HAL
open science

Advanced password-authenticated key exchanges

Pierre-Alain Dupont

To cite this version:

Pierre-Alain Dupont. Advanced password-authenticated key exchanges. Cryptography and Security [cs.CR]. PSL Research University, 2018. English. NNT : . tel-01868828v1

HAL Id: tel-01868828

https://inria.hal.science/tel-01868828v1

Submitted on 5 Sep 2018 (v1), last revised 13 Jan 2020 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École normale supérieure

Advanced password-authenticated key exchanges

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité Informatique

COMPOSITION DU JURY

M. ABDALLA Michel
École normale supérieure
Président du jury

M. POINTCHEVAL David
École normale supérieure
Directeur de thèse

M. FOUQUE Pierre-Alain
Université Rennes 1
Rapporteur

M. LIBERT Benoît
École normale supérieure de Lyon
Rapporteur

Mme CHEVALIER Céline
Université Panthéon-Assas Paris II
Membre du jury

M. POUPARD Guillaume
ANSSI
Membre du jury

M. WIEDLING Cyrille
Direction générale de l'armement
Membre du jury

Soutenue le 29 août 2018 par
Pierre-Alain Dupont

Dirigée par
David Pointcheval

ENS ÉCOLE NORMALE SUPÉRIEURE | PSL RESEARCH UNIVERSITY PARIS

Advanced password-authenticated key exchanges

Thèse de doctorat dirigée par David Pointcheval

Pierre-Alain Dupont

Abstract

Authenticated key exchange is probably the most widely deployed asymmetric cryptographic primitive, notably because of its inclusion in the TLS protocol. Its cousin, password-authenticated key exchange — where the authentication is done using a low-entropy password — while having been studied extensively as well has been much less used in practice. It is, however, a primitive much closer to actual authentication when at least one party is human.

In this thesis, we consider advanced primitives based on password-authenticated key exchange, with an eye toward practical applications. Specifically, we introduce fuzzy password-authenticated key exchange, where the authentication succeeds as long as the two passwords are close enough, and not necessarily equal. We provide a security model in the UC framework, as well as a construction based on regular password-authenticated key exchanges and robust secret-sharing schemes.

Secondly, we consider the practical problem of password leakage when taking into account sessions conducted on a corrupted device. As there is intrinsically no hope with regular password authentication, we extend the BPR security model to consider low-entropy challenge responses instead. We then provide several instantiations, some based on human-compatible function families, where the operation required to answer the challenge are simple enough to be conducted in one's head, allowing the actual authentication to be directly performed by the human being.

Résumé

L'échange de clef authentifié est probablement la primitive asymétrique la plus utilisée, notamment du fait de son inclusion dans le protocole TLS. Pour autant, son cousin, l'échange de clef authentifié par mot de passe, où l'authentification s'effectue par comparaison de mot de passe, l'est bien moins, bien qu'ayant déjà fait l'objet d'études considérables. C'est pourtant une primitive finalement bien plus proche d'une authentification réelle, dès lors qu'une des parties est humaine.

Dans cette thèse, nous considérons des primitives avancées fondées sur l'échange de clef authentifié par mot de passe, en gardant à l'œil ses applications pratiques. Spécifiquement, nous introduisons une nouvelle primitive, l'échange de clef authentifié par mot de passe approximatif, où la condition de succès de l'authentification est désormais d'avoir une distance suffisamment faible entre les deux mots de passe, et plus nécessairement l'égalité parfaite. Nous fournissons un modèle de sécurité dans le cadre du modèle de composabilité universelle (UC) ainsi qu'une construction reposant sur un partage de secret robuste et des échanges de clefs authentifiés par mot de passe exact.

Dans une seconde partie, nous considérons le problème pratique de la perte du mot de passe dès lors qu'une session est conduite sur un terminal compromis. Étant donné qu'il s'agit d'un problème intrinsèque à l'authentification par mot de passe, nous étendons le modèle BPR habituel pour prendre en compte, en lieu et place du mot de passe, des questions-réponses, toujours de faible entropie. Nous fournissons plusieurs protocoles dans ce modèle, dont certains reposent sur des familles de fonctions compatibles avec les humains, dans lesquelles les opérations requises pour dériver la réponse depuis la question sont suffisamment simples pour être faites de tête, permettant donc à l'humain de s'identifier directement.

Acknowledgments

En tout premier lieu, je tiens à adresser mes plus vifs remerciements à mon directeur de thèse, David Pointcheval. Les innombrables heures passées dans ton bureau à discuter marqueur en main des dernières difficultés découvertes dans nos travaux sont certainement le meilleur souvenir que je garderai de ces années de thèse. Il est certain que sans ton expertise et ta profonde maitrise de tant de sous-domaines de la cryptographie, cette thèse n'aurait jamais pu voir le jour.

Je témoigne également toute ma reconnaissance à mes deux rapporteurs, Pierre-Alain Fouque et Benoît Libert, pour avoir accepté d'entreprendre la lourde tâche de relire, mais surtout d'étudier ce manuscrit avec un œil frais et critique. Leurs remarques précises et nombreuses auront grandement contribué à le rendre plus clair et plus cohérent.

Je voudrais aussi remercier les autres examinateurs ayant accepté de prendre part à mon jury : Céline Chevalier et Michel Abdalla, avec qui j'ai eu plaisir à partager mon quotidien à l'ENS, Guillaume Poupard, mon illustre prédécesseur, ainsi que Cyrille Wiedling, qui a accepté au pied levé de représenter au sein de ce jury DGA MI. Je tiens par la même occasion à remercier la DGA elle-même, et plus particulièrement la commission de la recherche, qui m'a permis de dédier intégralement mes premières années d'Ingénieur de l'Armement à cette thèse, sans aucune contrepartie. Un grand merci à Julien Devigne, qui a assuré mon suivi scientifique, ainsi qu'à Philippe Leriche, mon unique point de contact au sein de la DGA pendant toutes ces années. Je souhaiterais en outre remercier Raphaël Bost, qui a contribué à m'aiguiller lors de mes débuts dans le monde de la cryptologie, et Bruno Bouzy, pour m'avoir confié une charge d'enseignement à l'université.

During this thesis, it was a great pleasure to collaborate with many talented researchers, and I therefore express my sincere thanks to my coauthors: Sasha (Alexandra Boldyreva), Shan Chen, Julia Hesse, David Pointcheval, Leonid Reyzin and Sophia Yakoubov. Among them a special thanks goes to Julia for all those times you discovered yet another issue in our construction and we had to fix it together. I'd also like to thank Guilhem Castagnos, Céline Chevalier, Patrick Towa Nguenewou and Damien Vergnaud for an incomplete work that was still quite interesting.

J'aimerais aussi remercier tous mes autres collègues de l'équipe crypto de l'ENS, pour avoir fait de cet environnement un endroit toujours agréable et plein de vie. Tout d'abord, Céline, Damien, David, Michel, Georg, Hoeteck et Vadim pour avoir su fournir le cadre nécessaire à cet apprentissage. Merci également à l'équipe administrative, et notamment Jacques Beigbeder, Lise-Marie Bivard, Stéphane Emery, Nathalie Gaudechoux, Joëlle Isnard, Sophie Jaudon et Valérie Mongiat. Et bien évidemment, mes collègues doctorants (ou jeunes docteurs) : Adrian, Alain, Anca, Aurélien, Chloé, Dahmun, Fabrice, Florian, Geoffroy, Houda, Julia, Jérémy, Louiza, Michele M., Michele O., Pierrick, Pooya, Rafaël, Răzvan, Romain, Sonia, Sylvain, Thierry, Thomas Pe. et Thomas Pr.

Je remercie également mes autres amis, sans qui mine de rien ces années auraient été longues et ternes. En premier lieu mes deux colocataires : Cyprien (*Cypi*) et Godeffroy (*god*). À refaire quand vous voulez. J'inclus évidemment tous les faërixiens, mais plus spécialement

Arnaud (*Nonal*), Corentin (*kangz*), Florian (*olotiar*), Guilhem, Guillaume (*Campi*), Henri, Jean-Baptiste (*Gueust*), Jeremy Bu. (*Aniem*), Jérémy Be. (*Bidou*), Lauriane (*lau*), Loïc (*Zed*), Matthias (*marteo*), Morgane (*Freeman*), Nicolas (*IooNag*), Pascal, Pierre-Étienne (*Karlos*), Raphaël (*Xelnor*), Sarah (*Goinfrex*), Thomas (*fulanor*), Victor (*Levans*) et Yann (*Drynn*). Je mentionne aussi Baptiste, François, Guillaume P., Matthieu, Maxime, Meredith, Morgane C., Myriam, Pauline, Richard et Zachary.

En remontant plus loin dans ma scolarité, je tiens à remercier Serge Vaudenay, pour ses excellents cours sur la cryptologie à l'EPFL, qui ne sont certainement pas étrangers à mon attrait initial pour ce domaine. Je remercie également Corentin et Nicolas pour cette première expérience, malheureusement infructueuse, de la recherche académique. Enfin, je ne saurais omettre Stéphane Olivier, Jean-Claude Sifre et Dominique Zann sans lesquels ma prépa aurait été bien différente.

Je dédie mes derniers remerciements à ma famille qui me supporte depuis toujours, mes deux parents Annie et Jacques et ma sœur Claire, ainsi qu'à Aurélie, qui me fait l'honneur de partager ma vie depuis plus de deux ans.

Contents

Abstract iii

Résumé v

Acknowledgments vii

1 Introduction 1
- 1.1 Personal contributions . . . 4
  - 1.1.1 Contributions in this thesis . . . 4
  - 1.1.2 Other contribution . . . 4
- 1.2 Organization of this thesis . . . 5

2 Preliminaries 7
- 2.1 Universal composability framework . . . 7
- 2.2 Password authenticated key exchange . . . 8
  - 2.2.1 Implicit or explicit authentication . . . 8
  - 2.2.2 PAKE in the UC framework . . . 8
  - 2.2.3 PAKE in the BPR model . . . 11
- 2.3 Other building blocks . . . 12
  - 2.3.1 Commitment scheme . . . 12
  - 2.3.2 Authenticated encryption . . . 14
  - 2.3.3 Linear codes . . . 15
  - 2.3.4 Robust secret sharing . . . 16
  - 2.3.5 Models as UC functionalities . . . 18

3 Implicit-only PAKE 21
- 3.1 Motivation . . . 21
- 3.2 Definition . . . 21
- 3.3 Instantiation . . . 23

4 Fuzzy Password Authenticated Key Exchange 35
- 4.1 Model . . . 35
- 4.2 Construction . . . 38
  - 4.2.1 A naive idea . . . 38
  - 4.2.2 Improved idea . . . 39
  - 4.2.3 Removing modeling assumptions . . . 48

5 Human authenticated key exchange 49
- 5.1 Human-compatible function family . . . 50
  - 5.1.1 Syntax . . . 50
  - 5.1.2 Security . . . 50

5.2 HCFF instantiations . . . 53
5.2.1 Token-based HCFF . . . 53
5.2.2 Only-human HCFF . . . 53
5.3 HAKE definition . . . 57
5.4 HAKE security model . . . 58
5.5 Generic HAKE constructions . . . 61
5.5.1 The Basic HAKE . . . 62
5.5.2 The Confirmed HAKE . . . 69
5.6 Token-based HAKE constructions . . . 74
5.6.1 Simplified Basic HAKE . . . 74
5.6.2 Time-Based HAKE . . . 75

6 Conclusion and open questions **85**
6.1 Conclusion . . . 85
6.2 Open questions . . . 86

Abbreviations **89**

List of Illustrations **91**
Figures . . . 91
Tables . . . 91

Bibliography **93**

Chapter 1

Introduction

At its beginning, cryptography was mainly about the design — and breaking — of ciphers, which we nowadays call symmetric encryption schemes. Basically, this refers to the ability to communicate a message without it being understood by anyone but its intended recipient. A prerequisite for all schemes is to have agreed upon some sort of the secret with the receiver at the beginning (be it the cipher itself or a smaller part of it, the key). While schemes got more and more secure and complicated as time went by, nothing much changed about this basic assumption until the age of the Internet.

In the last half-century, however, with the advent of computers and widely available remote communications, this basic setting has been changing. Not only do we require more and more security properties from the cryptographic scheme, but the prerequisite of initially sharing a common secret became a much bigger problem. How can I initially share a secret with every possible entity I will be communicating with?

Then came asymmetric cryptography, or public-key cryptography, in which there are still secrets but they no longer have to be shared with anybody. In an asymmetric encryption scheme, the receiver would simply generate a key pair, comprised of one public key and one private key. He would keep the private key for himself, and share the public key with not just the sender, but anybody. The security properties of the scheme would guarantee that, even knowing the public key, intercepting a message that had been encrypted by it would give no clue as to the actual message. Obviously the receiver would not be able to reply using the same key pair, for the sender could not read the response, but this can easily be solved by using another key pair, this time generated by the initial sender, for the reverse direction.

This leads to the first really new challenge of modern cryptography: authentication. Since now anyone can securely write to me, how do I know that the one I'm actually communicating with is who he pretends to be? With symmetric cryptography, this was quite easy, since he must be one that knows the shared secret[1], but not so much now. And, similarly, how do I know that the public key that is being presented to me as the one of my intended recipient, and therefore with which I'll send my message, is actually the correct one. Fortunately, asymmetric cryptography also holds the answer, in the form of cryptographic signatures, a primitive by which one does not protect the content of the message from being read, but protects the identity of its author, as well as the message's integrity, from being manipulated.

[1] Actually, the modern definition of authentication would not be quite happy with just that, since even not knowing shared key, it is possible to manipulate a message (e.g. truncating it) which should be detected by the authentication property.

Obviously it does need a source of trust, but we'll talk about that a bit later.

Asymmetric cryptography, for all its advantages, unfortunately comes with one drawback: it is very slow. This is due to the fact that, at its core, it works on a different kind of assumption. Symmetric cryptography iterates many rounds of quite simple operations (some linear, other non-linear). Asymmetric cryptography works on a hard mathematical problem, which typically involves one very complex operation (e.g. exponentiating a number). This simply does not scale up with the amount of data that need to be secured.

This is why most online communications actually compose an asymmetric primitive, called *authenticated key exchange* (AKE), with a symmetric encryption scheme. An AKE is an interactive protocol where two parties sharing an authentication means get, at the end, a shared secret, especially suitable to be used in a symmetric encryption scheme. This allows to mitigate the slowness of asymmetric cryptography, since it is only used at the beginning of the protocol on a fixed amount of data, regardless of the size of the data thereafter exchanged, while not requiring that the particular two parties involved actually initially share a secret, which is not practical.

As noted previously, asymmetric cryptography does not completely free one of having some trusted data beforehand. Indeed, without trusting anything *a priori*, it is not possible to do anything meaningful. In practice, the trust one must have is in the *public-key infrastructure* (PKI), a somewhat complex system of authorities, that can authenticate the fact that a public key is indeed associated with the entity you know of. On the Internet, this typically means bridging the gap between the website's name (e.g. www.example.org) and its public key (that will subsequently be used to secure the communication with it). This is done through a concept of trust store, where the public keys of some of those entities are populated by the vendor of the software used (typically, the browser or the operating system). This mechanism is usually not symmetric, in the sense that while websites are authenticated by the PKI, users are not and, whenever a website needs to know who one of its users is, it uses an ad hoc system to do it (such as registration, and a login/password field), completely independent of the actual cryptography involved.

This PKI is also, unfortunately, one of the biggest failure points of the security of the Internet as a whole. Repeatedly, malicious parties have proven able to obtain trusted certificates, either by corrupting an authority or by taking advantage of insufficiently validated credentials. Even when everything works as it should, it is easy to obtain a certificate for a domain close enough for an unsuspecting user to be oblivious to.

A different approach that does not require any PKI is undertaken in *password authenticated key exchange* (PAKE), a variant of AKE where the authentication means is a password — that is, some secret information that both parties know in advance. Basically, assuming both parties hold a password (of low entropy) at the beginning, a PAKE is an interactive process through which they will each receive a session key (of high entropy) that will — assuming they did input the same password — be identical, while no-one else can learn it. Since by definition there is a small but non-negligible chance of simply guessing the password, this translates into a small but non-negligible probability of defeating PAKE security.

The main advantage of a PAKE is that the authentication is symmetric. From the server's point of view, the user will be authenticated while at the same time the server itself is authenticated from the user's point of view. No external trust is required and the password does not need to be handed over to the server, it has to know it already.

From a cryptographic perspective, the PAKE primitive has been studied extensively, beginning with the seminal paper [BM92] to the first formal models of [BPR00; BMP00]

and eventually its formalization in the generic *universal composability* (UC) model [Can01; CHK+05]. Numerous constructions exist, but only the original EKE construction from [BM92] is of particular interest to this thesis.

While a PAKE guarantees that — if the same password was entered — the generated session key is the same for both parties, it is not necessarily the case that they know at the end of the protocol whether it is so or not. A PAKE is said to have explicit authentication when the party knows, at the end, if the protocol succeeded or not, and implicit authentication if they don't. Most applications actually care about explicitly authenticated PAKE, but fortunately, one can turn an implicitly authenticated PAKE into an explicitly authenticated PAKE, at the cost of one additional flow (basically to demonstrate knowledge of the correct key by encrypting/decrypting some value, see [BPR00]).

One could wonder, however, if there aren't situations where the uncertainty of implicitly authenticated PAKE isn't useful. This is usually not considered by regular PAKE models, that allow the adversary to learn whether authentication succeeded or not, even though no actual user has that capability. Hence we introduce *implicit-only password authenticated key exchange* (iPAKE), which formalizes a PAKE where even the adversary remains oblivious to the fact that authentication succeeded or not, and show that the seminal EKE construction [BM92] satisfies this functionality. Obviously, this is only useful if we find applications where an iPAKE is used as a building block, but a careful approach is taken regarding the generated session keys, so as not to leak whether authentication succeeded or not. We introduce such an application next.

Taking a look back at PAKE, a natural extension would be to consider an authentication mechanism that would succeed if the two passwords were close enough, instead of equal. We call it a *fuzzy password authenticated key exchange* (fPAKE). This is especially useful for biometric authentication, since its process of secret reconstruction is naturally imprecise. Quite an easy way to do it would be to use a fuzzy extractor [DRS04; Boy04] to reconstruct a deterministic secret from the fuzzy data, and to compose a regular PAKE with it. However, the fuzzy extractor requires some information to reconstruct the secret, and this information must be provided publicly, which reduces drastically the actual entropy of the password.

Assuming that this closeness of passwords is measured by the Hamming distance, two passwords are close enough if they only differ on a few number of characters. This reminds of *error-correcting code* (ECC), which can correct transmission errors or, in the case of secret values, of *robust secret sharing* (RSS). Hence we'll propose a fPAKE construction where we use a secret sharing to reconstruct a secret, while secret shares are actually masked by many session keys created through iPAKE protocols. For each character in the password, an independent iPAKE instance is run. When the iPAKE succeeded, you can remove the mask and hence learn the correct share, but shares at positions where the iPAKE failed are completely masked. Note that for this to work, it is essential that the positions where the iPAKE succeeded are not known in advance, otherwise more efficient RSS reconstruction algorithms exist. Hence a regular PAKE would not provide good-enough security guarantees.

