decrypt certain messages encrypted under earlier versions of the key. That way, previous sessions are safe against future compromises of the evolved key.

In this project we lay the theoretic foundation of puncturable symmetric-key

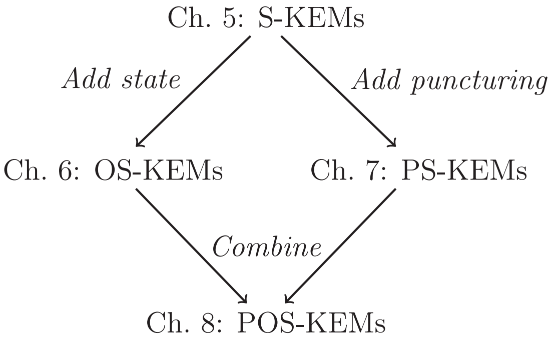

Figure 1.1: Overview of project.

KEMs, as well as an extension to ordered symmetric-key KEMs which capture the transmission reliability expected of the network across which the communication occurs. We introduce new cryptographic objects, defining both their functionality and the security goals they are expected to meet. The result of the project is a theoretical framework which can be used to reason about these objects, see how they fit with the intended application and help implementers construct secure schemes. We do give some examples of constructions in this work, as well discuss potential optimizations and ideas for instantiations to explore in the future, but we stress that the main contribution is the theory developed and not the particular constructions. To implement and optimize instantiations of puncturable symmetric-key KEMs will have to be the subject of future work.

1.2 Overview

First, in Chapter 2, we give some background and intuitive explanations of notions necessary for understanding the results of this project. A reader familiar with cryptography can skip it without any loss. Chapter 3 provides a more detailed overview of the results to come. In Chapter 4 we introduce the notation used in this text, explain briefly what a cryptographic game is as well as recap the definitions of some fundamental cryptographic objects. In Chapter 5 we define syntax, correctness and security of symmetric-key KEMs (S-KEMs), and give a construction which we prove is secure. With the foundations laid, we add properties to the basic S-KEM notion to bring us closer to our end goal. First, in Chapter 6, we add state (which can be used as a way to keep track of the order in which sessions are initiated) and define *ordered* S-KEMs (OS-KEMs). Next, we go back to the original stateless S-KEMs and add puncturing to get *puncturable* S-KEMs (PS-KEMs) as defined in Chapter 7. Finally, in Chapter 8 we combine the two branches to obtain *puncturable ordered* S-KEMs (POS-KEMs). Figure 1.1 shows an overview of the notions and illustrates their relations.

Chapter 2

Background

2.1 Symmetric-Key Cryptography

In symmetric-key cryptography, or *secret-key cryptography*, the parties wishing to communicate securely (e.g. Alice and Bob) share a secret key which they can use both to encrypt and, among other things, authenticate messages they exchange. The key can be a code word (as in the historic Vigenère cipher) or some piece of information which gives those holding it a mutual way of transforming plaintext into ciphertext (and reversing it). For further reading on historical ciphers, see for example [48]. In modern cryptography, a symmetric-key cipher could for example consist of a large set of permutations on a set of strings, which is both the message and ciphertext space. The secret key is then the precise permutation used to encrypt. Without it, the large number of possible permutations makes decryption infeasible. With it, it is a simple matter of evaluating the permutation on the ciphertext to get back the plaintext. Such a cipher is commonly referred to as a block cipher, and famous examples include DES (Data Encryption Standard) [21] and AES (Advanced Encryption Standard) [1].

The important thing to remember about secret-key cryptography is the symmetry; it requires that the involved parties have access to a common secret key. If Alice and Bob want to communicate, they need one such key. If Alice also wants to communicate privately with Charlie, Alice and Charlie need a different secret key. Any parties involved in private communication need a shared secret key for that specific session, meaning that the number of keys a user needs grows linearly with the number of communication partners they have. A difficulty associated with this is key distribution, i.e., ensuring that the parties are informed of the secret key in a secure way. This is a non-trivial problem and the difficulties associated to it spurred the invention of a different type of cryptography: public-key cryptography.

2.2 Public-Key Cryptography

In public-key cryptography, or *asymmetric cryptography*, a user (Alice) has a secret (or private) key sk and a public key pk. The public key is like an address; it is openly available to anyone who wishes to communicate with Alice, and (in the case of encryption) when applied to a message it encrypts it so that only Alice, who holds the secret key, can decrypt. Because of the asymmetry of the keys, Alice only needs one key-pair (sk, pk). Both Bob and Charlie can use Alice's public key to encrypt their messages to her, and neither will be able to read the other's communication because only Alice has the secret key that allows decryption. This effectively solves the key distribution problem, since each user simply broadcasts their public key openly. However public-key cryptography has its own drawbacks, such as being less efficient than symmetric primitives. For this reason, the public-key infrastructure is often used as a means of distributing symmetric keys, which are then used for the actual communication. One way to do this is by using a *key encapsulation mechanism*, which is a kind of encryption scheme for keys. We discuss it in more detail in Section 2.4.

2.3 Cryptographic Schemes: Syntax, Correctness and Security

This project revolves around the description of new cryptographic objects. The description is given in terms of schemes. A "scheme" is a plan, program or strategy. A cryptographic scheme is the description of a cryptographic object; a design for achieving a cryptographic goal. The base layer of a cryptographic scheme is its *syntax*, which specifies the structure and rules of the scheme. Typically we will define a scheme in terms of some mathematical object; it might be a function, in which case the syntax specifies the domain and range, or it might be a tuple consisting of algorithms, then the syntax specifies the possible input and output of each algorithm. An example could be a symmetric encryption scheme, which consists of a key generation algorithm, an encryption algorithm and a decryption algorithm. Syntax tells us what type the keys produced by key generation are (e.g. integers, strings of some specified length, prime numbers), what arguments the encryption algorithm takes (a secret key and a message for example) and whether each algorithm is deterministic or randomized.

On top of syntax, one typically specifies the *functionality* or *correctness* expected from the scheme. If the scheme is a tuple of algorithms, the functionality might say something about how the different algorithms interact with each other. In the symmetric encryption scheme example, a common correctness requirement would be that the decryption algorithm always recovers a plaintext message which has been encrypted using the encryption algorithm. This is necessary to rule out schemes which follow the specified syntax, but lack some important piece of functionality which renders them useless. Note that at this point, we still do not know anything about how the algorithms of a scheme work. We know what the domains and ranges are, and something about how they interact under specific circumstances, but the inner workings of each function or algorithm is not specified. This "black box" approach is used to obtain the greatest possible generality—to cover as many constructions that could potentially meet the intended goals as possible. A *construction* or *instantiation*, in contrast, is a specification of the procedures of a scheme and details exactly how the scheme works. An instantiation which has the specified syntax of a certain cryptographic object is simply called a scheme of that type. E.g., a construction that has the syntax of symmetric encryption is called a symmetric encryption scheme. If it additionally meets the correctness requirements it is called a "correct symmetric encryption scheme".

Not all correct schemes are of practical interest. Some may be inefficient or, more importantly, insecure. To know whether a scheme is insecure one needs to define what *security* means, which ties back to the original aim when designing the scheme. Security goals come in many flavours, such as privacy, authenticity, or unforgeability. In the symmetric encryption scheme example, the basic aim is likely to achieve privacy, i.e. that no one except the intended recipient can read an encrypted message. Security definitions are more fine-grained than so, however, and typically specify a precise goal such as "ciphertext indistinguishability from random bits" (which says that the goal is for encrypted messages to be difficult to tell apart from strings of bits chosen uniformly at random). To formally capture the security notion and be able to reason about whether a scheme meets the goals, security is often given in the form of a *game*. In the game, the scheme is subjected to an attack by an adversary who, while adhering to the rules of syntax and correctness, tries to break the security of the scheme. For more details on games, see Section 4.2. The abilities of the adversary are also specified in the game, and contribute with the attack scenario part of the security notion. That is, if in the symmetric encryption scheme example the adversary is assumed to have access to the encryption algorithm (to which it can input any plaintext message) and its target is to distinguish the resulting ciphertext from random for a message of its choice, the security notion might be called "ciphertext indistinguishability under chosen plaintext attack". Often the security notion is given some handy abbreviation "xxx" (such as ind-cpa, for $\underline{\text{ind}}$istinguishability under $\underline{\text{c}}$hosen $\underline{\text{p}}$laintext $\underline{\text{a}}$ttack), and a scheme that withstands all attacks in the specified scenario is called xxx secure (ind-cpa secure in our example).

2.4 Key Encapsulation Mechanisms

A key encapsulation mechanism (KEM) is a cryptographic primitive used to transport a symmetric key to a designated party in a public-key infrastructure. It is similar to a public-key encryption scheme with the difference that the encrypted (encapsulated) plaintexts are not arbitrary messages, but keys generated by the scheme. KEMs are typically used in combination with a symmetric encryption scheme to give an efficient public-key encryption scheme via so called "hybrid encryption". The concept was introduced and formalized by Cramer and Shoup [15, 18]. We give here an overview of a public-key KEM scheme following their definition.

Definition 1. A *key encapsulation mechanism*, $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encap}, \mathsf{Decap})$, is a triple of algorithms with four associated sets: the secret-key space $\mathcal{SK}$, the public-key space $\mathcal{PK}$, the key space $\mathcal{K}$ and the ciphertext space $\mathcal{C}$. The algorithms operate as follows.

- The probabilistic key generation algorithm, KeyGen, takes no input and produces a public key/secret key pair (sk, pk) with $sk \in \mathcal{SK}$, $pk \in \mathcal{PK}$.
- The randomized encapsulation algorithm, Encap, takes as input a public key $pk \in \mathcal{PK}$ and produces a pair (K, C) consisting of a key $K \in \mathcal{K}$ and an associated ciphertext $C \in \mathcal{C}$.
- The deterministic decapsulation algorithm, Decap, on input the secret key $sk \in \mathcal{SK}$ and a ciphertext $C \in \mathcal{C}$, returns either a key $K \in \mathcal{K}$ or the special symbol $\perp$ to indicate failure.

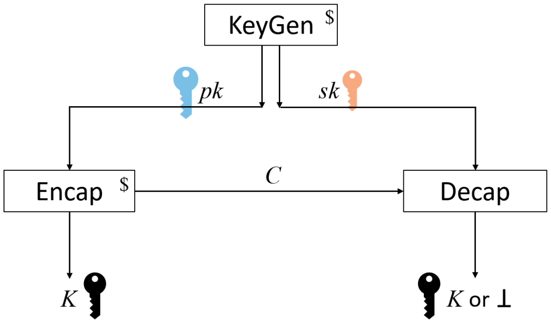

Figure 2.1: Illustration of the functioning of a public-key key encapsulation mechanism (KEM). A KEM scheme consists of three algorithms; KeyGen, Encap and Decap, the first two of which are randomized (indicated by $ in the figure.)

This specifies the syntax of a KEM scheme, illustrated in Figure 2.1. The functionality is captured in the following correctness requirement: For scheme KEM to be called correct, we require that decapsulation of a key $K \in \mathcal{K}$ under the secret key corresponding to the public key with which it was encapsulated always succeeds. Formally, for all (sk, pk) produced by KeyGen and all (K, C) output by $\mathsf{Encap}(pk)$ it holds that $\Pr[\,\mathsf{Decap}(sk, C) = K\,] = 1$, where the probability is over the randomness ("coins") of KeyGen and Encap.

In this project, we introduce a symmetric-key version of KEMs, which can be used to exchange fresh keys in a setting where the parties already share a secret. The problem we address bears similarities to that of *key wrapping*, posed by NIST in the 1990's and formally addressed by Rogaway and Shrimpton [43]. The goal of a key wrap algorithm is, as phrased by the American Standards Committee Working Group X9F1 in their request for comments, "to provide privacy and integrity protection for specialized data such as cryptographic keys ... without the use of nonces" [24]. Essentially, it is a primitive similar to symmetric encryption, but instead of encrypting arbitrary messages the plaintexts are cryptographic keys. The motivation for such algorithms is, for example, secure storage of keys, where one key is used to encrypt other keys so that they can be safely stored on an insecure disk.

Despite the similar goals, there are some fundamental differences that set key wrapping primitives apart from symmetric-key KEMs. For example, key wrap algorithms are deterministic, which S-KEMs are not required to be (in fact, none of our suggested constructions are). Secondly, the security goal of key wrap is privacy and integrity combined, and Rogaway and Shrimpton [43] hence define a security notion which covers both concurrently. For S-KEMs, we are mainly interested in privacy, and although we do formalize an integrity notion, we keep it separate from privacy. Lastly, the difference between key wrap algorithms and S-KEMs lies in their intended usage and hence in the extensions we consider. A key wrap algorithm could potentially be used to instantiate an S-KEM scheme, but the key wrapping problem is not concerned with forward secrecy nor transport of encapsulated keys across a network. Hence both ordering and puncturing is outside the scope of key wrap schemes.

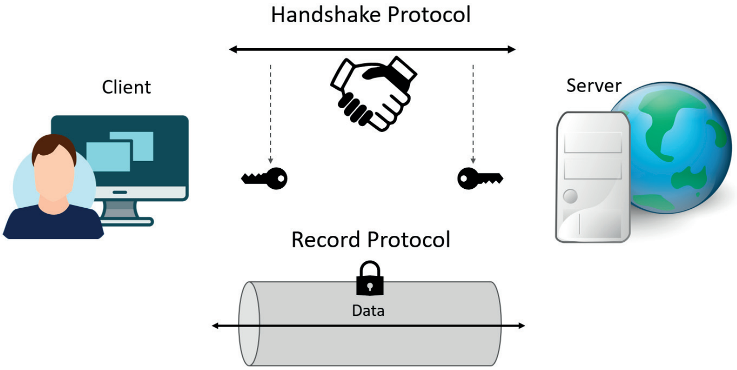

Figure 2.2: TLS is a cryptographic protocol for secure Internet communication. It consists of a handshake protocol (for key establishment) and a record protocol (for payload data exchange).

2.5 Network Protocols and TLS

A network protocol defines a set of rules which govern the communication between parties on a network. The protocol specifies how to format, transmit and receive data and ensures that the parties involved adhere to the common procedures. This enables communication between a multitude of devices regardless of their inherent differences.

Transport Layer Security (TLS) is a cryptographic network protocol designed to provide security for Internet communication. The Internet Engineering Task Force (IETF), responsible for publishing TLS as a standard, describe the protocol in the latest request for comments (TLS 1.3, RFC 8446) [42] as "TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery." They also state that "The primary goal of TLS is to provide a secure channel between two communicating peers." We will often refer to the peers as a client and a server. On a high level, the TLS protocol consists of two parts: the *handshake protocol* and the *record protocol*. In the handshake protocol, the parties negotiate security parameters and agree on the protocol version to use. The server authenticates itself (and the client optionally does too) and they agree on shared key material to be used in the record protocol, which is where the actual information (payload data) is exchanged. See Figure 2.2 for an illustration.

2.6 Sessions and Resumption

A session is a finite sequence of data transactions between communicating parties. In particular, a TLS session is a connection between a client and a server. It is initiated by the client connecting to the server, starts with the handshake phase and then moves on to the subsequent data transfer phase. The session ends when the connection is closed. Two entities can have several sessions running between them, both sequentially (one after the other) and concurrently (in parallel).

A resumed session is a new session between a client and server who have previously communicated and already share a long-term secret (a so called “pre-shared secret”). In session resumption, the time spent in the handshake phase before payload data is sent may be shorter than in a completely fresh session due to the use of previously established secrets. The same long-term pre-shared secret can be used to exchange fresh session keys for multiple resumed sessions within its lifetime. Session resumption and pre-shared secrets is what allows the zero round-trip time mode recently introduced in TLS 1.3. With this novel mode enabled, no round-trips are needed in the handshake of a resumed session before the first payload data is sent. Instead, the pre-shared secret is used to ensure the confidentiality of the very first data flow from client to server. Unfortunately, this has the drawback that the data encrypted with the long-term secret is not secure if this secret is later exposed. Addressing issues like this is the goal of a security notion called *forward secrecy*, which we introduce next.

2.7 Forward Secrecy

A cryptographic scheme provides forward secrecy if exposure of the secret key does not compromise the privacy of past communication. The term was first introduced by Günther [31] in the setting of authenticated key exchange, and was later mentioned as a desirable protocol characteristic by Diffie et al. [23]. Today it is considered an essential security goal for key exchange and has been studied extensively for such protocols following the work of Canetti and Krawczyk [14]. However, forward secrecy has also expanded beyond key exchange and been studied in the context of private-key cryptography [9], asymmetric cryptography [36, 13], digital signatures [2, 6] and secure channels [33].

In the setting of key encapsulation mechanisms (where to date there are surprisingly few studies of it), forward secrecy requires that corruption of the secret key does not compromise the confidentiality of previously encapsulated keys. In TLS session resumption – our intended application – this means that exposure of the long-term secret should not imply a loss of privacy for messages exchanged in past sessions.

Chapter 3

Overview of Results

Having laid the foundation we are now ready to provide a more detailed overview of the results of this work. To model the way TLS uses a resumption key to allow session resumption in 0-RTT, we introduce and define symmetric-key key encapsulation mechanisms (S-KEMs) in Chapter 5. Like KEMs, the idea behind S-KEMs is to enable two parties to establish a key in a one-shot message without further interaction, but from a shared symmetric key rather than in the public-key setting. Beyond giving the syntax and functionality of S-KEM schemes, we also define three security goals—two privacy notions and one integrity notion—and show that they relate in the same way as in symmetric encryption. We also give an example of a secure instantiation based on so-called 'pseudorandom functions' and describe how it captures a 0-RTT TLS handshake.

In TLS, and also in the S-KEM model we have devised, a pre-shared key allows for more than one subsequent resumed session to be established. In fact, not only multiple sequential sessions are possible, but also several ones running in parallel. Since the connection requests are sent across a network, they may or may not arrive to the recipient in the same order they were created. Depending on the degree of transmission reliability, the communicating parties could potentially leverage the ordering of their resumed sessions to obtain more efficient constructions. To enable this, we add state and define ordered S-KEMs (OS-KEMs) in Chapter 6.

To capture settings with varying network reliability—and hence varying degrees of ordering expected—we give three levels of correctness for OS-KEMs: no ordering, weak ordering and perfect ordering. The first essentially has the functionality of stateless S-KEM schemes, and demands correct handling under any (re-)ordering. Perfect ordering expects the network to be completely reliable, and only handles received messages if they arrive in the exact order sent. Such a setting would not need any communication between client and server, as they could simply follow a predetermined sequence of sessions keys (e.g. using a hash chain). In reality, however, we expect that some connection requests may cross in flight or be lost in transmission, and to handle such "local re-ordering" we introduce a class called weak ordering. In this correctness class, re-ordering within some sliding window is permitted, but from a bird's-eye view we still expect the communication to appear ordered. This allows some of the efficiency optimizations possible under perfect ordering to carry over. In this chapter, we also explore how OS-KEM schemes could be made robust, meaning that they are required to recover from out of order, unsupported connection requests. Finally, we give a construction for each correctness class.

The main goal of this work is to define primitives for forward-secret 0-RTT key exchange, but neither S-KEMs nor OS-KEMs enable this. To obtain the desired forward secrecy, we add a puncturing operation to S-KEMs and define puncturable S-KEMs (PS-KEMs) in Chapter 7. As forward secrecy is the core target of the project, some time is spent on developing a construction that achieves this type of security. The idea is to use the Goldreich-Goldwasser-Micali pseudorandom function construction [28], which is explained in detail in Section 7.4. Since PS-KEMs are stateless, there is no concept of ordering. To enable more efficient constructions, we finally tie together the theory developed for OS-KEMs and PS-KEMs to define puncturable ordered S-KEMs (POS-KEMs) in Chapter 8. Like for OS-KEMs, three correctness classes with associated privacy notions are introduced. Due to time constraints, no formal analysis of a POS-KEM construction is given, but we envision instantiations based on the OS-KEM and PS-KEM constructions given in earlier chapters.

Chapter 4

Basic Definitions

4.1 Notation and Conventions

If a is a string then $|a|$ denotes its length. A prefix of string $a = b_1 b_2 \dots b_n$ (where b_i are individual characters) is a string $a' = b_1 b_2 \dots b_m$ where $m \leq n$. We use $a_1 \| a_2 \| \cdots \| a_n$ as shorthand for the concatenation of strings $a_1, a_2, \dots, a_n$. By $\{0,1\}^n$ we denote the set of all binary strings of length n. By $\{0,1\}^*$ we denote the set of all binary strings of any length, including the empty string, which is denoted ε.

If S is a finite set, we let $x \leftarrow\!\!{\scriptstyle\$}\; S$ denote picking an element of S uniformly at random and assigning it to x, and we let $|S|$ denote the size of S. All sets/spaces associated to cryptographic schemes are assumed non-empty unless otherwise specified. $\mathbb{N}$ is the set of positive integers $\{1, 2, \dots\}$. By $\wedge$ we denote the logical AND operator, and by $\vee$ inclusive OR. If x is an n-tuple, then by $(a_1, a_2, \dots, a_n) \leftarrow x$ we mean that x is parsed into its constituents, which are then individually accessible through variables $a_1, a_2, \dots, a_n$. If we assign a tuple to a variable through $x \leftarrow (a_1, a_2, \dots, a_n)$, then we assume an implicit encoding which allows the individual elements to be recovered from x by parsing it into its constituents.

We use $\perp$ (bot) as a special symbol to denote rejection, and it is assumed to not be in $\{0,1\}^*$. Both inputs and outputs to algorithms can be $\perp$. We adopt the convention that if any input to an algorithm is $\perp$, then its output is $\perp$ as well. When specifying syntax, we sometimes write $y/\perp \leftarrow A()$ to explicitly show that the output of algorithm A is a string y or $\perp$. We use ♇ (pluto) to denote puncturing and assume that it is not in $\{0,1\}^*$. Algorithms may be randomized unless otherwise indicated. Some procedures maintain a table (e.g. $\mathsf{T}[\cdot]$) all of whose entries are initially assumed to be $\perp$.

4.2 Games

We use the code-based game-playing framework of Bellare and Rogaway [8]. A game $\mathbf{G}$ is a program written in pseudocode and is used to define a cryptographic security notion. It is often parameterized by a cryptographic scheme S and is run together with an *adversary*, $\mathcal{A}$, which in the language of games is an algorithm which plays the game with the goal of winning. The game should be seen as a "test" of the scheme, and the security of S is judged by the probability that any adversary can “break it” (win). A game consists of a main part (the “body” of the game) and possibly a number of *oracles*, which are procedures specified in the code of the game that provide certain functions to the adversary. (See Figure 4.1 and Figure 5.1 for examples.) Both games and adversaries keep *state*, i.e. internal memory that allows them to store variables. What it means to “win” a game is defined in terms of the so called *advantage* function. The advantage of an adversary is a numerical value (often between 0 and 1 or -1 and 1), and the higher the advantage the more successful the adversary is considered at playing the game. Hence a scheme is more secure the lower the highest advantage of any adversary playing the security game is.

Games are used to define security goals for cryptographic schemes, but also appear ubiquitously in security proofs for specific constructions. In the latter case, when games are used in proofs, it is often to showcase a *reduction* of the security of the scheme at hand to some other object, which is already known to be secure. Such a proof usually goes along the lines of “Object S_1 is secure, and object S_2 is built from S_1. We can show that if S_2 is insecure, then so must S_1 be. But this is a contradiction, hence S_2 is secure.” When games are used in reductions, they often appear as a sequence $(\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2 \ldots)$ with slight modifications separating a game in the sequence from the next. The differences are used to bound the advantage of an adversary playing the original game by the advantage change from each game in the sequence to the next, together with the advantage of the final game which is often easy to determine (e.g. because it is impossible to win). For a more thorough explanations of how games are used in security proofs, see e.g. [47].

In the game body, which is run exactly once, the first stage is *initialization* in which game variables are initialized. This stage ends by a call to the adversary, potentially on some input produced in the initialization of $\mathbf{G}$. The adversary is then allowed to run and make queries (procedure calls) to the oracles, each from which it may obtain some result as output from the procedure call. The adversary eventually halts and potentially returns some output. If $\mathcal{A}$ is an algorithm, such as an adversary, we let $y \leftarrow \mathcal{A}^{\mathrm{O}_1,\cdots}(x_1, \ldots ; r)$ denote running $\mathcal{A}$ on inputs $x_1, \ldots$ and coins r, with oracle access to $\mathrm{O}_1, \ldots$, and assigning the output to y. Coins here refers to a uniformly random bit string r, from which the algorithm can obtain randomness in the form of “coin tosses”. We use the shorthand notation $y \leftarrow\!\!\$\; \mathcal{A}^{\mathrm{O}_1,\cdots}(x_1, \ldots)$ to denote picking r at random and letting $y \leftarrow \mathcal{A}^{\mathrm{O}_1,\cdots}(x_1, \ldots ; r)$. We let $[\mathcal{A}^{\mathsf{O}_1,\cdots}(x_1, \ldots ; r)]$ denote the set of all possible outputs of $\mathcal{A}$ when run on inputs $x_1, \ldots$, coins r and with oracle access to $\mathsf{O}_1, \ldots$. If r is omitted, the set includes the possible outputs for all coins $r \in \{0, 1\}^*$.

The output of the adversary is used in the second part of the game body, *finalization*, after which the game halts and returns the game outcome x, where x is typically a string or boolean variable. By $\Pr[\,\mathbf{G}(\mathcal{A}) \Rightarrow x\,]$ we denote the probability that the execution of game $\mathbf{G}$ with adversary $\mathcal{A}$ results in the game outcome taking value x. By true and false we denote the boolean values of true and false. A $\neg$ preceding a boolean variable denotes negation, e.g. $\neg\mathsf{true} = \mathsf{false}$ and $\neg\mathsf{false} = \mathsf{true}$. Sometimes the shorthand notation $\Pr[\,\mathbf{G}(\mathcal{A})\,]$ will be used as an abbreviation for $\Pr[\,\mathbf{G}(\mathcal{A}) \Rightarrow \mathsf{true}\,]$. In games, integer variables, strings, set variables and boolean variables (such as the win flag) are assumed initialized, respectively, to 0, the empty string ε, the empty set $\emptyset$, and false, unless otherwise specified.

```
Game $\mathbf{G}^{\mathrm{prg}}_{1,\mathsf{G}}(\mathcal{A})$
  $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Fn}}()$
  Return $b^*$

$\textsc{Fn}()$:
  $\mathrm{s} \leftarrow_{\$} \{0,1\}^k$
  Return $\mathsf{G}(\mathrm{s})$
```

```
Game $\mathbf{G}^{\mathrm{prg}}_{0,\mathsf{G}}(\mathcal{A})$
  $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Fn}}()$
  Return $b^*$

$\textsc{Fn}()$:
  $y \leftarrow_{\$} \{0,1\}^{k+l}$
  Return $y$
```

Figure 4.1: Games formalizing PRG security of pseudorandom generator $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{k+l}$. Game $\mathbf{G}^{\mathrm{prg}}_{1,\mathsf{G}}$ is the "real world", where oracle $\textsc{Fn}$ returns real evaluations of G on a random seed. Game $\mathbf{G}^{\mathrm{prg}}_{0,\mathsf{G}}$ is the "random world". There oracle $\textsc{Fn}$ returns a string sampled uniformly at random from $\{0,1\}^{k+l}$.