Another issue we tackle with PAKE is the fact that, while the password-based authentication has the nice property of directly involving the actual (human) user, who has to remember it, it also means that this password could be used on potentially insecure devices. Consider the situation where one logs in on a public computer, which is potentially insecure. Since the computer (which we'll call the terminal) learns the password, were it to be malicious not only is this session compromised, but no security could be offered even on future sessions. This is called a *strong* corruption, where the adversary learns all secrets. It seems that nothing can

be done cryptographically, because the long-term secret — the password — was leaked to the adversary. But what if the long-term secret was not what was entered in the terminal? We therefore extend PAKE into a primitive called *human authenticated key exchange* (HAKE), where we consider three actual parties: the human, the (potentially malicious) terminal and the server. The goal is to allow the human to establish a secure communication with the server whenever the terminal is honest, assuming not too many sessions have been previously conducted on malicious terminals. Instead of directly using the password, the authentication will be conducted via a challenge-response mechanism, keyed by a long-term secret.

This can be achieved either using a trusted device such as the physical tokens used by 2-Auth mechanism, or directly by the human himself using a human-computable function, i.e. a function simple enough to be computed in one's head, yet secure enough to prevent the forging of answers to other queries. Though quite a challenging topic, recent breakthrough allow to begin to consider it. Notably, the paper by Blocki et al. [BBDV17], based on the hardness arguments from [FPV15], provides a first idea on how to achieve this.

1.1 Personal contributions

1.1.1 Contributions in this thesis

While this thesis is not purely a compilation of papers, it is mainly based on two publications, which are briefly described below.

[**BCDP17**] This paper deals with the hard task of maintaining security of a password-based authentication while, in practice, a user may occasionally use it in a non-secure setting (typically, a computer he does not own). This is called strong corruption and, by design, the long-term secret used in PAKE (the password) is compromised, and thus no security can be offered from future sessions. By replacing the password with a challenge-response function, whose responses can be easily computed, we achieve some security for future sessions, as long as not too many sessions were compromised. We invite interested readers to go to Chapter 5, which includes most of those results.

[**DHP+18**] In this paper, we consider how to extend the classical PAKE into a fPAKE functionality, where the authentication will succeed as long as the passwords are close enough according to some metric. A natural way to construct it would have been to compose a fuzzy extractor [DRS04; Boy04] and a regular PAKE, but the fuzzy extractor leaks a lot of information thus reducing the actual password entropy drastically. In this paper, we offer two constructions that achieve our fPAKE functionality in the UC model. The first construction based on Yao's garbled circuits achieves it for any efficiently computable metric. The second construction, based on PAKE and RSS, is more efficient but limited to the Hamming distance metric. We present the latter construction in this thesis, on Chapter 4. We direct readers interested in the former construction to the actual paper.

1.1.2 Other contribution

In addition to the work presented in this thesis, and outlined above, we also worked during our thesis on functional encryption, which led to the publication below.

[**DP17**] In this paper, we consider practical trade-offs related to the inner-product functional encryption primitive. While recent papers, notably [ABDP15], have shown how to construct it quite efficiently, the functionality itself leaks much information on the ciphertext,

which restricts the number of functional keys that can be generated. Indeed, if you hold the key for a vector $\boldsymbol{f}$, you can (by the functionality) learn $\boldsymbol{f} \cdot \boldsymbol{x}$ for all plaintext $\boldsymbol{x}$, and therefore colluding with n other key holder for different functions is typically enough to completely reveal a plaintex of size n. We therefore introduce an additional online player called the helper, that will have to be interacted with for any decryption query. We let it learn just the minimum required to guarantee the confidentiality of the database (in particular, not the result of the decryption). This allows to distribute many functional keys, while preventing too many of them from being used on the same data, which would breach privacy.

1.2 Organization of this thesis

This thesis is organized in six chapters, each of which is briefly presented below.

Chapter 2 introduces notations and recalls all preliminaries necessary to understand the thesis.

Chapter 3 presents a slightly stronger notion of PAKE, called iPAKE, in which the outcome of the key exchange is not directly linked to the adversary. It also introduces a label version, *labeled implicit-only password authenticated key exchange* (liPAKE), where an additional information can be added by each party during the execution of the PAKE, to be authenticated as well. Lastly, it shows that a well-known PAKE construction, EKE2, actually satisfies this liPAKE notion, in an idealized model.

Chapter 4 presents a more general notion of PAKE, called fPAKE, where the password equality requirement for the authentication success is replaced by a password closeness notion. It also shows how construct a secure fPAKE protocol, using a liPAKE construction as well as a RSS.

Chapter 5 takes a step back on the choice of passwords as the authentication method to consider how to achieve some kind of security in a setting where passwords can sometimes be compromised. By introducing a challenge response answered directly by the actual human user (with or without help from an external device), it allows to maintain some security for future sessions, as long as not too many sessions occurred on a corrupted machine.

Lastly, Chapter 6 concludes this thesis by offering a brief summary of what was achieved and stating some related questions that are still open for investigation.

Personal publications

[BCDP17] Alexandra Boldyreva, Shan Chen, Pierre-Alain Dupont, and David Pointcheval. "Human Computing for Handling Strong Corruptions in Authenticated Key Exchange". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Aug. 2017, pp. 159–175. DOI: 10.1109/CSF.2017.31.

[DHP+18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. "Fuzzy Password-Authenticated Key Exchange". In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Heidelberg, Apr. 2018, pp. 393–424. DOI: 10.1007/978-3-319-78372-7_13.

[DP17] Pierre-Alain Dupont and David Pointcheval. "Functional Encryption with Oblivious Helper". In: *ASIACCS 17*. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. ACM Press, Apr. 2017, pp. 205–214.

Chapter 2

Preliminaries

2.1 Universal composability framework

In the following, we will often discuss cryptographic primitives presented in several models. Historically, primitives are often presented in ad hoc models that fall into two categories: game-based models and simulation-based models. In this thesis, we will make use of both of them.

A game-based model (or indistinguishability-based model) provides the adversary with a way to interact with an instance of the protocol, as well as ad hoc goals for each desired security property. A simulation-based model provides an idealized protocol that has all the desired security properties and allows to show how interacting with a real instance of the protocol is indistinguishable from interacting with the idealized protocol.

Among simulation-based models, the *universal composability* (UC) framework holds a particular place because of its generic composability property, allowing for an easy re-use of a cryptographic primitive. While most of the framework is fixed, a component called the ideal functionality $\mathcal{F}$ captures the precise properties the idealized instance should have.

Proving that a construction is secure in the UC framework means exhibiting a simulator $\mathcal{S}$ such that, anything an adversary $\mathcal{A}$ could do against honest players in the real protocol could be achieved by the same adversary $\mathcal{A}$ against the ideal functionality $\mathcal{F}$, with the simulator $\mathcal{S}$ as an interface between $\mathcal{A}$ and $\mathcal{F}$. But the ideal functionality is secure, by definition, and the combination of $\mathcal{S}$ and $\mathcal{A}$ cannot do anything harmful against the honest players using $\mathcal{F}$. As a consequence, $\mathcal{A}$ cannot do anything harmful against the honest players in the real protocol execution.

The security of a UC protocol is thus measured by the advantage a distinguisher $\mathcal{Z}$ could get in distinguishing the real world (the interactive protocol between honest players with an adversary $\mathcal{A}$) and the ideal world (the honest players directly dealing with the ideal functionality $\mathcal{F}$, while the simulator $\mathcal{S}$ makes the interface with the adversary $\mathcal{A}$).

Composability. The UC framework allows for composability. This means that once a construction has been proven secure in the UC framework, using it as a building block becomes easier.

Indeed, it is possible to substitute the construction itself with the ideal functionality in the broader proof. Since the simulator $\mathcal{S}$ allows to convert an adversary designed for the construction to an adversary against the ideal functionality, the broader proof can likewise

be generically converted to a proof using all the actual constructions.

It is therefore possible to use the ideal functionality itself as a building block in future proofs, which is way easier.

2.2 Password authenticated key exchange

An *authenticated key exchange* (AKE) protocol is an interactive protocol between two parties who share an authentication means. At the end of the protocol, the parties should output a session key, a session id and partner id. The correctness requires that any honest AKE execution results in the parties outputting the same session key and session id, and the partner id being the identity of the other party. The goal of the protocol is to guarantee privacy and authentication of the session key in the presence of an attacker.

A *password authenticated key exchange* (PAKE) is a specific AKE where the authentication means is a common low entropy secret (called the *password*). It essentially allows to generate a high entropy key from a low entropy secret, interactively. Given the low-entropy nature of the password, it can be guessed, which constitutes a trivial attack against the protocol. Hence the security should only show that one cannot do better than this trivial attack.

2.2.1 Implicit or explicit authentication

AKE protocols have been widely described as having an implicit or explicit authentication. The simplest form is the implicit authentication where, at the end of the protocol, it is unclear if a party knows whether the authentication succeeded or not. We will present in Chapter 3 a new notion, called *implicit-only password authenticated key exchange* (iPAKE), where no-one can know this without comparing the two sessions keys. The usual goal, however, is explicit authentication where every party involved learns whether the authentication succeeded or not. An implicit AKE can be generically turned into an explicit AKE, at the cost of one additional key confirmation flow [BPR00].

2.2.2 PAKE in the UC framework

Historical functionality. The original PAKE functionality has been defined by Canetti et al. in [CHK+05]. To distinguish it from the simplified functionnality we'll introduce next, we denote it by $\mathcal{F}_{\mathsf{pake}}^{\mathtt{H}}$ and recall it in Figure 2.1.

Three queries are used to model the ideal world: `NewSession` is used to demonstrate willingness by one of the parties P_i to participate in the PAKE. If the adversary S lets the protocol complete, `NewKey` should be called and the session key sent to the corresponding party. However, the adversary S may also take advantage of the session to try to guess the password itself instead, using the `TestPwd` query. It may also have corrupted one of the parties, which may change the output of `NewKey`.

We stress that while this functionality models implicit authentication, it also immediately leaks the result of the `TestPwd`-query to the adversary: when the adversary tries a password, it learns whether the guess was correct or not but a regular party does not. We will discuss in Chapter 3 a functionality that is less permissive.

The main idea is the following: If neither party is corrupted and the adversary does not attempt any password guess, then the two players both end up with either the same uniformly distributed session key if the passwords are the same, or uniformly distributed independent

The functionality $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$ is parameterized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties $P_1,\ldots,P_n$ via the following queries:

- **Upon receiving a query $(\mathtt{NewSession}, \mathsf{sid}, P_i, P_j, \mathsf{pw}, \mathtt{role})$ from party P_i:**

  Send $(\mathtt{NewSession}, \mathsf{sid}, P_i, P_j, \mathtt{role})$ to $\mathcal{S}$. If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, P_j, P_i, \mathsf{pw}')$, then record $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$ and mark this record `fresh`.

- **Upon receiving a query $(\mathtt{TestPwd}, \mathsf{sid}, P_i, \mathsf{pw}')$ from the adversary S:**

  If there is a record of the form (P_i, P_j, pw) which is `fresh`, then do: If $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised` and reply to $\mathcal{S}$ with "correct guess". If $\mathsf{pw} \neq \mathsf{pw}'$, mark the record `interrupted` and reply with "wrong guess".

- **Upon receiving a query $(\mathtt{NewKey}, \mathsf{sid}, P_i, \mathsf{sk})$ from the adversary S:**

  If there is a record of the form $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$, and this is the first `NewKey` query for P_i, then:

  - If this record is `compromised`, or either P_i or P_j is corrupted, then output $(\mathsf{sid}, \mathsf{sk})$ to player P_i.
  - If this record is `fresh`, and there is a record (P_j, P_i, pw') with $\mathsf{pw}' = \mathsf{pw}$, and a key sk' was sent to P_j, and (P_j, P_i, pw) was `fresh` at the time, then output $(\mathsf{sid}, \mathsf{sk}')$ to P_i.
  - In any other case, pick a new random key sk' of length k and send $(\mathsf{sid}, \mathsf{sk}')$ to P_i.

  Either way, mark the record $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$ as `completed`.

Figure 2.1: Ideal Functionality $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$ for PAKE (recalled from [CHK+05])

session keys if the passwords are distinct. However, if one party is corrupted (the adversary was given the password), or if the adversary successfully guessed the player's password (the session is then marked as `compromised`), the adversary is granted the right to fully determine its session key. In case of wrong guess (the session is then marked as `interrupted`), the two players are given independently chosen random keys. A session that is neither `compromised` nor `interrupted` is called `fresh`, which is its initial status.

Remember than in the UC framework, security is determined by how a distinguisher can distinguish the construction from the ideal functionality. $\mathsf{Adv}^{\mathsf{pake}}_{\mathsf{pake}}(\mathcal{S}, \mathcal{A}, \mathcal{Z})$ thus denotes the advantage the distinguisher $\mathcal{Z}$ can get in distinguishing the two worlds. Note that in the ideal functionnality's description, S is described as the adversary and A doesn't appear. This is because it is described from the point of view of the ideal world's security game. However in the distinguishing game S — while still acting as an adversary against the ideal functionnality — is a simulator, A is the real-world adversary and $\mathcal{Z}$ is the distinguisher that should be fooled.

When using a UC-PAKE in black box, we will usually assume the existence of a simulator $\mathcal{S}$ which makes this advantage negligible for any adversary $\mathcal{A}$, and any distinguisher $\mathcal{Z}$, for the ideal functionality $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$ recalled in Figure 2.1.

The functionality $\mathcal{F}_{\mathsf{pake}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i)$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i)$ to $\mathcal{S}$;
  - If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ and mark this record `fresh`.
- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_i)$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$, then do:
  - If $\mathsf{pw}_i = \mathsf{pw}'_i$, mark the record `compromised` and reply to $\mathcal{S}$ with "correct guess".
  - If $\mathsf{pw}_i \neq \mathsf{pw}'_i$, mark the record `interrupted` and reply with "wrong guess".
- **Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is no record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ or this is not the first `NewKey` query for $\mathcal{P}_i$, ignore this query. Otherwise:
  - If the record is `compromised`, $\mathcal{P}_i$ is corrupted or both $\mathcal{P}_{1-i}$ is corrupted and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \mathsf{sk})$ to player $\mathcal{P}_i$.
  - If the record is `fresh`, both parties are honest and a key sk' was sent to $\mathcal{P}_{1-i}$, at which time there was a `fresh` record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
  - In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.

  Either way, mark the record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 2.2: Ideal Functionality $\mathcal{F}_{\mathsf{pake}}$ for PAKE (simplified from $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$)

Note that the classical EKE [BM92] protocol that encrypts a Diffie-Hellman key exchange, using the password as encryption key, is UC-secure [ACCP08], under the *computationnal Diffie-Hellman* (CDH) assumption in the *ideal cipher* (IC) model. In addition, it is quite efficient. But other constructions also exist in the standard model [CHK+05; ACP09], under the *decisional Diffie-Hellman* (DDH) assumption.

Simpler functionality. The Canetti et al. functionality can actually be simplified in several ways.

The first simplification concerns the `role` notion. While many protocols actually require a party to be identified as the initiator (or client) and another as the responder (or server), this is actually of little importance. It is perfectly acceptable for the UC functionality not to have this information (which is then actually masked by the simulator $\mathcal{S}$). This gives slightly less power to the adversary, but does not have much impact.

The second simplification concerns the setting. The $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$ functionality is defined as a two-party protocol operating in a setting with n parties. However, the n-party setting is not

required, as the composability property is enough to allow to extend the functionality to such a setting. With only two parties, the functionality is slightly simpler.

Lastly, we introduce another change in the functionality, by restricting the power of the adversary to fully determine the key for a party when it has corrupted the other party to the instances for which he holds the correct password. Indeed, it makes no sense for the adversary to be able to select it (and, thus, defeat the authentication) otherwise. Note that this makes our final functionality not strictly equivalent to the one from Canetti et al. [CHK+05], though it shouldn't have a large impact.

We present in Figure 2.2 a simplified PAKE functionality, denoted $\mathcal{F}_{\mathsf{pake}}$, that will be used in all the following.

2.2.3 PAKE in the BPR model

The UC framework, for all its usefulness, can be difficult to prove[1], and a simpler model can allow more efficient constructions to be shown. It is also often a good first step to prove a construction in a gamed-based model and then to try to show it is also secure in the UC framework.

Description. The most well-known game-based model for PAKE is the BPR model first introduced in [BPR00]. While we do not make use of it directly, we will define in Chapter 5, a notion of *human authenticated key exchange* (HAKE), whose security will be based on it. We thus present it succinctly here.

In the BPR model, the adversary is being given access to oracles representing the parties through the following queries:

- `Execute` The adversary receives a complete honest session between the two parties (passive network adversary)
- `Send` The adversary can forge messages to one party (active network adversary)
- `Reveal` (not allowed if `Test` was queried on the same session) The adversary receives the final session key (representing a leakage from the subsequent use of the session key).
- `Test` (not allowed if `Reveal` was queried on the same session) The adversary is being given either the session key or a random value (depending on a choice bit) and must distinguish.

The goal of the adversary is to either have a non-negligible advantage in distinguishing the two cases for the `Test` query (privacy property) or to make one oracle think it succeeded the protocol while the other didn't participate (authentication property).

Several variants exist, and notably the *real-or-random* technique allows to combine the `Reveal` and `Test` queries by allowing more than one `Test` query, with the same choice bit.

The `Execute` query problem. As one can note from the brief description above, the `Execute` query is perfectly simulable with `Send` queries. It actually only represents a commitment on the adversary's part to let an entire session run passively.

[1] A notable issue in UC-proofs is that the simulation must be correct even in situations where the adversary breaks the scheme, such as by guessing passwords, whereas weaker models allow to stop the simulation.

This distinction exists for counting purposes only. A passive session is much less dangerous and less costly to achieve than a session where some flows can be modified, and therefore a PAKE should be able to resist more of them.

However, it is actually hard to predict when the adversary is going to be passive and some adversaries may want to adaptively decide to modify flows, depending on the first flows. As such, `Execute` queries are not very well-suited to correctly count those requests, as this commitment to passivity may not exist in practice[2]. In Chapter 5, our BPR-based notion will not rely on `Execute` queries, but instead determines if a session is active or not (for counting purposes) after the session has occurred.

2.3 Other building blocks

2.3.1 Commitment scheme

In several constructions, we will make use of a commitment scheme, a primitive that allows the user to commit on a value x so that the receiver does not learn any information about x until a later point, but with the guarantee that the user will not be able to change his mind at that later point. UC-secure constructions for commitment scheme exist, but we'll instead introduce a weaker game-based model that will allow more efficient constructions in our protocols.

Syntax. A (non-interactive) commitment scheme $\mathcal{CS}$ is defined by Setup that defines the global public parameters, and two other algorithms:

- $\mathsf{Com}(x)$: on input a message x, and some internal random coins, it outputs a commitment c together with an opening value s;
- $\mathsf{Open}(c, s)$: on input a commitment c and then opening value s, it outputs either the committed value x or $\perp$ in case of invalid opening value.

The **correctness** condition requires that for every x, if $(c, s) = \mathsf{Com}(x)$, then $\mathsf{Open}(c, s)$ outputs x.

Game-based security. The usual **security notions** for commitment schemes are the *hiding* property, which says that x is hidden from c, and the *binding* property, which says that once c has been sent, no adversary can open it in more than one way. We respectively denote $\mathsf{Adv}^{\mathsf{hiding}}_{\mathcal{CS}}(\mathcal{A})$ (we will actually never use this one) and $\mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{A})$ the advantages an adversary may get against these two notions.

Two quite usual additional notions are the *extractability* and the *equivocality* properties which state that the simulator, using specially constructed setups, can defeat either the hidding or the binding properties, respectively. For those, we need trapdoors generated by an alternative setup algorithm and privately given to the simulator. And then, the hiding and binding properties are more delicate to be satisfied, hence the additional (probabilistic) algorithms: after a setup phase Setup' that defines the global public parameters available to

[2] If the adversary has no means of rewriting flows, it is obviously committed to passivity. However, a more reasonable assumption would be to assume it can write flows, but that it is more costly, notably because he then runs the risk of being detected.

everybody, with additional trapdoors we consider in the global private parameters, available to the simulator, we have:

- $\mathsf{SimCom}()$, that takes as input the trapdoor and outputs a pair (c, eqk) where c is a fake commitment and eqk is the equivocation key;
- $\mathsf{OpenCom}(c, \mathsf{eqk}, m)$ that takes as input a fake commitment, its equivocation key and a message, and outputs an opening value s;
- $\mathsf{ExtCom}(c)$ that takes as input a non-fake commitment and outputs the message m it commits to.

These algorithms must first satisfy the two following properties:

- **Trapdoor correctness**: (equivocality) for any message m, and any $(c, \mathsf{eqk}) \xleftarrow{\$} \mathsf{SimCom}()$ and $s \leftarrow \mathsf{OpenCom}(c, \mathsf{eqk}, m)$, we have $\mathsf{Open}(c, s) = m$; (extractability) for any message m and any $(c, s) \leftarrow \mathsf{Com}(m)$, we have $\mathsf{ExtCom}(c) = m$;
- **Setup indistinguishability**: the official setup Setup and the new one Setup' generate indistinguishable global public parameters. We denote by $\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A})$ the advantage an adversary $\mathcal{A}$ can get in distinguishing the global public parameters generated by the two setup phases.

In the trapdoor setting (when Setup' is used), the adversary is not given the trapdoors but just oracle access to the equivocation and extraction capabilities:

- $\mathsf{GenEquivCommit}()$ generates $(c, \mathsf{eqk}) \xleftarrow{\$} \mathsf{SimCom}()$, stores $(c, \mathsf{eqk}) \in \Psi$, and outputs c;
- $\mathsf{OpenEquivCommit}(c, m)$ first looks whether $c \in \Omega$ in which case it outputs $\perp$, otherwise it searches for $(c, \cdot) \in \Psi$, retrieves the matching eqk, stores $c \in \Omega$, and outputs $s \leftarrow \mathsf{OpenCom}(c, \mathsf{eqk}, m)$. It outputs $\perp$ if no (c, eqk) was found in Ψ;
- $\mathsf{ExtractCommit}(c)$ first looks whether $(c, \cdot) \in \Psi$ in which case it outputs $\perp$, otherwise it outputs $m \leftarrow \mathsf{ExtCom}(c)$.

The list Ψ is to keep track of the fake commitments, to exclude extraction on them, and the list Ω is to guarantee one opening only for any fake commitment. And then, with unlimited access to these oracles, we still expect the hiding and the binding properties to hold. They can be more formally modeled by the two following properties that, together with the setup indistinguishability, imply both the basic hiding and binding properties (see [ABB+13] for more details):

- **(Strong) commitment equivocality indistinguishability**: the real commitment algorithms $\mathsf{Com}/\mathsf{Open}$ and the fake-commitment algorithms $\mathsf{SimCom}/\mathsf{OpenCom}$ generate indistinguishable commitments and opening values. For any adversary $\mathcal{A}$, we denote by $\mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A})$ its advantage in distinguishing the commitments and opening values generated by the two kinds of algorithms (even with unlimited access to the oracles $\mathsf{GenEquivCommit}$, $\mathsf{OpenEquivCommit}$, and $\mathsf{ExtractCommit}$).
- **(Strong) binding extractability**: one cannot fool the extractor, *i.e.* produce a commitment c and a valid opening s to a message $m \neq \mathsf{ExtCom}(c)$. For any adversary $\mathcal{A}$, we denote by $\mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A})$ its advantage in generating a commitment c that it can open in a different way than the extraction algorithm (even with unlimited access to the oracles $\mathsf{GenEquivCommit}$, $\mathsf{OpenEquivCommit}$, and $\mathsf{ExtractCommit}$).

When a commitment scheme satisfies setup indistinguishability, strong commitment equivocality indistinguishability, and strong binding extractability, which additionally imply the basic hiding and binding properties, we say it is *strongly secure*.

Instantiation. As shown in [ABB+13], UC-secure commitment schemes [CF01; Can01] are strongly secure, and so are enough for us. As a consequence, the UC-secure non-interactive constructions from [FLM11; ABB+13] fulfill all our requirements, in the standard model, with negligible advantages for any adversary, under the Decisional Diffie-Hellman assumption. However they will not be efficient enough for our purpose. On the other hand, the simple commitment scheme that commits m with large enough random coins r into $c = \mathcal{H}(m, r)$ is quite efficient and also fulfill all the above requirements, in the random oracle model [BR93]:

Proof. Given a hash function $\mathcal{H}$ onto $\{0,1\}^\lambda$, the commitment scheme is defined as follows:

- $\mathsf{Com}(m)$: Generate $r \xleftarrow{\$} \{0,1\}^{2\lambda}$ and output $(c \leftarrow \mathcal{H}(m, r), s \leftarrow (m, r))$;
- $\mathsf{Open}(c, s = (m, r))$: if $\mathcal{H}(s) = c$, return m, otherwise, return $\perp$.

In the random oracle model, this simple scheme is trivially computationally *binding* ($\mathcal{H}$ is collision-resistant) and statistically *hiding* (for a large $r \in \{0,1\}^{2\lambda}$, there is almost the same number of possible r —actually 2^λ— for any m, that would lead to the commitment c).

Additionally, we can use the *programmability* of the random oracle for *equivocality* and the list of query answer for *extractability*:

- $\mathsf{SimCom}()$: Return $c \xleftarrow{\$} \{0,1\}^\lambda$.
- $\mathsf{OpenCom}(c, \cdot, m)$: Generate $r \xleftarrow{\$} \{0,1\}^{2\lambda}$, set $\mathcal{H}(m, r) \leftarrow c$, and return r.
- $\mathsf{ExtCom}(c)$: Search in list of queries to $\mathcal{H}$ which one was output to c and return the first corresponding m. If no such query exist, return $\perp$.

By construction, we have the equivocality correctness, unless the value $\mathcal{H}(m, r)$ has already been asked, which is quite unlikely, since r is a fresh random. Extractability correctness is also ensured, as the recovered value m effectively commits to c. We also have perfect *setup indistinguishability*, so $\mathsf{Adv}_{\mathcal{H}}^{\mathsf{setup\text{-}ind}}(\mathcal{A}) = 0$ for any adversary $\mathcal{A}$. The only way to distinguish a fake commitment from a real commitment would be to ask (m, r) to the oracle before $\mathsf{OpenCom}$, hence to guess r. Therefore $\mathsf{Adv}_{\mathcal{H}}^{\mathsf{s\text{-}eq}}(\mathcal{A}) \leq q_{\mathcal{H}} \times 2^{-2\lambda}$, for any adversary $\mathcal{A}$ asking at most $q_{\mathcal{H}}$ oracle queries and the scheme has *strong commitment equivocability indistinguishability*.

Moreover, this scheme has *strong binding extractability*. Suppose an adversary $\mathcal{A}$ is able to produce a tuple (c, s) that breaks the strong binding extractability. Hence if $(m', r') \leftarrow \mathsf{ExtCom}(c)$, $\mathcal{H}(m, r) = \mathcal{H}(m', r')$. This means that there is a collision between true random values, which is bounded by the birthday paradox. Hence: $\mathsf{Adv}_{\mathcal{H}}^{\mathsf{s\text{-}binding}}(\mathcal{A}) < q_{\mathcal{H}}^2 \times 2^{-\lambda}$.

Therefore, this commitment scheme is *strongly secure* in the random oracle model. □

2.3.2 Authenticated encryption

Eventually, for *explicit* authentication of the players, we will sometimes make use of an authenticated encryption scheme [BN00] $\mathcal{ES} = (\mathsf{Enc}, \mathsf{Dec})$, where decryption should fail when the cipher text has not been properly generated under the appropriate key. This will

thus provide a kind of key confirmation, as usually done to achieve explicit authentication. However, some critical data will have to be sent, hence a simple MAC would not be enough, privacy of the content is important too.

For an authenticated encryption scheme, there are two main simulation-based security notions: The *semantic security*, aka *indistinguishability under chosen plaintext attacks* (IND-CPA), prevents any information being leaked about the plain texts, while the *integrity of cipher texts*, aka `INT-CTXT`, essentially says that no valid cipher text can be produced without the key. The definitions of the corresponding advantages $\mathsf{Adv}^{\mathtt{int\text{-}ctxt}}_{\mathcal{ES}}(\mathcal{A})$ and $\mathsf{Adv}^{\mathtt{ind\text{-}cpa}}_{\mathcal{ES}}(\mathcal{B})$, for any adversaries $\mathcal{A}, \mathcal{B}$ can be found in [BN00]. In addition, for adversaries $\mathcal{A}, \mathcal{B}$ there exists an adversary $\mathcal{C}$ for which we define $\mathsf{Adv}^{\mathtt{authenc}}_{\mathcal{ES}}(\mathcal{C}) = \max\{\mathsf{Adv}^{\mathtt{int\text{-}ctxt}}_{\mathcal{ES}}(\mathcal{A}), \mathsf{Adv}^{\mathtt{ind\text{-}cpa}}_{\mathcal{ES}}(\mathcal{B})\}$.

One simple way to achieve secure authenticated encryption is by using a generic Encrypt-then-MAC approach [BN00] or by using a dedicated scheme such as OCB [RBBK01].

2.3.3 Linear codes

A linear q-ary code of length n and rank k is a subspace C with dimension k of the vector space $\mathbb{F}_q^n$. The vectors in C are called codewords. The size of a code is the number of codewords, and is thus equal to q^k. The weight of a word $w \in \mathbb{F}_q^n$ is the number of non-zero components and the distance between two words is the Hamming distance between them (equivalently, the weight of their difference). The minimal distance d of a linear code C is the minimum weight of its non-zero codewords, or equivalently, the minimum distance between any two distinct codewords.

A code for an alphabet of size q, of length n, rank k, and minimal distance d is called a $(n, k, d)_q$-code. Such a code can be used to detect up to $d-1$ errors (if a codeword is sent and fewer than $d-1$ errors occur, it cannot get transformed into another codeword), or correct up to $\lfloor (d-1)/2 \rfloor$ errors (for any received word, there is a unique codeword within distance $\lfloor (d-1)/2 \rfloor$). For linear codes, encoding of a (row vector) word $W \in \mathbb{F}_q^k$ is performed by an algorithm $C.\mathsf{Encode} : \mathbb{F}_q^k \to \mathbb{F}_q^n$, which is the multiplication of W by a matrix $G \in \mathbb{F}_q^{k \times n}$, called the "generating matrix" (which defines an injective linear map). This leads to a row-vector codeword $c \in C \subset \mathbb{F}_q^n$.

We also denote $C.\mathsf{Decode} : \mathbb{F}_q^n \to \mathbb{F}_q^k$ the decoding algorithm for the correcting property mentioned before, that is $\forall W \in \mathbb{F}_q^k, \forall c' \in F_q^n$ and $c = C.\mathsf{Encode}(W)$, if c and c' differ by at most $\lfloor (d-1)/2 \rfloor$ coordinates, $C.\mathsf{Decode}(c') = W$. The Singleton bound states that for any linear code, $k + d \leq n + 1$, and a *maximum distance separable* (MDS) code satisfies $k + d = n + 1$. Hence, MDS codes are fully described by the parameters (q, n, k). Such a $(n, k)_q$-MDS code can correct up to $\lfloor (n-k)/2 \rfloor$ errors; it can detect if there are errors, whenever there are no more than $n - k$ of them.

For a thorough introduction to linear codes and proof of all statements in this short overview we refer the reader to [Rot06].

Observe that a linear code, due to the linearity of its encoding algorithm, is not a primitive designed to hide anything about the encoded message. However, we show in Lemma 2.3.5 that a MDS code can be turned into a *robust secret sharing* (RSS) scheme, which is designed just for that and is presented next.

2.3.4 Robust secret sharing

Notion. We recall the definition of a RSS, slightly simplified for our purposes from [CDD+15]. $\forall c \in \mathbb{F}_q^n, \forall A \subset [\![1, n]\!]$, we denote by $c_A = (c_i)_{i \in A}$ the projection of c into $\mathbb{F}_q^{|A|}$.

Definition 2.3.1. *Let $\mathbb{F}_q$ be a finite field. A (n, t, r) – RSS consists of two probabilistic algorithms* $\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n$ *and* $\mathsf{Reconstruct} : \mathbb{F}_q^n \to \mathbb{F}_q$ *with the following properties:*

- *t-privacy: $\forall s, s' \in \mathbb{F}_q, \forall A \subset [\![1, n]\!]$ with $|A| \leq t$, the restrictions c_A and c'_A of $c \leftarrow \mathsf{Share}(s)$ and $c' \leftarrow \mathsf{Share}(s')$ to the coordinates in A are identically distributed.*

- *r-robustness $\forall s \in \mathbb{F}_q, \forall A \subset [\![1, n]\!]$ with $|A| = r$, $c \leftarrow \mathsf{Share}(s)$, if $\tilde{c}$ is such that $\tilde{c}_A = c_A$ it holds that $\mathsf{Reconstruct}(\tilde{c}) = s$.*

In other words, a RSS is able to reconstruct the shared secret even if the adversary tampered with up to $n - r$ shares, while each set of t shares is distributed independently of the shared secret, and thus reveals nothing about it. This definition allows for a gap, i.e. $r \geq t + 1$ and RSS that have $r > t + 1$ are called in the literature *ramp* RSS.

Among other things, our definition of robustness above is slightly simplified from the one in [CDD+15] which allowed for a slight probability δ of not reconstructing the secret. However, since we are interested in settings where $r \leq n/3$, such a provision is not required. Indeed, we will construct a suitable RSS based on MDS linear codes, which has perfect robustness.

Remark 2.3.2. *In Definition 2.3.1, $\tilde{c}_{\bar{A}}$ — that is, the coordinates of $\tilde{c}$ that are not in A — is determined by the adversary. A slightly weaker notion of robustness could be constructed by randomly sampling it instead. Such a weaker notion would be enough for all purposes in this thesis but, for the sake of simplicity we will retain the notion defined above in the following.*

We now introduce a new RSS property called smoothness. Intuitively, it means that getting a set of shares with a small number of them correct and the others random leaks no information (computationally) on the shared secret. It can be considered either for any secret, or for a randomly chosen secret as well.

In the following, let $\bar{A}$ be the complement of A in $[\![1, n]\!]$, i.e. $\bar{A} = [\![1, n]\!]$.

Definition 2.3.3 (Smoothness)**.** *We say that an $(n, t, r)_q$-RSS is*

- m-smooth *if for any $s \in \mathbb{F}_q, A \subset [\![1, n]\!]$ with $|A| \leq m$, any c output by $\mathsf{Share}(s)$, and any $\tilde{c}$ such that $\tilde{c}_A = c_A$, $\tilde{c}_{\bar{A}} \xleftarrow{\$} \mathbb{F}_q^{n-|A|}$, for all PPT $\mathcal{A}$ it holds that*

$$|\Pr[1 \leftarrow \mathcal{A}(1^\lambda, \mathsf{Reconstruct}(\tilde{c}))] - \Pr[1 \leftarrow \mathcal{A}(1^\lambda, u)]|$$

  is negligible in λ, where the probability is taken over the random coins of $\mathcal{A}$ and $\mathsf{Reconstruct}$ *and $u \xleftarrow{\$} \mathbb{F}_q$.*

- m-smooth on random secrets *if it is m-smooth for randomly chosen $s \xleftarrow{\$} \mathbb{F}_q$ and the probabilities are additionally taken over the coins consumed by this choice.*

We also introduce strong t-privacy, a slightly stronger notion of privacy than the one from the origin RSS definition above.

Definition 2.3.4 (Strong t-privacy)**.** *We say that an $(n,t,r)_q$-RSS has* strong t-privacy*, if for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \leq t$, the projection c_A of $c \xleftarrow{\$} \mathsf{Share}(s)$ is distributed uniformly randomly in $\mathbb{F}_q^{|A|}$.*

Note that while strong t-privacy implies t-privacy, the opposite does not necessarily hold (imagine a Share algorithm creating shares that start with "I'm a share!"). Note that, in case of random errors occuring, as long as there are fewer then t undisturbed shares, a strong t-private scheme actually hides the locations (and with this also the number) of errors.

The following Lemma shows that we can construct RSS directly from MDS linear codes.

Lemma 2.3.5. *Let C be a $(n+1,k)_q$-MDS code. We set L to be the last column of the generating matrix G of the code C and we denote by C' the $(n,k)_q$-MDS code whose generating matrix G' is G without the last column. Let further algorithm* Decode *of the MDS code C' be of the following form:*

1. *On input a word $c \in \mathbb{F}_q^n$,* Decode *chooses $D \subseteq [n]$ with $|D| = k$.*
2. *Let G'_D denote the matrix obtained from G' by eliminating all columns with indices not in D.* Decode *now outputs $c_D \cdot G'^{-1}_D$.*

Let Share *and* Reconstruct *work as follows:*

- $\mathsf{Share}(s)$ *for $s \in \mathbb{F}_q$ first chooses a random row vector $W \in \mathbb{F}_q^k$ such that $W \cdot L = s$, and outputs $c \leftarrow C'.\mathsf{Encode}(W)$ (equivalently, we can say that $\mathsf{Share}(s)$ chooses a uniformly random codeword of C whose last coordinate is s, and outputs the first n coordinates as c).*
- $\mathsf{Reconstruct}(w)$ *for $w \in \mathbb{F}_q^n$ first runs $C'.\mathsf{Decode}(w)$. If it gets a vector W', then output $s = W' \cdot L$, otherwise output $s \xleftarrow{\$} \mathbb{F}_q$.*

Then Share *and* Reconstruct *form a t-smooth, strongly t-private $(n,t,r)_q$-RSS for $t = k-1$ and $r = \lceil (n+k)/2 \rceil$ that is $(r-1)$-smooth on random secrets.*

Proof. Let us consider the two properties from Definition 2.3.1.

- strong t-privacy: Assume $|A| = t$ (privacy for smaller A will follow immediately by adding arbitrary coordinates to it to get to size t). Let $J = A \cup \{n+1\}$; note that $|J| = t+1 = k$. Note that for the code C, any k coordinates of a codeword determine uniquely the input to Encode that produces this codeword (otherwise, there would be two codewords that agreed on k elements and thus had distance $n-k+1$, which is less than the minimum distance of C). Therefore, the mapping given by $\mathsf{Encode}_J : \mathbb{F}_q^k \to \mathbb{F}_q^{|J|}$ is bijective; thus coordinates in J are uniform when the input to Encode is uniform. The algorithm Share chooses the input to Encode uniformly subject to fixing the coordinate $n+1$ of the output. Therefore, the remaining coordinates (i.e., the coordinates in A) are uniform.
- r-robustness: Note that C has minimum distance $n-k+2$, and therefore C' has minimum distance $n-k+1$ (because dropping one coordinate reduces the distance by at most 1). Therefore, C' can correct $\lfloor (n-k)/2 \rfloor = n-r$ errors. Since $c_A = \tilde{c}_A$ and $|A| \geq r$, there are at most $n-r$ errors in $\tilde{c}$, so the call to $C'.\mathsf{Decode}(\tilde{c})$ made by $\mathsf{Reconstruct}(\tilde{c})$ will output $\tilde{W} = W$. Then $\mathsf{Reconstruct}(\tilde{c})$ will output $s = \tilde{W} \cdot L = W \cdot L$.