4.3 Pseudorandom Generators

True randomness (in the sense of bits sampled from a completely uniform distribution) is difficult to obtain in practice, yet crucial for the security of many cryptographic primitives. To resolve this, we settle for the next best thing, namely *pseudorandomness*. A pseudorandom distribution appears random in all tests (at least of polynomial-time complexity), but bits from it can be generated much faster than "truly random" bits from just a small source of real randomness. This expansion of randomness is performed by a so called pseudorandom generator.

Definition 2. A *pseudorandom generator* (PRG) is a function $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{k+l}$ taking as input a seed $s \in \{0,1\}^k$ and producing output $\mathsf{G}(s) \in \{0,1\}^{k+l}$. The number l of additional bits in the output is called the *stretch* of the PRG.

A PRG is *secure* if the output is indistinguishable from strings sampled randomly from the uniform distribution. To formally define security, we give two games and define the advantage of an adversary playing them. The games are shown in Figure 4.1. They can be viewed as a test of the strength of the PRG, reminiscent of the "Turing test" for artificial intelligence. The idea is that an adversary $\mathcal{A}$ is placed either in the "real world" (if playing game $\mathbf{G}^{\mathrm{prg}}_{1,\mathsf{G}}$) or the "random world" (if playing game $\mathbf{G}^{\mathrm{prg}}_{0,\mathsf{G}}$) and by making queries to oracle $\textsc{Fn}$—which behaves differently in the real and random world—its goal is to determine which world it is in. It does so by returning a bit b^*, which should match the bit parameterizing the game it "thinks" it is playing; 1 for $\mathbf{G}^{\mathrm{prg}}_{1,\mathsf{G}}$ and 0 for $\mathbf{G}^{\mathrm{prg}}_{0,\mathsf{G}}$. We define the advantage of any adversary $\mathcal{A}$ against the PRG security of G as

$$\mathbf{Adv}^{\mathrm{prg}}_{\mathsf{G}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\mathrm{prg}}_{1,\mathsf{G}}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{G}^{\mathrm{prg}}_{0,\mathsf{G}}(\mathcal{A}) \Rightarrow 1\right].$$

4.4 Pseudorandom Functions

Sometimes, access to pseudorandomness in the form of bit strings generated by a PRG is not enough, and we want instead a *function* which is close to a "random function". For this purpose we use pseudorandom functions.

Definition 3. A *pseudorandom function* (PRF) is a function $\mathsf{F} : \{0,1\}^k \times \{0,1\}^{\text{in}} \to \{0,1\}^{\text{out}}$ which takes two inputs, a key $K \in \{0,1\}^k$ and a label $x \in \{0,1\}^{\text{in}}$ and gives some output in the range $\{0,1\}^{\text{out}}$.

We can equivalently view F as a family of keyed functions, which for each $K \in \{0,1\}^k$ defines $\mathsf{F}_K : \{0,1\}^{\text{in}} \to \{0,1\}^{\text{out}}$ where we let $\mathsf{F}_K(x) = \mathsf{F}(K, x)$ for all $x \in \{0,1\}^{\text{in}}$.

A "good" PRF produces output indistinguishable from random to an adversary without knowledge of the key. Formally, we define the advantage $\mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{A})$ of an adversary $\mathcal{A}$ attacking the PRF security of F to be the probability that it distinguishes between the games $\mathbf{G}^{\text{prf}}_{1,\mathsf{F}}$ (real) and $\mathbf{G}^{\text{prf}}_{0,\mathsf{F}}$ (random):

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{prf}}_{1,\mathsf{F}}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{G}^{\text{prf}}_{0,\mathsf{F}}(\mathcal{A}) \Rightarrow 1\right].$$

We omit the code and briefly explain the games instead. In both games, the adversary has access to oracle $\textsc{Fn}$ which returns either real evaluations of F or lazily sampled random strings depending on which game it is. Game $\mathbf{G}^{\text{prf}}_{1,\mathsf{F}}$ is initialized by picking $K \leftarrow_{\$} \{0,1\}^k$ and answers oracle query $\textsc{Fn}(x)$ from $\mathcal{A}$ with the real output of $\mathsf{F}_K(x)$. Game $\mathbf{G}^{\text{prf}}_{0,\mathsf{F}}$ responds to oracle query $\textsc{Fn}(x)$ with a random element $y \in \{0,1\}^{\text{out}}$. However, a repeated query is always given the same response. Formally the game maintains table $\mathsf{T}_{\mathsf{F}}[\cdot]$, and answers query $\textsc{Fn}(x)$ by checking if $\mathsf{T}_{\mathsf{F}}[x] = \bot$ (meaning x has not been queried before). If the entry is uninitialized, the game picks $y \leftarrow_{\$} \{0,1\}^{\text{out}}$ and assigns $\mathsf{T}_{\mathsf{F}}[x] \leftarrow y$, then returns the value stored in $\mathsf{T}_{\mathsf{F}}[x]$. (This is referred to as "lazy sampling", since the table entry is only initialized at position x and given a randomly sampled string after x is queried to $\textsc{Fn}$.) In both games, the adversary is run until it halts and outputs a decision bit, which is also the output from the game. Intuitively, the adversary attempts to "guess" which game it is in, and the value of the bit corresponds to its decision; 1 if it "thinks" it is in the real game $\mathbf{G}^{\text{prf}}_{1,\mathsf{F}}$ and 0 if it "thinks" it is in the random game $\mathbf{G}^{\text{prf}}_{0,\mathsf{F}}$.

4.5 Birthday Bound

In some applications, cryptographic schemes built on PRFs are susceptible to so called "birthday attacks". These attacks are named after the birthday paradox, which is a fancy name for the fact that among a relatively small group of people, the probability that any two share the same birthday is surprisingly high. Imagine that the group consists of q people, and let's for a moment assume that the year has exactly $n = 365$ days. The "paradox" arises because it seems intuitive that the collision probability (the chance that any two birthday's collide) should be roughly $\frac{q}{365}$, but in fact it is much closer to $\frac{q^2}{365}$, with the consequence that the probability increases much more quickly with the size of the group than one would intuitively expect. Why is this? Let's investigate the math behind the birthday paradox a bit closer.

Let $C(n,q)$ denote the probability that out of q elements drawn uniformly at random from a set of size n, any two elements coincide. We call this the *collision* probability. It is usually easier to determine the complement of the collision probability, i.e. the likelihood that all q elements are distinct. Imagine drawing one element at a time from the set (with replacement). The probability that the first is distinct is 1, as there is no other element it could collide with. Drawing the second element, the probability that it does not collide with the first is $\frac{n-1}{n}$. The chance that the third element drawn is distinct given that the two previous elements are unique is $\frac{n-2}{n}$. By simple combinatorics, continuing the reasoning along these lines gives

$$1 - C(n,q) = 1 \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-(q-1)}{n} = \prod_{i=1}^{q-1} \left(\frac{n-i}{n} \right).$$

Hence $C(n,q) = 1 - \prod_{i=1}^{q-1} \left(\frac{n-i}{n} \right)$. This formula gives the exact collision probability, however often an approximation will suffice for our purposes, or we are interested only in bounding the chance of collision. It can then be useful to know the following bounds:

$$0.3 \cdot \frac{q(q-1)}{n} \leq C(n,q) \leq 0.5 \cdot \frac{q(q-1)}{n}. \tag{4.1}$$

The upper bound is unconditional. The lower holds if $1 \leq q \leq \sqrt{2n}$. For a derivation of the bounds, see for example the appendix of [4].

Chapter 5

Symmetric-key KEMs

In this chapter, we introduce a new cryptographic object which we call symmetric-key key encapsulation mechanisms (S-KEMs). S-KEMs are an abstraction of the key exchange in a 0-RTT handshake in TLS 1.3. The syntax draws heavily on key encapsulation mechanisms in the public-key setting [15, 16], see Section 2.4.

5.1 Syntax

Definition 4. A *symmetric-key key encapsulation mechanism*, $\mathsf{S\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$, is a triple of algorithms with three associated sets; the secret-key space $\mathcal{SK}$, the key space $\mathcal{K}$ and the ciphertext space $\mathcal{C}$. The algorithms of $\mathsf{S\text{-}KEM}$ operate as follows.

- Via $sk \leftarrow_{\$} \mathsf{KG}()$, the probabilistic key generation algorithm KG, taking no input, produces the secret key $sk \in \mathcal{SK}$.
- Via $(K, C) \leftarrow_{\$} \mathsf{E}(sk)$, the randomized encapsulation algorithm, E, produces a key $K \in \mathcal{K}$ and an associated ciphertext $C \in \mathcal{C}$ on input the secret key.
- Via $K \leftarrow \mathsf{D}(sk, C)$, the deterministic decapsulation algorithm, D, on input a secret key $sk \in \mathcal{SK}$ and a ciphertext $C \in \mathcal{C}$ returns either a key $K \in \mathcal{K}$ or, to indicate failure, $\perp$.

For correctness we require that decapsulation of a key $K \in \mathcal{K}$ under the same secret key with which it was encapsulated always succeeds. Formally, for $sk \leftarrow_{\$} \mathsf{KG}()$ and $(K, C) \leftarrow_{\$} \mathsf{E}(sk)$ it holds that $\Pr[\, \mathsf{D}(sk, C) = K \,] = 1$, where the probability is over the coins of KG and E.

5.2 Security

For the security of S-KEMs we consider first two standard privacy notions: indistinguishability under *chosen-plaintext* attack (ind-cpa) and indistinguishability under *chosen-ciphertext* attack (ind-cca). In the chosen-plaintext scenario, the adversary is assumed to be passively eavesdropping and is only given access to an encapsulation oracle, which returns either a real encapsulation (a ciphertext and the corresponding key) or a ciphertext together with a randomly chosen independent key. The ind-cpa security notion requires that an attacker cannot distinguish

```
Game $\mathbf{G}^{\mathrm{cpa}}_{b,\mathsf{S\text{-}KEM}}(\mathcal{A})$
    $sk \leftarrow_{\$} \mathsf{KG}()$
    $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Enc}_b}()$
    Return $b^*$

$\textsc{Enc}_b()$:
    $(K_1, C) \leftarrow_{\$} \mathsf{E}(sk)$
    $K_0 \leftarrow_{\$} \mathcal{K}$
    Return $(K_b, C)$
```

```
Game $\mathbf{G}^{\mathrm{cca}}_{b,\mathsf{S\text{-}KEM}}(\mathcal{A})$
    $sk \leftarrow_{\$} \mathsf{KG}();\ S \leftarrow \emptyset$
    $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Enc}_b,\textsc{Dec}}()$
    Return $b^*$

$\textsc{Enc}_b()$:
    $(K_1, C) \leftarrow_{\$} \mathsf{E}(sk)$
    $K_0 \leftarrow_{\$} \mathcal{K}$
    $S \leftarrow S \cup \{C\}$
    Return $(K_b, C)$

$\textsc{Dec}(C)$:
    If $C \in S$ return $\perp$
    $K \leftarrow \mathsf{D}(sk, C)$
    Return $K$
```

Figure 5.1: Games formalizing ind-cpa and ind-cca security of symmetric-key KEM scheme S-KEM.

between the real and the random world. In a chosen-ciphertext attack, the adversary is additionally given access to a decapsulation oracle which returns honest decapsulations on any ciphertext queries which the attacker has not previously obtained from its encapsulation oracle. Once again security requires that no adversary can tell apart a game played in the real world from one in the random world.

We formalize the security notions via the games $\mathbf{G}^{\mathrm{cpa}}_{b,\mathsf{S\text{-}KEM}}$ and $\mathbf{G}^{\mathrm{cca}}_{b,\mathsf{S\text{-}KEM}}$ shown in Figure 5.1. The bit $b \in \{0,1\}$ is a parameter of the game which determines whether it is the "real" or the "random" version of the game that is being played. The games take an adversary as input. For cxa $\in \{\mathrm{cpa}, \mathrm{cca}\}$ we let $\mathbf{Adv}^{\mathrm{cxa}}_{\mathsf{S\text{-}KEM}}(\mathcal{A})$ denote the advantage of an adversary $\mathcal{A}$ playing game $\mathbf{G}^{\mathrm{cxa}}_{b,\mathsf{S\text{-}KEM}}$ and define it to be

$$\mathbf{Adv}^{\mathrm{cxa}}_{\mathsf{S\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\mathrm{cxa}}_{1,\mathsf{S\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\mathrm{cxa}}_{0,\mathsf{S\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right],$$

i.e. the probability that adversary $\mathcal{A}$ outputs 1 in the real world, minus the probability that it returns 1 in the random world. Note that ind-cca is a stronger notion than ind-cpa, since any adversary against the ind-cpa security of a scheme could use the same strategy in the ind-cca game (by simply not querying the decapsulation oracle), and this would preserve its advantage. Hence ind-cca security implies ind-cpa security, so a scheme which achieves indistinguishability under chosen-ciphertext attacks is naturally secure against chosen-plaintext attacks.

Indistinguishability is a common privacy goal for encryption primitives as it implies many desirable properties, and it turns out to cover well what we want from privacy in key encapsulation mechanisms too. The idea is that if a random key-ciphertext pairing is indistinguishable from a real encapsulated one, then the real key must be computationally independent from the ciphertext. This in turn means that transporting the key across the public channel by sending the ciphertext that encapsulates it, is equivalent to not sending anything at all and simply delivering the key in a trusted way to the intended recipient. As another implication, an adversary which cannot tell a random string from a real encapsulated key definitely cannot decapsulate the key and break the confidentiality of messages encrypted under it. Hence indistinguishability captures also this intuitive base for privacy.

The point of giving the adversary access to several "challenge" pairings through the encapsulation oracle is that we want to make it possible for many resumed sessions to be established securely from the same pre-shared secret. By giving the adversary access to honest decapsulations, the model captures that the sessions should be independent in the sense that learning a real key-ciphertext pairing for a non-challenge session doesn't reveal anything that helps determine if the challenge pairings are real or random. It also shouldn't help to intersect a ciphertext sent from client to server, modify it and then relay it to the recipient, who decapsulates and uses it. Hence, impersonating one party to the other should not provide an adversary with any information that helps it learn anything about honest communications between the real client and server. If one wants additional protection against modifications of ciphertexts, it can be formalized as a separate security goal. We look more closely at this in Section 5.4, but first we give an instantiation of a S-KEM scheme which achieves privacy in both the ind-cpa and the stronger ind-cca sense.

5.3 Instantiation using Pseudorandom Functions

In the 0-RTT mode of TLS 1.3 [42], the previously connected client and server share a symmetric *pre-shared key* (PSK), which is modelled in the S-KEM abstraction by the secret key *sk*. As part of the handshake, the client, modelled by the encapsulator, draws a random *nonce*. To derive a shared fresh session key, the client sends the nonce to the server, and both parties apply the HKDF [39, 40] key derivation function (KDF) to the PSK and client nonce, together with some auxiliary data. See Figure 5.2 for an illustration. A description of how HKDF is used in TLS 1.3 can be found in [41]. Here, HKDF essentially works as a pseudorandom function [25], allowing us to capture the TLS handshake just described in an S-KEM scheme built from a PRF. We give the construction next.

Let $\mathsf{F} : \{0,1\}^k \times \{0,1\}^{\text{in}} \to \{0,1\}^{\text{out}}$ be a pseudorandom function. We build the symmetric-key KEM $\mathsf{S\text{-}KEM[F]} = (\mathsf{S\text{-}KEM[F].KG}, \mathsf{S\text{-}KEM[F].E}, \mathsf{S\text{-}KEM[F].D})$ as shown in Figure 5.3. Here the secret key *sk* represents the PSK in the TLS handshake. The random nonce drawn by the client is the ciphertext C picked at random in encapsulation, and the key derivation function HKDF is modelled by the PRF F.

The privacy of S-KEM[F] follows from the PRF security of F. Intuitively, a good pseudorandom function produces output indistinguishable from random strings. Hence the session keys of S-KEM[F] are infeasible to tell apart from random strings if F is secure. This gives the construction ind-cca security (which means that by implication it is also ind-cpa secure). We formalize the result as theorems,

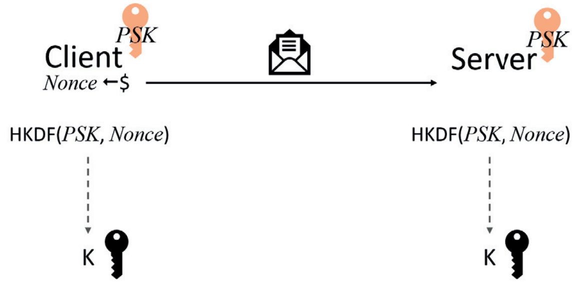

Figure 5.2: Simplified illustration of 0-RTT key derivation in TLS 1.3 handshake. Client and server initially share symmetric key PSK. The client picks a random nonce and sends it to the server across the network. Both client and server compute the session key K via $K \leftarrow \mathsf{HKDF}(PSK, Nonce)$.

```
S-KEM[F].KG():
    $sk \leftarrow\!\!\$\ \{0,1\}^k$
    Return $sk$

S-KEM[F].E($sk$):
    $C \leftarrow\!\!\$\ \{0,1\}^{\mathrm{in}}$
    $K \leftarrow \mathsf{F}(sk, C)$
    Return $(K, C)$

S-KEM[F].D($sk, C$):
    $K \leftarrow \mathsf{F}(sk, C)$
    Return $K$
```

Figure 5.3: Algorithms of S-KEM[F], an instantiation of a symmetric-key KEM using pseudorandom function F.

first for ind-cpa security to establish intuition and then for the stronger ind-cca security.

Theorem 1. S-KEM[F] *is* ind-cpa *secure assuming* F *is a secure PRF. More explicitly, for any adversary $\mathcal{A}$ in the* ind-cpa *game making q encapsulation queries, there exists an adversary $\mathcal{B}$ against the PRF security of* F *making at most q queries to oracle* FN *such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{B}) + 0.5 \cdot \frac{q(q-1)}{2^{\mathrm{in}}}.$$

Proof. Let $\mathcal{A}$ be an adversary in the ind-cpa game. We construct an adversary $\mathcal{B}$ against the PRF-security of F to show the advantage bound. Adversary $\mathcal{B}$ simulates the ind-cpa game for $\mathcal{A}$ using its FN-oracle and then forwards the output of $\mathcal{A}$ as its own, as follows.

When $\mathcal{A}$, being run by $\mathcal{B}$, makes a query to oracle ENC, $\mathcal{B}$ picks $C \leftarrow\!\!\$\ \{0,1\}^{\mathrm{in}}$. If C is new (distinct from the strings drawn in all previous queries) $\mathcal{B}$ queries its own FN oracle on C and returns $(\mathrm{FN}(C), C)$. Otherwise $\mathcal{B}$ aborts and returns $\perp$.

As long as $\mathcal{B}$ draws a new C, this perfectly simulates the ind-cpa game for $\mathcal{A}$. If $\mathcal{B}$ is in Game $\mathbf{G}^{\mathrm{prf}}_1$, then $\mathrm{FN}(C)$ mimics an honest encapsulation in S-KEM[F], simulating ENC_1. If $\mathcal{B}$ is in Game $\mathbf{G}^{\mathrm{prf}}_0$, the simulation likewise follows the expected behavior of ENC_0, returning a randomly chosen element $K \in \{0,1\}^{\mathrm{out}}$. When $\mathcal{A}$

halts and outputs a bit b^*, $\mathcal{B}$ uses this bit as its own guess in the PRF game. It wins if $\mathcal{A}$ has correctly guessed whether it is in the real or random world, hence $\mathbf{Adv}^{\text{ind-cpa}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) - \Pr[\,\mathcal{B} \text{ aborts}\,] = \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B})$.

The likelihood that $\mathcal{B}$ aborts at any point during the simulation is precisely the collision probability $C(2^{\text{in}}, q)$, where $2^{\text{in}} = |\{0,1\}^{\text{in}}|$. Using the birthday bound (from Equation (4.1)), we have $C(2^{\text{in}}, q) \leq 0.5 \cdot \frac{q(q-1)}{2^{\text{in}}}$ which shows the theorem statement. □

Next we look at security under chosen-ciphertext attacks. Recall that in the ind-cca game $\mathbf{G}^{\text{cca}}_{b,\mathsf{S\text{-}KEM}}$, the decapsulation oracle only decapsulates ciphertexts that have not previously been given to the adversary by the encapsulation oracle (Figure 5.1, line 8). We will call queries on such "new" ciphertexts *valid.*

Theorem 2. $\mathsf{S\text{-}KEM[F]}$ *is* ind-cca *secure assuming* F *is a secure PRF. More explicitly, for any adversary* $\mathcal{A}$ *in the* ind-cca *game making* q_e *encapsulation queries and* q_d *distinct and valid decapsulation queries, there exists an adversary* $\mathcal{B}$ *against the PRF security of* F *making at most* $q_e + q_d$ *queries to oracle* $\textsc{Fn}$ *such that*

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{q_e(q_e-1)}{2^{\text{in}} - q_d} + 2\left(1 - \left(\frac{2^{\text{in}} - q_d}{2^{\text{in}}}\right)^{q_e}\right).$$

Proof. Let $\mathcal{A}$ be an adversary in the ind-cca game. We construct an adversary $\mathcal{B}$ against the PRF-security of F and show the advantage bound through a sequence of game hops via games $\mathbf{G}_0$, $\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$, $\mathbf{G}_4$ and $\mathbf{G}_5$ shown in Figure 5.4.

Game $\mathbf{G}_0$ runs $\mathcal{A}_{\text{cca}}$ as per $\mathbf{G}^{\text{cca}}_{b,\mathsf{S\text{-}KEM[F]}}$, with the difference that the bit b is no longer a parameter of the game, but instead picked randomly at the start of the game. Observe that

$$\begin{aligned}\mathbf{Adv}^{\text{cca}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) &= \Pr\left[\mathbf{G}^{\text{cca}}_{1,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{G}^{\text{cca}}_{0,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\right] \\ &= 2 \cdot \Pr[\,\mathbf{G}_0(\mathcal{A}) \Rightarrow \mathsf{true}\,] - 1.\end{aligned}$$

This is clear if we let b^* denote the output of $\mathcal{A}$, which allows us to write the probability that $\mathcal{A}$ returns 1 in the real game as the probability that $b^* = 1$ given that $b = 1$, i.e. $\Pr\left[\mathbf{G}^{\text{cca}}_{1,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\right] = \Pr\left[\,b^* = 1 \,\middle|\, b = 1\,\right]$. Analogously in the random game we have $\Pr\left[\mathbf{G}^{\text{cca}}_{0,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\right] = \Pr\left[\,b^* = 1 \,\middle|\, b = 0\,\right]$. We use

```
Game $\mathbf{G}_0(\mathcal{A})$, $\boxed{\mathbf{G}_1(\mathcal{A})}$
  $sk \leftarrow\$ \{0,1\}^k$
  $b \leftarrow\$ \{0,1\}$
  $\mathcal{S}_e \leftarrow \emptyset$; $\mathcal{S}_d \leftarrow \emptyset$
  $b^* \leftarrow\$ \mathcal{A}^{\textsc{Enc},\textsc{Dec}}()$
  Return $(b^* = b)$

$\textsc{Enc}()$:
  $C \leftarrow\$ \{0,1\}^{\mathrm{in}}$
  If $C \in \mathcal{S}_d$ then:
      bad $\leftarrow$ true; $\boxed{\text{Ret. } \bot}$
  $\mathcal{S}_e \leftarrow \mathcal{S}_e \cup \{C\}$
  $K_1 \leftarrow \mathsf{F}(sk, C)$
  $K_0 \leftarrow\$ \{0,1\}^{\mathrm{out}}$
  Return $(K_b, C)$

$\textsc{Dec}(C)$:
  If $C \in \mathcal{S}_e$ then return $\bot$
  $\mathcal{S}_d \leftarrow \mathcal{S}_d \cup \{C\}$
  $K \leftarrow \mathsf{F}(sk, C)$
  Return $K$
```

```
Game $\mathbf{G}_2(\mathcal{A})$, $\boxed{\mathbf{G}_3(\mathcal{A})}$
  $sk \leftarrow\$ \{0,1\}^k$
  $b \leftarrow\$ \{0,1\}$
  $\mathcal{S}_e \leftarrow \emptyset$; $\mathcal{S}_d \leftarrow \emptyset$
  $b^* \leftarrow\$ \mathcal{A}^{\textsc{Enc},\textsc{Dec}}()$
  Return $(b^* = b)$

$\textsc{Enc}()$:
  $C \leftarrow\$ \{0,1\}^{\mathrm{in}}$
  If $C \in \mathcal{S}_d$ then return $\bot$
  If $C \in \mathcal{S}_e$ then:
      bad $\leftarrow$ true; $\boxed{\text{Ret. } \bot}$
  $\mathcal{S}_e \leftarrow \mathcal{S}_e \cup \{C\}$
  $K_1 \leftarrow \mathsf{F}(sk, C)$
  $K_0 \leftarrow\$ \{0,1\}^{\mathrm{out}}$
  Return $(K_b, C)$

$\textsc{Dec}(C)$:
  If $C \in \mathcal{S}_e$ then return $\bot$
  $\mathcal{S}_d \leftarrow \mathcal{S}_d \cup \{C\}$
  $K \leftarrow \mathsf{F}(sk, C)$
  Return $K$
```

```
Game $\mathbf{G}_4(\mathcal{A})$, $\boxed{\mathbf{G}_5(\mathcal{A})}$
  $sk \leftarrow\$ \{0,1\}^k$
  $b \leftarrow\$ \{0,1\}$
  $\mathcal{S}_e \leftarrow \emptyset$; $\mathcal{S}_d \leftarrow \emptyset$
  $b^* \leftarrow\$ \mathcal{A}^{\textsc{Enc},\textsc{Dec}}()$
  Return $(b^* = b)$

$\textsc{Enc}()$:
  $C \leftarrow\$ \{0,1\}^{\mathrm{in}}$
  If $C \in \mathcal{S}_d$ then return $\bot$
  If $C \in \mathcal{S}_e$ then return $\bot$
  $\mathcal{S}_e \leftarrow \mathcal{S}_e \cup \{C\}$
  $K_1 \leftarrow \mathsf{F}(sk, C)$
  $\boxed{K_1 \leftarrow\$ \{0,1\}^{\mathrm{out}}}$
  $K_0 \leftarrow\$ \{0,1\}^{\mathrm{out}}$
  Return $(K_b, C)$

$\textsc{Dec}(C)$:
  If $C \in \mathcal{S}_e$ then return $\bot$
  $\mathcal{S}_d \leftarrow \mathcal{S}_d \cup \{C\}$
  $K \leftarrow \mathsf{F}(sk, C)$
  $\boxed{K \leftarrow\$ \{0,1\}^{\mathrm{out}}}$
  Return $K$
```

Figure 5.4: Games $\mathbf{G}_0$, $\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$, $\mathbf{G}_4$ and $\mathbf{G}_5$ for the proof of Theorem 2. The code in boxes is executed in $\mathbf{G}_1$, $\mathbf{G}_3$ and $\mathbf{G}_5$, but not $\mathbf{G}_0$, $\mathbf{G}_2$ or $\mathbf{G}_4$. Ret. stands for Return.