- t-smoothness: to prove this, we show that disturbing one share uniformly random already randomizes the output of Reconstruct. Let D denote the set chosen by Decode. Since every codeword is uniquely determined by k elements, the mappings $f_i : \mathbb{F}_q \to \mathbb{F}_q$, $x \mapsto \mathsf{Encode}(G'^{-1}_D(c))_{n+1}$ with $c \leftarrow \mathbb{F}_q^k, c_i = x$ are bijective for all $i \in [k]$. Since $t = k-1$, $\tilde{c}_D$ contains at least one entry that is chosen uniformly at random and thus the claim follows from the fact that the output of Reconstruct is computed as $\mathsf{Encode}(G'^{-1}_D(\cdot))$.

- $(r-1)$-smoothness on random secrets: first, it holds that $r-1 > k$ and thus $\tilde{c}$ contains more than k undisturbed shares. We distinguish two cases. Either D chosen by Decode contains only undisturbed shares (i.e., $D \subseteq A$), then Reconstruct will output s which is distributed uniformly random in $\mathbb{F}_q$. Else, $D \not\subseteq A$. In this case, at least one element of $\tilde{c}_D$ is distributed uniformly random and the randomness of the output of Reconstruct follows as in the proof of t-smoothness.

□

Note that the Shamir's secret-sharing scheme is exactly the above construction with Reed-Solomon codes [MS81].

2.3.5 Models as UC functionalities

We recall in this section several classical models in cryptography, viewed as UC ideal functionality. While they are not usually instantiable, which means that proofs based on those have less implications, they still give a good intuition as of to why a protocol ought to be secure. We will therefore make use of them in several security proofs.

Common reference string. The *common reference string* (CRS) functionality was defined in [Can07]. We recall it in Figure 2.3 for completeness. Note that we do not let $\mathcal{F}_{\mathsf{CRS}}$ check whether a party is allowed to obtain the CRS, but the latter can be assumed public.

The functionality $\mathcal{F}^{\mathcal{D}}_{\mathsf{CRS}}$ is parametrized with a distribution $\mathcal{D}$ and proceeds as follows:

- Upon receiving $(\mathsf{sid}, \mathtt{crs})$, if there is no value r recorded, then choose and record a value $r \xleftarrow{\$} \mathcal{D}$ and reply with (sid, r).

Figure 2.3: Functionality $\mathcal{F}_{\mathsf{CRS}}$

Random oracle. The *random oracle* (RO) functionality was defined by Hofheinz and Müller-Quade in [HM04]. We recall it in Figure 2.4 for completeness. It is clear that the random oracle model UC-emulates this functionality.

Ideal cipher. An *ideal cipher* (IC) [BPR00] is a block cipher that takes a plain text or a cipher text as input. We describe the IC functionality $\mathcal{F}_{\mathsf{IC}}$ in Figure 2.5, in the same vein as the above RO functionality. Notice that the ideal cipher model UC-emulates this functionality. Note that this functionality characterizes a perfectly random permutation, by ensuring injectivity for each query simulation.

The functionality $\mathcal{F}_{\mathsf{RO}}$ proceeds as follows, running on security parameter λ, with a set of (dummy) parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{\mathsf{RO}}$ keeps a list L (which is initially empty) of pairs of bit strings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from some party P_i or from $\mathcal{S}$, do:
  - If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^\lambda$ in the list L, set $h := \tilde{h}$.
  - If there is no such pair, choose uniformly $h \in \{0,1\}^\lambda$ and store the pair $(m, h) \in L$.

  Once h is set, reply to the activating machine (either P_i or $\mathcal{S}$) with (sid, h).

Figure 2.4: Functionality $\mathcal{F}_{\mathsf{RO}}$

The functionality $\mathcal{F}_{\mathsf{IC}}$ takes as input the security parameter κ, and interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $P_1, \ldots, P_n$ by means of these queries:

- $\mathcal{F}_{\mathsf{IC}}$ keeps an (initially empty) list L containing 3−tuples of bit strings and a number of (initially empty) sets C_{ek} and M_{ek}.
- **Upon receiving a query $(\mathsf{sid}, \mathcal{E}, \mathsf{ek}, m)$ (with $m \in \{0,1\}^\kappa$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{ek}, m, \tilde{c})$ for some $\tilde{c} \in \{0,1\}^\kappa$ in the list L, set $c := \tilde{c}$.
  - If there is no such record, choose uniformly c in $\{0,1\}^\kappa \backslash C_{\mathsf{ek}}$ which is the set consisting of cipher texts not already used with ek. Next, it stores the 3−tuple $(\mathsf{ek}, m, c) \in L$ and sets both $M_{\mathsf{ek}} \leftarrow M_{\mathsf{ek}} \cup \{m\}$ and $C_{\mathsf{ek}} \leftarrow C_{\mathsf{ek}} \cup \{c\}$.

  Once c is set, reply to the activating machine with (sid, c).
- **Upon receiving a query $(\mathsf{sid}, \mathcal{D}, \mathsf{ek}, c)$ (with $c \in \{0,1\}^\kappa$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{ek}, \tilde{m}, c)$ for some $\tilde{m} \in \{0,1\}^\kappa$ in L, set $m := \tilde{m}$.
  - If there is no such record, choose uniformly m in $\{0,1\}^\kappa \backslash M_{\mathsf{ek}}$ which is the set consisting of plain texts not already used with ek. Next, it stores the 3−tuple $(\mathsf{ek}, m, c) \in L$ and sets both $M_{\mathsf{ek}} \leftarrow M_{\mathsf{ek}} \cup \{m\}$ and $C_{\mathsf{ek}} \leftarrow C_{\mathsf{ek}} \cup \{c\}$.

  Once m is set, reply to the activating machine with (sid, m).

Figure 2.5: Functionality $\mathcal{F}_{\mathsf{IC}}$

Chapter 3

Implicit-only PAKE

3.1 Motivation

Password authenticated key exchange (PAKE) is a cryptographic primitive that aims at allowing two parties sharing a low-entropy secret (the *password*) to interactively establish a high-entropy random session key. Two types of authentication have been considered: implicit authentication, where at the end of the protocol the two parties share the same key if they used the same password and random independent keys otherwise; or explicit authentication where, in addition, they actually know which of the two situations happened. A PAKE protocol that only achieves implicit authentication can be enhanced to have explicit authentication by adding key-confirmation flows [BPR00]. For more details about this primitive, one can refer to Section 2.2 of Chapter 2.

Its standard *universal composability* (UC) ideal functionality $\mathcal{F}_{\mathsf{pake}}$ based on [CHK+05] (see Figure 2.2), while only achieving implicit authentication, was designed with explicit authentication in mind. Thus, it allows the adversary (but not the player) to know whether a password guess attempt was successful or not. In most settings this is reasonable, because success or failure can anyway be determined by trying to use the key established by the PAKE. However, some applications, such as the one we design in the next chapter, could make use of a PAKE that does not internally provide any feedback, even to the adversary.

3.2 Definition

Implicit-only. Hence, we introduce a new notion, called *implicit-only password authenticated key exchange* (iPAKE), as a stronger notion of PAKE. Its ideal functionality, presented in Figure 3.1, models the complete absence of feedback: the two players get back the session keys, but neither them nor the adversary can know if the protocol succeeded. Of course, in many cases, the players can later check whether the keys match or not, and so whether the passwords were the same or not, but this would actually depend on the protocol using the keys. Hence, we stress that this is not a leakage from the iPAKE protocol itself, but from the global system.

In terms of functionalities, there is only one relatively minor differences from $\mathcal{F}_{\mathsf{pake}}$ to $\mathcal{F}_{\mathsf{iPAKE}}$, namely the fact that `TestPwd`-query now silently updates the internal state of the record, meaning that its outcome is not given to the adversary $\mathcal{S}$ as before. We stress that a `NewKey`-query can only be asked for a player $\mathcal{P}_i$ that has previously issued a `NewSession`-

The functionality $\mathcal{F}_{\mathsf{iPAKE}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i)$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i)$ to $\mathcal{S}$;
  - If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ and mark this record `fresh`.
- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_i)$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$, then do:
  - If $\mathsf{pw}_i = \mathsf{pw}'_i$, mark the record `compromised`;
  - If $\mathsf{pw}_i \neq \mathsf{pw}'_i$, mark the record `interrupted`.
- **Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is no record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ or this is not the first `NewKey` query for $\mathcal{P}_i$, ignore this query. Otherwise:
  - If the record is `compromised`, $\mathcal{P}_i$ is corrupted or both $\mathcal{P}_{1-i}$ is corrupted and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \mathsf{sk})$ to player $\mathcal{P}_i$.
  - If the record is `fresh`, both parties are honest and a key sk' was sent to $\mathcal{P}_{1-i}$, at which time there was a `fresh` record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
  - In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.

  Either way, mark the record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 3.1: Functionality $\mathcal{F}_{\mathsf{iPAKE}}$

query. If its partner $\mathcal{P}_{1-i}$ is corrupted, either a `NewSession`-query has been sent for it, which means that $\mathcal{P}_{1-i}$ got adaptively corrupted afterwards, and only matching passwords could provide it some control on the session key computed by $\mathcal{P}_i$, or no `NewSession`-query has been issued for $\mathcal{P}_{1-i}$ then only a `TestPwd`-query can impact the key received by $\mathcal{P}_{1-i}$. Without a `TestPwd`-query, in the latter case, $\mathcal{P}_{1-i}$ receives a random session key.

Labeled implicit-only. We can also slightly alter this functionality to allow for public labels in Figure 3.2. We call this new functionality *labeled implicit-only password authenticated key exchange* (liPAKE), resembling the notion of labeled public-key encryption as formalized in [Sho01]. In a nutshell, labels are public authenticated strings that are chosen by each user individually for each execution of the protocol. Here authenticated is relative to the authentication of the PAKE, meaning that tampering can be efficiently detected by the other user. i.e. if a user chooses label ℓ, it is required for the authentication to succeed (which, in the context of a iPAKE, only means that the final keys are equal) that the other user actually receives ℓ. In the next chapters, those labels will be used to distribute public information (such as public keys) reliably over an unauthenticated channel.

The functionality $\mathcal{F}_{\mathsf{liPAKE}}$ is parameterized by a security parameter λ and makes use of two initially empty lists $\Lambda_{\mathcal{P}}$ and $\Lambda_{\mathcal{L}}$, storing passwords and labels, respectively. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \ell)$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \ell)$ to $\mathcal{S}$;
  - If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$:
    - Record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$ and mark this record `fresh`.
    - Unless there exists a record $(\mathsf{sid}, \mathcal{P}_i, \cdot)$, record $(\mathsf{sid}, \mathcal{P}_i, \ell)$ in $\Lambda_{\mathcal{L}}$.

- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_i, \ell')$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, then do:
  - If $\mathsf{pw}_i = \mathsf{pw}'_i$, mark the record `compromised`; else mark it `interrupted`;
  - Remove any previously existing record of the form $(\mathsf{sid}, \mathcal{P}_{1-i}, \cdot)$ and store $(\mathsf{sid}, \mathcal{P}_{1-i}, \ell')$ in $\Lambda_{\mathcal{L}}$.

- **Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \ell)$ in $\Lambda_{\mathcal{L}}$, extract ℓ from it; otherwise set $\ell \leftarrow \perp$
  If there is no record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$ or this is not the first `NewKey` query for $\mathcal{P}_i$, ignore this query. Otherwise:
  - If the record is `compromised`, $\mathcal{P}_i$ is corrupted or both $\mathcal{P}_{1-i}$ is corrupted and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \ell, \mathsf{sk})$ to player $\mathcal{P}_i$.
  - If the record is `fresh`, both parties are honest and a key sk' was sent to $\mathcal{P}_{1-i}$, at which time there was a `fresh` record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, then output $(\mathsf{sid}, \ell, \mathsf{sk}')$ to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \ell, \mathsf{sk}')$ to $\mathcal{P}_i$.

  Either way, mark the record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 3.2: Functionality $\mathcal{F}_{\mathsf{liPAKE}}$

In the next section, we will present a protocol (Figure 3.3) and prove it is a UC-secure liPAKE.

3.3 Instantiation

In the seminal paper by Bellovin and Merritt [BM92] that introduced PAKE, they propose the Encrypted Key Exchange protocol (EKE), which is essentially a Diffie-Hellman [DH76] key exchange, with the two flows encrypted under the password with an appropriate symmetric encryption scheme. This EKE protocol has been formalized by Bellare et al. [BPR00] under EKE2. We present its labeled variant in Figure 3.3. The idea of appending the label to the

symmetric key is taken from [ACCP08].

$A(\mathsf{pw} \in \mathbb{P}, \ell \in \mathcal{L})$		$B(\mathsf{pw}' \in \mathbb{P}, \ell' \in \mathcal{L})$
$x \xleftarrow{\$} \mathbb{Z}_q,\ X \leftarrow g^x$		
$X^* \leftarrow \mathcal{E}_{\mathsf{pw}\|\|\ell}(X)$	$\xrightarrow{\ell, X^*}$	$y \xleftarrow{\$} \mathbb{Z}_q,\ Y \leftarrow g^y$
	$\xleftarrow{\ell', Y^*}$	$Y^* \leftarrow \mathcal{E}_{\mathsf{pw}'\|\|\ell'}(Y)$
$Z \leftarrow \mathcal{D}_{\mathsf{pw}\|\|\ell'}(Y^*)^x$		$Z' \leftarrow \mathcal{D}_{\mathsf{pw}'\|\|\ell}(X^*)^y$
$sk \leftarrow H(X^*, Y^*, Z)$		$sk' \leftarrow H(X^*, Y^*, Z')$
output (ℓ', sk)		output (ℓ, sk')

Figure 3.3: Protocol EKE2, in a group $\mathbb{G} = \langle g \rangle$ of prime order q, with a hash function $H : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \mathbb{G} \to \{0,1\}^\lambda$ and a symmetric cipher $\mathcal{E} : \mathbb{G} \to \{0,1\}^\kappa, \mathcal{D} : \{0,1\}^* \to \mathbb{G}$ for keys in $\mathbb{P} \times \mathcal{L}$.

The following theorem proves security in the $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{IC}}, \mathcal{F}_{\mathsf{CRS}}$-hybrid model, meaning that we use an ideal random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ as the hash function, an ideal cipher functionality $\mathcal{F}_{\mathsf{IC}}$ to model the encryption scheme and assume a publicly available common reference string modeled by its functionality $\mathcal{F}_{\mathsf{CRS}}$.

Theorem 3.3.1. *If the computationnal Diffie-Hellman (CDH) assumption holds in $\mathbb{G}$, the protocol EKE2 securely realizes $\mathcal{F}_{\mathsf{liPAKE}}$ in the $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{IC}}, \mathcal{F}_{\mathsf{CRS}}$-hybrid model with respect to static corruptions.*

We note that this result is not surprising given that other variants of EKE2 have already been proven to UC-emulate $\mathcal{F}_{\mathsf{pake}}$. Intuitively, a protocol with only two flows not depending on each other does not leak the outcome to the adversary via the transcript, which explains why EKE2 is implicit-only. Hashing of the transcript keeps the adversary from biasing the key unless he knows the correct password or breaks the ideal cipher. For completeness, we include the full proof below.

Proof. We proceed in a serie of games, starting with the real execution of the protocol and ending up with the ideal execution, with a simulator. For convenience, we refer to a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ from the adversary $\mathcal{S}$ as *due* when:

- $\mathcal{P}_i$ is honest
- there is a `fresh` record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$
- this is the first `NewKey` query for $\mathcal{P}_i$
- there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$
- a key sk' was sent to the other party, and $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time.

Game $\mathbf{G}_0$: The real protocol execution. This is the real execution where the environment $\mathcal{Z}$ runs the EKE2 protocol (see Figure 3.4) with parties $\mathcal{P}_0$ and $\mathcal{P}_1$, both having access to ideal *common reference string* (CRS), *random oracle* (RO), and *ideal cipher* (IC) functionalities, and an adversary $\mathcal{A}$ that, w.l.o.g., is assumed to be the dummy adversary as shown in [Can01, section 4.4.1].

The parties $\mathcal{P}_0$ and $\mathcal{P}_1$ are running with $\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{IC}}$.
Protocol Steps:

1. When a party $\mathcal{P}_i$ receives an input $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i, \ell)$ from $\mathcal{Z}$, it does the following:
   - If $\mathcal{P}_i = \mathcal{P}_1$, it does nothing and waits for a message from $\mathcal{P}_{1-i}$
   - If $\mathcal{P}_i = \mathcal{P}_0$, $\mathcal{P}_i$ first chooses $x \xleftarrow{\$} \mathbb{Z}_q$. $\mathcal{P}_i$ then
     - sends $(\mathsf{sid}, \texttt{crs})$ to $\mathcal{F}_{\mathsf{CRS}}$ and receives $(\mathsf{sid}, (g, q))$ back
     - sends $(\mathsf{sid}, \mathcal{E}, \mathsf{pw}_i || \ell, g^x)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, X^*) back
     - sends $(\mathsf{sid}, \ell, X^*)$ to $\mathcal{P}_{1-i}$ and waits for an answer
2. When $\mathcal{P}_1$, who already obtained an input $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_1, \mathsf{pw}_1, \ell')$, receives a message $(\mathsf{sid}, \ell, X^*)$ from $\mathcal{P}_0$, it first chooses $y \xleftarrow{\$} \mathbb{Z}_q$. $\mathcal{P}_1$ then
   - sends $(\mathsf{sid}, \texttt{crs})$ to $\mathcal{F}_{\mathsf{CRS}}$ and receives $(\mathsf{sid}, (g, q))$ back
   - sends $(\mathsf{sid}, \mathcal{E}, \mathsf{pw}_1 || \ell', g^y)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, Y^*) back
   - sends $(\mathsf{sid}, \ell', Y^*)$ to $\mathcal{P}_0$
   - sends $(\mathsf{sid}, \mathcal{D}, \mathsf{pw}_1 || \ell, X^*)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, X') back
   - sends $(\mathsf{sid}, X^*, Y^*, X'^y)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives $(\mathsf{sid}, \mathsf{sk})$ back

   $\mathcal{P}_1$ then outputs $(\mathsf{sid}, \ell, \mathsf{sk})$ towards $\mathcal{Z}$ and terminates the session.
3. When $\mathcal{P}_0$ obtains an answer $(\mathsf{sid}, \ell', Y^*)$ from $\mathcal{P}_1$, it
   - sends $(\mathsf{sid}, \mathcal{D}, \mathsf{pw}_0 || \ell', Y^*)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, Y) back
   - sends $(\mathsf{sid}, X^*, Y^*, Y^x)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives $(\mathsf{sid}, \mathsf{sk}')$ back

   $\mathcal{P}_0$ then outputs $(\mathsf{sid}, \ell', \mathsf{sk}')$ towards $\mathcal{Z}$ and terminates the session.

Figure 3.4: An UC Execution of EKE2

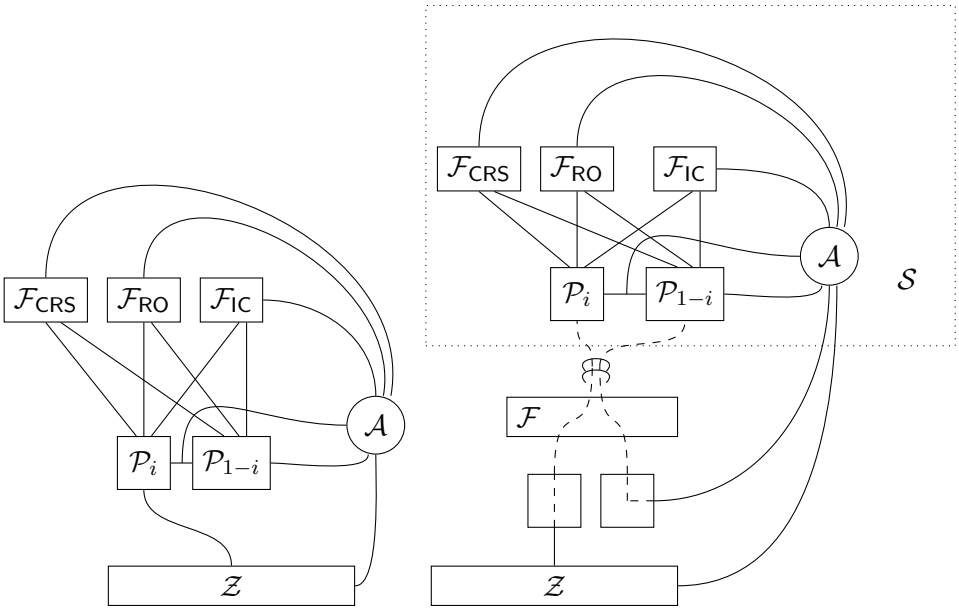

Figure 3.5: Transition from Game $\mathbf{G}_0$ (left) to Game $\mathbf{G}_1$ (right), showing a setting where $\mathcal{P}_{1-i}$ is corrupted.

Game $\mathbf{G}_1$: Modeling the ideal layout. We first make some purely conceptual changes that do not modify the input/output interfaces of $\mathcal{Z}$. We add one relay (also referred to as the *dummy party*) on each of the wires between $\mathcal{Z}$ and a party. We also add one relay covering all the wires between the dummy parties and real parties and call it $\mathcal{F}$ (and let $\mathcal{F}$ relay messages according to the original wires). We group all the formerly existing instances except for $\mathcal{Z}$ into one machine and call it $\mathcal{S}$. Note that this implies that $\mathcal{S}$ executes the code of the CRS, RO and IC functionalities as depicted in Figures 2.3, 2.4, and 2.5.

Game $\mathbf{G}_2$: Simulating the ideal functionalities. We modify simulation of $\mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{IC}}$ as follows. We let $\mathcal{S}$ implement Figure 2.5 by maintaining a list Λ_{IC} with entries of the form $(\mathsf{sid}, k, m, \alpha, \mathcal{E}|\mathcal{D}, c)$. $\mathcal{S}$ handles encryption and decryption queries as follows:

- Upon receiving $(\mathsf{sid}, \mathcal{E}, \mathsf{ek}, m)$ (for shortness of notation, we will also write $\mathcal{E}_{\mathsf{ek}}(m)$ for this query) if $\mathsf{ek} \notin \mathbb{P} \times \mathcal{L}$ or $m \notin \mathbb{G}$ then abort. Else, if there is an entry $(\mathsf{sid}, \mathsf{ek}, m, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ replies with (sid, c). Else, $\mathcal{S}$ chooses $c \xleftarrow{\$} \{0,1\}^\kappa$. If there is already a record $(*, *, *, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds $(\mathsf{sid}, \mathsf{sk}, m, \perp, \mathcal{E}, c)$ to Λ_{IC} and replies with (sid, c).
- Upon receiving $(\mathsf{sid}, \mathcal{D}, \mathsf{ek}, c)$ (or $\mathcal{D}_{\mathsf{ek}}(c)$, for short), if $\mathsf{ek} \notin \mathbb{P} \times \mathcal{L}$ or $c \notin \{0,1\}^\kappa$ then abort. Else, if there is an entry $(\mathsf{sid}, \mathsf{ek}, m, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ replies with (sid, m). Else, $\mathcal{S}$ chooses $\alpha \leftarrow \mathbb{Z}_q$. If there is already a record $(*, *, g^\alpha, *, *, *)$ in Λ_{IC}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds $(\mathsf{sid}, \mathsf{sk}, g^\alpha, \alpha, \mathcal{D}, c)$ to Λ_{IC} and replies with (sid, g^α).

Similarly, let Λ_{RO} denote the list that $\mathcal{S}$ maintains upon implementing Figure 2.4, containing entries of the form (sid, m, h). We let $\mathcal{S}$ handle queries to $\mathcal{F}_{\mathsf{RO}}$ as follows:

- Upon receiving $H(m)$, if $m \notin \{0,1\}^\kappa \times \{0,1\}^\kappa \times \mathbb{G}$, then abort. Else, if there is

an entry (sid, m, h) in Λ_{RO}, $\mathcal{S}$ replies with (sid, h). Else, $\mathcal{S}$ chooses $h \xleftarrow{\$} \{0,1\}^\lambda$. If there is already a record $(*, *, h)$ in Λ_{RO}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds (sid, m, h) to Λ_{RO} and replies with (sid, h).

We note that these modifications later allow $\mathcal{S}$ to extract unique inputs from values obtained using the two functionalities. In particular, note that Λ_{IC} will never contain $(\mathsf{sid}, \mathsf{ek}, *, *, \mathcal{E}, c), (\mathsf{sid}', \mathsf{ek}', *, *, \mathcal{E}, c)$ with $\mathsf{ek} \neq \mathsf{ek}'$. The entry α serves $\mathcal{S}$ as a trapdoor for solving discrete-log type problems.

Since q is greater than 2^λ, if the oracles are only queried a polynomial number of times, the birthday problem states that Game $\mathbf{G}_1$ and Game $\mathbf{G}_2$ are indistinguishable with probability overwhelming in λ.

Game $\mathbf{G}_3$: Building $\mathcal{F}_{\mathsf{iPAKE}}$. In this game, we start modeling $\mathcal{F}_{\mathsf{liPAKE}}$. First, we let $\mathcal{F}$ maintain two initially empty lists: $\Lambda_{\mathcal{P}}$, a list of tuples of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ and $\Lambda_{\mathcal{L}}$, a list of tuples of the form $(\mathsf{sid}, \mathcal{P}_i, \ell)$. Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \ell)$ from (dummy) party $\mathcal{P}_i$, if this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i}, \ell')$, then $\mathcal{F}$ records $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$ and marks this record as `fresh`. If $\Lambda_{\mathcal{L}}$ does not contain any record $(\mathsf{sid}, \mathcal{P}_i, \cdot)$ so far, $\mathcal{F}$ also records $(\mathsf{sid}, \mathcal{P}_i, \ell)$ in $\Lambda_{\mathcal{L}}$. Then, $\mathcal{F}$ relays the query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \ell)$ to $\mathcal{S}$. Now that $\mathcal{F}$ knows about passwords and labels, we can add an `TestPwd` interface to $\mathcal{F}$ as described in Figure 3.1. We let $\mathcal{S}$ parse outputs $(\mathsf{sid}, \ell', \mathsf{sk})$ towards $\mathcal{F}$ to be of the form $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ by adding the `NewKey` tag and the name of the party who produced the output. Additionally, we let $\mathcal{F}$ translate this back to $(\mathsf{sid}, \ell', \mathsf{sk})$ and send it to $\mathcal{Z}$ via the dummy party $\mathcal{P}_i$.

Obviously none of these modifications change the output towards $\mathcal{Z}$ compared to the previous Game $\mathbf{G}_2$.

Game $\mathbf{G}_4$: $\mathcal{F}$ generates a random session key for an honest, interrupted session. Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, if $\mathcal{P}_i$ is not corrupted and there is a record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw})$ that is marked as `interrupted`, and this is the first `NewKey` query for $\mathcal{P}_i$, we let $\mathcal{F}$ choose a random session key sk^* of length λ. Additionally, $\mathcal{F}$ derives the label as follows: if there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$, extract ℓ^* from it; otherwise, set $\ell^* \leftarrow \perp$. Then, $\mathcal{F}$ outputs $(\mathsf{sid}, \ell^*, \mathsf{sk}^*)$ to P.

If there is no such interrupted record, $\mathcal{F}$ continues to relay sk and ℓ'.

Since the simulators described in Game $\mathbf{G}_3$ and Game $\mathbf{G}_4$ do not make use of the `TestPwd` interface, none of the records of $\mathcal{F}$ are marked as `interrupted` and thus the output towards $\mathcal{Z}$ is equally distributed in both games.

Game $\mathbf{G}_5$: $\mathcal{S}$ handles dictionary attacks against the client $\mathcal{P}_0$ using the `TestPwd` interface. If both $\mathcal{P}_0$ and $\mathcal{P}_1$ are honest, $\mathcal{P}_0$ obtained input and $\mathcal{Z}$ advises $\mathcal{A}$ to substitute $(\mathsf{sid}, \ell', Y^*)$ with $(\mathsf{sid}, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$, or if $\mathcal{P}_1$ is corrupted and produces $(\mathsf{sid}, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$ as first flow, then $\mathcal{S}$ will proceed with the simulation of $\mathcal{P}_0$ using $\ell_{\mathcal{Z}}$ and $Y^*_{\mathcal{Z}}$.

In this situation, we modify $\mathcal{S}$ as follows: upon receiving $(\mathsf{sid}, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$ if there is an entry[1] $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}} || \hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$, for any $\hat{\ell} \in \mathcal{L}$ in Λ_{IC}, $\mathcal{S}$ asks a `TestPwd` query

[1] This entry is unique due to the simulation of $\mathcal{F}_{\mathsf{IC}}$ as described in Game $\mathbf{G}_2$.

$(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_0, \mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}$. $\mathcal{S}$ then proceeds the simulation[2] using $\mathsf{pw}_{\mathcal{Z}}$ and $\ell_{\mathcal{Z}}$ instead of pw and ℓ'. If there is no entry $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}, $\mathcal{S}$ sends $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_0, \mathsf{pw}_0, \ell_{\mathcal{Z}})$ to $\mathcal{F}$.

Regarding the label, observe that $\mathcal{S}$'s `NewKey` query will contain $\ell_{\mathcal{Z}}$ which was contained in the output of the honest $\mathcal{P}_0$ (cf. Figure 3.4). Since `TestPwd` queries overwrite any existing labels, there will be an entry $(\mathsf{sid}, \mathcal{P}_1, \ell_{\mathcal{Z}})$ in $\Lambda_{\mathcal{L}}$ and thus, regarding the label, the output towards $\mathcal{Z}$ does not change compared to the previous game. Regarding the session key, we have to analyze different cases depending on whether $Y^*_{\mathcal{Z}}$ was generated using $\mathcal{F}_{\mathsf{IC}}$ or not. However, observe that the only changes of session keys between this and the previous game occur whenever a `TestPwd` query of $\mathcal{S}$ causes a record to be marked as `interrupted`.

- There is an entry $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}: if $\mathsf{pw}_{\mathcal{Z}} = \mathsf{pw}_0$, the record is marked `compromised` and the session key is not changed by $\mathcal{F}$. If $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$, on the other hand, the record is marked `interrupted` and $\mathcal{F}$ hands out a random session key, as opposed to Game $\mathbf{G}_4$. However, since the session key is distributed as before, $\mathcal{Z}$ can only detect this by reproducing $\mathcal{P}_0$'s input $(\mathsf{sid}, X^*, Y^*_{\mathcal{Z}}, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_0||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_0||\ell_{\mathcal{Z}}}(Y^*_{\mathcal{Z}}))$ to $\mathcal{F}_{\mathsf{RO}}$. Lemma 3.3.2 (see below) shows indistinguishability of Game $\mathbf{G}_4$ and Game $\mathbf{G}_5$.
- There is no entry $(\mathsf{sid}, *, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}: since the `TestPwd` query will result in a compromised record, the modified simulation has no impact on the output towards $\mathcal{Z}$ in this case.

The following lemma bounds the probability that an unsuccessful dictionary attack leads to a non-random looking session key. Since in this case the labels do not play any role (the encryption keys of the form *password*||*label* will not match regardless of the labels), we ignore them for the sake of simplicity.

Lemma 3.3.2. *If CDH holds in $\mathbb{G}$, then $\forall(\mathsf{pw}_0, \ell) \in \mathbb{P} \times \mathcal{L}, Y^*_{\mathcal{Z}} \leftarrow \mathcal{Z}$, where $Y^*_{\mathcal{Z}}$ is a cipher text generated through $\mathcal{F}_{IC}$ with some key $(\mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}}) \neq (\mathsf{pw}_0, \ell)$, it holds that*

$$\Pr_{\mathbf{G}_5}[\mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_0||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_0||\ell}(Y^*_{\mathcal{Z}})) \leftarrow \mathcal{Z}(X^*)] = negl(\lambda).$$

Proof. We create an attacker $\mathcal{B}_{\mathsf{CDH}}$ given a CDH instance $(g, A = g^a, B = g^b)$. $\mathcal{B}_{\mathsf{CDH}}$ runs $\mathcal{Z}$ simulating Game $\mathbf{G}_5$ as follows: $\mathcal{B}_{\mathsf{CDH}}$ internally runs all of the participating machines, i.e. $\mathcal{S}$, $\mathcal{F}$ and the dummy parties as in Game $\mathbf{G}_5$, but with some modifications. First, $\mathcal{B}_{\mathsf{CDH}}$ computes $X^* \leftarrow \mathcal{E}_{\mathsf{pw}_0||\ell}(A)$ and updates Λ_{IC} accordingly, aborting if there was already an entry $(\mathsf{sid}, \mathsf{pw}_0||\ell, A, *, \mathcal{E}, *)$. Upon receiving a query $\mathcal{D}_{\mathsf{pw}_0||\ell}(Y^*_{\mathcal{Z}})$, $\mathcal{B}_{\mathsf{CDH}}$ again aborts if there is already an entry $(\mathsf{sid}, \mathsf{pw}_0||\ell, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$. Otherwise, it draws $\beta \xleftarrow{\$} \mathbb{Z}_q$ and sets the answer to this query to be Bg^β. This can happen multiple times (for different $Y^*_{\mathcal{Z}}$), and $\mathcal{B}_{\mathsf{CDH}}$ keeps track of the pairs $(\beta, Y^*_{\mathcal{Z}})$ in a list Λ_{CDH}. The last modification concerns the part of the simulator's code of Game $\mathbf{G}_5$ where a value $Z \leftarrow \mathcal{D}_{\mathsf{pw}_0||\ell}(Y^*)^a$ needs to be computed, but note that $\mathcal{B}_{\mathsf{CDH}}$ does not know a. Instead, $\mathcal{B}_{\mathsf{CDH}}$ just sets $Z \leftarrow \perp$.

Finally, $\mathcal{B}_{\mathsf{CDH}}$ picks a random entry from Λ_{RO} asked by $\mathcal{Z}$, parses it into the tuple $(\mathsf{sid}, \mathsf{pw}_0||\ell, X^*, Y^*_{\mathcal{Z}}, Z), h)$, looks for an entry $(\beta, Y^*_{\mathcal{Z}})$ in Λ_{CDH} and outputs $Z/(g^a)^\beta$ as a CDH solution.

[2] Note that, since $\mathcal{F}$ does not leak any information at this point, $\mathcal{S}$ cannot depend on the outcome of a `TestPwd` query.

First, note that if $\mathcal{B}_{\mathsf{CDH}}$ does not abort, it perfectly emulates $\mathcal{Z}$'s view in Game $\mathbf{G}_5$, since A, Bg^{β} are random in $\mathbb{G}$ and the record for $\mathcal{P}_0$ will be interrupted, which means that $\mathcal{F}$ will output a random session key for $\mathcal{P}_0$, overwriting $\perp$. However, it is obvious that $\mathcal{B}_{\mathsf{CDH}}$ only has to abort if there is a collision upon choosing random values from $\mathbb{G}$.

Assume that $\mathcal{Z}$ outputs $\mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_0||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_0||\ell}(Y^*_{\mathcal{Z}}))$ with non-negligible probability. This is only possible if $\mathcal{Z}$ asked both corresponding decryption queries. Existence of $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\ell_{\mathcal{Z}}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ with $(\mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}}) \neq (\mathsf{pw}_0, \ell)$ in Λ_{IC} ensures that the answer to $\mathcal{D}_{\mathsf{pw}_0||\ell}(Y^*_{\mathcal{Z}})$ can be chosen by $\mathcal{B}_{\mathsf{CDH}}$ as described above. Thus, $\mathcal{B}_{\mathsf{CDH}}$ finds a correct CDH solution with probability $1/q_{\mathcal{Z}}$, where $q_{\mathcal{Z}}$ is the number of hash queries issued by $\mathcal{Z}$. □

Game $\mathbf{G}_6$: $\mathcal{S}$ handles dictionary attacks against the server $\mathcal{P}_1$ using the `TestPwd` interface. Analogously to Game $\mathbf{G}_5$, we let $\mathcal{S}$ use the `TestPwd` interface upon receiving adversarially generated $X^*_{\mathcal{Z}}, \ell_{\mathcal{Z}}$ upon simulating $\mathcal{P}_1$. Observe that the only difference is due to the order of flows: if $\mathcal{S}$ extracts an incorrect password, he produces Y^* using this wrong password. However, Y^* will be distributed as before and again, $\mathcal{Z}$ can only detect the change by reproducing $\mathcal{P}_1$'s input to $\mathcal{F}_{\mathsf{RO}}$, namely $(\mathsf{sid}, X^*_{\mathcal{Z}}, Y^*, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_1||\ell_{\mathcal{Z}}}(X^*_{\mathcal{Z}}), \mathcal{D}_{\mathsf{pw}_1||\ell'}(Y^*))$.

Using an analogous argument to Lemma 3.3.2, indistinguishability from Game $\mathbf{G}_5$ follows from the hardness of CDH in $\mathbb{G}$.

Game $\mathbf{G}_7$: $\mathcal{F}$ aligns session keys. Upon receiving a query $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ from $\mathcal{S}$ for a session where none of the players is corrupted, if this query is *due* then output $(\mathsf{sid}, \ell^*, \mathsf{sk}^*)$ to $\mathcal{P}_i$ where sk^* is the session key that was formerly sent to the other party and the label ℓ^* is derived as usual: if there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$, extract ℓ^* from it; otherwise, set $\ell^* \leftarrow \perp$.

We now analyze distinguishability of this game from Game $\mathbf{G}_6$. If $\mathcal{Z}$ tampered with the transcript, any player that received a modified message will not have a fresh record anymore (cf. simulation described in games $\mathbf{G}_5$ and $\mathbf{G}_6$) and the output of this player towards $\mathcal{Z}$ is not changed in this game. On the other hand, if $\mathcal{Z}$ does not advise $\mathcal{A}$ to tamper with any message, $\mathcal{F}$ did not overwrite any labels and thus $\ell^* = \ell'$. Additionally, perfect correctness of the EKE2 protocol ensures that in case of a due record $\mathsf{sk} = \mathsf{sk}^*$.

Note that $\mathcal{F}$ still differs from the functionality $\mathcal{F}_{\mathsf{liPAKE}}$ described in Figure 3.1 in some aspects. First, it does not output randomly generated session keys towards $\mathcal{Z}$ for honest sessions. Furthermore, it reports all passwords to $\mathcal{S}$. We will take care of these remaining differences in the next games.

Game $\mathbf{G}_8$: In some cases, $\mathcal{F}$ generates a random session key when the other party is corrupted. Upon receiving a `NewKey` query $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, and this is the first `NewKey` query for $\mathcal{P}_i$, $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ corrupted and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i \neq \mathsf{pw}_{1-i}$, we let $\mathcal{F}$ pick a new random key sk^* of length λ and send $(\mathsf{sid}, \ell^*, \mathsf{sk}^*)$ to $\mathcal{P}_i$, where ℓ^*, as usual, is taken from the list $\Lambda_{\mathcal{L}}$ or set to be $\perp$.

The simulation ensures that the record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ is either compromised or interrupted (cf. description of the simulator in games $\mathbf{G}_5$ and $\mathbf{G}_6$). Thus, the modification has no effect since it only concerns fresh records.