that for b chosen uniformly at random in $\mathbf{G}_0$, $\Pr[\,b=1\,] = \Pr[\,b=0\,] = 1/2$, giving

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{cca}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) &= \Pr\left[\,\mathbf{G}^{\mathrm{cca}}_{1,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\mathrm{cca}}_{0,\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) \Rightarrow 1\,\right] \\
&= \Pr\left[\,b^* = 1 \,\middle|\, b = 1\,\right] - \Pr\left[\,b^* = 1 \,\middle|\, b = 0\,\right] \\
&= \Pr\left[\,b^* = 1 \,\middle|\, b = 1\,\right] - \left(1 - \Pr\left[\,b^* = 0 \,\middle|\, b = 0\,\right]\right) \\
&= 2\Big(\Pr\left[\,b^* = b \,\middle|\, b = 1\,\right] \cdot \frac{1}{2} + \Pr\left[\,b^* = b \,\middle|\, b = 0\,\right] \cdot \frac{1}{2}\Big) - 1 \\
&= 2\Big(\Pr\left[\,b^* = b \,\middle|\, b = 1\,\right] \cdot \Pr[\,b = 1\,] \\
&\qquad\qquad + \Pr\left[\,b^* = b \,\middle|\, b = 0\,\right] \cdot \Pr[\,b = 0\,]\Big) - 1 \\
&= 2\Big(\Pr[\,b^* = b\,]\Big) - 1 \\
&= 2 \cdot \Pr[\,\mathbf{G}_0(\mathcal{A}) \Rightarrow \mathsf{true}\,] - 1.
\end{aligned}$$

Next, we apply the fundamental lemma of game playing [8] to bound the difference between $\Pr[\,\mathbf{G}_0(\mathcal{A})\,] - \Pr[\,\mathbf{G}_1(\mathcal{A})\,]$. The lemma lets us give an upper limit on the difference between the probability that $\mathcal{A}$ wins game $\mathbf{G}_0$ and game $\mathbf{G}_1$, because the games are *identical until* bad. This means that the code of the games is identical until the bad flag is set to true, hence the outcome will always be the same when $\mathcal{A}$ is executed in game $\mathbf{G}_0$ and $\mathbf{G}_1$, unless the bad flag is triggered.

$$\begin{aligned}
\Pr[\,\mathbf{G}_0(\mathcal{A})\,] &= \Pr[\,\mathbf{G}_1(\mathcal{A})\,] + \left(\Pr[\,\mathbf{G}_0(\mathcal{A})\,] - \Pr[\,\mathbf{G}_1(\mathcal{A})\,]\right) \\
&\le \Pr[\,\mathbf{G}_1(\mathcal{A})\,] + \Pr[\,\mathbf{G}_0(\mathcal{A}) \text{ sets } \mathsf{bad}\,]. \qquad (5.1)
\end{aligned}$$

The probability that $\mathbf{G}_0$ sets the bad flag to true is the likelihood that out of the q_e Enc-queries that $\mathcal{A}$ makes, at least one ciphertext drawn by the Enc-oracle is the same as a previous Dec-query that $\mathcal{A}$ has made. This can be bounded by the probability that at least one of the ciphertexts drawn by the Enc-oracle is the same as the ciphertext in any of the total q_d Dec-queries that $\mathcal{A}$ makes (previous or future). The complement event is that all q_e ciphertexts produced by the Enc-oracle are distinct from the q_d Dec-queries that $\mathcal{A}$ makes. Hence

$$\Pr[\mathbf{G}_0(\mathcal{A}) \text{ sets } \mathsf{bad}] = 1 - \left(\frac{2^{\mathrm{in}} - q_d}{2^{\mathrm{in}}}\right)^{q_e}.$$

We continue to the next game hop, noting that $\mathbf{G}_1$ and $\mathbf{G}_2$ are equivalent. Applying the game playing lemma again, we obtain

$$\Pr[\,\mathbf{G}_2(\mathcal{A})\,] \le \Pr[\,\mathbf{G}_3(\mathcal{A})\,] + \Pr[\,\mathbf{G}_2(\mathcal{A}) \text{ sets } \mathsf{bad}\,].$$

Here, the probability that $\mathbf{G}_2$ triggers the bad flag is the collision probability of the ciphertexts drawn at random by the Enc-oracle. This is bounded by the birthday bound,

$$\Pr[\mathbf{G}_2(\mathcal{A}) \text{ sets } \mathsf{bad}] \le 0.5 \cdot \frac{q_e(q_e - 1)}{2^{\mathrm{in}} - q_d},$$

where the size of the set the q_e ciphertexts are drawn from is $2^{\mathrm{in}} - q_d$, since the q_d ciphertexts queried to Dec by $\mathcal{A}$ are excluded. Hence, what we have currently

```
Adversary $\mathcal{B}^{\text{Fn}}()$:
    $b \leftarrow\$ \{0,1\}$
    $\mathcal{S}_e \leftarrow \emptyset$; $\mathcal{S}_d \leftarrow \emptyset$
    $b^* \leftarrow\$ \mathcal{A}^{\text{Enc}^*,\text{Dec}^*}()$
    If $b^* = b$ return 1
    Else return 0

Enc$^*()$:
    $C \leftarrow\$ \{0,1\}^{\text{in}}$
    If $C \in \mathcal{S}_d$ then return $\bot$
    If $C \in \mathcal{S}_e$ then return $\bot$
    $\mathcal{S}_e \leftarrow \mathcal{S}_e \cup \{C\}$
    $K_1 \leftarrow \text{Fn}(C)$
    $K_0 \leftarrow\$ \{0,1\}^{\text{out}}$
    Return $(K_b, C)$

Dec$^*(C)$:
    If $C \in \mathcal{S}_e$ then return $\bot$
    $\mathcal{S}_d \leftarrow \mathcal{S}_d \cup \{C\}$
    $K \leftarrow \text{Fn}(C)$
    Return $K$
```

Figure 5.5: Adversary $\mathcal{B}$ for the proof of Theorem 2. A star superscript to a procedure indicates that it is a subroutine in the code of adversary $\mathcal{B}$, constructed by it to simulate an oracle expected by $\mathcal{A}$.

shown is that

$$\begin{aligned}
\mathbf{Adv}^{\text{cca}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) &= 2 \cdot \Pr[\,\mathbf{G}_0(\mathcal{A}) \Rightarrow \mathsf{true}\,] - 1 \\
&\leq 2\big(\Pr[\,\mathbf{G}_1(\mathcal{A})\,] + \Pr[\,\mathbf{G}_0(\mathcal{A}) \text{ sets } \mathsf{bad}\,]\big) - 1 \\
&\leq 2\left(\Pr[\,\mathbf{G}_2(\mathcal{A})\,] + 1 - \left(\frac{2^{\text{in}} - q_d}{2^{\text{in}}}\right)^{q_e}\right) - 1 \\
&\leq 2\left(\Pr[\,\mathbf{G}_3(\mathcal{A})\,] + \Pr[\,\mathbf{G}_2(\mathcal{A}) \text{ sets } \mathsf{bad}\,] + 1 - \left(\frac{2^{\text{in}} - q_d}{2^{\text{in}}}\right)^{q_e}\right) - 1 \\
&\leq 2 \cdot \Pr[\,\mathbf{G}_3(\mathcal{A})\,] + 2 \cdot 0.5 \cdot \frac{q_e(q_e-1)}{2^{\text{in}} - q_d} + 2\left(1 - \left(\frac{2^{\text{in}} - q_d}{2^{\text{in}}}\right)^{q_e}\right) - 1 \\
&= 2 \cdot \Pr[\,\mathbf{G}_3(\mathcal{A})\,] - 1 + 2\left(0.5 \cdot \frac{q_e(q_e-1)}{2^{\text{in}} - q_d} + 1 - \left(\frac{2^{\text{in}} - q_d}{2^{\text{in}}}\right)^{q_e}\right).
\end{aligned}$$

Now, game $\mathbf{G}_4$ is equivalent to $\mathbf{G}_3$, so $\Pr[\,\mathbf{G}_3(\mathcal{A})\,] = \Pr[\,\mathbf{G}_4(\mathcal{A})\,]$. Game $\mathbf{G}_5$ is similar to $\mathbf{G}_4$, except that it always gives random strings from $\{0,1\}^{\text{out}}$ as output on both Enc- and Dec-queries, regardless of the value of bit b. Intuitively, if F is a secure PRF, the difference is indistinguishable. Formally, this is captured by a reduction to the PRF security of F via PRF adversary $\mathcal{B}$, the code of which is shown in Figure 5.5. Adversary $\mathcal{B}$ begins by drawing a random bit and assigning it to b. If b is 1, then $\mathcal{B}$ uses its own PRF oracle Fn to answer Enc-queries from adversary $\mathcal{A}$. If $b = 0$ it instead picks keys uniformly at random. Decapsulation queries from $\mathcal{A}$ are relayed by $\mathcal{B}$ to its own Fn oracle. When adversary $\mathcal{A}$ halts and output a bit b^*, adversary $\mathcal{B}$ checks if $b^* = b$, and if so returns 1 in the PRF game. Else it returns 0.

This setup means that if adversary $\mathcal{B}$ is playing the real PRF game (so that $\text{Fn}(C) = \mathsf{F}(sk, C)$), then $\mathcal{B}$ simulates game $\mathbf{G}_4$ for $\mathcal{A}$. If on the other hand $\mathcal{B}$ is in the random PRF game, and Fn returns values from a table with lazily sampled random entries, then $\mathcal{B}$ simulates game $\mathbf{G}_5$ for adversary $\mathcal{A}$. To see this, we note that computing key K as $K \leftarrow\$ \text{Fn}(C)$ when Fn is a lazily sampled random table with values in $\{0,1\}^{\mathrm{out}}$ is equivalent to drawing K uniformly at random from $\{0,1\}^{\mathrm{out}}$, as long as the same ciphertext C is never repeated. I.e., the only difference between $\textsc{Fn}(C)$ and a random string is that $\textsc{Fn}$ will return the same string each time it is evaluated on the same input C. But with the previous game hops—and the assumption (w.l.o.g.) of distinct decapsulation queries—we assured precisely that no C needs to be handled in more than one query from $\mathcal{A}$ in either game $\mathbf{G}_4$ or $\mathbf{G}_5$ (see lines 7, 8 and 14 in Figure 5.4). Since the simulations are sound, we can directly relate the PRF advantage of $\mathcal{B}$ to the probability that adversary $\mathcal{A}$ wins games $\mathbf{G}_4$ and $\mathbf{G}_5$.

$$\Pr[\,\mathbf{G}_4(\mathcal{A})\,] = \Pr[\,b^* = b \text{ in game } \mathbf{G}_4\,] = \Pr\left[\,\mathbf{G}^{\mathrm{prf}}_{1,\mathsf{F}}(\mathcal{B}) \Rightarrow 1\,\right]$$

$$\Pr[\,\mathbf{G}_5(\mathcal{A})\,] = \Pr[\,b^* = b \text{ in game } \mathbf{G}_5\,] = \Pr\left[\,\mathbf{G}^{\mathrm{prf}}_{0,\mathsf{F}}(\mathcal{B}) \Rightarrow 1\,\right]$$

Hence

$$\Pr[\,\mathbf{G}_4(\mathcal{A})\,] - \Pr[\,\mathbf{G}_5(\mathcal{A})\,] = \Pr\left[\,\mathbf{G}^{\mathrm{prf}}_{1,\mathsf{F}}(\mathcal{B}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\mathrm{prf}}_{0,\mathsf{F}}(\mathcal{B}) \Rightarrow 1\,\right] = \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{B}),$$

so we end up with

$$\begin{aligned}\Pr[\,\mathbf{G}_4(\mathcal{A})\,] &= (\Pr[\,\mathbf{G}_4(\mathcal{A})\,] - \Pr[\,\mathbf{G}_5(\mathcal{A})\,]) + \Pr[\,\mathbf{G}_5(\mathcal{A})\,] \\ &\leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{1}{2},\end{aligned}$$

where $\Pr[\,\mathbf{G}_5(\mathcal{A})\,] \leq \frac{1}{2}$. This is motivated by the fact that in game $\mathbf{G}_5$, adversary $\mathcal{A}$ gets random strings instead of keys in both encapsulation and decapsulation queries regardless of the value of bit b, so the best it can do is guess the value of b, which succeeds with probability $\frac{1}{2}$. Using that $\Pr[\,\mathbf{G}_3(\mathcal{A})\,] = \Pr[\,\mathbf{G}_4(\mathcal{A})\,]$ and putting this together with the previously showed bound, we have

$$\begin{aligned}\mathbf{Adv}^{\mathrm{cca}}_{\mathsf{S\text{-}KEM[F]}}(\mathcal{A}) &\leq 2 \cdot \Pr[\,\mathbf{G}_4(\mathcal{A})\,] - 1 + 2\left(0.5 \cdot \frac{q_e(q_e-1)}{2^{\mathrm{in}} - q_d} + 1 - \left(\frac{2^{\mathrm{in}} - q_d}{2^{\mathrm{in}}}\right)^{q_e}\right) \\ &\leq 2 \cdot \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{B}) + 2\left(0.5 \cdot \frac{q_e(q_e-1)}{2^{\mathrm{in}} - q_d} + 1 - \left(\frac{2^{\mathrm{in}} - q_d}{2^{\mathrm{in}}}\right)^{q_e}\right).\end{aligned}$$

This is precisely the bound in the theorem statement. □

5.4 Integrity of Ciphertexts

To exemplify how S-KEMs could be protected against ciphertext forgeries and impersonation attacks, we formalize the standard notion of *integrity of ciphertexts* (int-ctxt) by the game in Figure 5.6. Integrity of ciphertext security aims to provide a measure of certainty in the authenticity of encapsulated keys. What this means is that no-one without access to the secret key can construct (forge) a new, valid ciphertext, even when seeing honest prior encapsulations. In the game, an adversary is given access to both real encapsulations and decapsulations via oracles $\textsc{Enc}$ and $\textsc{Dec}$, and wins if it can produce new ciphertexts which decapsulate to

```
Game $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{S\text{-}KEM}}(\mathcal{A})$:
  $sk \leftarrow\!\!\$\ \mathsf{KG}()$
  $S \leftarrow \emptyset$; $\mathsf{win} \leftarrow \mathsf{false}$
  $\mathcal{A}^{\text{Enc},\text{Dec}}()$
  Return $\mathsf{win}$

Enc():
  $(K, C) \leftarrow\!\!\$\ \mathsf{E}(sk)$
  $S \leftarrow S \cup \{C\}$
  Return $(K, C)$

Dec($C$):
  $K \leftarrow \mathsf{D}(sk, C)$
  If $C \notin S$ and $K \neq \bot$:
      $\mathsf{win} \leftarrow \mathsf{true}$
  Return $K$
```

Figure 5.6: Game formalizing integrity of ciphertexts, int-ctxt, for symmetric-key KEM scheme S-KEM.

something other than $\bot$. We define the advantage of adversary $\mathcal{A}$ in the int-ctxt game to $\mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{S\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{int-ctxt}}_{\mathsf{S\text{-}KEM}}(\mathcal{A}) \Rightarrow \mathsf{true}\,\right]$, the probability that the game returns true.

Integrity of ciphertexts can also help "lift" the weaker ind-cpa privacy notion to the stronger ind-cca security. That is, if a symmetric-key KEM scheme is secure against chosen-plaintext attacks and has integrity of ciphertexts, then it is ind-cca secure. The intuition behind this is that in a scheme which is secure against forgeries, a decapsulation oracle will not help because the adversary cannot construct valid ciphertexts to query to it. This is the result of our third theorem, which is reminiscent of a generic composition for encryption given by Bellare and Namprempre [7].

Theorem 3. *Let* $\mathsf{S\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$ *be a symmetric-key KEM scheme. If* S-KEM *is* ind-cpa *and* int-ctxt *secure, then it is* ind-cca *secure. Formally, let* $\mathcal{A}_{\text{cca}}$ *be an* ind-cca *adversary against* S-KEM*. Then there exists an* ind-cpa *adversary* $\mathcal{B}_{\text{cpa}}$ *and an* int-ctxt *adversary* $\mathcal{C}_{\text{ctxt}}$ *such that*

$$\mathbf{Adv}^{\text{cca}}_{\mathsf{S\text{-}KEM}}(\mathcal{A}_{\text{cca}}) \leq \mathbf{Adv}^{\text{cpa}}_{\mathsf{S\text{-}KEM}}(\mathcal{B}_{\text{cpa}}) + 2 \cdot \mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{S\text{-}KEM}}(\mathcal{C}_{\text{ctxt}}).$$

Proof. To show the statement we provide a sequence of games, Game $\mathbf{G}_0$, $\mathbf{G}_1$ and $\mathbf{G}_2$, as well as two adversaries, $\mathcal{B}_{\text{cpa}}$ and $\mathcal{C}_{\text{ctxt}}$, the codes of which are shown in Figure 5.7. Game $\mathbf{G}_0$ is equivalent to $\mathbf{G}^{\text{cca}}_{b,\mathsf{S\text{-}KEM}}$, except that the bit b is not a parameter of the game, but rather picked at the start of the game. We use that for b chosen uniformly at random $\Pr[\,b = 1\,] = \Pr[\,b = 0\,] = 1/2$, giving

$$\mathbf{Adv}^{\text{cca}}_{\mathsf{S\text{-}KEM}}(\mathcal{A}_{\text{cca}}) = 2 \cdot \Pr[\,\mathbf{G}_0(\mathcal{A}_{\text{cca}}) \Rightarrow \mathsf{true}\,] - 1. \tag{5.2}$$

The equality follows from a probability argument similar to that given in the proof of Theorem 2. For the games, we have that

$$\begin{aligned}\Pr[\,\mathbf{G}_0(\mathcal{A}_{\text{cca}})\,] &= \Pr[\,\mathbf{G}_1(\mathcal{A}_{\text{cca}})\,] + \big(\Pr[\,\mathbf{G}_0(\mathcal{A}_{\text{cca}})\,] - \Pr[\,\mathbf{G}_1(\mathcal{A}_{\text{cca}})\,]\big)\\ &\leq \Pr[\,\mathbf{G}_1(\mathcal{A}_{\text{cca}})\,] + \Pr[\,\mathbf{G}_0(\mathcal{A}_{\text{cca}}) \text{ sets } \mathsf{win}\,]\,. \end{aligned} \tag{5.3}$$

The last inequality is given by the fundamental lemma of game playing [8] applied to games $\mathbf{G}_0$ and $\mathbf{G}_1$, which are identical until the win flag is set. In game $\mathbf{G}_1$, we have that $K_1 = \bot$ at the time procedure Dec_b returns, so

$$\Pr[\,\mathbf{G}_1(\mathcal{A}_{\text{cca}})\,] = \Pr[\,\mathbf{G}_2(\mathcal{A}_{\text{cca}})\,]\,. \tag{5.4}$$

```
Game $\mathbf{G}_0(\mathcal{A}_{\mathrm{cca}})$, $\boxed{\mathbf{G}_1(\mathcal{A}_{\mathrm{cca}})}$
  $sk \leftarrow\$ \mathsf{KG}()$
  $b \leftarrow\$ \{0,1\}$; $\mathcal{S} \leftarrow \emptyset$
  $b^* \leftarrow\$ \mathcal{A}_{\mathrm{cca}}^{\textsc{Enc}_b,\textsc{Dec}}()$
  Return $(b^* = b)$

$\textsc{Enc}_b()$:
  $(K_1, C) \leftarrow\$ \mathsf{E}(sk)$
  $K_0 \leftarrow\$ \mathcal{K}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
  Return $(K_b, C)$

$\textsc{Dec}(C)$:
  If $C \notin \mathcal{S}$ then $K_1 \leftarrow \mathsf{D}(sk, C)$
  Else $K_1 \leftarrow \bot$
  If $K_1 \neq \bot$ then win $\leftarrow$ true; $\boxed{K_1 \leftarrow \bot}$
  Return $K_1$
```

```
Game $\mathbf{G}_2(\mathcal{A}_{\mathrm{cca}})$
  $sk \leftarrow\$ \mathsf{KG}()$
  $b \leftarrow\$ \{0,1\}$
  $b^* \leftarrow\$ \mathcal{A}_{\mathrm{cca}}^{\textsc{Enc}_b,\textsc{Dec}}()$
  Return $(b^* = b)$

$\textsc{Enc}_b()$:
  $(K_1, C) \leftarrow\$ \mathsf{E}(sk)$
  $K_0 \leftarrow\$ \mathcal{K}$
  Return $(K_b, C)$

$\textsc{Dec}(C)$:
  Return $\bot$
```

```
Adversary $\mathcal{B}_{\mathrm{cpa}}^{\textsc{Enc}}()$:
  $b^* \leftarrow\$ \mathcal{A}_{\mathrm{cca}}^{\mathrm{Enc}^*,\mathrm{Dec}^*}()$
  Return $b^*$

$\mathrm{Enc}^*()$:
  $(K, C) \leftarrow\$ \textsc{Enc}()$
  Return $(K, C)$

$\mathrm{Dec}^*(C)$:
  Return $\bot$
```

```
Adversary $\mathcal{C}_{\mathrm{int\text{-}ctxt}}^{\textsc{Enc},\textsc{Dec}}()$:
  $b \leftarrow\$ \{0,1\}$
  $\mathcal{S} \leftarrow \emptyset$
  $\mathcal{A}_{\mathrm{cca}}^{\mathrm{Enc}_b^*,\mathrm{Dec}^*}()$

$\mathrm{Enc}_b^*()$:
  $(K_1, C) \leftarrow\$ \textsc{Enc}()$
  $K_0 \leftarrow\$ \mathcal{K}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
  Return $(K_b, C)$

$\mathrm{Dec}^*(C)$:
  $K_1 \leftarrow \textsc{Dec}(C)$
  If $C \in S$ return $\bot$
  Return $K_1$
```

Figure 5.7: Top: Games $\mathbf{G}_0$, $\mathbf{G}_1$ and $\mathbf{G}_2$ for the proof of Theorem 3. The code in the box is executed in $\mathbf{G}_1$ but not $\mathbf{G}_0$. **Bottom:** Adversaries for the proof of Theorem 3. A star superscript to a procedure indicates that it is a subroutine in the code of the corresponding adversary, constructed by it to simulate an oracle expected by $\mathcal{A}_{\mathrm{cca}}$.

We claim that for adversaries $\mathcal{B}_{\mathrm{cpa}}$ and $\mathcal{C}_{\mathrm{ctxt}}$ in Figure 5.7 it holds that

$$2 \cdot \Pr\left[\, \mathbf{G}_2(\mathcal{A}_{\mathrm{cca}}) \,\right] - 1 \leq \mathbf{Adv}^{\mathrm{cpa}}_{\textsf{S-KEM}}(\mathcal{B}_{\mathrm{cpa}}) \tag{5.5}$$

$$\Pr\left[\, \mathbf{G}_0(\mathcal{A}_{\mathrm{cca}}) \text{ sets } \textsf{win} \,\right] \leq \mathbf{Adv}^{\text{int-ctxt}}_{\textsf{S-KEM}}(\mathcal{C}_{\mathrm{ctxt}}). \tag{5.6}$$

Combining Equations (5.2), (5.3), (5.4), (5.5) and (5.6) gives the theorem statement. It remains to prove the claims. We begin with Equation (5.5).

Adversary $\mathcal{B}_{\mathrm{cpa}}$ playing game $\mathbf{G}^{\mathrm{cpa}}_{b,\textsf{S-KEM}}$ runs $\mathcal{A}_{\mathrm{cca}}$ as per game $\mathbf{G}_2$, relaying encapsulation queries to its own ENC-oracle via the subroutine Enc^* and responding to all decapsulation queries with $\perp$. When adversary $\mathcal{A}_{\mathrm{cca}}$ halts and returns bit b^*, adversary $\mathcal{B}_{\mathrm{cpa}}$ uses b^* as its own output. When $\mathcal{B}_{\mathrm{cpa}}$ is in the real world, the simulation perfectly captures game $\mathbf{G}_2$ with $b = 1$. Likewise when $\mathcal{B}_{\mathrm{cpa}}$ is in the random world, it perfectly simulates $\mathbf{G}_2$ with $b = 0$ for $\mathcal{A}_{\mathrm{cca}}$. Because $\Pr\left[\, b = 1 \,\right] = \Pr\left[\, b = 0 \,\right] = 1/2$ in game $\mathbf{G}_2$, this shows that $\mathbf{Adv}^{\mathrm{cpa}}_{\textsf{S-KEM}}(\mathcal{B}_{\mathrm{cpa}}) = 2 \cdot \Pr\left[\, \mathbf{G}_2(\mathcal{A}_{\mathrm{cca}}) \,\right] - 1$. We move on to Equation (5.6).

Adversary $\mathcal{C}_{\mathrm{ctxt}}$ simulates game $\mathbf{G}_0$ for adversary $\mathcal{A}_{\mathrm{cca}}$. Each time $\mathcal{A}_{\mathrm{cca}}$ submits a query $\mathrm{Dec}^*(C)$, adversary $\mathcal{C}_{\mathrm{ctxt}}$ forwards the ciphertext to its DEC-oracle. It wins the int-ctxt game precisely if flag win is set to true in game $\mathbf{G}_0$, justifying claim (5.6). □

We do not give any S-KEM instantiation which fulfills int-ctxt, but will touch more upon protection against forgeries in the constructions of ordered S-KEM schemes in Chapter 6. However, the focus in the rest of this work is on privacy, and integrity is only discussed when it appears naturally as a means of achieving security against chosen-ciphertext attacks.

Chapter 6

Ordered Symmetric-key KEMs

The symmetric-key KEMs introduced in the previous chapter allow us to capture a PSK 0-RTT TLS key exchange, which can be used in the handshake of a resumed session. But they tell us little about the relation between separate sessions. Recall that in TLS, client and server can have multiple sessions running concurrently, and may also use the same PSK to establish several subsequent resumed sessions. The connection requests for these sessions are sent across a network, and if the network is somewhat reliable, they should arrive in approximately the same order that they were sent in. If this is the case, the ordering can potentially be used to give more efficient constructions, also with respect to forward secrecy. But in order to leverage ordering, the syntax of S-KEMs must be extended to include state. This is the purpose of ordered symmetric-key KEMs (OS-KEMs), which we introduce next.

Stateful schemes have the benefit of being more flexible than stateless ones, in the sense that they naturally allow more diverse functionality. To take advantage of this, we define several versions of correctness for OS-KEMs. We refer to the versions as "levels", since they are connected to the level of transmission reliability expected of the network. The correctness demands specified by the levels range from a strict requirement equivalent to that of S-KEMs, to a less stringent condition requiring successful decapsulation only when the ciphertexts sent between encapsulator and decapsulator are relayed in perfect order.

6.1 Syntax

Definition 5. An *ordered symmetric-key key encapsulation mechanism* scheme $\mathsf{OS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$ is a triple of algorithms, the first two of which may be randomized. Algorithms E and D are stateful and use variables st_e (the encapsulation state) and st_d (the decapsulation state), respectively, to keep state. Associated to the scheme is secret key space $\mathcal{SK}$, key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$ as well as two non-empty sets of states, $\mathcal{S}_e$ and $\mathcal{S}_d$, which we call the encapsulation state space and decapsulation state space, respectively.