Game $\mathbf{G}_9$: $\mathcal{F}$ generates a random session key for an honest session. Upon receiving a `NewKey` query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, and this is the first `NewKey` query for $\mathcal{P}_i$, both parties are honest and the `NewKey` query is not *due*, we let $\mathcal{F}$ pick a new random key sk^* of length λ and send $(\mathsf{sid}, \ell^*, \mathsf{sk}^*)$ to $\mathcal{P}_i$, where ℓ^*, as usual, is taken from the list $\Lambda_{\mathcal{L}}$ or set to be $\perp$.

In other words, $\mathcal{F}$ now generates a random session key upon a first `NewKey` query for an honest party $\mathcal{P}_i$ with fresh record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ where the other party is also honest if (at least) one of the following events happens:

1. There is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i \neq \mathsf{pw}_{1-i}$;
2. No output was sent to the other party yet;
3. If there was output to the other party, the record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ was not fresh and thus interrupted or compromised at that time

In all of these cases, $\mathcal{S}$ chose a fresh sk following a uniform distribution and ℓ' was contained in the `NewSession` query of $\mathcal{P}_i$'s partner, thus $\ell' = \ell^*$. Regarding the session key, $\mathcal{Z}$ can only notice a difference if it reproduces sk by computing $\mathcal{P}_i$'s input $(\mathsf{sid}, X^*, Y^*, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_i||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_i||\ell'}(Y^*))$ to $\mathcal{F}_{\mathsf{RO}}$ and sending it to $\mathcal{F}_{\mathsf{RO}}$ via the adversary $\mathcal{A}$.

The following lemma bounds the probability that a session key of an unattacked session does not look random.

Lemma 3.3.3. *If CDH holds in $\mathbb{G}$, then $\forall \mathsf{pw}, \ell, \ell' \leftarrow \mathcal{Z}$ with $\mathsf{pw} \in \mathbb{P}, \ell, \ell' \in \mathcal{L}$ it holds that*

$$\Pr_{\boldsymbol{G}_9}[\mathit{CDH}(\mathcal{D}_{\mathsf{pw}||\ell}(X^*), \mathcal{D}_{\mathsf{pw}||\ell'}(Y^*)) \leftarrow \mathcal{Z}(X^*, Y^*)] = negl(\lambda).$$

Proof. We only sketch the proof since it is similar to the proof of Lemma 3.3.2. Namely, the strategy of embedding (randomized versions of) a CDH challenge into the simulation of Game $\mathbf{G}_9$ is done by just encrypting both CDH challenge elements to obtain X^* and Y^*. For the final argument, note that $\perp$ is not seen by $\mathcal{Z}$ since it is either replaced using a random session key or a previously computed key. □

It follows that Game $\mathbf{G}_8$ and Game $\mathbf{G}_9$ are indistinguishable.

Game $\mathbf{G}_{10}$: $\mathcal{F}$ always takes all labels from the list $\Lambda_{\mathcal{L}}$. We modify $\mathcal{F}$ as follows: if $\mathcal{F}$ outputs $(\mathsf{sid}, \ell', \mathsf{sk})$ towards $\mathcal{P}_i$ where ℓ', sk are taken from a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \ell', \mathsf{sk})$ from $\mathcal{S}$, $\mathcal{F}$ extracts ℓ^* from a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$ or sets $\ell^* \leftarrow \perp$ if such a record does not exist. $\mathcal{F}$ then outputs $(\mathsf{sid}, \ell^*, \mathsf{sk})$ towards $\mathcal{P}_i$. We additionally modify $\mathcal{S}$ to remove the labels from the `NewKey` queries altogether.

First observe that we can remove the labels from the `NewKey` queries because, in this and the past games, we ensured that $\mathcal{F}$ never accesses this label. However, we still have to argue indistinguishability of this and the previous game. The cases where sk of $\mathcal{S}$ is relayed by $\mathcal{F}$ towards $\mathcal{P}_i$ are the following:

- $\mathcal{P}_i$ has a compromised record
- $\mathcal{P}_i$ is corrupted
- $\mathcal{P}_i$ has a fresh record, its partner is corrupted and has a record with a matching password

In the first case, we have that $\ell' = \ell^*$ since the label ℓ' output by $\mathcal{P}_i$ was also contained in a `TestPwd` query by $\mathcal{S}$ and overwrote any existing label send by $\mathcal{P}_i$'s partner. For the second case, observe that since we restrict to static corruption, corrupted players will not have records in $\Lambda_{\mathcal{P}}$ and thus this case will never happen. In the third case, corruption of the partner ensures that $\mathcal{S}$ issued a `TestPwd` query which overwrote any existing label with ℓ', so $\ell' = \ell^*$ as well.

Observe that now $\mathcal{F}$ acts like $\mathcal{F}_{\mathsf{liPAKE}}$ regarding the output of session keys. The only remaining difference is that the `NewSession` queries still contain the passwords of the parties. In the next games, we will make the simulation independent of these passwords.

Game $\mathbf{G}_{11}$: Simulate without pw_1 if the server $\mathcal{P}_1$ is honest. In case of receiving a $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_1, \ell')$ from an honest $\mathcal{P}_1$ playing the role of a server, we modify $\mathcal{F}$ by forwarding only $(\texttt{NewSession}, \mathsf{sid}, \ell')$ to $\mathcal{S}$. We now have to modify $\mathcal{S}$ to proceed with the simulation without knowing pw_1. Upon receiving $(\texttt{NewSession}, \mathsf{sid}, \ell')$ from $\mathcal{F}$ for an honest $\mathcal{P}_1$, we let $\mathcal{S}$ draw uniformly at random a "dummy" password $\mathsf{pw}_{\mathcal{S}}$ and proceed with the simulation of $\mathcal{P}_1$ using $\mathsf{pw}_{\mathcal{S}}$ as a password.

We first note that in this case of both parties being honest, if at any time $\mathcal{S}$ sends a `NewKey` query to $\mathcal{F}$ containing a session key sk' for $\mathcal{P}_1$, this session key is only seen by $\mathcal{Z}$ if the corresponding record is compromised. Otherwise, we thus only have to argue indistinguishability of the transcripts of Game $\mathbf{G}_{10}$ and Game $\mathbf{G}_{11}$.

- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, X^*_{\mathcal{Z}}$, there is a record $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, X, *, \mathcal{E}, X^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} = \mathsf{pw}_1$: since $\mathcal{S}$ will issue a `TestPwd` query that will result in a compromised record (cf. simulation described in Game $\mathbf{G}_6$), nothing is changed since $\mathsf{pw}_{\mathcal{S}}$ was never used, and Y^* is generated using the correct password $\mathsf{pw}_{\mathcal{Z}}$.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, X^*_{\mathcal{Z}}$, there is a record $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, X, *, \mathcal{E}, X^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_1$: since $\mathcal{P}_1$ will receive a random session key from $\mathcal{F}$ in this case (the record will be interrupted), we only have to argue indistinguishability of Y^* generated with $\mathsf{pw}_{\mathcal{Z}}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. Obviously, Y^* is distributed uniformly random as before. Observe that here it is important that even for a corrupted session, an interrupted record lets the functionality hand out a random session key.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, X^*_{\mathcal{Z}}$ and no $\mathcal{E}$ record: the simulation described in Game $\mathbf{G}_6$ tells $\mathcal{S}$ to issue a `TestPwd` query, but now using $\mathsf{pw}_{\mathcal{S}}$ instead of pw_1. If, coincidentally, $\mathsf{pw}_1 = \mathsf{pw}_{\mathcal{S}}$, nothing changes. On the other hand, if $\mathsf{pw}_1 \neq \mathsf{pw}_{\mathcal{S}}$, $\mathcal{P}_1$ obtains a random session key from $\mathcal{F}$ as opposed to the game before and Y^* is created using $\mathsf{pw}_{\mathcal{S}}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. This can only be detected if $\mathcal{Z}$ reproduces $\mathcal{P}_1$'s input to $\mathcal{F}_{\mathsf{RO}}$ from Game $\mathbf{G}_{10}$, which happens only with negligible probability according to Lemma 3.3.4 (see below).
- both parties honest and no injections: $\mathcal{P}_1$ will obtain a uniformly random session key from $\mathcal{F}$ in this case, and thus the only difference is that Y^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. Again, this is indistinguishable since Y^* is distributed exactly as before.

The following lemma bounds the probability that an injected X^* that was not obtained using encryption leads to a non-random looking session key.

Lemma 3.3.4. *If CDH holds in $\mathbb{G}$, then $\forall \mathsf{pw}_1, \ell', \ell_{\mathcal{Z}}, X^*_{\mathcal{Z}} \leftarrow \mathcal{Z}$, where $X^*_{\mathcal{Z}}$ was not generated using $\mathcal{F}_{IC}$, $\ell', \ell_{\mathcal{Z}} \in \mathcal{L}$ and $\mathsf{pw}_1 \in \mathbb{P}$, it holds that*

$$\Pr_{\boldsymbol{G}_{11}}[\mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_1||\ell_{\mathcal{Z}}}(X^*_{\mathcal{Z}}), \mathcal{D}_{\mathsf{pw}_1||\ell'}(Y^*)) \leftarrow \mathcal{Z}(Y^*)] = negl(\lambda).$$

Proof. Note that the only difference to Lemma 3.3.2 is that this time, no record $(*, *, *, \mathcal{E}, X^*_{\mathcal{Z}})$ exists so the fact that $\mathcal{B}_{\mathsf{CDH}}$ is able to embed an element of its CDH challenge into $\mathcal{D}_{\mathsf{pw}_1}(X^*_{\mathcal{Z}})$ is even more obvious. The rest of the proof is analogous to Lemma 3.3.2. □

Game $\mathbf{G}_{12}$: Simulate without pw_0 if the client $\mathcal{P}_0$ is honest. In a similar fashion, we now let $\mathcal{F}$ cut the password from `NewSession` queries to an honest $\mathcal{P}_0$. We again have to modify $\mathcal{S}$ to proceed with the simulation without knowing pw_0. Upon receiving $(\texttt{NewSession}, \mathsf{sid}, \ell)$ from $\mathcal{F}$ for an honest $\mathcal{P}_0$, we let $\mathcal{S}$ draw uniformly at random a "dummy" password $\mathsf{pw}_{\mathcal{S}}$. $\mathcal{S}$ proceeds the simulation of $\mathcal{P}_0$ using $\mathsf{pw}_{\mathcal{S}}$ as a password.

Additionally, we further change $\mathcal{S}$ in case of a dictionary attack against client $\mathcal{P}_0$, i.e., upon receiving $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$ from $\mathcal{Z}$. After submitting a `TestPwd` query with an extracted $\mathsf{pw}_{\mathcal{Z}}$, we let $\mathcal{S}$ now choose $x' \xleftarrow{\$} \mathbb{Z}_q$ and add $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\ell, g^{x'}, x', \perp, X^*)$ to Λ_{IC} and proceed with the simulation of $\mathcal{P}_0$ using x' instead of x.

- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$, there is a record $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, Y, *, \mathcal{E}, Y^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} = \mathsf{pw}_0$: $\mathcal{S}$ will issue a `TestPwd` query that will result in a compromised record (cf. simulation described in Game $\mathbf{G}_5$), resulting in a session key that is computed using $\mathsf{pw}_{\mathcal{Z}}$ instead of $\mathsf{pw}_{\mathcal{S}}$. Additionally, X^* is generated using the incorrect password $\mathsf{pw}_{\mathcal{S}}$. However, adjusting Λ_{IC} as described above still allows $\mathcal{S}$ to know the exponent of $\mathcal{D}_{\mathsf{pw}_{\mathcal{Z}}||\ell}(X^*)$ and continue the simulation, making it look like $\mathsf{pw}_{\mathcal{Z}}$ was used from the beginning. $\mathcal{Z}$'s view is distributed exactly as before since x', x are both uniformly random in $\mathbb{Z}_q$.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$, there is a record $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, Y, *, \mathcal{E}, Y^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$: since $\mathcal{P}_0$ will receive a random session key from $\mathcal{F}$ in this case (the record will be interrupted), we only have to argue indistinguishability of X^* generated with $\mathsf{pw}_{\mathcal{S}}|\ell$ instead of $\mathsf{pw}_0||\ell$. Obviously, Y^* is distributed uniformly random as before. Observe that here it is crucial that even for a corrupted session, an interrupted record lets the functionality hand out a random session key.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$ and no $\mathcal{E}$ record: the simulation described in Game $\mathbf{G}_5$ tells $\mathcal{S}$ to issue a `TestPwd` query, but now using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. If, coincidentally, $\mathsf{pw}_0 = \mathsf{pw}_{\mathcal{S}}$, nothing changes. On the other hand, if $\mathsf{pw}_0 \neq \mathsf{pw}_{\mathcal{S}}$, $\mathcal{P}_0$ obtains a random session key from $\mathcal{F}$ as opposed to the game before and X^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. This can only be detected if $\mathcal{Z}$ reproduces $\mathcal{P}_0$'s input to $\mathcal{F}_{\mathsf{RO}}$ from Game $\mathbf{G}_{11}$, which happens only with negligible probability using an argument very similar to Lemma 3.3.4.
- both parties honest and no injections: $\mathcal{P}_0$ will obtain a uniformly random session key from $\mathcal{F}$ in this case, and thus the only difference is that X^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. Again, this is indistinguishable since X^* is distributed exactly as before.

Observe that in Game $\mathbf{G}_{12}$, $\mathcal{F} = \mathcal{F}_{\mathsf{liPAKE}}$[3], and thus the theorem follows. The complete description of the simulator of Game $\mathbf{G}_{12}$ interacting with $\mathcal{F}_{\mathsf{iPAKE}}$ and $\mathcal{Z}$ is given in Figure 3.6.

□

[3]We note that we can, w.l.o.g, assume that there is no `NewSession` queries from $\mathcal{Z}$ to corrupted parties. Thus, it is enough to remove the passwords from the `NewSession` queries given as input from $\mathcal{Z}$ to honest parties.

The simulator $\mathcal{S}$, initialized with a security parameter λ, first runs a group generation algorithm using λ to obtain a cyclic group $\mathbb{G}$ with generator g of order q with $\log_2(q) \geq \lambda$. Then, $\mathcal{S}$ initializes the dummy adversary $\mathcal{A}$. $\mathcal{S}$ then interacts with an ideal functionality $\mathcal{F}_{\mathsf{liPAKE}}$ and an environment $\mathcal{Z}$ via the following queries:

- **Upon receiving a query $(\mathtt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \ell)$ from $\mathcal{F}_{\mathsf{liPAKE}}$:**
  - initialize a party $\mathcal{P}_i$, connect it to $\mathcal{A}$ and proceed the UC protocol execution described in Figure 3.4 using $\mathsf{pw}_{\mathcal{S}} \xleftarrow{\$} \mathbb{P}$ as password and $\mathcal{S}$'s random coins.
- **Upon receiving a RO query (sid, m) from any entity:**
  If $m \notin \{0,1\}^\kappa \times \{0,1\}^\kappa \times \mathbb{G}$, then abort. Else:
  - if there is an entry (sid, m, h) then reply with (sid, h).
  - else, choose $h \xleftarrow{\$} \{0,1\}^\lambda$ and abort if there is already an entry $(*, *, h)$ in Λ_{RO}. Else, reply with (sid, h).
- **If an internally simulated party $\mathcal{P}_i$ produces an output $(\mathsf{sid}, \ell', \mathsf{sk})$:**
  Send $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ to $\mathcal{F}_{\mathsf{liPAKE}}$.
- **If $\mathcal{Z}$ sends $(\mathsf{sid}, \ell_{\mathcal{Z}}, Z^*_{\mathcal{Z}})$ to an honest party $\mathcal{P}_i$:**
  - if $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}} || \hat{\ell}, *, *, \mathcal{E}, Z^*_{\mathcal{Z}}) \in \Lambda_{IC}$ for any $\hat{\ell} \in \mathcal{L}$, then send the tuple $(\mathtt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}_{\mathsf{liPAKE}}$ and proceed with the simulation of P with $\mathsf{pw}_{\mathcal{Z}}$.
  - if $(\mathsf{sid}, *, *, *, \mathcal{E}, Z^*_{\mathcal{Z}}) \notin \Lambda_{IC}$, then send $(\mathtt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_{\mathcal{S}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}_{\mathsf{liPAKE}}$.
  - if P was started with input $(\mathtt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \ell)$, choose $x' \xleftarrow{\$} \mathbb{Z}_q$, add $(\mathsf{sid}, \mathsf{pw}_{\mathcal{Z}} || \ell, g^{x'}, x', \perp, Z^*)$ to Λ_{IC} and proceed as if x' was the value drawn uniformly random from $\mathbb{Z}_q$ at the beginning of the simulation.
- **Upon receiving a query $(\mathsf{sid}, \mathtt{Encrypt}, ek, m)$ from any entity:**
  If $ek \notin \mathbb{P} \times \mathcal{L}$ or $m \notin \mathbb{G}$ then abort. Else:
  - if there is an entry $(\mathsf{sid}, ek, m, *, *, c)$ in Λ_{IC}, reply with (sid, c)
  - else, choose $c \xleftarrow{\$} \{0,1\}^*$. If there is already a record $(\mathsf{sid}, *, *, *, *, c)$ then abort. Else, add $(\mathsf{sid}, ek, m, \perp, \mathcal{E}, c)$ to Λ_{IC} and reply with (sid, c).
- **Upon receiving a query $(\mathsf{sid}, \mathtt{Decrypt}, ek, c)$ from any entity:**
  If $ek \notin \mathbb{P} \times \mathcal{L}$ or $c \notin \{0,1\}^*$ then abort. Else:
  - if there is an entry $(\mathsf{sid}, ek, m, *, *, c)$ in Λ_{IC}, reply with (sid, m).
  - else, choose $\alpha \xleftarrow{\$} \mathbb{G}$. If there is already a record $(\mathsf{sid}, *, g^\alpha, *, *, *)$ then abort. Else, add $(\mathsf{sid}, ek, g^\alpha, \alpha, \mathcal{D}, c)$ to Λ_{IC} and reply with (sid, g^α).
- **Upon receiving a query $(\mathsf{sid}, \mathtt{crs})$ from any entity:**
  - reply with $(\mathsf{sid}, (g, q))$.

Additionally, $\mathcal{S}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any $\mathtt{NewSession}$ query from $\mathcal{F}_{\mathsf{liPAKE}}$.

Figure 3.6: The Simulator $\mathcal{S}$ for the EKE2 Protocol indistinguishability from $\mathcal{F}_{\mathsf{liPAKE}}$

Chapter 4

Fuzzy Password Authenticated Key Exchange

4.1 Model

We now present a new notion, called *fuzzy password authenticated key exchange* (fPAKE), which is a relaxation of a PAKE protocol (see Section 2.2) in the sense that the authentication should succeed even if the passwords are slightly different.

More precisely, we define a threshold δ and require that the fPAKE session keys are identical if the distance between the two passwords, for a certain notion of distance, is smaller than δ. Otherwise, as in PAKE, the two session keys should each be indistinguishable from random.

Notation. In the following we assume the two passwords are strings of length n over some finite alphabet, and denote by $\mathsf{pw}[i]$ the i-th character of the string pw. We will use the Hamming distance as our choice of distance between two passwords, so

$$d(\mathsf{pw}, \mathsf{pw}') := \left|\{i \in [\![1, n]\!] \,:\, \mathsf{pw}[i] \neq \mathsf{pw}'[i]\}\right| .$$

Note that we do not restrict passwords to bit strings but view them as n-digit strings in any p-alphabet (e.g., coming from $\mathbb{Z}_p^n$) and the distance is defined as the number of non-matching password digits;

We also define a masking function, that reveals the positions of the identical bits, but nothing else:

$$m(\mathsf{pw}, \mathsf{pw}') := \{i \in [\![1, n]\!] : \mathsf{pw}[i] = \mathsf{pw}'[i]\} .$$

By construction $|m(\mathsf{pw}, \mathsf{pw}')| = n - d(\mathsf{pw}, \mathsf{pw}')$.

Functionalities. We will now present ideal functionality for the fPAKE in the UC framework, with the δ threshold hard-coded in. We proceed from $\mathcal{F}_{\mathsf{pake}}$ and thus allow the adversary one password guess against each player per session, to model the possibility of dictionary attacks. We refer the reader to Section 2.2 for more details about the basic modeling of PAKE.

As in the definition of Canetti et al. [CHK+05], we consider only static corruptions in the standard corruption model of Canetti [Can01]. Also as in their definition, we choose not to provide the players with confirmation that key agreement was successful. The players might obtain such confirmation from subsequent use of the key.

The functionality $\mathcal{F}_{\mathsf{fpake}}$ is parameterized by a security parameter λ and tolerances $\delta \leq \gamma$. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i)$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i)$ to $\mathcal{S}$;
  - If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ and mark this record `fresh`.
- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_i)$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$, then set $d \leftarrow d(\mathsf{pw}_i, \mathsf{pw}'_i)$ and do:
  - If $d < \delta$, mark the record `compromised`;
  - If $d \geq \delta$, mark the record `interrupted`.
- **Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is no record of the form $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ or this is not the first `NewKey` query for $\mathcal{P}_i$, ignore this query. Otherwise:
  - If the record is `compromised`, $\mathcal{P}_i$ is corrupted or both $\mathcal{P}_{1-i}$ is corrupted and there is a record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) < \delta$, then output $(\mathsf{sid}, \mathsf{sk})$ to player $\mathcal{P}_i$.
  - If the record is `fresh`, both parties are honest and a key sk' was sent to $\mathcal{P}_{1-i}$, at which time there was a `fresh` record $(\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) < \delta$, then output $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
  - In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.

  Either way, mark the record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 4.1: Ideal Functionality $\mathcal{F}_{\mathsf{fpake}}$ for fPAKE

Ideally, since we target implicit authentication we would like to achieve full implicit authentication, as in iPAKE from Chapter 3 (i.e., the adversary is no more powerful than the players, and does not know either whether authentication succeeded). This is what is presented in Figure 4.1.

However, given the difficulty of achieving this, we also present a more general `TestPwd` interface which can be substituted to allow for setting some leakage functions.

It includes three leakage functions that we will instantiate in different ways below—L_c if the guess is close enough to succeed, L_f if it is too far. Moreover, a third leakage function—L_m for medium distance—allows the adversary to get some information even if the adversary's guess is only somewhat close (closer than some parameter $\gamma \geq \delta$), but not close enough for successful key agreement. We thus decouple the distance needed for functionality from the (possibly larger) distance needed to guarantee security; the smaller the gap between these two distances, the better, of course.

Below, we list the specific leakage functions L_c, L_m and L_f that we consider in this work, in order of decreasing strength (or increasing leakage):

- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_i)$ from the adversary $\mathcal{S}$:** If there is a `fresh` record $(\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i)$, then set $d \leftarrow d(\mathsf{pw}_i, \mathsf{pw}'_i)$ and do:
  - If $d \leq \delta$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_c(\mathsf{pw}_i, \mathsf{pw}'_i)$;
  - If $\delta < d \leq \gamma$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_m(\mathsf{pw}_i, \mathsf{pw}'_i)$;
  - If $\gamma < d$, mark the record `interrupted` and reply to $\mathcal{S}$ with $L_f(\mathsf{pw}_i, \mathsf{pw}'_i)$.

Figure 4.2: A Modified `TestPwd` Interface to Allow for Different Leakage

1. The strongest option is to provide no feedback at all to the adversary, which is what we did in Figure 4.1. In the language of the generic `TestPwd` from Figure 4.2, this would mean setting
$$L_c(\mathsf{pw}_i, \mathsf{pw}'_i) = L_m(\mathsf{pw}_i, \mathsf{pw}'_i) = L_f(\mathsf{pw}_i, \mathsf{pw}'_i) = \perp .$$

2. A slightly weaker option would be, as with the regular PAKE functionality $\mathcal{F}_{\mathsf{pake}}$ to leak the correctness of the adversary's guess. We define $\mathcal{F}^D_{\mathsf{fpake}}$ (for decision) to be the functionality described in Figure 4.1, except with `TestPwd` is from Figure 4.2 with
$$\begin{aligned} L_c(\mathsf{pw}_i, \mathsf{pw}'_i) &= L^D_c(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“correct guess”}, \\ \text{and} \qquad L_m(\mathsf{pw}_i, \mathsf{pw}'_i) &= L^D_f(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}. \end{aligned}$$

3. We define fPAKE^M (for mask) to be the functionality described in Figure 4.1, except that `TestPwd` is from Figure 4.2, with L_c and L_m that leak the indices at which the guessed password differs from the actual one when the guess is close enough (we will call this leakage the *mask* of the passwords). That is,
$$\begin{aligned} L_c(\mathsf{pw}_i, \mathsf{pw}'_i) &= L^M_c(\mathsf{pw}_i, \mathsf{pw}'_i) = (\{j \text{ s.t. } \mathsf{pw}_i[j] = \mathsf{pw}'_i[j]\}, \text{“correct guess”}), \\ L_m(\mathsf{pw}_i, \mathsf{pw}'_i) &= L^M_m(\mathsf{pw}_i, \mathsf{pw}'_i) = (\{j \text{ s.t. } \mathsf{pw}_i[j] = \mathsf{pw}'_i[j]\}, \text{“wrong guess”}) \\ \text{and} \qquad L_f(\mathsf{pw}_i, \mathsf{pw}'_i) &= L^M_f(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}. \end{aligned}$$

   Note that this functionality fPAKE^M leaks enough information that a single guess within distance γ of the actual password will then enable the adversary to get within distance δ (and thus complete key agreement) through multiple on-line attempts, each guided by the leakage from the previous (by changing all the characters where the two passwords differ, in at most as many attempts as the size of the alphabet). Even so, this functionality can provide *one-time* security against guesses within distance γ. That is, a guess within distance γ, but not within distance δ, will not violate security if the honest party never reuses a password after an unsuccessful attempt. We do not define such security formally and do not pursue this direction further.

4. The weakest definition — or the strongest leakage — reveals the entire actual password to the adversary if the password guess is close enough. We define fPAKE^P (for password) to be the functionality described in Figure 4.1, except that `TestPwd` is from Figure 4.2, with
$$L^P_c(\mathsf{pw}_i, \mathsf{pw}'_i) = L^P_m(\mathsf{pw}_i, \mathsf{pw}'_i) = \mathsf{pw}_i \quad \text{and} \quad L^P_f(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}.$$

Here, L_c^P and L_m^P do not need to include "correct guess" and "wrong guess", respectively, because this is information that can be easily derived from pw_i itself.

The first two functionalities are the strongest, but there is no known construction that realize them, other than through generic two-party computation secure against malicious adversaries, which is an inefficient solution. The last two functionalities, though weaker, still provide meaningful security, especially when $\gamma = \delta$ (i.e., there is no L_m situation). Intuitively, this is because strong leakage only occurs when an adversary guesses a "close" password, which enables him to authenticate as though he knows the real password anyway.

In the next section, we present a construction satisfying fPAKE^M with $\gamma = 2\delta$. A more generic construction satisfying fPAKE^P, but with $\gamma = \delta$ and for any efficiently computable notion of distance (not just Hamming Distance) is presented in [DHP+18].

4.2 Construction

4.2.1 A naive idea

We now give an intuition of the strength of our fPAKE security notion. For this, we look at a fPAKE from the literature and show that it is inherently not UC-secure, regardless of the amount of leakage from $\mathcal{F}_{\mathsf{fpake}}$ that we allow, i.e., the result holds even w.r.t. our weakest notion $\mathcal{F}_{\mathsf{fpake}}^P$.

A natural idea for building a fPAKE is the use of a fuzzy extractor [DRS04; Boy04], that allows to extract a common secret from two strings close enough, and to compose it with a regular PAKE. This approach was introduced in [BDK+05] (Section 4). Their protocol uses the code-offset construction of a fuzzy sketch [DRS04], aka fuzzy commitment [JW99], to implement a fuzzy extractor as a two-party primitive. In Figure 4.3, we present it while substituting the more generic *robust secret sharing* (RSS) notation that we presented in Section 2.3.4 for the error-correcting code.

Theorem 4.2.1. *The construction from Figure 4.3 cannot securely realize $\mathcal{F}_{\mathsf{fpake}}^P$.*

Proof. Consider the following attack by $\mathcal{Z}$. $\mathcal{Z}$ sends a randomly chosen pw as input to an honest $\mathcal{P}_0$ and obtains a sketch s from $\mathcal{A}$. It then computes $c \leftarrow s - \mathsf{pw}$ and outputs 1 if c is in the image of Share. In the real world, this happens with probability 1. Now assume there is a simulator $\mathcal{S}$ outputting a simulated sketch $s^{\mathcal{S}}$ in the ideal world. Since $\mathcal{S}$ does not get to learn pw unless it succeeds at a `TestPwd` query, observe that this output may not depend on pw except with some small (but non-negligible) probability p, namely the probability of guessing

$\mathcal{P}_0(\mathsf{pw} \in \mathbb{F}_q^n)$		$\mathcal{P}_1(\mathsf{pw}' \in \mathbb{F}_q^n)$
$w \xleftarrow{\$} \mathbb{F}_q^k, c \leftarrow \mathsf{Share}(w) \in \mathbb{F}_q^n$		
$s \leftarrow c + \mathsf{pw}$	$\xrightarrow{s}$	$c' \leftarrow s + \mathsf{pw}'$
		$w' \leftarrow \mathsf{Reconstruct}(c')$
K	$\xleftrightarrow{\mathsf{PAKE}(w, w')}$	K'
$\mathsf{sk} \leftarrow K$		$\mathsf{sk}' \leftarrow K'$

Figure 4.3: A First Natural Construction (with code-offset fuzzy sketch and PAKE)

a password that makes $\mathcal{F}^P_{\mathsf{fpake}}$ output pw. Thus, with probability $1-p \approx 1$, $c^{\mathcal{S}} := s^{\mathcal{S}} - \mathsf{pw}$ is randomly distributed in $\mathbb{F}_q^n$ and lies in the image of Share only with probability $1/q^{n-1}$. More formally, the probability that $\mathcal{Z}$ outputs 1 in the ideal world is

$$\begin{aligned}\Pr[c^{\mathcal{S}} \in Im(\mathsf{Share})] &= \Pr[c^{\mathcal{S}} \in Im(\mathsf{Share}) | \mathcal{S} \text{ depends on } \mathsf{pw}] \cdot p \\ &\quad + \Pr[c^{\mathcal{S}} \in Im(\mathsf{Share}) | \mathcal{S} \text{ does not depend on } \mathsf{pw}] \cdot (1-p) \\ &\leq p + 1/q^{n-1}(1-p) \approx p.\end{aligned}$$

□

4.2.2 Improved idea

In this section we show how to keep the full entropy of the password when some public data is seen (contrarily to the previous construction from Figure 4.3) and to avoid any leakage when the two passwords are away from each other. To this aim, we still use a RSS together with a signature scheme and our liPAKE from Chapter 3. Our protocol is depicted in Figure 4.4.

Intuitively, our protocol works as follows: in the first phase, the two parties aim at enhancing their passwords to a vector of session keys with good entropy. For this, passwords

Sender ($\mathsf{pw} \in \mathbb{F}_p^n$)		Receiver ($\mathsf{pw}' \in \mathbb{F}_p^n$)
$(\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{SigGen}(1^\lambda)$		
	$\xrightarrow{(\mathsf{vk}, \mathsf{pw}[t])_t}$	
	$\xleftarrow{(\varepsilon, \mathsf{pw}'[t])_t}$	
for $t \in [\![1, n]\!]$	$\boxed{\mathsf{liPAKE}}$	for $t \in [\![1, n]\!]$
	$\xleftarrow{(\varepsilon, K_t)_t}$	abort if $\ell_r \neq \ell_s$ for any r, s or $\ell_1 \notin \mathcal{VK}$
	$\xrightarrow{(\ell_t, K'_t)_t}$	
Let $K := (K_t)_{t \in [\![1,n]\!]}$		Let $K' := (K'_t)_{t \in [\![1,n]\!]}$
$U \xleftarrow{\$} \mathbb{F}_q, C \leftarrow \mathsf{Share}(U)$		
$E \leftarrow C + K$		
$\sigma_E \leftarrow \mathsf{Sign}(\mathsf{vk}, E)$	$\xrightarrow{E, \sigma_E, \mathsf{vk}}$	abort if $\mathsf{vk} \neq \ell_1$ or $\mathsf{Vfy}(\mathsf{vk}, \sigma_E, E) = 0$
output $\mathsf{k} \leftarrow U$		
		$U' \leftarrow \mathsf{Reconstruct}(E - K')$
		output $\mathsf{k}' \leftarrow U'$

$q \approx 2^\lambda$ is a prime number and $+$ denotes the group operation in $\mathbb{F}_q^n$. ε denotes the empty string. $(\mathsf{Share}, \mathsf{Reconstruct})$ is a Robust Secret Sharing scheme with $\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n$, and $(\mathsf{SigGen} \to \mathcal{VK} \times \mathcal{SK}, \mathsf{Sign}, \mathsf{Vfy})$ is a signature scheme. The parties repeatedly execute a liPAKE protocol with label space $\mathcal{VK}$ and key space $\mathbb{F}_q$, which takes inputs from $\mathcal{VK} \times \mathbb{F}_p$. If at any point an expected message fails to arrive (or arrives malformed), the parties output a random key.

Figure 4.4: $\mathsf{FPAKE}^{\mathsf{rss}}$ Protocol

are viewed as vectors of characters. Then, the parties repeatedly execute a PAKE on each one of these characters. The PAKE will ensure that the key vectors held by the two parties match in all positions where their passwords matched, and are uniformly random in all other positions.

Next, in the second phase of the protocol, one party — the sender — will pick the final session key uniformly at random and send it in such a way that it reaches the other party only if enough of the key vector matches. This is done by applying a RSS to the key, and send it to the other party using the key vector as a one-time pad. The robustness property of the RSS ensures that a few non-matching password digits do not prevent the receiver from recovering the sender's key.

When using the RSS derived from *maximum distance separable* (MDS) codes described in Lemma 2.3.5, the one-time pad encryption of the shares (which forms a codeword) can be viewed as the code-offset construction for information reconciliation (aka secure sketch) [JW99; DRS04] applied to the key vectors. While our presentation goes through RSS as a separate object, we could instead present this construction using information reconciliation. The syndrome construction of secure sketches can also be used here instead of the code-offset construction.

Security of FPAKE^{rss}. We show that our protocol realizes functionality $\mathcal{F}^M_{\mathsf{fpake}}$ in the $\mathcal{F}_{\mathsf{liPAKE}}$-hybrid model. In a nutshell, the idea is to simulate without the passwords by adjusting the keys outputted by $\mathcal{F}_{\mathsf{liPAKE}}$ to the mask of the passwords, which is leaked by $\mathcal{F}^M_{\mathsf{fpake}}$.

Theorem 4.2.2. *If RSS* $:= (\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n, \mathsf{Reconstruct} : \mathbb{F}_q^n \to \mathbb{F}_q)$ *is a t-smooth $(n,t,r)_q$-RSS that in addition is $(r-1)$-smooth on random secrets, and* $(\mathsf{SigGen}, \mathsf{Sign}, \mathsf{Vfy})$ *is a* existencially unforgeable under adaptive chosen message attack *(EUF-CMA) secure one-time signature scheme, protocol FPAKE^{rss} securely realizes $\mathcal{F}^M_{\mathsf{fpake}}$ with $\gamma = n - t - 1$ and $\delta = n - r$ in the $\mathcal{F}_{\mathsf{liPAKE}}$-hybrid model with respect to static corruptions.*

In particular, if we wish key agreement to succeed as long as there are fewer than δ errors, we instantiate RSS using the construction of Lemma 2.3.5 based on a $(n+1,k)_q$ MDS code, with $k = n - 2\delta$. This will give $r = \lceil (n+k)/2 \rceil = n - \delta$, so δ will be equal to $n - r$, as required. It will also give $\gamma = n - t - 1 = 2\delta$.

We thus obtain the following corollary:

Corollary 4.2.3. *For any δ and $\gamma = 2\delta$, given a $(n+1,k)_q$-MDS code for $k = n - 2\delta$ (with minimal distance $d = n - k + 2$) and a EUF-CMA secure one-time signature scheme, protocol FPAKE^{rss} securely realizes $\mathcal{F}^M_{\mathsf{fpake}}$ in the $\mathcal{F}_{\mathsf{liPAKE}}$-hybrid model with respect to static corruptions.*

Proof sketch. We start with the real execution of the protocol and indistinguishably switch to an ideal execution with dummy parties relaying their inputs to and obtaining their outputs from $\mathcal{F}^M_{\mathsf{fpake}}$. To preserve the view of the distinguisher, the environment $\mathcal{Z}$, a simulator $\mathcal{S}$ plays the role of the real world adversary by controlling the communication between $\mathcal{F}^M_{\mathsf{fpake}}$ and $\mathcal{Z}$. During the proof, we build $\mathcal{F}^M_{\mathsf{fpake}}$ and $\mathcal{S}$ by subsequently randomizing passwords (since the final simulation has to work without them) and session keys (since $\mathcal{F}^M_{\mathsf{fpake}}$ hands out random session keys in certain cases). We have to tackle the following difficulties, which we will describe in terms of attacks.

- Passive attack: in this attack, $\mathcal{Z}$ picks two passwords and then observes the transcript and outputs of the protocol, without having access to any internal state of the parties. We show that $\mathcal{Z}$ cannot distinguish between transcript and outputs that were either produced using $\mathcal{Z}$'s passwords or random passwords. Regarding the outputs, we argue that even in the real execution the session keys were chosen uniformly at random (with $\mathcal{Z}$ not knowing the coins consumed by this choice) as long as the distance check is reliable. Using properties of the RSS, we show that this is the case with overwhelming probability. Regarding the transcript, randomization is straightforward using properties of the one-time pad.

- Man-in-the-middle attack: in this attack, $\mathcal{Z}$ injects a malicious message into a session of two honest parties. There are several ways to secure protocols that have to run in unauthenticated channels and are prone to this attack. Basically, all of them introduce methods to bind messages together to prevent the adversary from injecting malicious messages. To do this, we need the *labeled* version of our iPAKE and a one-time signature scheme[1,2]. Unless $\mathcal{Z}$ is able to break a one-time-signature scheme, this attack always results in an abort.

- Honest-but-curious/Active attack: in this attack, $\mathcal{Z}$ corrupts one of the parties. The simulator will get help from $\mathcal{F}^M_{\mathsf{fpake}}$ by issuing a `TestPwd` query, which will inform him whether the passwords used by both parties are close and, if so, in which positions they match (i.e., their mask).
  - If the sender is honest, we show how to use this information to simulate the transcript. Note that knowledge of the mask is necessary since, due to corruption, $\mathcal{Z}$ can now actually decrypt the one-time pad and thus the transcript reveals the positions of the errors in the passwords, which are, of course, already known to $\mathcal{Z}$. If the simulator does not learn a mask, then the passwords are too far away and it follows from the privacy of the RSS that real and simulated transcript are indistinguishable from $\mathcal{Z}$'s view.
  - If the receiver is honest and $\mathcal{Z}$ injects a malicious message on behalf of the sender, the simulator uses the mask to compute the output of the honest receiver. If no mask is obtained then again, the passwords are too far away from each other, and the smoothness property of the RSS (for arbitrarily chosen secrets) says that the receiver's output can be simulated by choosing it uniformly at random.