- Initially via $(sk, st_e, st_d) \leftarrow\!\!\$\ \mathsf{KG}()$ the scheme produces $sk \in \mathcal{SK}$, $st_e \in \mathcal{S}_e$ and $st_d \in \mathcal{S}_d$.
- Via $(K, C, \hat{st_e}) \leftarrow\!\!\$\ \mathsf{E}(sk, st_e)$, algorithm E, taking as input sk and st_e, updates the state and returns a key $K \in \mathcal{K}$ and a corresponding ciphertext $C \in \mathcal{C}$.

- Via $(K, \hat{st_d}) \leftarrow \mathsf{D}(sk, st_d, C)$, algorithm D on input sk, a ciphertext $C \in \mathcal{C}$ and decapsulation state st_d returns the updated state $\hat{st_d}$ and $K \in \mathcal{K} \cup \{\perp\}$. If $K \neq \perp$ we say that D *accepts* C.

In the following, the term "stateful symmetric KEM" will be used interchangeably with ordered symmetric-key KEM when referring to OS-KEMs.

6.2 Correctness

Correctness for stateful schemes can be a messy affair. If we imagine an application in which some party, Alice, performs a sequence of n encapsulations and labels the resulting key-ciphertext pairs with the position in the order she produced them, i.e. $1, 2, \dots, n$, and then sends the ciphertexts in this order to a receiver, Bob, then it is possible that Bob receives them in some altogether different order. For example, he might obtain ciphertexts $1, 3, 6, \dots$ if some are dropped, or he might receive $2, 3, 1, \dots$ if there has been a reordering in transmission. Depending on the requirements from the application one can imagine a full spectrum of correctness notions, ranging from those demanding correct decapsulation of all possible ciphertext sequences (including permutations, omissions and repetitions), to those that only require it for sequences that have not been reordered at all. As noted for example by Rogaway and Zhang in their innovative work on *indistinguishability up to correctness* [44], the constraints posed by varying levels, types or classes of correctness have led to complicated definitions that are difficult to understand and even harder to verify. In their own treatment of stateful authenticated encryption (originally introduced by [5]), which in its correctness demands bears similarities to stateful S-KEMs, they introduce what they call "level sets" which specify the permissible (re)ordering of ciphertexts on the recipient side. They associate to each such set of orderings a correctness class, and use the classes to specify security in the ind-cca sense.

Although security follows nicely from correctness when using indistinguishability up to correctness, and hence saves some work and potential mistakes in the definitions, correctness itself is still a complicated object. In an admirable quest to cover all cases and formulate unambiguous classes, Rogaway and Zhang [44] develop very technical correctness requirements that nonetheless require a lot from the reader to decipher. By doing so, they in some sense shift the burden that used to fall on security over to correctness. In an attempt to reap the benefits of the idea behind indistinguishability up to correctness, while still keeping definitions somewhat intuitive, we choose to instead use a game-playing framework to define correctness requirements for OS-KEMs. This approach has been taken previously by, among others, Jaeger and Stepanovs [35] in the setting of forward-secure messaging, although not as a means of clarifying exceptions in the security games. We will see later that this method benefits us also in the definitions of puncturable S-KEMs (PS-KEMs) and, our final goal, puncturable ordered S-KEMs (POS-KEMs).

Correctness of scheme $\mathsf{X\text{-}KEM}$ ($\mathsf{X} \in \{\mathsf{OS}, \mathsf{PS}, \mathsf{POS}\}$) will hence on be determined by the *correctness advantage* of $\mathsf{X\text{-}KEM}$ in game $\mathbf{G}^{\text{correct}}_{\mathsf{X\text{-}KEM}}$ associated to the scheme. For an adversary $\mathcal{A}$ playing game $\mathbf{G}^{\text{correct}}_{\mathsf{X\text{-}KEM}}$ we define the advantage as

$$\mathbf{Adv}^{\text{correct}}_{\mathsf{X\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{correct}}_{\mathsf{X\text{-}KEM}}(\mathcal{A}) \Rightarrow \mathsf{true}\,\right].$$

We say that scheme $\mathsf{X\text{-}KEM}$ is *correct* if $\mathbf{Adv}^{\text{correct}}_{\mathsf{X\text{-}KEM}}(\mathcal{A}) = 0$ for all, even unbounded, adversaries $\mathcal{A}$. Specifically for OS-KEMs, for $\text{xO} \in \{\text{nO}, \text{wO}, \text{pO}\}$ we say that scheme $\mathsf{OS\text{-}KEM}$ is "xO-correct" if $\mathbf{Adv}^{\text{xO-correct}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A}) = 0$ for all adversaries $\mathcal{A}$.[1]

For an example of a correctness game, see e.g. the left hand column of Figure 6.2. Returning to the rationale behind the introduction of ordered S-KEMs, we see that our treatment of stateful schemes demands varying strictness in the correctness requirement depending on the expectations of the receiver/decapsulator. Therefore there will not only be a specific correctness game for each type of scheme, but also for each level of correctness support. For OS-KEMs we introduce three such classes of correctness, each allowing a certain degree of reordering, omissions and repetitions between encapsulation and decapsulation. The classes are: correctness under *no ordering* ($\mathbf{G}^{\text{nO-correct}}_{\mathsf{OS\text{-}KEM}}$ in Figure 6.2 to the left), correctness under *weak ordering* ($\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM}}$ in Figure 6.3 to the left) and correctness under *perfect ordering* ($\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}$ in Figure 6.4 to the left). Before taking a closer look at the games, we explain briefly what functionality each class aims to capture.

No Ordering. This correctness class places no requirements on the reliability of transmission. Ciphertexts may be reordered, dropped and repeated to decapsulation. For an OS-KEM scheme, correctness under *no ordering* is the strictest requirement as it demands decapsulation to correctly recover any sequence of encapsulated keys. This gives correctness equivalent to that of standard S-KEMs, meaning that a stateless scheme could fulfill correctness with only slight syntax modifications.

Weak Ordering. This class disallows repetitions, but allows local reordering and potentially dropped ciphertexts in the sequence. By local reordering we mean that the ciphertexts may arrive out of order within some bounded interval (sliding window). Correctness under *weak ordering* is parameterized by a *forward window* $w_f \geq 1$ which gives an upper bound on the number of potentially dropped ciphertexts (by bounding the "jumps" ahead), and a *backward window* $w_b \geq 0$, which bounds how "old" ciphertexts should be correctly decapsulated. Old here refers to ciphertexts encapsulated prior to the most recently encapsulated one received so far. For an illustration of a sliding window, see Figure 6.1.

Note that if all ciphertexts produced by encapsulation are assumed to be distinct[2], then with $w_b = 0$ and $w_f = 1$ this class is equivalent to perfect ordering defined below. Compared to no ordering, weak ordering imposes a weaker requirement as schemes in this class only need to handle potential drops and reordering, but never repeated ciphertexts. This means that even with $w_b = w_f = \infty$, weak ordering is still not equal to no ordering because repetitions are not permitted.

[1] For weak ordering, there is an additional condition on the encapsulation procedure not covered by the correctness game. See Weak Ordering in Section 6.2.1 below.

[2] This is actually not a pre-requisite for perfect ordering, making weak and perfect ordering disjoint classes.

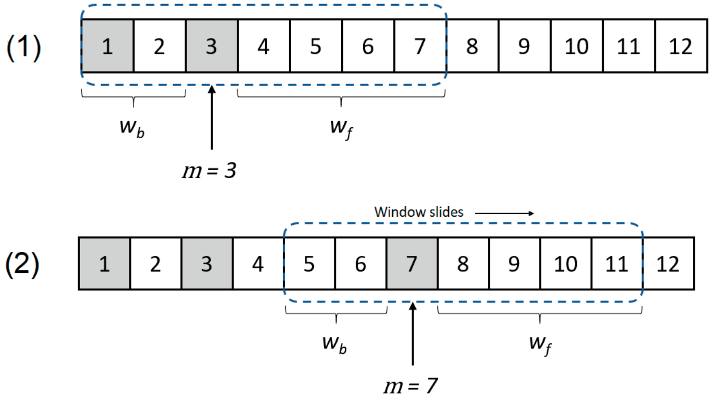

Figure 6.1: A sliding window for correctness under weak ordering with forward window $w_f = 4$ and backward window $w_b = 2$.
(1) Ciphertexts number 1 and 3 have arrived and been accepted. The highest received ciphertext index is $m = 3$.
(2) Ciphertext with encapsulation index 7 has arrived and been accepted (it is within the window in (1)). The sliding window has moved right and updated to $m = 7$. At this point, ciphertexts 2 and 4 are permanently dropped, but 5 and 6 could still arrive and be accepted.

Perfect Ordering. This class requires successful decapsulations only when the input to the decapsulator is a prefix of the sequence of outputs of the encapsulator. That is, the decapsulator must only accept and correctly handle ciphertexts that arrive in the exact sequence they were produced by encapsulation. Hence no replays, reordering or omissions of ciphertexts are covered.

6.2.1 Correctness Games

The correctness games $\mathbf{G}^{\text{nO-correct}}_{\mathsf{OS\text{-}KEM}}$, $\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ and $\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}$ (shown on the left side in Figures 6.2, 6.3 and 6.4) provide access to the encapsulation and decapsulation algorithms of scheme OS-KEM via oracles Enc and Dec. At the start of the game, the key generation algorithm is run and the adversary is given the secret key as well as the initial encapsulation and decapsulation states. The adversary is then allowed to query the oracles an unbounded number of times in any order it sees fit. From each query it obtains the output of the underlying scheme algorithm, including the updated state. Games $\mathbf{G}^{\text{nO-correct}}_{\mathsf{OS\text{-}KEM}}$ and $\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ keep a table $\mathsf{T}[\cdot]$ where each key K corresponding to ciphertext C produced in an encapsulation query is stored and can be obtained through $\mathsf{T}[C]$. Game $\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}$ also keeps a table $\mathsf{T}[\cdot]$, but stores both ciphertext and encapsulated key at the position of the encapsulation index n_e for reasons that will become apparent later. In addition to the table, games $\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ and $\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}$ keep some internal variables used to determine whether the sequence of ciphertexts input to the decapsulation oracle is supported by the correctness level.

No Ordering. In correctness under *no ordering* all sequences are supported,

```
Game $\mathbf{G}^{\text{nO-correct}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A})$
1  $(sk, st_e, st_d) \leftarrow_{\$} \mathsf{KG}()$
2  $\mathcal{A}^{\textsc{Enc},\textsc{Dec}}(sk, st_e, st_d)$
3  Return $\mathsf{win}$

$\textsc{Enc}()$:
4  $(K, C, st_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$
5  $\mathsf{T}[C] \leftarrow K$
6  Return $(K, C, st_e)$

$\textsc{Dec}(C)$:
7  If $C \notin \mathcal{C}$ return $\perp$
8  Else $(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
9  If $\mathsf{T}[C] \neq \perp$:
10     If $K \neq \mathsf{T}[C]$ then $\mathsf{win} \leftarrow \mathsf{true}$
11 Return $(K, st_d)$
```

```
Game $\mathbf{G}^{\text{nO-ind-cca}}_{b,\mathsf{OS\text{-}KEM}}(\mathcal{A})$
1  $(sk, st_e, st_d) \leftarrow_{\$} \mathsf{KG}()$
2  $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Enc}_b,\textsc{Dec}}()$
3  Return $b^*$

$\textsc{Enc}_b()$:
4  $(K_1, C, st_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$;
   $K_0 \leftarrow_{\$} \mathcal{K}$
5  $\mathsf{T}[C] \leftarrow K_b$
6  Return $(K_b, C)$

$\textsc{Dec}(C)$:
7  $(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
8  If $\mathsf{T}[C] \neq \perp$:
9      Return $\perp$
10 Return $K$
```

Figure 6.2: Left: Correctness game for OS-KEMs under no ordering. **Right:** Game defining nO-ind-cca security of OS-KEMs under nO-correctness.

so for all C in the ciphertext space $\mathcal{C}$, the key K produced by the decapsulation algorithm should match the one stored at $\mathsf{T}[C]$ – if there is a key stored there. This is precisely the meaning of lines 7-10 on the left-hand side of Figure 6.2. On line 7, we assert that C is a valid ciphertext. Line 9 checks whether it has been previously obtained in an encapsulation query (remember that all table entries $\mathsf{T}[\cdot]$ are initially set to $\perp$, and that a key $K \neq \perp$ is stored at $\mathsf{T}[C]$ only if (K, C) is output by the encapsulation algorithm in an $\textsc{Enc}$-query). If so, correctness is violated and the adversary wins if the key output by the decapsulation algorithm is different from the one stored at $\mathsf{T}[C]$ (line 10).

Weak Ordering. Correctness under *weak ordering* is slightly more complex. First of all, game $\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ has parameters w_f and w_b which bound the omissions and reordering of permissible ciphertext sequences by setting the size of the sliding window. To determine whether a ciphertext is supported, the game must also keep track of the order in which ciphertexts are produced by encapsulation, whether a ciphertext has been previously queried to decapsulation and the index of the highest labelled ciphertext received and accepted so far. The encapsulation order is maintained by encapsulation index n_e and stored in table $\mathrm{I}[\cdot]$ at location $\mathrm{I}[C]$ for ciphertext C. To check that the ciphertext is new (not replayed), the game keeps set $\mathcal{S}$ to which a ciphertext is added after is has been received and accepted by decapsulation. The highest received ciphertext index (counted by when it was encapsulated) is stored in variable m and updated when a new, more recently encapsulated ciphertext, is received and accepted.

```
$\boxed{\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}(\mathcal{A}),}$ $\mathbf{G}^{\text{wO-r-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}(\mathcal{A})$

$n_e \leftarrow 0;\ m \leftarrow 0$
$\mathcal{S} \leftarrow \emptyset;\ \mathsf{sync} \leftarrow \mathsf{true}$
$(sk, st_e, st_d) \leftarrow_\$ \mathsf{KG}()$
$\mathcal{A}^{\text{Enc},\text{Dec}}(sk, st_e, st_d)$
Return $\mathsf{win}$

$\underline{\text{Enc}():}$
$n_e \leftarrow n_e + 1$
$(K, C, st_e) \leftarrow_\$ \mathsf{E}(sk, st_e)$
$\mathrm{I}[C] \leftarrow n_e$
$\mathsf{T}[C] \leftarrow K$
Return $(K, C, st_e)$

$\underline{\text{Dec}(C):}$
If $C \notin \mathcal{C}$ return $\bot$
$(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
If $\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S})$
  $\boxed{\text{and } \mathsf{sync} = \mathsf{true}}$ then:
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
    $m \leftarrow \max(\mathrm{I}[C], m)$
    If $\mathsf{T}[C] \neq K$ then $\mathsf{win} \leftarrow \mathsf{true}$
$\boxed{\text{Else } \mathsf{sync} \leftarrow \mathsf{false}}$
Return $(K, st_d)$
```

```
$\boxed{\mathbf{G}^{\text{wO-ind-cca}}_{b,\mathsf{OS\text{-}KEM},w_f,w_b}(\mathcal{A})}$ , $\mathbf{G}^{\text{wO-r-ind-cca}}_{b,\mathsf{OS\text{-}KEM},w_f,w_b}(\mathcal{A})$

$n_e \leftarrow 0;\ m \leftarrow 0;\ \mathcal{S} \leftarrow \emptyset;\ \mathsf{sync} \leftarrow \mathsf{true}$
$(sk, st_e, st_d) \leftarrow_\$ \mathsf{KG}()$
$b^* \leftarrow_\$ \mathcal{A}^{\text{Enc}_b,\text{Dec}}()$
Return $b^*$

$\underline{\text{Enc}_b():}$
$n_e \leftarrow n_e + 1$
$(K_1, C, st_e) \leftarrow_\$ \mathsf{E}(sk, st_e)$
$\mathrm{I}[C] \leftarrow n_e$
$K_0 \leftarrow_\$ \mathcal{K}$
Return $(K_b, C)$

$\underline{\text{Dec}(C):}$
$(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
If $\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S})$
  $\boxed{\text{and } \mathsf{sync} = \mathsf{true}}$ then:
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
    $m \leftarrow \max(\mathrm{I}[C], m)$
    Return $\bot$
$\boxed{\text{Else } \mathsf{sync} \leftarrow \mathsf{false}}$
Return $K$
```

Figure 6.3: Left: Correctness and robust correctness game for OS-KEMs under weak ordering. **Right:** Games for ind-cca security of OS-KEMs under weak ordering and robust weak ordering.
All games are parameterized by $w_b \geq 0$ and $w_f \geq 1$. Predicate $\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S})$ checks that C is not replayed and within the sliding window. We assume that if $\mathrm{I}[C] = \bot$ then $\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S})$ evaluates to false. The code in boxes is executed in the games for ordinary correctness, but not for robust correctness.

To make the code of the game more readable we introduce a predicate, which given the current ciphertext C, the highest accepted index m and the set of previously received ciphertexts $\mathcal{S}$ determines if the ciphertext is *Supported.*

$$\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S}) = (C \notin \mathcal{S}) \wedge (m - w_b \leq \mathrm{I}[C] \leq m + w_f)\,.$$

The predicate is parameterized by the forward and backward windows. It evaluates to true if C is not in the set of previously queried ciphertexts and if the encapsulation index of C is within the bounds of the current sliding window, which extends backwards from m by w_b and forward by w_f. We assume that if $\mathrm{I}[C] = \perp$, then conditions such as $(m - w_b \leq \mathrm{I}[C] \leq m + w_f)$ are false. Hence $\mathit{Supported}_{w_f,w_b}(C, m, \mathcal{S})$ implicitly checks that C has been produced by a previous encapsulation query, ensuring that correctness only covers such ciphertexts.

In addition to checking that the currently queried ciphertext is supported, the game also needs to ensure that all previous queries have been supported. This is done with the *synchronization* flag sync, which is initialized to true at the start of the game, and remains so as long as all queries to the Dec-oracle are within the current sliding window. For standard wO-correctness, a scheme only needs to handle ciphertext sequences that do not trigger the sync-flag. In Section 6.3 we introduce a stricter form of correctness called *robust correctness* in which schemes are required to recover from unsupported procedure calls. Analogous to *no ordering*, an adversary wins the weak ordering correctness game if it can provoke the scheme's decapsulation algorithm into producing a different key than the one stored in $\mathsf{T}[C]$ on a supported ciphertext C while sync is true.

Finally, in order for scheme $\mathsf{OS\text{-}KEM}$ to be considered correct under weak ordering, we enforce an additional condition outside of the correctness game, namely that there is a one-to-one function mapping keys to ciphertexts. This means that the encapsulation algorithm cannot "re-use" ciphertexts across multiple keys, rather each ciphertext produced by encapsulation must correspond to a unique key. For stateless schemes, this is implicit from correctness. (E.g., for ordinary S-KEMs it is clear that there can only be one key associated to each ciphertext, otherwise correct decapsulation would be impossible). However, in the stateful setting it is possible to have identical ciphertexts for different keys because of the extra information contained in the state; it is enough that there is a unique key associated to each ciphertext for each decapsulation state. But in weak ordering this becomes quite complicated, as it is unclear which ciphertext the decapsulator should mark as accepted (which index should be used to update the sliding window) if the same ciphertext is expected at multiple positions within the current window.

In fact, we require something stronger than an injective key-ciphertext relationship, because if the same ciphertext is received twice within the sliding window, the decapsulator will not know which position to mark as received, regardless of whether the ciphertext was encapsulated to encode the same key or two different keys. Therefore, correctness for OS-KEMs under weak ordering demands both an injective key-ciphertext relationship, and that all ciphertexts produced by encapsulation are unique within each sliding window. For simplicity, we require unique ciphertexts produced by encapsulation overall, which directly gives the one-to-one mapping. Formally, the condition is that for scheme $\mathsf{OS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$ to be

```
Game $\boxed{\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A}),}$ $\mathbf{G}^{\text{pO-r-correct}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A})$
$n_e \leftarrow 0;\ n_d \leftarrow 0;\ \mathsf{sync} \leftarrow \mathsf{true}$
$(sk, st_e, st_d) \leftarrow_{\$} \mathsf{KG}()$
$\mathcal{A}^{\text{Enc},\text{Dec}}(sk, st_e, st_d)$
Return $\mathsf{win}$

$\text{Enc}()$:
$n_e \leftarrow n_e + 1$
$(K, C, st_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$
$\mathsf{T}[n_e] \leftarrow (K, C)$
Return $(K, C, st_e)$

$\text{Dec}(C)$:
If $C \notin \mathcal{C}$ return $\bot$
$(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
$(K', C') \leftarrow \mathsf{T}[n_d]$
If $C = C'$ $\boxed{\text{and } \mathsf{sync} = \mathsf{true}}$:
    $n_d \leftarrow n_d + 1$
    If $K \neq K'$ then $\mathsf{win} \leftarrow \mathsf{true}$
$\boxed{\text{Else } \mathsf{sync} \leftarrow \mathsf{false}}$
Return $(K, st_d)$
```

```
Game $\boxed{\mathbf{G}^{\text{pO-ind-cca}}_{b,\mathsf{OS\text{-}KEM}}(\mathcal{A})}$, $\mathbf{G}^{\text{pO-r-ind-cca}}_{b,\mathsf{OS\text{-}KEM}}(\mathcal{A})$
$n_e \leftarrow 0;\ n_d \leftarrow 0;\ \mathsf{sync} \leftarrow \mathsf{true}$
$(sk, st_e, st_d) \leftarrow_{\$} \mathsf{KG}()$
$b^* \leftarrow_{\$} \mathcal{A}^{\text{Enc}_b,\text{Dec}}()$
Return $b^*$

$\text{Enc}_b()$:
$n_e \leftarrow n_e + 1$
$(K_1, C, st_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$
$K_0 \leftarrow_{\$} \mathcal{K}$
$\mathsf{T}[n_e] \leftarrow C$
Return $(K_b, C)$

$\text{Dec}(C)$:
$(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
If $C = \mathsf{T}[n_d]$ $\boxed{\text{and } \mathsf{sync} = \mathsf{true}}$:
    $n_d \leftarrow n_d + 1$
    Return $\bot$
$\boxed{\text{Else } \mathsf{sync} \leftarrow \mathsf{false}}$
Return $K$
```

Figure 6.4: Left: Correctness game for OS-KEMs under perfect ordering and robust perfect ordering. **Right:** Games for pO-ind-cca security of OS-KEMs under perfect ordering and robust perfect ordering.
If $\mathsf{T}[n_d] = \bot$ then K' and C' both parse as $\bot$. The code in boxes is executed in the games for ordinary correctness, but not for robust correctness.

called wO-correct, in addition to having zero advantage for all adversaries in the correctness game it also holds that:

$$\text{For all } sk \in [\mathsf{KG}()] \text{ and all } st_e, st'_e \in \mathcal{S}_e, \text{ if } st_e \neq st'_e \text{ then } \Pr[\, C \neq C' \,] = 1,$$

where C and C' have been produced, respectively, by $(K, C, \hat{st}_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$ and $(K', C', \hat{st}'_e) \leftarrow_{\$} \mathsf{E}(sk, st'_e)$. The probability is over the coins of E.

This extra requirement on the encapsulator is also made for example by Kohno, Palacio and Black [38] in the setting of stateful encryption when they define what they call a "Type 3 cryptographic transform", which is a correctness class similar to our weak ordering.

Perfect Ordering. In correctness under *perfect ordering*, the game needs only to keep track of and compare the encapsulation index n_e of a certain ciphertext,

C, to the decapsulation index n_d when C is queried to oracle Dec. To do this, a Table $\mathsf{T}[\cdot]$ (which—because of its potentially unbounded size—we assume is allocated memory based on need and as usual has all entries initialized to $\perp$) is used to store the key-ciphertext pairs produced by encapsulation. On query Dec(C) the game checks that C is a valid ciphertext (line 9) and that C is equal to the ciphertext stored in $\mathsf{T}[n_d]$ (line 12). Note that this will be the case if n_d is equal to n_e at the time of encapsulation. The game also checks that all previously queried ciphertexts have matched up (using the sync-flag, which is initialized to true). If so, a correct scheme must decapsulate C to the key stored in $\mathsf{T}[n_d]$.

6.3 Robustness

In all three games ($\mathbf{G}^{\text{nO-correct}}_{\mathsf{OS\text{-}KEM}}$, $\mathbf{G}^{\text{wO-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ and $\mathbf{G}^{\text{pO-correct}}_{\mathsf{OS\text{-}KEM}}$), the adversary may attempt to break correctness multiple times and is successful if the scheme fails at any point. However, it can not make queries outside of the supported sequence(s) (e.g. outside of the sliding window for weak ordering, or a repeated query in weak or perfect ordering). If it does so, the game goes out of sync (the sync flag is set to false), and the adversary has lost its chance of breaking correctness. Depending on perspective, this might be considered overly punitive. One could for example imagine a scenario in which a network protocol responsible for transmission is expected to re-transmit a rejected package at some later point. If the package is then within the supported window, it should be accepted and correctly handled by the scheme.

To meet the expectations in such scenario, we introduce a stricter kind of correctness called *robust correctness* for weak ordering and perfect ordering. In the robust correctness games $\mathbf{G}^{\text{wO-r-correct}}_{\mathsf{OS\text{-}KEM},w_f,w_b}$ and $\mathbf{G}^{\text{pO-r-correct}}_{\mathsf{OS\text{-}KEM}}$ shown in Figure 6.3 and 6.4, an adversary that submits an unsupported query can still win if it later manages to make the scheme fail to correctly decapsulate a supported ciphertext. This means that the scheme will be considered incorrect if it does not recover from unsupported procedure calls. Robust correctness will be even more valuable for stateful schemes providing the ability to puncture specific ciphertexts so that they are no longer covered by correctness (see Section 8 on puncturable ordered S-KEMs).

6.4 Security

For the security of OS-KEMs we consider standard ind-cpa as well as an adaption of ind-cca to the three correctness classes. (Note that correctness only plays a role in the security games when a decryption oracle is present.) Like for S-KEMs, indistinguishability under chosen plaintext attack (ind-cpa) aims to capture the scenario of a passively eavesdropping attacker that sees either all real encapsulations (key-ciphertext pairs), or all random (ciphertexts together with randomly sampled keys). The aim of the adversary is to distinguish the real world from the random one, and an OS-KEM scheme is said to be ind-cpa secure if this is infeasible. The notion is captured formally by game $\mathbf{G}^{\text{cpa}}_{b,\mathsf{OS\text{-}KEM}}$ in Figure 6.5. We

```
Game $\mathbf{G}^{\mathrm{cpa}}_{\textsf{OS-KEM}}(\mathcal{A})$
    $(sk, st_e, st_d) \leftarrow_{\$} \mathsf{KG}()$
    $b^* \leftarrow_{\$} \mathcal{A}^{\textsc{Enc}_b}()$
    Return $b^*$

$\textsc{Enc}_b()$:
    $(K_1, C, st_e) \leftarrow_{\$} \mathsf{E}(sk, st_e)$
    $K_0 \leftarrow_{\$} \mathcal{K}$
    Return $(K_b, C)$
```

Figure 6.5: Game formalizing the privacy notion *indistinguishability under chosen-plaintext attack* (ind-cpa) for ordered S-KEMs.

define the advantage as

$$\mathbf{Adv}^{\mathrm{cpa}}_{\textsf{OS-KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\mathrm{cpa}}_{1,\textsf{OS-KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\mathrm{cpa}}_{0,\textsf{OS-KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

For the chosen ciphertext (ind-cca) notion, the adversary is additionally provided with honest decapsulations, *except* on queries where correctness demands that the decapsulation algorithm returns a key previously given to the adversary in an Enc-query. These queries are aborted and the adversary gets only $\perp$ back. The reason is that returning the actual key would give the adversary a "trivial win", which must be disallowed in a security game.

To explain what this means, consider the ind-cca game $\mathbf{G}^{\mathrm{cca}}_{b,\textsf{S-KEM}}$ for standard S-KEMs in Figure 5.1. What, really, is the purpose of the first line in the code of the decryption oracle (line 8), where we check if the queried ciphertext C is a member of the set S? The answer is that S contains the ciphertexts previously output by the encapsulation oracle, and if an adversary was to obtain an honest decapsulation of such a ciphertext, it could tell if the previously performed encapsulation was honest (real) too. By simply comparing the output from such a decapsulation query to the key it was given by the encapsulation oracle in the earlier query, it would with high probability be able to tell if it is playing the real or the random game. Hence the condition "If $C \in S$ return $\perp$" on line 8 in the decapsulation oracle prevents a trivial win.

In general, trivial wins are directly related to the correctness of the underlying scheme, since correctness imposes restrictions on the decapsulation algorithm which fixes its output on certain queries. Returning to S-KEMs, imagine a modified version of game $\mathbf{G}^{\mathrm{cca}}_{b,\textsf{S-KEM}}$ (Figure 5.1) without the membership check on C in a $\textsc{Dec}(C)$ query (line 8). An adversary playing this game could make an $\textsc{Enc}()$ query and get back (K^e, C) and then perform decapsulation query $K^d \leftarrow \textsc{Dec}(C)$ without penalty. Since the adversary knows that correctness holds, K^d must be the honest decapsulation of C, so if $K^d \neq K^c$ it is playing the random game. The important insight for the adversary is that *there is only one* valid response to its decapsulation query given correctness; the response to its query is *fixed*. We conclude that the extra condition in the decapsulation oracle, which prevents trivial wins such as the one described, is essential for a meaningful security notion.

As a generic method for specifying security notions which avoid trivial wins introduced by correctness, Rogaway and Zhang recently proposed the concept of *indistinguishability up to correctness* [44]. There, oracles are *silenced* whenever their output is fixed by correctness. We use the intuition behind their method when defining security for OS-KEMs under all three correctness classes on the right in Figure 6.2, 6.3 and 6.4. Note the similarities between the correctness game and the security game in each figure. The decapsulation oracle in the security game is silenced (only returns $\perp$) precisely when the correctness game could be won, i.e. when there is a correctness demand on the decapsulation. We call this approach *intuitive* indistinguishability up to correctness, since it builds on the ideas of Rogaway and Zhang, but without the full formalism they develop.

The advantage of an adversary $\mathcal{A}$ playing the xO-ind-cca security games for $\mathrm{xO} \in \{\mathrm{nO}, \mathrm{wO}, \mathrm{pO}\}$, is, as usual,

$$\mathbf{Adv}^{\text{xO-ind-cca}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{xO-ind-cca}}_{1,\mathsf{OS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\text{xO-ind-cca}}_{0,\mathsf{OS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

We also introduce an ind-cca notion for the robust versions of each correctness class. As in the correctness games, the difference lies in removing the synchronization condition for silencing. The games are shown on the right in Figure 6.2, 6.3 and 6.4, ignoring the code in boxes. For $\mathrm{xO} \in \{\mathrm{nO}, \mathrm{wO}, \mathrm{pO}\}$, we define the advantage of adversary $\mathcal{A}$ to be

$$\mathbf{Adv}^{\text{xO-r-ind-cca}}_{\mathsf{OS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{xO-r-ind-cca}}_{1,\mathsf{OS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\text{xO-r-ind-cca}}_{0,\mathsf{OS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

6.5 Instantiations

Having defined correctness and security for stateful symmetric KEM schemes we now give some examples of schemes which meet the requirements. We provide one scheme for each correctness level in increasing order of complexity, beginning with *no ordering*.

6.5.1 OS-KEM Instantiation under No Ordering

As pointed out previously, correctness under no ordering is essentially equivalent to that of stateless S-KEMs, so only syntax modifications are necessary to obtain a correct and secure stateful scheme in this correctness class. Let $\mathsf{S\text{-}KEM} = (\mathsf{S\text{-}KEM.KG}, \mathsf{S\text{-}KEM.E}, \mathsf{S\text{-}KEM.D})$ be a standard (stateless) symmetric KEM scheme. We construct stateful S-KEM scheme $\mathsf{nOS\text{-}KEM} = (\mathsf{nOS\text{-}KEM.KG}, \mathsf{nOS\text{-}KEM.E}, \mathsf{nOS\text{-}KEM.D})$ based on $\mathsf{S\text{-}KEM}$ as shown in Figure 6.6. The secret key space $\mathcal{SK}$, key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$ of $\mathsf{nOS\text{-}KEM}$ are the same as those of $\mathsf{S\text{-}KEM}$. The state spaces of $\mathsf{nOS\text{-}KEM}$ are $\mathcal{S}_e = \mathcal{S}_d = \{\varepsilon\}$.
Correctness of $\mathsf{nOS\text{-}KEM}$ follows trivially from that of $\mathsf{S\text{-}KEM}$ (a correctness adversary which breaks correctness under *no ordering* for $\mathsf{nOS\text{-}KEM}$ must have violated the correctness of the underlying $\mathsf{S\text{-}KEM}$ scheme). Similarly for security, the privacy of $\mathsf{S\text{-}KEM}$ carries over to $\mathsf{nOS\text{-}KEM}$. Albeit straightforward, we state the result as a theorem.

Theorem 4. *Ordered S-KEM scheme* $\mathsf{nOS\text{-}KEM} = (\mathsf{nOS\text{-}KEM.KG}, \mathsf{nOS\text{-}KEM.E}, \mathsf{nOS\text{-}KEM.D})$ *given in Figure 6.6 is* ind-cpa (ind-cca) *secure given that underlying stateless scheme* $\mathsf{S\text{-}KEM} = (\mathsf{S\text{-}KEM.KG}, \mathsf{S\text{-}KEM.E}, \mathsf{S\text{-}KEM.D})$ *is* ind-cpa (ind-cca) *secure.*

```
nOS-KEM.KG():
    $sk \leftarrow\!\!\$\; \mathsf{S\text{-}KEM.KG}()$
    $st_e \leftarrow \varepsilon;\ st_d \leftarrow \varepsilon$
    Return $(sk, st_e, st_d)$

nOS-KEM.E($sk, st_e$):
    $(K, C) \leftarrow\!\!\$\; \mathsf{S\text{-}KEM.E}(sk)$
    Return $(K, C, st_e)$

nOS-KEM.D($sk, st_d, C$):
    $K \leftarrow \mathsf{S\text{-}KEM.D}(sk, C)$
    Return $(K, st_d)$
```

Figure 6.6: Algorithms of nOS-KEM, an instantiation of an ordered symmetric-key KEM under *no ordering* based on the stateless S-KEM scheme S-KEM.

Proof. The result follows from a simple reduction, which we outline for the ind-cca case. The ind-cpa case is analogous and obtained by excluding decapsulation oracles. (Privacy in the ind-cpa sense also follows directly from the stronger ind-cca notion).

Given adversary $\mathcal{A}$ attacking the ind-cca security of nOS-KEM we construct adversary $\mathcal{B}$ against the ind-cca security of S-KEM such that

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathsf{nOS\text{-}KEM}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{ind-cca}}_{\mathsf{S\text{-}KEM}}(\mathcal{B})$$

which proves the theorem. Adversary $\mathcal{B}$ runs $\mathcal{A}$ and relays any queries to its own oracles. Since both adversaries are subject to the same restrictions (decapsulation queries are silenced if the queried ciphertext has been previously output by encapsulation) this perfectly simulates the ind-cca game for $\mathcal{A}$. When $\mathcal{A}$ halts and outputs bit b^*, $\mathcal{B}$ uses b^* as its own guess. It wins precisely when $\mathcal{A}$ has made a correct guess, hence $\mathbf{Adv}^{\text{ind-cca}}_{\mathsf{nOS\text{-}KEM}}(\mathcal{A}) = \mathbf{Adv}^{\text{ind-cca}}_{\mathsf{S\text{-}KEM}}(\mathcal{B})$, showing the claimed bound. □

6.5.2 OS-KEM Instantiation under Perfect Ordering

Next, we move on to the more interesting case of perfect ordering. Here, the decapsulator should only accept ciphertexts sequences that have not been modified at all after being produced by the encapsulator. To do this, the scheme needs to ensure that the decapsulator only accepts ciphertexts that have been produced by encapsulation, that it only accepts the next ciphertext expected in the sequence and that all previous ciphertexts have arrived in order. To meet these demands, ciphertexts must be hard to create for an adversary without access to the encapsulator, and the scheme must keep track of encapsulation and decapsulation indices, as well as a synchronization flag. The first requirement (the "unforgeability" of ciphertexts) is handled by clever use of a pseudorandom function. The rest is covered by the encapsulator and decapsulator states.

We construct the scheme pOS-KEM[F] $=$ (KG, E, D)[3] directly from a PRF $\mathsf{F} : \{0,1\}^k \times \{0,1\}^{\text{in}} \rightarrow \{0,1\}^{\text{out}}$. The algorithms of pOS-KEM[F] are shown

[3] When referring to algorithms of a construction, we use dot notation. For example pOS-KEM[F].KG refers to the key generation algorithm KG of scheme pOS-KEM[F]. From here on we will omit the scheme name preceding the procedure name in the tuple when defining the scheme.

```
pOS-KEM[F].KG():
$sk_1 \leftarrow\$ \{0,1\}^k$
$sk_2 \leftarrow\$ \{0,1\}^k$
$sk \leftarrow (sk_1, sk_2)$
$st_e \leftarrow 0$; $n_d \leftarrow 1$
$\mathsf{sync} \leftarrow \mathsf{true}$
$st_d \leftarrow (n_d, \mathsf{sync})$
Return $(sk, st_e, st_d)$

pOS-KEM[F].E($sk, st_e$):
$st_e \leftarrow st_e + 1$
$(sk_1, sk_2) \leftarrow sk$
$K \leftarrow \mathsf{F}(sk_1, st_e)$
$C \leftarrow \mathsf{F}(sk_2, st_e)$
Return $(K, C, st_e)$

pOS-KEM[F].D($sk, st_d, C$):
$(n_d, \mathsf{sync}) \leftarrow st_d$
$(sk_1, sk_2) \leftarrow sk$
If $C \neq \mathsf{F}(sk_2, n_d)$ or
$\mathsf{sync} \neq \mathsf{true}$ then:
    $\boxed{\mathsf{sync} \leftarrow \mathsf{false}}$
    $st_d \leftarrow (n_d, \mathsf{sync})$
    Return $(\perp, st_d)$
$K \leftarrow \mathsf{F}(sk_1, n_d)$
$st_d \leftarrow (n_d + 1, \mathsf{sync})$
Return $(K, st_d)$
```

Figure 6.7: Algorithms of pOS-KEM[F]; an instantiation of an ordered symmetric-key KEM under *perfect ordering* based on pseudorandom function F. The code in the box is executed under standard correctness, but left out for robust correctness.

in Figure 6.7. The PRF is used doubly, both to derive the keys and to ensure the unforgeability of ciphertexts by making them unpredictable. Therefore the scheme will need a pair of secret keys, so the secret-key space of pOS-KEM[F] is $\mathcal{SK} = \{0,1\}^k \times \{0,1\}^k$. As in the S-KEM PRF construction in Figure 5.3, the key space of pOS-KEM is the range of the PRF, i.e. $\mathcal{K} = \{0,1\}^{\mathrm{out}}$. In contrast to the stateless setting, the ciphertexts of pOS-KEM will also be computed using the PRF. This is to avoid attacks where the adversary guesses the next ciphertext, which is necessary to obtain ind-cca security in the stateful case. Therefore the ciphertext space is also $\mathcal{C} = \{0,1\}^{\mathrm{out}}$. The state spaces are $\mathcal{S}_e = \{0,1\}^{\mathrm{in}}$ and $\mathcal{S}_d = \{0,1\}^{\mathrm{in}} \times \{\mathsf{true}, \mathsf{false}\}$, the latter to accommodate both an index and the boolean sync-flag. Note that the encapsulation and decapsulation state contain indices, which in the code of the game are treated as numbers. We assume here an implicit encoding of the integers in $\{0, \ldots, 2^{\mathrm{in}} - 1\}$ as strings in $\{0,1\}^{\mathrm{in}}$. To perform for example addition, the string st_e is parsed as an integer, the operation is completed and the resulting number is converted back to a string.

Privacy. Security of pOS-KEM[F] is derived from the pseudorandom function on which it is based. Because privacy in the ind-cpa sense is independent of the correctness class (see Figure 6.5) and weaker than pO-ind-cca, we here onward focus on the stronger chosen ciphertext security.

Theorem 5. *Let* $\mathsf{F} : \{0,1\}^k \times \{0,1\}^{\mathrm{in}} \to \{0,1\}^{\mathrm{out}}$ *be a pseudorandom function. Let* pOS-KEM[F] = (KG, E, D) *be the ordered symmetric-key KEM scheme based on* F *as defined in Figure 6.7. Then* pOS-KEM[F] *is* pO-ind-cca*-secure given that* F *is a secure PRF.*

Proof. The proof is by reduction. Given adversary $\mathcal{A}$ against the pO-ind-cca security of pOS-KEM[F] we construct adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ attacking the PRF security of F such that

$$\mathbf{Adv}^{\text{pO-ind-cca}}_{\mathsf{pOS\text{-}KEM[F]}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_1) + \frac{1}{2^{\text{out}}} + \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_2).$$

If F is a secure PRF with a large output space, then all terms on the right hand side will be small, making the advantage of $\mathcal{A}$ small as well. To show the claimed bound, we partition the probability that adversary $\mathcal{A}$ correctly guesses bit b in game $\mathbf{G}^{\text{pO-ind-cca}}_{b,\mathsf{OS\text{-}KEM}}$ into two distinct cases.

Case 1: $\mathcal{A}$ correctly guesses b and has managed to *forge* a ciphertext.

Case 2: $\mathcal{A}$ correctly guesses b and has not forged a ciphertext.

By *forge* a ciphertext we mean that $\mathcal{A}$ has created a ciphertext $C \in \{0,1\}^{\text{out}}$ which it has not obtained from its encapsulation oracle, but which the decapsulation algorithm of the scheme accepts. (Recall that $\mathsf{pOS\text{-}KEM[F].D}$ is said to *accept* ciphertext C if the key it returns is not $\perp$.) We claim that the following equations hold.

$$\mathbf{Adv}^{\text{pO-ind-cca}}_{\mathsf{pOS\text{-}KEM[F]}}(\mathcal{A}) = \Pr[\,\text{Case } 1\,] + \Pr[\,\text{Case } 2\,] \tag{6.1}$$

$$\Pr[\,\mathcal{A} \text{ correctly guesses } b \text{ and forges}\,] \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_1) + \frac{1}{2^{\text{out}}} \tag{6.2}$$

$$\Pr[\,\mathcal{A} \text{ correctly guesses } b \text{ and does not forge}\,] \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_2) \tag{6.3}$$

Equation (6.1) is basic probability theory since the events "$\mathcal{A}$ has forged" and "$\mathcal{A}$ has not forged" are complementary. To show Equation (6.2), we first use that $\Pr[\,\mathcal{A}$ correctly guesses b and forges$\,] \leq \Pr[\,\mathcal{A}$ forges$\,]$, so it's enough to bound the probability that $\mathcal{A}$ manages to forge. For this, we can intuitively justify the two terms in Equation (6.2) by arguing that either F is a "bad" PRF (meaning not close to an ideal random function with completely unpredictable output), in which case the fact that $\mathcal{A}$ can forge means that $\mathcal{B}_1$ can detect a difference between F and an ideal random function, contributing to $\mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_1)$. Or F is actually a perfect random function and $\mathcal{A}$ has simply managed to guess the next expected ciphertext, which it can do with probability $1/2^{\text{out}}$. To formally justify Equation (6.2) we would need to formalize what it means for a PRF to be *unforgeable*, which is outside the scope of this work. (The interested reader is referred to e.g. [4].)

To show Equation (6.3) we construct PRF adversary $\mathcal{B}_2$ that runs $\mathcal{A}$, simulating game $\mathbf{G}^{\text{pO-ind-cca}}_{b,\mathsf{pOS\text{-}KEM[F]}}$ using its own FN-oracle and returns the guess of $\mathcal{A}$ as its own. The code is given in Figure 6.8. Adversary $\mathcal{B}_2$ begins by drawing a random element sk_2 from the key space of the PRF. It then runs $\mathcal{A}$. On encapsulation queries from $\mathcal{A}$ it uses its own FN-oracle to compute the key. It computes the ciphertext by itself using sk_2 and stores it in table $\mathsf{T}[\cdot]$. On decapsulation queries from $\mathcal{A}$, adversary $\mathcal{B}_2$ checks if the queried ciphertext is a forgery (i.e., if it is the next ciphertext in the decapsulation order, but has not previously been given to $\mathcal{A}$ by $\mathcal{B}_2$ in an encapsulation query). If so, $\mathcal{B}_2$ aborts and gives up its own game. If the query is not a forgery, $\mathcal{B}_2$ simply returns $\perp$. This perfectly simulates the game for $\mathcal{A}$ as long as $\mathcal{A}$ does not query a forged ciphertext. The reason is that $\mathsf{pOS\text{-}KEM[F]}$ is constructed to only return a key $K \neq \perp$ when the decapsulator is given the next expected ciphertext in order, but as long as this ciphertext has

```
Adversary $\mathcal{B}_2^{\mathrm{FN}}()$:
    $n_e \leftarrow 0$; $n_d \leftarrow 1$
    $sk_2 \leftarrow_\$ \{0,1\}^k$
    $b^* \leftarrow_\$ \mathcal{A}^{\mathrm{ENC}^*,\mathrm{DEC}^*}()$
    Return $b^*$

Enc$^*()$:
    $n_e \leftarrow n_e + 1$
    $K \leftarrow \mathrm{FN}(n_e)$
    $C \leftarrow \mathsf{F}(sk_2, n_e)$
    $\mathsf{T}[n_e] \leftarrow C$
    Return $(K, C)$

Dec$^*(C)$:
    If $(\mathsf{T}[n_d] \neq C)$ and
       $(C = \mathsf{F}(sk_2, n_d))$:
        Abort
    Else:
        $n_d \leftarrow n_d + 1$
        Return $\perp$
```

Figure 6.8: Code of PRF-adversary $\mathcal{B}_2$ for the proof of Theorem 5. A star superscript to a procedure indicates that it is a subroutine in the code of $\mathcal{B}_2$, used to simulate an oracle expected by adversary $\mathcal{A}$. "Abort" means that adversary $\mathcal{B}_2$ stops simulating the game for $\mathcal{A}$, halts and returns 0.

been previously produced by encapsulation, game $\mathbf{G}^{\text{pO-ind-cca}}_{b,\mathsf{pOS\text{-}KEM[F]}}$ will silence such queries. Only if C is the next expected ciphertext, but not previously given in an encapsulation query, will the scheme give an honest decapsulation without being silenced by the game. But this is precisely when C is a forgery. Since we are trying to bound the probability that $\mathcal{A}$ correctly guesses b and *does not* forge, $\mathcal{B}_2$ is therefore perfectly simulating decapsulation queries for all adversaries $\mathcal{A}$ of interest. From the reasoning above it is clear that

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_2) = \Pr\left[\mathcal{A} \text{ correctly guesses } b \text{ in } \mathbf{G}^{\text{pO-ind-cca}}_{b,\mathsf{pOS\text{-}KEM[F]}} \text{ and does not forge}\right],$$

which directly shows the bound in Equation (6.3). □

The above result is for ind-cca-security of pOS-KEM[F] under standard correctness. If one adds the requirements of robustness (i.e. removes the synchronization flags from the games and construction), the bound changes slightly. The difference lies in the fact that an adversary in the robust version is given multiple attempts at forging a ciphertext, so the advantage relating to successful forgeries increases.

6.5.3 OS-KEM Instantiation under Weak Ordering

We continue using pseudorandom functions as building blocks to construct a scheme for the most complex correctness class: *weak ordering.* The scheme has parameters w_f and w_b, the *forward window* and *backward window* respectively, which determine the size of the sliding window in which the decapsulator accepts ciphertexts. Other than that, it functions much the same as pOS-KEM[F]; the PRF $\mathsf{F} : \{0,1\}^k \times \{0,1\}^{\text{in}} \to \{0,1\}^{\text{out}}$ is used both to produce keys and ensure that ciphertexts are unforgeable. We call the scheme $\mathsf{wOS\text{-}KEM}_{w_f,w_b}[\mathsf{F}]$, and as usual it consists of three procedures; $\mathsf{wOS\text{-}KEM}_{w_f,w_b}[\mathsf{F}] = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$. The algorithms of wOS-KEM are shown in Figure 6.9. In addition to the code there, the decapsulator uses a predicate called *Accept* to check if a ciphertext should be accepted.

```
wOS-KEM[F].KG():
$sk_1 \leftarrow_{\$} \{0,1\}^k$
$sk_2 \leftarrow_{\$} \{0,1\}^k$
$sk \leftarrow (sk_1, sk_2)$
$st_e \leftarrow 0;\ m \leftarrow 0$
$\mathcal{S} \leftarrow \emptyset$
$\mathsf{sync} \leftarrow \mathsf{true}$
$st_d \leftarrow (m, \mathsf{sync}, \mathcal{S})$
Return
$(sk, st_e, st_d)$
```

```
wOS-KEM[F].E($sk, st_e$):
$st_e \leftarrow st_e + 1$
$(sk_1, sk_2) \leftarrow sk$
$K \leftarrow \mathsf{F}(sk_1, st_e)$
$C \leftarrow \mathsf{F}(sk_2, st_e)$
$\tilde{C} \leftarrow (C, st_e)$
Return $(K, \tilde{C}, st_e)$
```

```
wOS-KEM[F].D($sk, st_d, \tilde{C}$):
$(m, \mathsf{sync}, \mathcal{S}) \leftarrow st_d$
$(sk_1, sk_2) \leftarrow sk$
$(C, st_e) \leftarrow \tilde{C}$
If $Accept(sk_2, C, st_e, m, \mathcal{S})$
and $\mathsf{sync} = \mathsf{true}$ then:
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
    $m \leftarrow \max(m, st_e)$
    $st_d \leftarrow (m, \mathsf{sync}, \mathcal{S})$
    $K \leftarrow \mathsf{F}(sk_1, st_e)$
Else:
    $\boxed{\mathsf{sync} \leftarrow \mathsf{false}}$
    $st_d \leftarrow (m, \mathsf{sync}, \mathcal{S})$
    $K \leftarrow \perp$
Return $(K, st_d)$
```

Figure 6.9: Algorithms of wOS-KEM$_{w_f,w_b}$; an instantiation of an ordered symmetric-key KEM under *weak ordering* based on pseudorandom function F. Predicate $Accept$ ensures that C has not been replayed, that $\tilde{C}$ is authentic and that st_e is within the current sliding window. The code in the box is executed under standard correctness, but left out for robust correctness.

The predicate is defined as

$$Accept(sk_2, C, st_e, m, \mathcal{S}) = (C \notin \mathcal{S}) \wedge (C = \mathsf{F}(sk_2, st_e)) \wedge (m - w_b \leq st_e \leq m + w_f)\,.$$

The forward and backward windows w_b, w_f are implicit parameters to the predicate, and F is assumed publicly available and hence not needed as input. Note the similarities to the *Supported*-predicate used in game $\mathbf{G}^{\text{wO-ind-cca}}_{b,w_f,w_b}$ (Figure 6.3). The predicate checks for replays through condition $C \notin \mathcal{S}$ and ensures that the ciphertext is within the sliding window via $m - w_b \leq st_e \leq m + w_f$. A difference to the *Supported*-predicate is that the scheme has no internal memory shared by the encapsulator and the decapsulator, so in predicate *Accept* the encapsulation index cannot be fetched from a table, like it is in the game. A solution to this could be to let the decapsulator scan through the current window and for each position i check if $\mathsf{F}(sk_2, i) = C$ (recall that ciphertext C is produced by $C \leftarrow \mathsf{F}(sk_2, st_e)$). To make the construction slightly more efficient, we include the encapsulation index in the ciphertext as a “hint” to the decapsulator. To ensure that the index has not been tampered with, predicate *Accept* checks that $C = \mathsf{F}(sk_2, st_e)$.

Because of the twofold functionality of PRF F, wOS-KEM$_{w_f,w_b}$[F] uses two distinct secret keys, so $\mathcal{SK} = \{0,1\}^k \times \{0,1\}^k$. The session keys are elements in the range of the F, i.e. $\mathcal{K} = \{0,1\}^{\text{out}}$. To incorporate the encapsulation state in the ciphertexts, the ciphertext space is $\mathcal{C} = \{0,1\}^{\text{out}} \times \{0,1\}^{\text{in}}$, because the encapsulation state space is $\mathcal{S}_e = \{0,1\}^{\text{in}}$. The decapsulation state needs to include not only the highest accepted index m, but also a boolean synchronization flag sync and the set of previously accepted ciphertexts $\mathcal{S}$. Hence $\mathcal{S}_d = \{0,1\}^{\text{in}} \times \{\mathsf{true}, \mathsf{false}\} \times \mathcal{P}(\mathcal{C})$, where $\mathcal{P}(\mathcal{C})$ is the power set of the ciphertext space. As in the pO-correct instantiation, we assume that strings in $\{0,1\}^{\text{in}}$ are parsed as integers for the purpose of addition operations.

Privacy. We will not prove the security of $\mathsf{wOS\text{-}KEM}_{w_f,w_b}[\mathsf{F}]$; the reasoning in such a proof is similar to that of Theorem 5 because of the similarities to $\mathsf{pOS\text{-}KEM}[\mathsf{F}]$. We do however outline why $\mathsf{wOS\text{-}KEM}_{w_f,w_b}[\mathsf{F}]$ is correct under weak ordering, as defined by game $\mathbf{G}^{\text{wO-correct}}_{\mathsf{wOS\text{-}KEM}[\mathsf{F}],w_f,w_b}$ in Figure 6.3 (now parameterized by the constructed scheme). First of all, we note that ciphertexts are unique (meaning each $\tilde{C} \in \mathcal{C}$ is output at most once by encapsulation) because $\tilde{C}$ includes the encapsulation index st_e which is incremented in each call to $\mathsf{wOS\text{-}KEM}[\mathsf{F}].\mathsf{E}$. The inclusion of the encapsulation index also ensures correct decapsulation if $\tilde{C}$ is a supported ciphertext; since PRF F is a *function*, it will always give the same output when evaluated on a fixed input. Hence $K = \mathsf{F}(sk_1, st_e)$ will be returned when the decapsulator is given ciphertext $\tilde{C} = (\mathsf{F}(sk_2, st_e), st_e)$, if st_e is within the current sliding window.

Optimizations. As noted previously, including the encapsulation index in the ciphertext enables a more efficient construction because the decapsulator does not have to scan through the entire sliding window to see if the ciphertext should be accepted. An even greater optimization can be achieved if the construction is tweaked further. Recall that the construction checks for replays by storing the set of previously decapsulated ciphertexts in the decapsulation state. The memory needed for this check can easily be reduced by only storing decapsulated ciphertexts *within the current sliding window.* This suffices, because ciphertexts outside of the window will anyways be rejected. An even more efficient solution is to simply keep a bit vector which represents the indices within the current window, and to mark a received ciphertext by flipping a bit in the vector. With this method there is no need to store whole ciphertexts and the decapsulation state is brought down to only a few bits of memory.

Chapter 7

Puncturable Symmetric-key KEMs

In the last chapter we discussed ordered S-KEMs and the necessity of state to realistically model network communication and handle the variable reliability of message transmission. The ideas developed there will be of importance later on, when we explore optimizations and efficient constructions. First, however, we present a tool that enables an essential step toward forward-secret zero round-trip time key exchange, namely the concept of *puncturing*. First introduced by Green and Miers [30] as a means of revoking the decryption capability for specific messages in a public-key encryption scheme, the suitability of puncturable schemes for forward-secret 0-RTT key exchange was observed by Günther et al. [32]. It has later been honed further to this purpose by Derler et al. [20] and Aviram, Gellert and Jager [3]. The original idea was in turn based on forward-secret public-key encryption, for which the first construction was proposed by Canetti, Halevi and Katz [13].

Intuitively, puncturing is similar to a limited blackout. The decrypting party (or in our case, decapsulating) forgets some part of the secret used to convert a specific ciphertext into plaintext, thereby losing the capability to decrypt (decapsulate) that ciphertext. Compared to other methods of obtaining forward secrecy, which often depend on an update of the secret key that renders all prior ciphertexts indecipherable, puncturing provides fine-grained revocation of the decryption capability; the ciphertext being punctured on can no longer be deciphered, but all others are unaffected. This is especially suitable for applications such as TLS key exchange, where encapsulated session keys may arrive out of order due to network issues and there is a benefit to removing access to the messages of a session as soon as it is closed, without having to wait for earlier or parallel sessions to finish.

7.1 Syntax

Formally, extending the notion of a stateless symmetric-key KEM to that of a puncturable S-KEM consists of adding a puncturing algorithm, P, by which the secret key can be altered to no longer allow decapsulation of specific ciphertexts. The other algorithms operate the same way as for S-KEM schemes described in Definition 4, except for a slight revision of the encapsulation algorithm to allow that it outputs $\perp$. This is necessary because excessive puncturing could cause a scheme to run out of valid, non-punctured ciphertexts, which the scheme may then choose to indicate by returning $\perp$.

Definition 6. A *puncturable symmetric-key key encapsulation mechanism*, $\mathsf{PS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P})$ is a 4-tuple of algorithms. Associated to PS-KEM are a secret key space $\mathcal{SK}$, a key space $\mathcal{K}$ and a ciphertext space $\mathcal{C}$. The algorithms operate as follows:

- The probabilistic key generation algorithm, KG, takes no input and produces a secret key $sk \in \mathcal{SK}$. We write $sk \leftarrow_{\$} \mathsf{KG}()$.
- Via $(K, C) \leftarrow_{\$} \mathsf{E}(sk)$, the randomized encapsulation algorithm, E, produces a pair (K, C) consisting of a key $K \in \mathcal{K}$ and an associated ciphertext $C \in \mathcal{C}$, or $(\perp, \perp)$ to indicate failure.
- Via $K \leftarrow \mathsf{D}(sk, C)$ the deterministic decapsulation algorithm, D, on input the secret key $sk \in \mathcal{SK}$ and a ciphertext $C \in \mathcal{C}$, returns either a key $K \in \mathcal{K}$ or $\perp$ to indicate failure.
- Via $\hat{sk} \leftarrow_{\$} \mathsf{P}(sk, C)$ a punctured secret key, $\hat{sk} \in \mathcal{SK}$, is produced on input the previous secret key $sk \in \mathcal{SK}$ and a ciphertext-key $C \in \mathcal{C}$. The puncturing algorithm may be randomized.

7.2 Correctness

Correctness of PS-KEMs states that, in addition to the correct decapsulation requirements of standard S-KEMs, puncturing a secret key on a specific ciphertext does not impair the ability to decapsulate any other ciphertext. This means that regardless of the order in which procedures E, D and P are called, previously encapsulated ciphertexts must always be correctly decapsulated if the secret key has not been punctured on that specific ciphertext. Although this consistency requirement is intuitively easy to grasp, it turns out to be difficult to formalize. Prior work on puncturable KEMs has treated this by sticking to informal notions, speaking about "arbitrary interleaved" sequences [20, 32], or only handling certain sub-cases and not specifying what happens outside of those. Although that is of course a possible approach, we choose to instead exercise the machinery developed for ordered S-KEMs and give correctness in the form of a game. In doing so, we avoid vague quantifying statements such as "for any arbitrary interleaved sequences" and instead formalize correctness in a straightforward manner which leaves no risk of uncovered cases.

The correctness game $\mathbf{G}^{\mathrm{correct}}_{\mathsf{PS\text{-}KEM}}$ is given to the left in Figure 7.1. Initially, the game runs the key generation algorithm to produce the original secret key sk which is given to the adversary. The adversary is then allowed to interact with the encapsulation, decapsulation and puncture algorithms via the corresponding oracles, making any number of queries in any order. The game keeps track of the encapsulation history in table $\mathsf{T}[\cdot]$ where it stores key K at location $\mathsf{T}[C]$ when (K, C) is the output from an encapsulation query. On query $\textsc{Punc}(C)$, a special flag ♇ ("pluto") is written to $\mathsf{T}[C]$, erasing a potential key stored there earlier. Correctness is violated (the adversary wins) if it manages to make a query $\textsc{Dec}(C)$ such that ciphertext C has previously been given to the adversary in

```
Game $\mathbf{G}^{\text{correct}}_{\textsf{PS-KEM}}(\mathcal{A})$
1  $sk \leftarrow\$ \textsf{KG}()$
2  $\mathcal{A}^{\textsc{Enc},\textsc{Dec},\textsc{Punc}}(sk)$
3  Return $\textsf{win}$

$\textsc{Enc}()$:
4  $(K, C) \leftarrow\$ \textsf{E}(sk)$
5  If $C = \bot$ return $\bot$
6  If $\textsf{T}[C] \neq \unrhd$ then $\textsf{T}[C] \leftarrow K$
7  Return $(K, C)$

$\textsc{Dec}(C)$:
8  If $C \notin \mathcal{C}$ return $\bot$
9  $K \leftarrow \textsf{D}(sk, C)$
10 If $(\textsf{T}[C] \neq \bot) \wedge (\textsf{T}[C] \neq \unrhd)$:
11     If $K \neq \textsf{T}[C]$ then $\textsf{win} \leftarrow \textsf{true}$
12 Return $K$

$\textsc{Punc}(C)$:
13 If $C \notin \mathcal{C}$ return $\bot$
14 $sk \leftarrow\$ \textsf{P}(sk, C)$
15 $\textsf{T}[C] \leftarrow \unrhd$
16 Return $sk$
```

```
Game $\mathbf{G}^{\text{fs-cpa}}_{b,\textsf{PS-KEM}}(\mathcal{A})$, $\boxed{\mathbf{G}^{\text{fs-cca}}_{b,\textsf{PS-KEM}}(\mathcal{A})}$
1  $sk \leftarrow\$ \textsf{KG}()$
2  $b^* \leftarrow\$ \mathcal{A}^{\textsc{Enc},\boxed{\textsc{Dec},}\textsc{Punc},\textsc{Corrupt}}()$
3  Return $b^*$

$\textsc{Enc}_b()$:
4  If $\textsf{corrupt}$ then return $\bot$
5  $(K_1, C) \leftarrow\$ \textsf{E}(sk)$; $K_0 \leftarrow\$ \mathcal{K}$
6  If $C = \bot$ return $\bot$
7  If $\textsf{T}[C] \neq \unrhd$ then $\textsf{T}[C] \leftarrow K_b$
8  Return $(K_b, C)$

┌─────────────────────────────────────────────
│ $\textsc{Dec}(C)$:
│ 9  $K \leftarrow \textsf{D}(sk, C)$
│ 10 If $(\textsf{T}[C] \neq \bot) \wedge (\textsf{T}[C] \neq \unrhd)$:
│ 11     Return $\bot$
│ 12 Return $K$
└─────────────────────────────────────────────

$\textsc{Punc}(C)$:
13 $sk \leftarrow\$ \textsf{P}(sk, C)$
14 $\textsf{T}[C] \leftarrow \unrhd$

$\textsc{Corrupt}()$:
15 If $(\forall C \in \mathcal{C})$ $\textsf{T}[C] \in \{\bot, \unrhd\}$ then:
16     $\textsf{corrupt} \leftarrow \textsf{true}$
17     Return $sk$
18 Else return $\bot$
```

Figure 7.1: Left: Correctness game for PS-KEMs. **Right:** Games for fs-cpa and fs-cca security of puncturable symmetric-key KEMs. The code in the box is executed in $\mathbf{G}^{\text{fs-cca}}_{b,\textsf{PS-KEM}}$, but not in $\mathbf{G}^{\text{fs-cpa}}_{b,\textsf{PS-KEM}}$.

an $\textsc{Enc}$-query (first condition on line 10), the adversary has not punctured on C (second check on line 10) and the decapsulation algorithm fails to (re-)produce the key which the encapsulation algorithm associated to C (line 11).

7.3 Security

Analogously to S-KEMs we consider two security metrics for PS-KEMs; fs-cpa and fs-cca. The notions are extended from ind-cpa and ind-cca to capture the forward secrecy (fs) we wish to obtain via puncturing. Essentially, forward secrecy demands that puncturing a secret key on a ciphertext removes the capability to decapsulate that specific ciphertext. We formalize this in games $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{PS\text{-}KEM}}$ and $\mathbf{G}^{\text{fs-cca}}_{b,\mathsf{PS\text{-}KEM}}$ to the right in Figure 7.1.

In game $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{PS\text{-}KEM}}$, the adversary is given access to an encapsulation oracle which provides either all real encapsulations (if $b = 1$) or key-ciphertext pairs where the key is drawn uniformly at random independently of the ciphertext (if $b = 0$). The attacker is also given a puncturing oracle, Punc, to which it can submit ciphertexts $C \in \mathcal{C}$ on which it wishes to puncture the secret key stored by the game. When the secret key is punctured, the game's copy of the key is overwritten by the new version, which is then used for all subsequent computations. Finally in the fs-cpa game, the adversary has access to a Corrupt-oracle. A Corrupt()-query gives the adversary a copy of the current secret key, given that it has previously queried Punc(C) on all ciphertexts C which it obtained from prior Enc-queries. However, after querying Corrupt, no more encapsulations are provided to the adversary. Security of a puncturable S-KEM scheme in the fs-cpa sense requires that an attacker cannot distinguish the real from the random world using Enc, Punc and Corrupt.

In game $\mathbf{G}^{\text{fs-cca}}_{b,\mathsf{PS\text{-}KEM}}$, an attacker is given access to all of the above described oracles, together with an additional decapsulation oracle Dec. The Dec-oracle lets the adversary decapsulate ciphertexts of its choice, except those that it has previously obtained from the Enc-oracle and has not punctured on. (As for OS-KEMs, the decapsulation oracle is silenced and returns only $\perp$ when correctness demands a fixed response. Hence the identical conditions on line 10 in $\mathbf{G}^{\text{fs-cca}}_{b,\mathsf{PS\text{-}KEM}}$ and $\mathbf{G}^{\text{correct}}_{\mathsf{PS\text{-}KEM}}(\mathcal{A})$.) Security still demands that an adversary cannot tell the real from the random world. For cxa $\in \{\text{cpa}, \text{cca}\}$ we define the advantage of adversary $\mathcal{A}$ playing game $\mathbf{G}^{\text{fs-cxa}}_{b,\mathsf{PS\text{-}KEM}}$ as

$$\mathbf{Adv}^{\text{cxa}}_{\mathsf{PS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{fs-cxa}}_{1,\mathsf{PS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\text{fs-cxa}}_{0,\mathsf{PS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

As usual, the advantage is the difference in probability that the adversary returns 1 in the real and random worlds.

7.4 Instantiation

GGM PRFs. One way of obtaining a secure PS-KEM scheme is to build it similarly to a Goldreich-Goldwasser-Micali pseudorandom function [28], which we call a *GGM PRF*. A GGM PRF is a pseudorandom function constructed from a length-doubling pseudorandom generator where the expansion of the PRG is used repeatedly in a clever way to obtain a PRF. The use of the GGM PRF construction toward puncturing is inspired by the similar concept of constrained PRFs [11, 12, 37]. Puncturable PRFs are not only useful for key exchange, they also appear for example in the context of indistinguishability obfuscation [10, 34, 45]. In the following we give a high-level description of the GGM construction, enough to let us utilize the idea toward our construction of a secure PS-KEM scheme. For a full definition of GGM PRFs see e.g. [27].

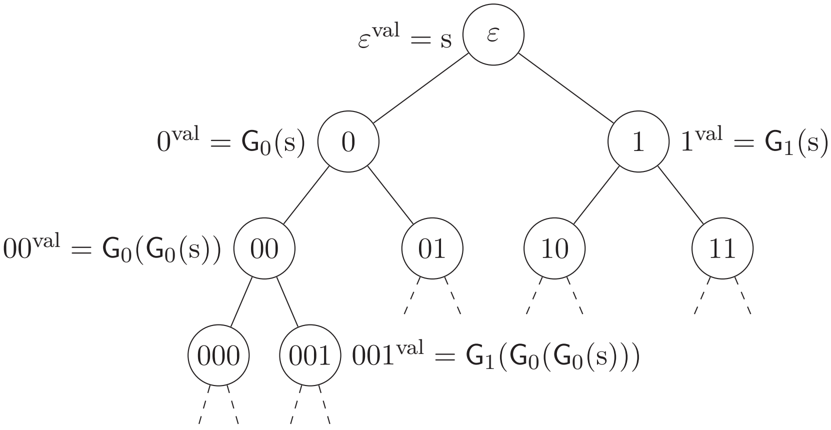
$\varepsilon^{\mathrm{val}} = \mathrm{s}$
ε
$0^{\mathrm{val}} = \mathsf{G}_0(\mathrm{s})$
0
1
$1^{\mathrm{val}} = \mathsf{G}_1(\mathrm{s})$
$00^{\mathrm{val}} = \mathsf{G}_0(\mathsf{G}_0(\mathrm{s}))$
00
01
10
11
000
001
$001^{\mathrm{val}} = \mathsf{G}_1(\mathsf{G}_0(\mathsf{G}_0(\mathrm{s})))$

Figure 7.2: Illustration of a GGM tree $\mathsf{T}_{\mathsf{GGM}}[\mathrm{s}]$ with root value $\mathrm{s} \in \{0,1\}^k$. The values of the nodes in the tree are computed using the pseudorandom generator $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{2k}$. G_0 denotes the left half of the output of G and G_1 the right half. I.e. $\mathsf{G}(x) = \mathsf{G}_0(x) \| \mathsf{G}_1(x)$ for each $x \in \{0,1\}^k$.

The GGM PRF construction works as follows: Let $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{2k}$ be a length-doubling PRG. For $\mathrm{s} \in \{0,1\}^k$ let $\mathsf{G}_0(\mathrm{s})$ be the first k bits of $\mathsf{G}(\mathrm{s})$ and let $\mathsf{G}_1(\mathrm{s})$ be the remaining k bits (the second half of $\mathsf{G}(\mathrm{s})$). That is, $\mathsf{G}(\mathrm{s}) = \mathsf{G}_0(\mathrm{s}) \| \mathsf{G}_1(\mathrm{s})$. For each $\mathrm{s} \in \{0,1\}^k$ we define a binary tree $\mathsf{T}_{\mathsf{GGM}}[\mathrm{s}]$ consisting of nodes which each have a label in $\{0,1\}^*$ and a value in $\{0,1\}^k$. The root is labeled ε and has value s. Its left child has label 0 and value $\mathsf{G}_0(\mathrm{s})$. The right child is labeled 1 and has value $\mathsf{G}_1(\mathrm{s})$. We continue recursively: For each node with label n and value n^{val}, its left child is labeled $n\|0$ and has value $\mathsf{G}_0(n^{\mathrm{val}})$. Its right child has label $n\|1$ and value $\mathsf{G}_1(n^{\mathrm{val}})$. When we write *node* in the following we implicitly refer to the label of the node. For the node value we will explicitly state that it is the value we are referring to. As an example, node 001 in $\mathsf{T}_{\mathsf{GGM}}[\mathrm{s}]$ has value $\mathsf{G}_1(\mathsf{G}_0(\mathsf{G}_0(\mathrm{s})))$. Figure 7.2 illustrates the concept. We call a binary tree constructed in this way a *GGM tree*.

In the GGM construction, PRF $\mathsf{F} : \{0,1\}^k \times \{0,1\}^h \to \{0,1\}^k$ is built by assigning the values of the nodes at depth h in the tree as outputs of F. We use b to denote a single bit, i.e. $b \in \{0,1\}$. For each $\mathrm{s} \in \{0,1\}^k$ and each node $n = b_1 b_2 \ldots b_h \in \{0,1\}^h$, we set $\mathsf{F}(\mathrm{s}, n) = \mathsf{G}_{b_h}(\ldots(\mathsf{G}_{b_2}(\mathsf{G}_{b_1}(\mathrm{s})))\ldots)$. Using Figure 7.2 to illustrate, a PRF $\mathsf{F}_2 : \{0,1\}^k \times \{0,1\}^2 \to \{0,1\}^k$ with labels in $\{0,1\}^2 = \{00, 01, 10, 11\}$ would have the values of the nodes at depth 2 as outputs. I.e. $\mathsf{F}_2(\mathrm{s}, 00) = 00^{\mathrm{val}} = \mathsf{G}_0(\mathsf{G}_0(\mathrm{s}))$, and so on.

This is the idea of a GGM PRF, and it is a well-known fact that F is secure given that G is a secure PRG. (This is easily shown using a hybrid argument, for a full-fledged proof see [27].) An important thing to note is that given the value of any node in the tree (such as the root), one can compute the value of each descendant node (its children and children's children etc.). However, security tells us that without the value of an ancestor of node n, the value of n is indistinguishable from random. Hence the values of the leaves (the nodes of $\mathsf{T}_{\mathsf{GGM}}[\mathrm{s}]$ at depth

h) "look random" to anyone not in possession of the value of the nodes higher up in the tree. We will use this in our PS-KEM construction, by letting the session keys be the values of the leaf nodes and the ciphertexts the corresponding node labels. The secret key will contain some nodes in the tree that allow the values of all non-punctured leaf node labels to be computed. Initially it consists only of the root and hence allows computation of all node values in the tree. Puncturing on a ciphertext will remove all ancestors of the corresponding node from the secret key, hence ensuring that the value (session key) of the node (ciphertext) cannot be computed and appears random. Instead the sibling nodes on the path from the node to the root will be added to ensure that the values of rest of the leaves can still be computed.

To construct our puncturable S-KEM scheme we will use the idea behind a GGM PRF, but not directly the PRF F defined above. Instead we will build the scheme from the pseudorandom generator G, since this allows us to perform intermediate computations necessary for puncturing. The ciphertexts of the scheme will be labels of the leaf nodes in the associated GGM tree. In key generation, a random seed will be drawn to determine the value of the root node, and from that the session keys can be derived as the values of the leaf nodes.

PS-KEM construction. From the length doubling pseudorandom generator $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{2k}$ and integer $h \in \mathbb{N}$ we construct scheme

$$\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h] = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P}).$$

The ciphertexts are the labels of nodes at depth h, which we will refer to as the leaf nodes. Hence $\mathcal{C} = \{0,1\}^h$ and h determines the number of ciphertexts that can be generated (namely 2^h). The keys are leaf node values, which are the left or right output of the PRG, so $\mathcal{K} = \{0,1\}^k$. The secret key is a set of nodes in the GGM tree, and each node has a label in $\{0,1\}^*$ and a value in $\{0,1\}^k$, giving $\mathcal{SK} = \mathcal{P}(\{0,1\}^* \times \{0,1\}^k)$.

The algorithms of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ will be given in pseudocode, and we use an object-oriented notation. The secret key sk, for example, is going to contain pairs of node labels and values, and these will be accessed through method calls similar to those of a hash map. E.g., if $sk = \{(n_1, n_1^{\mathrm{val}}), (n_2, n_2^{\mathrm{val}})\}$, then $sk.\mathsf{Labels} = \{n_1, n_2\}$ and $sk.\mathsf{Values} = \{n_1^{\mathrm{val}}, n_2^{\mathrm{val}}\}$. The predicate $\mathit{Ancestor}(n_1, n_2)$ is true if node n_1 is an ancestor of node n_2, which we define as the label n_1 being a prefix of n_2. (Note that this includes the case $n_1 = n_2$, i.e. a node is an ancestor of itself.) We stress that a property which will always be true for this construction is that for any leaf node in the tree, there exists at most one node in the secret key which is an ancestor of that leaf. Furthermore, a leaf node will have an ancestor in the secret key if and only if the leaf has not been punctured on. The reader is encouraged to verify for themselves that the following algorithms uphold these properties.

Key Generation. The key generation algorithm determines the node values in the GGM tree by assigning the root (which we recall has label ε) a value drawn at random from $\{0,1\}^k$. The algorithm is defined in Figure 7.4.

Encapsulation. The encapsulation algorithm of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ works by drawing a non-punctured leaf node uniformly at random. By the above assumption, the set of non-punctured leaf nodes is precisely the set of ciphertexts/leaf nodes which have an ancestor in the secret key. I.e.

$$\mathcal{C}_{np} = \{C \in \mathcal{C} \mid (\exists n \in sk.\mathsf{Labels})\ \mathit{Ancestor}(n, C)\},$$

where $\mathcal{C}_{np} \subseteq \{0,1\}^h$ denotes the set of non-punctured ciphertexts. The label of the leaf (lets call it $\ell_1\ell_2\ldots\ell_h$) is assigned to C. The value of the leaf node is computed as previously described by repeatedly applying PRG G to the value of the ancestor of C stored in the secret key. This is what subroutine `CompVal` does, on input sk (containing the ancestor node) and the label C. For an ancestor $n_a = \ell_1\ell_2\ldots\ell_{\mathsf{depth}}$ at level depth, it first computes the path $\ell_{\mathsf{depth}+1}\ell_{\mathsf{depth}+2}\ldots\ell_h$ from the ancestor to C and then computes the value of C as $\mathsf{G}_{\ell_h}(\ldots(\mathsf{G}_{\ell_{\mathsf{depth}+2}}(\mathsf{G}_{\ell_{\mathsf{depth}+1}}(n_a^{\mathrm{val}})))\ldots)$, where n_a^{val} is retrieved from sk. The value is assigned to K and the algorithm returns the pair (K, C). If the secret key is empty because all leaf nodes have been punctured on, the encapsulation algorithm returns $\perp$. The algorithm is shown in Figure 7.4.

We remark that it is not strictly necessary to choose C from $\mathcal{C}_{np}$. The restriction to non-punctured ciphertexts is made to increase the functionality of the scheme, but is not required by correctness nor security. In fact, it affects the security of the construction negatively, as the set of possible ciphertexts decreases by each puncturing. However, the effect should be negligible given that the ciphertext space is large enough.

Decapsulation. Decapsulation repeats the second half of encapsulation. I.e., on input secret key sk and ciphertext label C it checks if the secret key contains an ancestor node of C, and if so computes the value of node C. The value is assigned to variable K and returned. Otherwise (if the secret key has been punctured and no longer contains any ancestor of C), decapsulation returns $\perp$. The details are in Figure 7.4.

Puncturing. To define the puncturing algorithm of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ we will need some new notation. Let $\mathsf{T}_{\mathsf{GGM}}[\mathrm{s}]$ be a GGM tree of height h with root value $\mathrm{s} \in \{0,1\}^k$. For any node n at depth depth with label $n = b_1b_2\ldots b_{\mathsf{depth}}$ we define functions which give the label of the parent (direct ancestor) of n and the sibling (other child of node n's parent). For bit b we let $\overline{b}$ denote the complement of b (i.e. $\overline{0} = 1, \overline{1} = 0$).

- `Parent`$(n) = b_1b_2\ldots b_{\mathsf{depth}-1}$ if $\mathsf{depth} \geq 1$, else `Parent`$(n) = \perp$ (if $n = \varepsilon$, i.e. n is the root).
- `Sibling`$(n) = b_1b_2\ldots\overline{b_{\mathsf{depth}}}$.

The puncturing algorithm of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ updates the secret key by both adding and deleting nodes in the GGM tree, making sure that in the end at most one ancestor of each leaf node is in the secret key. (Recall that a node counts as an ancestor of itself). On call $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h].\mathsf{P}(sk, C)$, any ancestor of C is removed from the secret key. This ensures that the value of C (i.e. the corresponding key) can no longer be computed. Instead, the siblings of all nodes on the path from C up to, but excluding, its ancestor in sk are added to the secret key. This ensures that the values of the other leaf nodes in the subtree under the ancestor node can still be computed. The idea is illustrated in Figure 7.3 for a tree of height 3. For an extended example, see Figure A.1 in Appendix A.

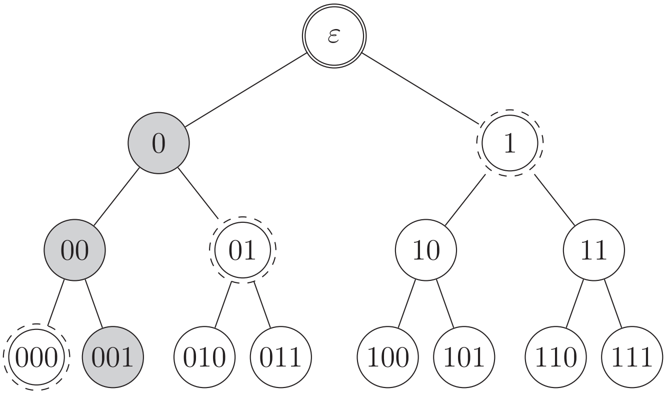

Figure 7.3: GGM tree $\mathsf{T}_{\mathsf{GGM}}[s]$ illustrating the puncturing algorithm of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}]$. The figure shows puncturing on node 001 where the secret key initially consists of the root only (double circle), which is also the ancestor of 001 in sk. Nodes on the path from 001 to its ancestor in the secret key are shaded. The siblings of the shaded nodes are added to the secret key (illustrated by a dashed double circle) and the ancestor (root) is removed.

For brevity we define an "add" operation and a "delete" method on the secret key. The add operation takes as input the label of the new node and its value. The delete operation takes only the label.

```
- $\underline{sk.\mathsf{Add}(n, n^{val})}$
  $sk \leftarrow sk \cup \{(n, n^{\mathrm{val}})\}$
  Return $sk$

- $\underline{sk.\mathsf{Delete}(n)}$
  If $n \notin sk.\mathsf{Labels}$ return $sk$
  $n^{\mathrm{val}} \leftarrow \mathtt{CompVal}(sk, n)$
  $sk \leftarrow sk \setminus \{(n, n^{\mathrm{val}})\}$
  Return $sk$
```

Finally, to define $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h].\mathsf{P}(sk, C)$, where C is the label of a leaf in $\mathsf{T}_{\mathsf{GGM}}$, we specify a recursive subroutine $\mathtt{RecursivePunc}(sk, n)$ which takes as input the secret key sk and a node n. The code of the helper function is given at the bottom of Figure 7.4. The recursive algorithm starts at the node to be punctured, traverses up the tree until it reaches the ancestor which is in sk (if such exists – if not, puncturing does not change the secret key), computes the values of both children and adds them to the secret key. It then steps back, the same path it traversed up, using the value of the current node which was added to sk to compute the values of both children. It adds the children to the secret key and then removes the current node as it steps down again. In the end, the secret key contains the sibling nodes of all nodes on the path from the punctured node up to the ancestor in sk, and the nodes on the path that were temporarily added have been removed. Finally, the node punctured on is itself removed from sk. The code of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h].\mathsf{P}$ is given at the top of Figure 7.4 together with the key generation, encapsulation and decapsulation algorithms described earlier.

```
PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.KG():
  $\mathrm{s} \leftarrow\!\!\$\ \{0,1\}^k$
  $sk \leftarrow \{(\varepsilon, \mathrm{s})\}$
  Return $sk$

PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.E($sk$):
  If $sk = \emptyset$ return $(\bot, \bot)$
  $C \leftarrow\!\!\$\ \mathcal{C}_{np}$
  $K \leftarrow$ CompVal($sk, C$)
  Return $(K, C)$

PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.D($sk, C$):
  If $C \in \mathcal{C}_{np}$ then $K \leftarrow$ CompVal($sk, C$)
  Else $K \leftarrow \bot$
  Return $K$

PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.P($sk, C$):
  If $C \notin \mathcal{C}_{np}$ return $sk$
  $sk \leftarrow$ RecursivePunc($sk, C$)
  $sk$.Delete($C$)
  Return $sk$
```

```
Subroutine RecursivePunc($sk, n$):
  If $n \in sk$.Labels return $sk$   // Base case
  $pn \leftarrow$ Parent($n$)
  $sk \leftarrow$ RecursivePunc($sk, pn$)   // Traverse up
  If $pn \in sk$.Labels then:
     $sn \leftarrow$ Sibling($n$)
     $sn^{\mathrm{val}} \leftarrow$ CompVal($sk, sn$)   // Compute value of sibling node
     $sk$.Add($sn, sn^{\mathrm{val}}$)   // Add sibling to $sk$
     $n^{\mathrm{val}} \leftarrow$ CompVal($sk, n$)   // Compute value of current node
     $sk$.Add($n, n^{val}$)   // Temp. add current node for use in next level
     $sk$.Delete($pn$)   // Delete parent from $sk$
  Return $sk$
```

Figure 7.4: Top: Algorithms of PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$, an instantiation of a puncturable symmetric-key KEM based on pseudorandom generator G.
Bottom: Code of subroutine RecursivePunc, used internally as helper method by procedure PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.P.

Privacy. A full-fledged security proof for $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ could be given using a *complexity leveraging* argument [11, 12], and better bounds could perhaps be attained through other techniques [26], but this is unfortunately outside the scope of this thesis. However, we do provide some intuition as to why the construction achieves fs-cca (and hence, by implication also fs-cpa) privacy. We begin with the weaker notion and build up our reasoning to the stronger. Recall that the goal of an adversary in the fs-cpa game is to distinguish real encapsulated keys from random ones, with the help of a puncturing oracle and potentially by corrupting the punctured secret key. If the construction had been a standard S-KEM scheme, without the ability to puncture, the keys (leaf node values) would have been indistinguishable from random given that the underlying PRG G is secure. The security of such an S-KEM scheme directly reduces to that of the GGM PRF which it would essentially be built of, which in turn reduces to PRG G by a standard result proved originally in [29]. The reduction from S-KEM to any PRF (including a GGM PRF) is shown in Theorem 1, Section 5.3.

Adding puncturing makes the analysis more complex, both because the accessible ciphertext space decreases with each puncture (recall that the encapsulation algorithm in our construction chooses ciphertexts among the non-punctured leaf nodes of the GGM tree) and because we are interested in forward security. The shrinking ciphertext space will not be an issue as long as the original ciphertext space is large compared to the number of punctures. It will show up in the birthday bound part of the reduction in Theorem 1, as the collision probability of future encapsulations increases slightly with each punctured ciphertext. For forward secrecy, the key point is that any leaf node in the GGM tree will have at most one ancestor node in the secret key at any time, and it will have an ancestor if and only if it has not been punctured on. In other words, the key value corresponding to a ciphertext can be computed if and only if that ciphertext has not previously been punctured on. This means that decapsulation will fail on all punctured ciphertexts. But the adversary in the fs-cpa game is only allowed to corrupt the secret key if it has punctured all its challenge encapsulations, and after corruption it is not allowed to query the encapsulation oracle again. Hence it will have no use of the secret key it obtains when corrupting, because any attempts to use it to decapsulate the challenge ciphertexts will fail.

Analogously, the fs-cca security of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ also reduces to the PRF security of the GGM PRF following Theorem 2, Section 5.3. Because of the random choice in the encapsulation algorithm of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$, we expect the probability that an adversary can use the decapsulation oracle to obtain an honest decapsulation of a future challenge ciphertext to be small if the ciphertext space is large. This is also covered by a term in the bound in Theorem 2. Note, however, that a deterministic choice of ciphertext in the encapsulation algorithm would have rendered the scheme insecure against chosen-ciphertext attacks. If the choice was not random, an attacker could use the knowledge of which ciphertexts the encapsulation oracle will give it as challenge in future queries and decapsulate them ahead of time.

Optimizations. An issue associated with using the GGM PRF construction for our PS-KEM scheme is the storage needed for the secret key. In contrast to the non-puncturable symmetric KEM schemes discussed in previous sections, the secret key of a PS-KEM scheme is not a fixed length binary string. Instead it is an evolving object that changes with each puncturing. In the GGM PRF instantiation shown above, the secret key starts small, containing only the root node of the GGM tree, but grows when a ciphertext is punctured and more nodes are added. In the worst case, the secret key grows (roughly) linearly with the number of punctured ciphertexts. Not all ciphertexts cause the same size increase when punctured though. As shown in Figure 7.3, the first puncturing will add $h-1$ nodes to the secret key, where h is the height of the tree. Each such node forms the root of a subtree of the whole GGM tree. If the second node punctured is in the largest such subtree (e.g. node 111 in the example), another $h-2$ nodes are added to the secret key. However, if the next node punctured is the sibling of the previous one (node 000 in the illustrated example), the secret key actually decreases in size, as the punctured node is simply removed. What this illustrates is that the issue of the growing secret key can be combated by puncturing ciphertexts in a certain order. We will call this a clever *puncturing pattern*.

To enable a clever puncturing pattern, we will focus on controlling the order in which ciphertexts are encapsulated. The reason for this approach is that in the intended applications of PS-KEM schemes, ciphertexts would be punctured only after they have been encapsulated. In TLS key exchange for example, we imagine that the client performs an encapsulation before initializing a resumed session with the server, who then decapsulates and punctures on the decapsulated ciphertext. Hence by regulating the encapsulation order, we indirectly control the puncturing pattern. Ideally, we would like the encapsulator in the GGM PRF PS-KEM instantiation to output ciphertexts in perfect order, starting at the leftmost leaf of the GGM tree and moving right one step at a time. Puncturing in that order would "prune" the tree from the left and keep the number of nodes in secret key at less than or equal to the height h of the tree. Unfortunately, that is not the case in the construction described in Figure 7.4. There the encapsulator picks a non-punctured ciphertext at random (to achieve fs-cca security), so there is no controlling the order. In fact, initially it is more likely that a "bad" ciphertext is chosen (in the sense that puncturing it would cause a large size increase of the secret key), because the large subtrees contain more non-punctured ciphertexts than the smaller subtrees. In order to enforce ordering of encapsulations, the scheme would need to be stateful. Toward this, we introduce puncturable ordered symmetric-key KEMs in the next chapter.

Chapter 8

Punctuable Ordered Symmetric-key KEMs

We have previously defined ordered symmetric-key KEMs, where the encapsulator and decapsulator are stateful, allowing us to specify more diverse correctness requirements than for stateless schemes. The varying correctness classes of OS-KEMs relate to the expected reliability of the network across which the key exchange happens, and enable optimizations that leverage the ordering as well as more lenient functionality demands. We also defined puncturable symmetric KEMs in the previous chapter, which give us the desired forward-secrecy property. It is now time to combine the two into what we call puncturable ordered symmetric-key KEMs, or POS-KEMs for short.

8.1 Syntax

Definition 7. A *puncturable ordered symmetric-key key encapsulation mechanism* scheme $\mathsf{POS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P})$ is a 4-tuple of algorithms. Algorithms KG, E and D operate as previously described for OS-KEMs (Definition 5), except that procedure E is modified to allow returning $\perp$ (used to indicate that there are no non-punctured ciphertexts left in the ciphertext space). Procedure P operates as defined for PS-KEMs (Definition 6). Associated to the scheme are secret key space $\mathcal{SK}$, key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$ as well as a set of encapsulation states, $\mathcal{S}_e$, and decapsulation states, $\mathcal{S}_d$.

Algorithmically we write

- $(sk, st_e, st_d) \leftarrow\$\ \mathsf{KG}()$, where $sk \in \mathcal{SK}$, $st_e \in \mathcal{S}_e$ and $st_d \in \mathcal{S}_d$,
- $(K, C, \hat{st}_e) \leftarrow\$\ \mathsf{E}(sk, st_e)$, for $sk \in \mathcal{SK}$, $K \in \mathcal{K} \cup \{\perp\}$, $C \in \mathcal{C} \cup \{\perp\}$ and $st_e, \hat{st}_e \in \mathcal{S}_e$,
- $(K, \hat{st}_d) \leftarrow \mathsf{D}(sk, st_d, C)$, where $K \in \mathcal{K} \cup \{\perp\}$ and $C \in \mathcal{C}$, $st_d, \hat{st}_d \in \mathcal{S}_d$,
- $\hat{sk} \leftarrow\$\ \mathsf{P}(sk, C)$ for $sk, \hat{sk} \in \mathcal{SK}$, $C \in \mathcal{C}$.

8.2 Correctness

To specify correctness of a POS-KEM scheme we define three correctness classes, *no ordering* (nO), *weak ordering* ($\mathrm{wO}_{\mathrm{w_f},\mathrm{w_b}}$) and *perfect ordering* (pO). Similar to the correctness levels introduced for OS-KEMs they capture the consistency we expect from a scheme under varying degrees of (un)reliable transmission. The weak ordering class wO_{w_f,w_b} has two parameters: $w_f \geq 1$ (the *forward window*) and $w_b \geq 0$ (the *backward window*) and in fact specifies many levels, one for each pair of parameter values. Like for OS-KEMs, we imagine that some party, Alice, has performed a sequence of n encapsulations $(K_1, C_1), \ldots (K_n, C_n)$. A receiver, Bob, obtains the encapsulated ciphertexts in some possibly different order $C_{i_1}, \ldots C_{i_m}$ (for $i_1, \ldots, i_m \in \{1, \ldots, n\}$), where some ciphertexts might also be repeated or have been lost in transmission. In correctness under *no ordering*, we expect the scheme to correctly decapsulate all previously encapsulated ciphertexts that have not been punctured on, regardless of potential reordering, replays or omissions. Correctness under *weak ordering* requires correct decapsulation on non-punctured ciphertexts only under certain bounds on the reordering and number of dropped ciphertexts, and disallows replays. In *perfect ordering*, only non-punctured ciphertexts that arrive in the same order they were encapsulated in are required to be correctly handled.

As discussed previously in the context of OS-KEMs, for stateful schemes there is a possibility to allow that ciphertexts may be "re-used" across multiple keys, i.e. each ciphertext may appear as output from encapsulation several times, coupled with different keys. (This is not the case in the stateless setting, because correctness demands that a unique key can be produced by decapsulation of each ciphertext.) However, especially in the case of weak ordering, analysis is simplified by a restriction to schemes with an injective function from keys to ciphertexts. In the setting of weakly ordered OS-KEMs, we made the stricter requirement that correct encapsulation outputs unique ciphertexts overall. Because puncturing is an operation on a specific ciphertext, there are further advantages to having unique ciphertexts in the setting of POS-KEMs, even outside of weak ordering. Hence we require that for $\mathsf{POS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P})$ to be *correct*, for any secret key $sk \in [\mathsf{KG}()]$ and any two encapsulations states $st_e, st'_e \in \mathcal{S}_e$, it must hold that if $st_e \neq st'_e$, then $\Pr[\, C \neq C' \,] = 1$ where $(K, C, \hat{st}_e) \leftarrow\!\!\$\ \mathsf{E}(sk, st_e)$ and $(K', C', \hat{st}'_e) \leftarrow\!\!\$\ \mathsf{E}(sk, st'_e)$. The probability is over the coins of E. We can phrase this mathematically as a class of schemes Corr_e which we call *encapsulation-correct.* If we denote the second output of E (i.e. the ciphertexts) by $\mathsf{E}(\cdot, \cdot)[2]$, and recall that for any procedure Proc, the expression $[\mathsf{Proc}(x_1, \ldots, x_n)]$ denotes the set of all possible outputs of Proc when run on inputs $x_1, \ldots, x_n$ over all coins, then the class is defined by

$$\begin{aligned}\mathrm{Corr}_e = \{&\mathsf{POS\text{-}KEM} = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P}) : \\ &(\forall sk \in [\mathsf{KG}()])\,(\forall st_e, st'_e)\,\big(\forall C \in \big[\mathsf{E}(sk, st_e)[2]\big]\big)\,\big(\forall C' \in \big[\mathsf{E}(sk, st'_e)[2]\big]\big) \\ &(st_e \neq st'_e) \implies (C \neq C')\}.\end{aligned}$$

In addition to unique encapsulations, scheme functionality places demands on decapsulation. As for OS-KEMs an PS-KEMs, we formalize this part of correctness via games. For each class $\mathrm{xO} \in \{\mathrm{nO}, \mathrm{wO}_{w_f,w_b}, \mathrm{pO}\}$ we define a predicate called *Supported* which, on input a ciphertext and some game variables, determines if the ciphertext is "supported" by the correctness class given the game history. A ciphertext is supported if it has been previously output by encapsulation, has not been punctured on and is received by the decapsulator in the order required by the correctness class. A scheme $\mathsf{POS\text{-}KEM}$ is called xO-correct if it correctly decapsulates all supported ciphertexts in class xO, and is in the class Corr_e of encapsulation-correct schemes.

The correctness games $\mathbf{G}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}$ for POS-KEMs are given on the left in Figure 8.1 and take the correctness class as parameter. Initially, the games run the key generation algorithm to produce the original secret key *sk* which is given to the adversary. The adversary is then allowed to interact with the encapsulation, decapsulation and puncture algorithms via the corresponding oracles, making any number of queries in any order. The game keeps a *synchronization* flag sync, which is used by the decapsulation oracle Dec to ensure that all previous queries have been supported. It is initialized to true at the start of the game, and remains so as long as all queries to the Dec-oracle are supported. For standard correctness, a scheme only needs to handle ciphertext sequences that do not trigger the sync-flag. The games keep tables $\mathsf{T}[\cdot]$ and $\mathrm{I}[\cdot]$ which we assume are accessible to predicates and subroutines in the code of the game (such as *Supported*). As usual we define the correctness advantage for adversary $\mathcal{A}$ to be

$$\mathbf{Adv}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A}) \Rightarrow \mathsf{true}\,\right].$$

and say that scheme $\mathsf{POS\text{-}KEM}$ is in correctness class xO (or "xO-*correct*") if $\mathsf{POS\text{-}KEM} \in \mathrm{Corr}_e$ and $\mathbf{Adv}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A}) = 0$ for all—even of unbounded time and resources—adversaries $\mathcal{A}$.

The support predicates for each correctness class are given in Figure 8.2. We examine each one, starting with the predicate for no ordering.

No ordering support predicate.

$$\mathit{Supported}^{\mathrm{nO}}(C, \mathcal{S}, n_d) = (\mathsf{T}[C] \neq \bot) \wedge (\mathsf{T}[C] \neq \unrhd).$$

The predicate $\mathit{Supported}^{\mathrm{nO}}$ captures that ciphertext C is supported if it has been previously produced by encapsulation ($\mathsf{T}[C] \neq \bot$) and it has not been punctured on ($\mathsf{T}[C] \neq \unrhd$). Recall that all table entries $\mathsf{T}[\cdot]$ are initialized to $\bot$ and that key K is stored in $\mathsf{T}[C]$ when K and C are produced by the encapsulation algorithm in an Enc-query. The flag $\unrhd$ is written to $\mathsf{T}[C]$ when C is the input to a Punc-query. The puncturing flag is never overwritten, so if puncturing on C has occurred prior to encapsulation of C, then $\mathsf{T}[C]$ will still contain $\unrhd$ after the encapsulation. This captures that there are no correctness requirements on the decapsulator when it receives a punctured ciphertext, independently of whether that ciphertext was encapsulated before or after puncturing (or not at all). We continue with the predicate for perfect ordering, which has one additional condition.

Perfect ordering support predicate.

$$\mathit{Supported}^{\mathrm{pO}}(C, \mathcal{S}, n_d) = (\mathsf{T}[C] \neq \bot) \wedge (\mathsf{T}[C] \neq \unrhd) \wedge (n_d = \mathrm{I}[C]).$$

As before, the first two checks ensure that the ciphertext has been previously output by the encapsulator and that it has not been punctured on. In addition

```
Game $\mathbf{G}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A})$
    $n_e \leftarrow 0$; $n_d \leftarrow 0$; $\mathcal{S} \leftarrow \emptyset$
    $\mathsf{sync} \leftarrow \mathsf{true}$
    $(sk, st_e, st_d) \leftarrow\!\!\$\ \mathsf{KG}()$
    $\mathcal{A}^{\textsc{Enc},\textsc{Dec},\textsc{Punc}}(sk, st_e, st_d)$
    Return $\mathsf{win}$

$\textsc{Enc}()$:
    $(K, C, st_e) \leftarrow\!\!\$\ \mathsf{E}(sk, st_e)$
    If $C = \bot$ return $\bot$
    $\mathrm{I}[C] \leftarrow n_e$
    If $\mathsf{T}[C] \neq ⊵$ then $\mathsf{T}[C] \leftarrow K$
    $n_e \leftarrow n_e + 1$
    Return $(K, C, st_e)$

$\textsc{Dec}(C)$:
    If $C \notin \mathcal{C}$ return $\bot$
    $(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
    If $\mathit{Supported}^{\text{xO}}(C, \mathcal{S}, n_d)$
    and $\mathsf{sync} = \mathsf{true}$ then:
        $n_d \leftarrow \mathit{Update}(n_d)$
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
        If $K \neq \mathsf{T}[C]$ then $\mathsf{win} \leftarrow \mathsf{true}$
    Else $\mathsf{sync} \leftarrow \mathsf{false}$
    Return $(K, st_d)$

$\textsc{Punc}(C)$:
    If $C \notin \mathcal{C}$ return $\bot$
    $sk \leftarrow\!\!\$\ \mathsf{P}(sk, C)$
    $\mathsf{T}[C] \leftarrow ⊵$
    Return $sk$
```

```
Game $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{POS\text{-}KEM}}(\mathcal{A})$, $\boxed{\mathbf{G}^{\text{xO-fs-cca}}_{b,\mathsf{POS\text{-}KEM}}(\mathcal{A})}$
    $\boxed{n_e \leftarrow 0;\ n_d \leftarrow 0;\ \mathcal{S} \leftarrow \emptyset}$
    $(sk, st_e, st_d) \leftarrow\!\!\$\ \mathsf{KG}()$
    $b^* \leftarrow\!\!\$\ \mathcal{A}^{\textsc{Enc}_b,\textsc{Dec},\textsc{Punc},\textsc{Corrupt}}()$
    Return $b^*$

$\textsc{Enc}_b()$:
    If $\mathsf{corrupt}$ then return $\bot$
    $(K_1, C, st_e) \leftarrow\!\!\$\ \mathsf{E}(sk, st_e)$; $K_0 \leftarrow\!\!\$\ \mathcal{K}$
    If $C = \bot$ return $\bot$
    $\boxed{\mathrm{I}[C] \leftarrow n_e}$
    If $\mathsf{T}[C] \neq ⊵$ then $\mathsf{T}[C] \leftarrow K_b$
    $\boxed{n_e \leftarrow n_e + 1}$
    Return $(K_b, C)$

┌─────────────────────────────────────────────
│$\textsc{Dec}(C)$:
│    $(K, st_d) \leftarrow \mathsf{D}(sk, st_d, C)$
│    If $\mathit{Supported}^{\text{xO}}(C, \mathcal{S}, n_d)$
│    and $\mathsf{sync} = \mathsf{true}$ then:
│        $n_d \leftarrow \mathit{Update}(n_d)$
│        $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
│        Return $\bot$
│    Else $\mathsf{sync} \leftarrow \mathsf{false}$
│    Return $K$
└─────────────────────────────────────────────

$\textsc{Punc}(C)$:
    $sk \leftarrow\!\!\$\ \mathsf{P}(sk, C)$
    $\mathsf{T}[C] \leftarrow ⊵$

$\textsc{Corrupt}()$:
    If $(\forall C \in \mathcal{C})\ \mathsf{T}[C] \in \{\bot, ⊵\}$ then:
        $\mathsf{corrupt} \leftarrow \mathsf{true}$
        Return $sk$
    Else return $\bot$
```

Figure 8.1: Left: Game defining xO-correctness of POS-KEMs for $\text{xO} \in \{\text{nO}, \text{wO}, \text{pO}\}$. **Right:** Games formalizing fs-cpa and fs-cca security for POS-KEM schemes under xO-correctness. The code in boxes is executed in $\mathbf{G}^{\text{xO-fs-cca}}_{b,\mathsf{POS\text{-}KEM}}(\mathcal{A})$, but not in $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{POS\text{-}KEM}}(\mathcal{A})$. If $\text{xO} = \text{wO}$ (weak ordering) the games are parameterized by $w_b \geq 0$ and $w_f \geq 1$. Predicate $\mathit{Supported}^{\text{xO}}$ is used by the decapsulation oracle to check if the queried ciphertext is supported. Subroutine $\mathit{Update}^{\text{xO}}$ increments the decapsulation index as specified for each correctness class.

Correctness class	Predicate $\mathit{Supported}^{\mathrm{xO}}(C, \mathcal{S}, n_d) =$
nO	$(\mathsf{T}[C] \neq \perp) \wedge (\mathsf{T}[C] \neq \underline{\mathsf{P}})$
wO_{w_f,w_b}	$(\mathsf{T}[C] \neq \perp) \wedge (\mathsf{T}[C] \neq \underline{\mathsf{P}}) \wedge (C \notin \mathcal{S})$ $\wedge\, (n_d - w_b \leq \mathrm{I}[C] \leq n_d + w_f)$
pO	$(\mathsf{T}[C] \neq \perp) \wedge (\mathsf{T}[C] \neq \underline{\mathsf{P}}) \wedge (n_d = \mathrm{I}[C])$

Correctness class	Subroutine $\mathit{Update}^{\mathrm{xO}}(n_d, C) =$
nO	n_d
wO_{w_f,w_b}	$\max(n_d, \mathrm{I}[C])$
pO	$n_d + 1$

Figure 8.2: Top: Support predicates for the correctness classes *no ordering* (nO), *weak ordering* (wO_{w_f,w_b}) and *perfect ordering* (pO) of POS-KEMs. The predicates are used in the correctness and security games in Figure 8.1. Arguments $C, \mathcal{S}$ and n_d are game variables. C is a ciphertext, n_d is the current decapsulation index and $\mathcal{S}$ the set of previously received and accepted ciphertexts. **Bottom:** Update function for the decapsulation index n_d for correctness classes $\mathrm{nO}, \mathrm{wO}_{w_f,w_b}$ and pO.
$\mathsf{T}[\cdot]$ and $\mathrm{I}[\cdot]$ are tables kept by the game and accessible by subroutines.

to this, $\mathit{Supported}^{\mathrm{pO}}$ contains the condition $n_d = \mathrm{I}[C]$, which checks that the encapsulation index of C stored in $\mathrm{I}[C]$ matches the current decapsulation index. This ensures that only perfectly ordered ciphertext sequences are supported, given that the decapsulation index is incremented by 1 for each accepted ciphertext. Although this is natural to expect, it will not be the case for correctness under weak ordering, so in order to unify the games we also include a subroutine called $\mathit{Update}^{\mathrm{xO}}(n_d, C)$ which updates the decapsulation index according to the needs of the correctness class. $\mathit{Update}^{\mathrm{xO}}$ is defined in Figure 8.2. Next we take a closer look at weak ordering.

Weak ordering support predicate. In addition to only allowing previously encapsulated and non-punctured ciphertexts, this class only supports ciphertexts that arrive within a sliding window determined by parameters $w_b \geq 0$ (the *backward window*) and $w_f \geq 1$ (the *forward window*). The window is updated each time an accepted ciphertext with a higher encapsulation index than the previously highest one is received. It then slides forward to this index, from which it extends backwards w_b positions and forward w_f positions. The idea is the same as in the weak ordering correctness class of OS-KEMs. See Section 6.2 for more details and Figure 6.1 for an illustration.

The support predicate for weak ordering is

$$\mathit{Supported}^{\mathrm{wO}_{w_f,w_b}}(C, \mathcal{S}, n_d) =$$
$$(\mathsf{T}[C] \neq \bot) \wedge (\mathsf{T}[C] \neq \underline{\mathsf{P}}) \wedge (C \notin \mathcal{S}) \wedge (n_d - w_b \leq \mathrm{I}[C] \leq n_d + w_f).$$

Here, set $\mathcal{S}$ includes all previously accepted ciphertexts, and is used to check for replays. The last condition ensures that ciphertext C is within the current window, which covers all encapsulation indices from $n_d - w_b$ to $n_d + w_f$. The decapsulation index is updated upon receipt of a supported ciphertext according to $\mathit{Update}^{\mathrm{wO}_{w_f,w_b}}(n_d, C) = \max(n_d, \mathrm{I}[C])$, where $\mathrm{I}[C]$ contains the encapsulation index of C.

8.3 Robustness

As for OS-KEMs, the correctness games $\mathbf{G}^{\text{xO-correct}}_{\mathsf{POS\text{-}KEM}}$ in Figure 8.1 keep a synchronization flag sync which is set to false if an unsupported ciphertext is received by the decapsulator. After that, there is no longer any correctness requirements on $\mathsf{POS\text{-}KEM}$. If one wants a robust version of the correctness game, this can be achieved by simply removing the code on line 18 (line 17 in the corresponding security games) where sync is set to false. Without the synchronization flag, the scheme is required to recover from unsupported ciphertexts and correctly decapsulate subsequent supported ciphertexts.

Robustness is useful for example in applications that use time-outs, i.e. where the secret key is punctured on all ciphertexts in an interval after a certain time limit, regardless of whether those ciphertexts have arrived to the decapsulator. Such constructions can be used to provide a coarse forward-secrecy guarantee, and also to "purge" the secret key and reduce its size. However, if a ciphertext from an earlier interval arrives after the time-out (when the secret key has already been punctured on the ciphertext), then it is reasonable to expect that the scheme should still keep functioning. But to correctly handle non-punctured ciphertexts belonging to the current time interval that arrive after a punctured ciphertext is outside the scope of standard correctness, hence the need for robustness.

8.4 Security

The security goal we are interested in for POS-KEMs is privacy with forward secrecy. This has been introduced and discussed in previous chapters, and the security notions we present here are simply adaptations of those treated earlier. We give two games, $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{POS\text{-}KEM}}$ and $\mathbf{G}^{\text{xO-fs-cca}}_{b,\mathsf{POS\text{-}KEM}}$, to the right in Figure 8.1. Both games combine privacy goals with forward secrecy. In $\mathbf{G}^{\text{fs-cpa}}_{b,\mathsf{POS\text{-}KEM}}$ the privacy notion is indistinguishability under chosen-plaintext attack, which in the POS-KEM setting means that the adversary has access to challenge encapsulations (either real and honest, or random ones), and is given the task of distinguishing the real case from the random one. To cover forward secrecy as well, the adversary is additionally bestowed the ability to puncture the secret key on ciphertexts of its choice, and can then ask to have the secret key revealed if it has punctured on all challenge ciphertexts that it has been given by the encapsulator. This is handled by the Corrupt-oracle, and sets the $\mathsf{corrupt}$-flag of the game to true. After the secret key has been revealed ("corrupted"), no more encapsulations will be given to the adversary. The scheme is forward-secret under chosen-plaintext attack if real encapsulations are indistinguishable from random ones even after the secret key has been corrupted. We define the fs-cpa-advantage of adversary $\mathcal{A}$ playing the fs-cpa-game with POS-KEM scheme $\mathsf{POS\text{-}KEM}$ to be

$$\mathbf{Adv}^{\text{fs-cpa}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{fs-cpa}}_{1,\mathsf{POS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\text{fs-cpa}}_{0,\mathsf{POS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

Game $\mathbf{G}^{\text{xO-fs-cca}}_{b,\mathsf{POS\text{-}KEM}}$ captures the stronger notion of indistinguishability under chosen-ciphertext attack by additionally giving the adversary access to honest decapsulations, except on ciphertexts which are supported by correctness and hence would give the adversary a trivial win. (See the discussion in Section 6.4.) Here too the adversary has the ability to puncture on ciphertexts of its choosing and to corrupt the secret key if there are no non-punctured challenge ciphertexts. Once again the goal for the adversary is to distinguish the real encapsulation world from the random one with the help of its Dec-, Punc- and Corrupt-oracles. Because the three POS-KEM correctness classes have different ciphertext supports, there is one fs-cca-security game for each class. We use the abbreviation nO to denote *no ordering*, wO_{w_f,w_b} for *weak ordering* with window parameters w_f and w_b and lastly pO for *perfect ordering*. For $\text{xO} \in \{\text{nO}, \text{wO}_{w_f,w_b}, \text{pO}\}$ we define the xO-fs-cca-advantage of an adversary $\mathcal{A}$ by

$$\mathbf{Adv}^{\text{xO-fs-cca}}_{\mathsf{POS\text{-}KEM}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}^{\text{xO-fs-cca}}_{1,\mathsf{POS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right] - \Pr\left[\,\mathbf{G}^{\text{xO-fs-cca}}_{0,\mathsf{POS\text{-}KEM}}(\mathcal{A}) \Rightarrow 1\,\right].$$

8.5 Instantiations

A simple way to obtain a construction of a POS-KEM scheme is to modify the syntax of the PS-KEM GGM instantiation in Section 7.4 so that it includes state, but does not use it. This gives a correct POS-KEM scheme under no ordering. Since the construction is completely trivial, we omit the details here. Of higher interest is a perfectly ordered POS-KEM scheme built from the PS-KEM GGM construction, where state is used to ensure that ciphertexts are produced in perfect order (traversing the leaves from left to right in the GGM tree), as discussed under Optimizations in Section 7.4.

Given PS-KEM scheme $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$ built from pseudorandom generator $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{2k}$ and integer h, we construct $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}] = (\mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{P})$. The algorithms are given in Figure 8.3. Key generation produces a secret key using the key generation procedure of $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}$, but also initializes a binary string which will work as a counter for the encapsulator and decapsulator. The string is initially 0^h (i.e. the label of the leftmost leaf in the underlying GGM tree) and is stored in the encapsulation state st_e and the decapsulation state st_d. The latter additionally includes the boolean synchronization flag sync, which is initially true. The encapsulation procedure begins by checking that st_e has not reached 10^h, i.e. that there are still unused leaf labels left in the tree. If so, it sets the current encapsulation state as ciphertext and computes the corresponding

```
POS-KEM[PS-KEM_GGM].KG():
 1 $sk \leftarrow_{\$}$ PS-KEM_GGM.KG()
 2 $st_e \leftarrow 0^h$
 3 $n_d \leftarrow 0^h$; sync ← true
 4 $st_d \leftarrow (n_d, \mathsf{sync})$
 5 Return $(sk, n_e, n_d)$

POS-KEM[PS-KEM_GGM].E($sk, st_e$):
 6 If $st_e = 10^h$ return $(\bot, \bot, st_e)$
 7 $C \leftarrow st_e$
 8 $K \leftarrow \mathtt{CompVal}(sk, C)$
 9 $st_e \leftarrow st_e + 1$
10 Return $(K, C, st_e)$

POS-KEM[PS-KEM_GGM].D($sk, st_d, C$):
11 $(n_d, \mathsf{sync}) \leftarrow st_d$
12 If $C = n_d$ and sync = true then:
13     $K \leftarrow$ PS-KEM_GGM.D($sk, C$)
14     $n_d \leftarrow n_d + 1$
15 Else $K \leftarrow \bot$; sync ← false
16 $st_d \leftarrow (n_d, \mathsf{sync})$
17 Return $(K, st_d)$

POS-KEM[PS-KEM_GGM].P($sk, C$):
18 PS-KEM_GGM.P($sk, C$)
19 Return $sk$
```

Figure 8.3: Top: Algorithms of POS-KEM[PS-KEM$_{\mathsf{GGM}}$], an instantiation of a puncturable ordered symmetric-key KEM based on PS-KEM scheme PS-KEM$_{\mathsf{GGM}}$. When strings $st_e \in \{0,1\}^h$ and $n_d \in \{0,1\}^h$ are incremented (e.g. on line 9), the binary value is interpreted as an integer, incremented and then converted back to a string.

key (value of the node with label C in the GGM tree) using subroutine CompVal described in Section 7.4. If all leaves have already been used ($st_e = 10^h$), procedure E returns $\bot$. The decapsulation algorithm, receiving ciphertext C, checks that C is the expected leaf node (next in order) and that the sync flag is still true. If so, it computes the key corresponding to C using PS-KEM$_{\mathsf{GGM}}$.D. If not, it sets sync to false and returns $\bot$ as the key. The puncturing procedure is identical to that of the underlying scheme PS-KEM$_{\mathsf{GGM}}$, see Section 7.4.

The benefits of POS-KEM[PS-KEM$_{\mathsf{GGM}}$] in comparison to PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$ lie in the possibility to leverage the perfect ordering of ciphertexts toward a leaner construction (memory-wise) when implementing the scheme. If used in an application with two parties, e.g. a client and a server, where the client acts as encapsulator and the server as decapsulator, the advantages are attained if the decapsulating party is implemented to puncture a ciphertext directly after receiving and decapsulating it. The secret key (which we recall consists of nodes in the GGM tree) then contains at most h nodes, where h is the height of the tree (and length of leaf node labels/ciphertexts). This allows less storage to be reserved for the secret key compared to an application with an arbitrary puncturing pattern.

In network applications such as TLS, it might be unrealistic to expect that the ciphertexts sent from client to server arrive in perfect order, which would cause trouble for the construction described above. (Remember that once the decapsulator receives an out-of-order ciphertext, the synchronization flag is triggered and all future decapsulations only return $\bot$.) A better solution is to use a POS-KEM scheme for weak ordering, where ciphertexts are allowed to be locally re-ordered and potentially dropped if lost in transmission. We will not give the full details of such a scheme, but it is essentially equivalent to $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}]$, except for slight modifications of procedure D and the decapsulation state. The most significant difference is that the decapsulation state now keeps track of a window in which ciphertexts are expected and allowed to arrive, instead of just the label of the next leaf node in the tree. The window extends forward and backward from the most recently encapsulated ciphertext that has been received and accepted, and is updated when a new ciphertext with a higher[1] label is received. See the instantiation of a weakly ordered OS-KEM scheme in Section 6.5.3 for details on this part of the construction.

Privacy. We expect both $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}]$ given in Figure 8.3 and a weak ordering (wO) version of it to be fs-cpa secure, on similar grounds as for $\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}[\mathsf{G}, h]$. A full security proof is outside the scope of this thesis, and we refer the reader to the discussion in Section 7.4 for an outline of the intuitive reasoning. Neither $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}]$ nor the wO-version described will be fs-cca secure, however, as the following attack on $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}]$ shows: An adversary $\mathcal{A}$ in game $\mathbf{G}^{\text{fs-cca}}_{b,\mathsf{POS\text{-}KEM}}$ submits $C = 0^h$ to oracle DEC. Because C is not a challenge ciphertext (it has not been previously given to the adversary by oracle ENC), the query is not silenced by the game. C is the ciphertext expected by the decapsulator, so the corresponding key K is computed and returned to $\mathcal{A}$. The adversary then proceeds to make an ENC-query. Since the encapsulation index is not affected by the DEC-query, $\mathcal{A}$ is given (C, K_b), where $C = 0^h$ and K_b is either the real key K or a random string. Adversary $\mathcal{A}$ checks if $K_b = K$; if true, it halts and returns 1, if false it halts and returns 0. This strategy gives $\mathcal{A}$ an fs-cca advantage of 1, hence $\mathsf{POS\text{-}KEM}[\mathsf{PS\text{-}KEM}_{\mathsf{GGM}}]$ is not secure against chosen-ciphertext attacks. The same attack works in the wO-version of the scheme.

The issue with the proposed instantiations is that ciphertexts are not authenticated by the decapsulator. Since the functionality of the scheme is public knowledge and the labels of the GGM tree are also known (the strings in $\{0,1\}^h$, in order), it is easy for an attacker to forge the next expected ciphertext. To combat this, the construction could be modified to also achieve integrity of ciphertexts (Section 5.4), for example by using a so called *message authentication code* or in a manner similar to that discussed for OS-KEM instantiations in Sections 6.5.2 and 6.5.3. Constructing an fs-cca secure POS-KEM scheme will be the subject of future work.

[1] For such comparisons the string is interpreted as an integer via the natural binary encoding.

Chapter 9

Conclusion

The necessity of secure Internet connections today is beyond question. At the same time, the demand for smooth and fast browsing is higher than ever, and performance enhancements are under constant development. Recently, the newest version of the cryptographic network protocol TLS (version 1.3) was released. Among the novelties was a pre-shared key (PSK) zero round-trip time (0-RTT) mode for resumed TLS sessions, in which the need for an interactive setup-phase is removed and private payload data can be sent in 0-RTT after the connection has been initiated. This leads to a significant speed-improvement, which is especially noticeable in cellular networks that have a high inherent latency.

Symmetric-key KEMs. In this work we have introduced a new cryptographic primitive which we call *symmetric-key key encapsulation mechanisms* (S-KEMs). S-KEMs make it possible to conceptually capture the 0-RTT key derivation step in for example PSK 0-RTT mode of TLS 1.3. In addition to formalizing symmetric-key KEMs, we have defined privacy by the notions of indistinguishability under chosen-plaintext and chosen-ciphertext attack, and have given a construction (modeling a real handshake) that we show fulfills both. However, the deployed PSK 0-RTT mode of TLS 1.3 fails to provide the communicating parties with additional privacy in the form of forward secrecy. In response to this, we extend the syntax of S-KEMs to *puncturable S-KEMs* (PS-KEMs), which make it possible to model a 0-RTT key exchange that fulfills forward security. We also suggest a construction based on a Goldreich-Goldwasser-Micali PRF which enjoys both the privacy of S-KEMs and gives forward security. In summary, the abstraction to S-KEMs and PS-KEMs allows us to treat the privacy of the key exchange in a resumed TLS session in PSK 0-RTT mode, and propose ways to achieve stronger forward secrecy properties through puncturing.

Ordering. S-KEMs were introduced to model session resumption in TLS, and specifically the handshake part of the session. The connection requests which initiate a TLS handshake are sent from client to server (or one communicating party to the other) over a network. In general, networks do not have perfect reliability, so it is reasonable to assume that some of the connection requests will be lost or reordered in transmission. To capture this, we extend the stateless S-KEMs to a stateful primitive which we call *ordered S-KEMs*. Introducing state also allows us to leverage ordering toward optimizations in instantiations, because more diverse (less strict) functionality demands are possible. We define three levels of correctness, from the strictest correctness under *no ordering*, via *weak ordering* to *perfect ordering*, and discuss how they model varying degrees of transmission reliability. To reap the benefits of optimizations from ordering in the setting of puncturable S-KEMs as well, we introduce the stateful *puncturable ordered S-KEMs* (POS-KEMs). Finally, we envision potential constructions of POS-KEM schemes.

Results. The result of the project is a framework for symmetric-key KEMs which enable unilateral key exchange between two parties who already share a symmetric key, but wish to derive fresh and independent keys for separate communication sessions with forward secrecy. Additionally, the project explores ways of defining correctness and security for stateful and forward secret schemes via games and the concept of indistinguishability up to correctness [44]. The results can be viewed as a pre-study for more in-depth investigations into forward secrecy for 0-RTT key exchanges and key encapsulation in the symmetric setting. Hopefully, the abstraction of symmetric-key KEMs can find applications outside of TLS handshakes as well.

9.1 Future Work

Several interesting problems have presented themselves during the course of this project, and in the following we describe a few of the directions which future research on this topic might take. First and foremost, an extension of the project to give GGM PRF instantiations of weakly ordered POS-KEMs and prove them secure is the natural next step. Once such constructions are defined, the question of how efficient they are and what further optimizations can potentially be made could be studied. It would also be interesting to see if one can construct POS-KEM schemes that are not based on the GGM construction, for example by using Bloom filter encryption which was introduced in a recent paper by Derler et al. [20]. Furthermore, to give secure generic transforms from S-KEMs to OS-KEMs and from PS-KEMs to POS-KEMs—so that any instantiation of a secure (P)S-KEM scheme directly gives a secure (P)OS-KEM scheme—would be grand.

In studying the optimizations from leveraging ordering in POS-KEM schemes, it would be specifically interesting to look at what storage bounds on the secret key can be proven. Especially, it might be possible to reduce such bounds to the integrity of ciphertexts, thereby giving a connection between security and memory. As a means of ensuring that a clever puncturing pattern is obtained in an optimized construction, the syntax of PS-KEMs and POS-KEMs could also potentially be modified so that puncturing happens automatically after decapsulation. How this affects correctness and security notions would need to be explored first, however.

Forward secrecy of public-key KEMs has only recently been discussed, and then only via puncturing in the setting of 0-RTT key exchange in [32]. Another research direction is therefore to formally define and analyze forward secrecy of KEMs in general, in a manner similar to what e.g. Bellare and Yee [9] have done for symmetric-key encryption.

References

[1] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[2] Ross Anderson. Two remarks on public-key cryptology. Manuscript. Relevant material presented by the author in an invited lecture at the 4th ACM Conference on Computer and Communications Security, CCS 1997, Zurich, Switzerland, April 1–4, 1997, September 2000.

[3] Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 117–150, Cham, 2019. Springer International Publishing.

[4] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362 – 399, 2000.

[5] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In Vijayalakshmi Atluri, editor, *ACM CCS 02: 9th Conference on Computer and Communications Security*, pages 1–11. ACM Press, November 2002.

[6] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer, Heidelberg, August 1999.

[7] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, Heidelberg, December 2000.

[8] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, Heidelberg, May / June 2006.

[9] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Heidelberg, April 2003.

[10] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. On the existence of extractable one-way functions. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 505–514. ACM Press, May / June 2014.

[11] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 280–300. Springer, Heidelberg, December 2013.

[12] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 501–519. Springer, Heidelberg, March 2014.

[13] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 255–271. Springer, Heidelberg, May 2003.

[14] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474. Springer, Heidelberg, May 2001.

[15] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, Heidelberg, August 1998.

[16] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. Cryptology ePrint Archive, Report 2001/108, 2001. `http://eprint.iacr.org/2001/108`.

[17] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 45–64. Springer, Heidelberg, April / May 2002.

[18] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[19] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, Heidelberg, December 2003.

[20] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 425–455. Springer, Heidelberg, April / May 2018.

[21] Data encryption standard. National Bureau of Standards, NBS FIPS PUB 46, U.S. Department of Commerce, January 1977.

[22] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[23] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[24] Morris Dworkin. Request for review of key wrap algorithms. Cryptology ePrint Archive, Report 2004/340, 2004. `http://eprint.iacr.org/2004/340`.

[25] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 60–75, Paris, France, April 2017. IEEE Computer Society Press.

[26] Georg Fuchsbauer, Momchil Konstantinov, Krzysztof Pietrzak, and Vanishree Rao. Adaptive security of constrained PRFs. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 82–101. Springer, Heidelberg, December 2014.

[27] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

[28] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science*, pages 464–479. IEEE Computer Society Press, October 1984.

[29] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[30] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE Computer Society Press, May 2015.

[31] Christoph G. Günther. An identity-based key-exchange protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology – EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 29–37. Springer, Heidelberg, April 1990.

[32] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 519–548. Springer, Heidelberg, April / May 2017.

[33] Felix Günther and Sogol Mazaheri. A formal treatment of multi-key channels. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 587–618. Springer, Heidelberg, August 2017.

[34] Susan Hohenberger, Amit Sahai, and Brent Waters. Replacing a random oracle: Full domain hash from indistinguishability obfuscation. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 201–220. Springer, Heidelberg, May 2014.

[35] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 33–62. Springer, Heidelberg, August 2018.

[36] Jonathan Katz. A forward-secure public-key encryption scheme. Cryptology ePrint Archive, Report 2002/060, 2002. `http://eprint.iacr.org/2002/060`.

[37] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 669–684. ACM Press, November 2013.

[38] Tadayoshi Kohno, Adriana Palacio, and John Black. Building secure cryptographic transforms, or how to encrypt and MAC. Cryptology ePrint Archive, Report 2003/177, 2003. `http://eprint.iacr.org/2003/177`.

[39] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, Heidelberg, August 2010.

[40] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[41] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P 2016)*, pages 81 – 96, Saarbrucken, Germany, May 2016. IEEE Computer Society Press.

[42] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[43] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, Heidelberg, May / June 2006.

[44] Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 3–32. Springer, Heidelberg, August 2018.

[45] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 475–484. ACM Press, May / June 2014.

[46] Victor Shoup. Using hash functions as a hedge against chosen ciphertext attack. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 275–288. Springer, Heidelberg, May 2000.

[47] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/2004/332`.

[48] Nigel P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer, Heidelberg, 2016.

[49] Nick Sullivan. Introducing zero round trip time resumption (0-RTT). `https://blog.cloudflare.com/introducing-0-rtt/`, 2017.

Appendix A

Illustration of the PS-KEM$_{\mathsf{GGM}}$ Puncturing Algorithm

We give here a more detailed illustration of the puncturing algorithm in the PS-KEM construction PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$ from Section 7.4. The example is given for a tree of height $h = 3$.

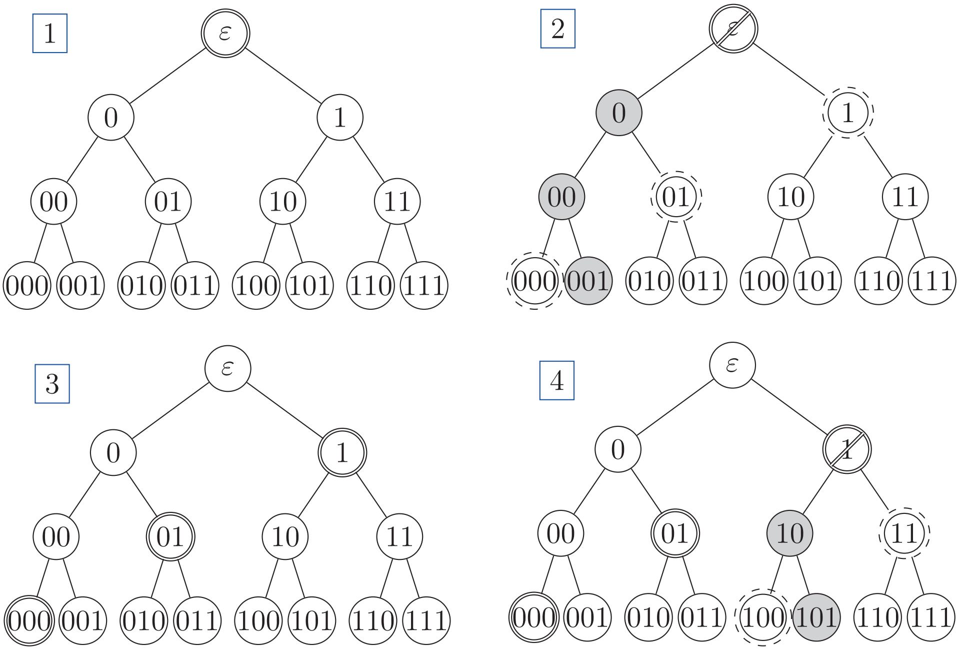

Figure A.1: GGM tree illustrating the puncturing algorithm of PS-KEM$_{\mathsf{GGM}}[\mathsf{G}, h]$.
(1) The secret key initially consists of the root only.
(2) Puncturing on node 001 removes the root from the secret key and adds the nodes with a dashed extra circle (siblings of the shaded nodes).
(3) The updated secret key consists of nodes 1, 01 and 000.
(4) Puncturing on node 101 removes node 1 from the secret key, and adds node 11 and node 100.

Appendix B

Populärvetenskaplig sammanfattning

Framtidssäkrat internetsurfande - både snabbt och tryggt

Integritet på nätet eller snabb uppkoppling? Med dagens krypteringsmetoder är det antingen eller, men i framtiden behöver vi kanske inte kompromissa.

I takt med att samhället digitaliseras flyttas alltmer kommunikation till internet, vilket i många fall gör livet smidigare. Samtidigt blir vi mer sårbara då data som skickas över nätet eller lagras online riskerar att läcka ut och bli allmänt känd. För att värna om vår integritet och säkerhet vidtas åtgärder för att dölja information som skickas via nätet, men dessa insatser är tidskrävande och innebär extra överföringar vilket gör uppkopplingen långsammare.

Mer specifikt skyddas internetanslutningar till webbadresser som inleds med 'https' idag av kryptering som gör kommunikationen oläslig för alla utomstående. Utan tillgång till nycklar som "låser upp" och dechiffrerar de krypterade meddelandena är de inget mer än rappakalja. För att de kommunicerande parterna ska kunna ta del av innehållet krävs därför att de utbyter krypteringsnycklar. Detta sker i början av uppkopplingen för att se till att krypteringen i varje anslutning är oberoende av tidigare sessioner. Man kan säga att nycklarna är som engångsartiklar. De förhandlas fram, används i en session och slängs sedan. Själva utbytet, när parterna kommer överens om nycklarna, bromsar uppkopplingen och gör att anslutningen upplevs som långsam. Men det finns snabbare sätt.

Nyligen lanserades en snabbuppkopplingsfunktion som gör det möjligt att återansluta utan det inledande nyckelutbytet. Förenklat kan man säga att om de kommunicerande parterna har varit i kontakt tidigare och redan delar nycklar från en tidigare session, så kan dessa återanvändas för att kryptera även framtida anslutningar. Effekten blir en markant prestandaökning.

Tyvärr ger den nya funktionen lägre säkerhetsgarantier än vid ett fullt nyckelutbyte. Detta eftersom återanvändningen av kryptografiskt material potentiellt leder till att alla sessioner där samma nycklar har använts blir oskyddade om nycklarna läcker ut. Det finns naturligtvis alltid en risk att någon knäcker krypteringen eller kommer över de nycklar som används, men då engångsnycklar används leder det bara till att kommunikationen i den pågående sessionen blir synlig för förövaren. Om nycklarna däremot har använts till flera sessioner, riskerar alla dessa att bli oskyddade.

I det här arbetet undersöks möjligheten att behålla den snabba uppkopplingen som den nya funktionen ger, men utan att behöva kompromissa om säkerheten. Idén bygger på ett smart sätt att återanvända nycklar. Istället för att använda exakt samma nyckel till flera sessioner uppdateras nyckeln mellan varven, så att tidigare sessioners kommunikation inte går att dekryptera om den nya, förändrade nyckeln läcker ut. Operationen kallas för punktering, och man kan föreställa sig att förmågan att dekryptera vissa meddelanden försvinner när nyckeln punkteras, ungefär som när en biljett klipps efter användning. Den går fortfarande att använda till framtida sessioner, men avslutad kommunikation som nyckeln har "punkterats på" är lika oläsbar med den punkterade nyckeln som utan den.

Det här projektet har gått ut på att skapa en modell som visar hur punkterade nycklar skulle kunna användas för att ge både snabb uppkoppling och den önskade säkerheten. Modellen består av en ny matematisk abstraktion som fångar hur den nya funktionen tillåter snabba uppkopplingar. Den innehåller också en påbyggnad som beskriver hur punktering kan användas för högre säkerhetsgarantier. Modellen visar att det i teorin är möjligt att genom punktering uppnå både hög säkerhet och prestanda, men att det finns vissa nackdelar med tillvägagångssättet. Framförallt är det problematiskt att de punkterade nycklarna kräver mer lagringsutrymme än vanliga nycklar, vilket i vissa fall kan vara ett hinder. I arbetet har det därför också undersökts hur ordningsföljden av återanslutna sessioner skulle kunna utnyttjas för att hålla ned storleken på de punkterade nycklarna. Detta genom att designa algoritmer så att det vid normal användning skapas platseffektiva "punkteringsmönster". Studien visar att det finns potential hos förslaget. För att avgöra om idén med punkterbara nycklar är värd att lansera i stor skala krävs dock fortsatta undersökningar där konstruktioner implementeras och testas på verkliga scenarion.

Printed by Tryckeriet i E-huset, Lund 2019

LUND
UNIVERSITY

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-699
http://www.eit.lth.se