One interesting subtlety that arises is the usage of the iPAKE. Observe that the UC security notion for a regular PAKE as defined in [CHK+05] and in Section 2.2 provides an interface to the adversary to test a password once and learn whether it is right or wrong. Using this notion, our simulator would have to answer to such queries from $\mathcal{Z}$. Since this is not possible without $\mathcal{F}^M_{\mathsf{fpake}}$ leaking the mask all the time, it is crucial to use the iPAKE variant that we introduced in Chapter 3. Using this stronger notion, the adversary is still allowed one password guess which may affect the output, but the adversary learns nothing more about the outcome of his guess than he can infer from whatever access he has to the outputs alone.

[1] Instead, one could just sign all the messages, as would be done using the split functionality [BCL+05], but this would be less efficient.

[2] This trade-off is especially useful when we use a PAKE that admits adding labels basically for free, as it is the case with the special PAKE protocol we use.

Since our protocol uses the outputs of the PAKE as one-time pad keys, it is intuitively clear that by preventing $\mathcal{Z}$ from getting additional leakage about these keys, we protect the secrets of honest parties.

Proof. In this proof, we describe an honest execution of the protocol $\mathsf{FPAKE}^{\mathsf{rss}}$ in the UC framework in Figure 4.5. See [FHH14] for a detailed description on how to execute protocols within the UC framework. This real protocol execution will be the starting point for our proof. We then proceed in a series of games, to end up with the ideal execution running with only dummy parties, a simulator and the ideal functionality $\mathcal{F}^M_{\mathsf{fpake}}$. For convenience, we refer to a received protocol message as *adversarially generated* if it was not produced by either $\mathcal{P}_0$ or $\mathcal{P}_1$. We also refer to a query $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ from the adversary $\mathcal{S}$ with an honest party $\mathcal{P}_i$ as *due* if

- there is a `fresh` record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$
- this is the first `NewKey` query for $\mathcal{P}_i$
- there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$ and $\mathcal{P}_{1-i}$ is honest
- a key k_{1-i} was sent to the other party while $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time.

We also recall the masking function that reveals the positions of the identical bits:

$$m(\mathsf{pw}, \mathsf{pw}') := \{i : \mathsf{pw}[i] = \mathsf{pw}'[i], i \in [\![1, n]\!]\} .$$

Game $\mathbf{G}_0$: The real protocol execution. This is the real execution of $\mathsf{FPAKE}^{\mathsf{rss}}$ where the environment $\mathcal{Z}$ runs the protocol (cf. Figure 4.4) with parties $\mathcal{P}_0$ and $\mathcal{P}_1$, both having access to an ideal liPAKE functionality $\mathcal{F}_{\mathsf{liPAKE}}$, and an adversary $\mathcal{A}$ that, w.l.o.g., can be assumed to be the dummy adversary as shown in [Can01, section 4.4.1].

Game $\mathbf{G}_1$: Modeling the ideal layout. We first make some purely conceptual changes that do not modify the input/output interfaces of $\mathcal{Z}$. We add one relay (also referred to as the *dummy party*) on each of the wires between $\mathcal{Z}$ and a party. We also add one relay covering all the wires between the dummy parties and real parties and call it $\mathcal{F}$ (and let $\mathcal{F}$ relay messages according to the original wires). We group all the formerly existing instances except for $\mathcal{Z}$ into one machine and call it $\mathcal{S}$. Note that this implies that $\mathcal{S}$ executes the code of the liPAKE functionality $\mathcal{F}_{\mathsf{liPAKE}}$. The differences are depicted in Figure 4.6 with $\mathcal{F}_{\mathsf{OT}}$ replaced by $\mathcal{F}_{\mathsf{liPAKE}}$.

Game $\mathbf{G}_2$: Building $\mathcal{F}^M_{\mathsf{fpake}}$. In this game, we start modeling $\mathcal{F}^M_{\mathsf{fpake}}$.

First, we let $\mathcal{F}$ maintain a list of tuples of the form $(\mathcal{P}_i, \mathsf{pw}_i)$. Upon receiving a query $(\mathtt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \mathtt{role})$ from party $\mathcal{P}_i$, if this is the first `NewSession`-query, or if this is the second `NewSession`-query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then $\mathcal{F}$ records $(\mathcal{P}_i, \mathsf{pw}_i)$ and marks this record as `fresh`.

In any case the query $(\mathtt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_i, \mathtt{role})$ is relayed to $\mathcal{S}$. Now that $\mathcal{F}$ knows about passwords, we can add a `TestPwd` interface to $\mathcal{F}$ as described in Figure 4.2, using leakage functions L^M_c, L^M_m and L^M_f. We let $\mathcal{S}$ parse outputs towards $\mathcal{F}$ to be of the form $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ by adding the `NewKey` tag and the name of the party who

The parties $\mathcal{P}_0, \mathcal{P}_1$ are running with $\mathcal{F}_{\mathsf{liPAKE}}$.
Protocol Steps:

1. When a party $\mathcal{P}_i$, $i \in \{0,1\}$, receives an input $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{sender})$ from $\mathcal{Z}$, it does the following:
   - compute $(\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{SigGen}(1^\lambda)$
   - query n times $\mathcal{F}_{\mathsf{liPAKE}}$ with $(\texttt{NewSession}, \mathsf{sid}, (\mathsf{pw}_i)_t, \mathsf{vk})$, $t = 1, ..., n$, receiving back $(\mathsf{sid}, \ell_t, K_t)$ as answer
   - choose $U \xleftarrow{\$} \mathbb{F}_q$
   - compute $C \leftarrow \mathsf{Share}(U)$
   - compute $E \leftarrow C + (K_t)_{t=1}^n$
   - compute $\sigma_E \leftarrow \mathsf{Sign}(\mathsf{vk}, E)$
   - send $(\mathsf{sid}, E, \sigma_E, \mathsf{vk})$ to $\mathcal{P}_{1-i}$
   - set $\mathsf{k} \leftarrow U$
   - send $(\mathsf{sid}, \mathsf{k})$ towards $\mathcal{Z}$ and terminate the session.
2. When a party $\mathcal{P}_i$, $i \in \{0,1\}$, receives an input $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{receiver})$ from $\mathcal{Z}$, it does the following:
   - query n times $\mathcal{F}_{\mathsf{liPAKE}}$ with $(\texttt{NewSession}, \mathsf{sid}, (\mathsf{pw}_i)_t, \varepsilon)$, $t = 1, ..., n$, receiving back $(\mathsf{sid}, \ell_t, K'_t)$ as answer
   - if not all ℓ_t are equal or $\ell_1 \neq \mathcal{VK}$, then abort
3. When $\mathcal{P}_i$, who already obtained an input $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{receiver})$ and thus holds a vector $(\mathsf{sid}, \ell_t, K'_t)$ obtained from $\mathcal{F}_{\mathsf{liPAKE}}$, receives a message $(\mathsf{sid}, E, \sigma_E, \mathsf{vk})$ from $\mathcal{P}_{1-i}$, it does the following:
   - set $K' := (K'_t)_{t \in [n]}$
   - abort if $\mathsf{vk} \neq \ell_1$
   - abort if $\mathsf{Vfy}(\mathsf{vk}, \sigma_E, E) = 0$
   - compute $U' \leftarrow \mathsf{Reconstruct}(E - K')$
   - set $\mathsf{k} \leftarrow U'$
   - send $(\mathsf{sid}, \mathsf{k})$ towards $\mathcal{Z}$ and terminate the session.

Figure 4.5: A UC Execution of $\mathsf{FPAKE}^{\mathsf{rss}}$

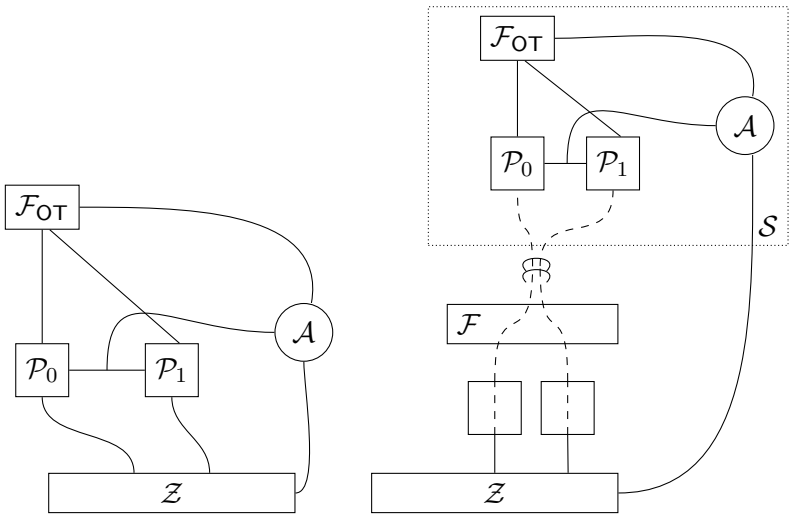

Figure 4.6: Transition from Game $\mathbf{G}_0$ (left) to Game $\mathbf{G}_1$ (right), showing a setting where both parties are honest.

produced the output. Additionally, we let $\mathcal{F}$ translate this back to $(\mathsf{sid}, \mathsf{k}_i)$, send it to $\mathcal{Z}$ via $\mathcal{P}_i$ and mark the corresponding record as `completed`.

None of these modifications changes the output towards $\mathcal{Z}$ compared to the previous Game $\mathbf{G}_1$.

Game $\mathbf{G}_3$: $\mathcal{F}$ generates a random session key for an interrupted session. Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ that is marked as `interrupted`, and this is the first `NewKey` query for $\mathcal{P}_i$, we let $\mathcal{F}$ output a random session key of length λ to $\mathcal{P}_i$. Otherwise, it continues to relay k_i.

Since the simulators described in Game $\mathbf{G}_2$ and Game $\mathbf{G}_3$ do not make use of the `TestPwd` interface, none of the records of $\mathcal{F}$ are marked as `interrupted` and thus the output towards $\mathcal{Z}$ is equally distributed in both games.

Game $\mathbf{G}_4$: $\mathcal{S}$ handles dictionary attacks using the `TestPwd` interface. In this game, we only change the simulation. Consider the following setting: $\mathcal{P}_i$ obtained input $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{role})$ and $\mathcal{P}_{1-i}$ is corrupted and already provided its inputs to $\mathcal{F}_{\mathsf{liPAKE}}$. In this situation, $\mathcal{S}$ will proceed with the simulation of $\mathcal{P}_i$ as follows:

First, $\mathcal{S}$ assembles $\mathsf{pw}_{\mathcal{Z}} \in \mathbb{F}_p^n$ from the queries to $\mathcal{F}_{\mathsf{liPAKE}}$ that $\mathcal{P}_{1-i}$ issued. $\mathcal{S}$ sends $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}})$ to $\mathcal{F}$, obtaining either "wrong guess", "correct guess" and perhaps also a mask $M \subseteq [\![1, n]\!]$ from $\mathcal{F}$. If $\mathcal{S}$ does not receive a mask, $\mathcal{S}$ is not modified further. Else, let $I := [\![1, n]\!] \setminus M$ the set of mismatching indices, and $d := |I| \leq \gamma$ their number. $\mathcal{S}$ sets up keys $K, K' \in \mathbb{F}_q^n$ with $K_t = K'_t \xleftarrow{\$} \mathbb{F}_q$ for the matching indices $t \in M$ and $K_t, K'_t \xleftarrow{\$} \mathbb{F}_p^2$ for the mismatching indices $t \in I$, where K' denotes the $\mathcal{F}_{\mathsf{liPAKE}}$ output of $\mathcal{P}_{1-i}$. $\mathcal{S}$ now continues the simulation of $\mathcal{P}_i$ using K as output of $\mathcal{F}_{\mathsf{liPAKE}}$.

We have to analyze different cases depending on the different outcomes of `TestPwd`. However, note that the modifications only have an impact on the output k_i of $\mathcal{P}_i$ if the record gets `interrupted`, and only affect the transcript if the answer to the `TestPwd` query contains a mask. Considering the case where `TestPwd`

- outputs $\boldsymbol{m}$ and sets the record `compromised`, i.e. $d \leq \gamma$ since the distribution of K, K' only depends on the mask of the passwords, the view of $\mathcal{Z}$ is identically distributed in Game $\mathbf{G}_4$ and Game $\mathbf{G}_3$;
- outputs "wrong guess" and sets the record `interrupted`, i.e. $d > \gamma$: $\mathcal{P}_i$ will now obtain a randomly chosen session key from $\mathcal{F}$, substituting the key k_i computed by $\mathcal{S}$. If $\mathcal{P}_i$ obtained `role` = `sender`, the output in Game $\mathbf{G}_4$ and Game $\mathbf{G}_3$ is equally distributed since the honest sender outputs a random $\mathbb{F}_q$ value according to the protocol description. If $\mathcal{P}_i$ obtained `role` = `receiver`, both outputs are indistinguishable with overwhelming probability due to the smoothness of the RSS, since in Game $\mathbf{G}_3$ at least $\gamma + 1$ shares are random.

Game $\mathbf{G}_5$: Excluding man-in-the-middle attacks. Again, in this game, we only change the simulation. We now consider the case where $\mathcal{Z}$ injects a message into a session where both parties are honest. We modify $\mathcal{S}$ as follows: upon receiving an adversarially generated $(\mathsf{sid}, M_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, \mathsf{vk}_{\mathcal{Z}})$ from $\mathcal{Z}$ intended for party $\mathcal{P}_i$, $\mathcal{S}$ aborts.

Observe that the simulation is only changed compared to the previous game if it is not aborted due to protocol instructions. This means that both games are equal unless all checks pass, especially $\mathsf{Vfy}(\mathsf{vk}_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, M_{\mathcal{Z}}) = 1$. Any distinguisher between Game $\mathbf{G}_5$ and Game $\mathbf{G}_4$ can thus be turned into a forger of a valid message w.r.t the verification key of an honest party. Indinstinguishability thus follows from the security of the one-time signature scheme.

Game $\mathbf{G}_6$: $\mathcal{F}$ aligns session keys. Upon receiving a query $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if this query is *due* then output $(\mathsf{sid}, \mathsf{k}_{1-i})$ to $\mathcal{P}_i$ where k_{1-i} is the session key that was formerly sent to the other party.

We now analyze distinguishability of this game from Game $\mathbf{G}_5$. If $\mathcal{Z}$ tampered with the transcript, the simulation in Game $\mathbf{G}_5$ ensures that the simulation aborts and there is thus no `NewKey` query for $\mathcal{P}_i$. On the other hand, if $\mathcal{Z}$ does not advise $\mathcal{A}$ to tamper with any message, perfect correctness of $\mathsf{FPAKE}^{\mathsf{rss}}$ protocol ensures that, in case of a due record where the parties hold close passwords $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq n - r$, the output of $\mathcal{F}$ towards $\mathcal{Z}$ is the same as in the previous Game $\mathbf{G}_5$. Observe that perfect correctness directly follows from the perfect correctness of $\mathcal{F}_{\mathsf{liPAKE}}$ and the r-robustness of the secret sharing, which is *always* able to correct up to $n - r$ errors.

Note that $\mathcal{F}$ still differs from the functionality $\mathcal{F}^M_{\mathsf{fpake}}$ in some aspects. First, it does not output randomly generated session keys towards $\mathcal{Z}$ for honest sessions. Furthermore, it reports all passwords to $\mathcal{S}$. We will take care of these remaining differences in the next games.

Game $\mathbf{G}_7$: In some cases, $\mathcal{F}$ generates a random session key when the other party is corrupted. Upon receiving a `NewKey` query $(\mathtt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, and this is the first `NewKey` query for $\mathcal{P}_i$, $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ corrupted and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \delta$, we let $\mathcal{F}$ pick a new random key k from $\mathbb{F}_q$ and send $(\mathsf{sid}, \mathsf{k})$ to $\mathcal{P}_i$.

The simulation ensures that the record $(\mathcal{P}_i, \mathsf{pw}_i)$ is either compromised or interrupted (cf. description of the simulator in Game $\mathbf{G}_4$). Thus, the modification has no effect since it only concerns fresh records.

Game $\mathbf{G}_8$: $\mathcal{F}$ generates a random session key for an honest session. Upon receiving a `NewKey` query $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, and this is the first `NewKey` query for $\mathcal{P}_i$, both parties are honest and the `NewKey` query is not *due*, we let $\mathcal{F}$ pick a new random key k from $\mathbb{F}_q$ and send $(\mathsf{sid}, \mathsf{k})$ to $\mathcal{P}_i$.

In other words, $\mathcal{F}$ now generates a random session key upon a first `NewKey` query for an honest party $\mathcal{P}_i$ with fresh record $(\mathcal{P}_i, \mathsf{pw}_i)$ where $\mathcal{P}_{1-i}$ is also honest if (at least) one of the following events happens:

1. There is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \delta$; then, the probability that k_i was already random in Game $\mathbf{G}_7$ is overwhelming due to the $r-1$-smoothness of the RSS on random secrets. Note that to apply this property it is crucial that both parties are honest and thus the value U is randomly chosen.
2. No session key was sent to $\mathcal{P}_{1-i}$ yet; we just have to consider the case where there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$ since we already dealt with the other case in the first event. Due to the r-robustness of the RSS, the session key in the previous game was U, which is distributed uniformly random in $\mathbb{F}_q$.
3. If there was a session key sent to $\mathcal{P}_{1-i}$, the record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was not fresh and thus interrupted or compromised at that time; since our simulation never issues `TestPwd` queries for honest sessions (in fact, Game $\mathbf{G}_5$ states that $\mathcal{S}$ aborts upon man-in-the-middle attacks with overwhelming probability), this event can not happen in our simulation.

Game $\mathbf{G}_9$: Simulating without password if both parties are honest. In case of receiving a $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{role})$ from an honest $\mathcal{P}_i$, we modify $\mathcal{F}$ by forwarding only $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \texttt{role})$ to $\mathcal{S}$. We now have to modify $\mathcal{S}$ to proceed with the simulation without knowing pw. Upon receiving $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \texttt{role})$ from $\mathcal{F}$ for an honest $\mathcal{P}_i$, we let $\mathcal{S}$ draw uniformly at random a "dummy" password $\mathsf{pw}_{\mathcal{S}}$ and proceed with the simulation of $\mathcal{P}_i$ using $\mathsf{pw}_{\mathcal{S}}$ as a password.

We first observe that $\mathcal{Z}$ is oblivious of k_i contained in the $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ query that $\mathcal{S}$ will eventually send to $\mathcal{F}$ during the simulation (since $\mathcal{F}$ never lets the simulator determine k_i for an honest session). This means that, informally, we have to show that $\mathcal{Z}$, knowing $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ and seeing two transcripts, cannot tell which one was generated using $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ and which one was generated using $\mathsf{pw}_{\mathcal{S}}, \mathsf{pw}_{1-i}$ with a random $\mathsf{pw}_{\mathcal{S}}$ unknown to $\mathcal{Z}$. But this is trivial since the distribution of the values U, K does not depend on the passwords: U is randomly chosen from $\mathbb{F}_q$. $\mathcal{F}_{\mathsf{liPAKE}}$ ensures that K is randomly chosen from $\mathbb{F}_q^n$.

Game $\mathbf{G}_{10}$: Simulating without password if someone is corrupted. Upon receiving $(\texttt{NewSession}, \mathsf{sid}, \mathsf{pw}_i, \texttt{role})$ from $\mathcal{P}_i$ where $\mathcal{P}_{1-i}$ is corrupted, we modify $\mathcal{F}$ to only relay $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \texttt{role})$ to $\mathcal{S}$. Additionally, we let $\mathcal{S}$ draw uniformly at random a "dummy" password $\mathsf{pw}_{\mathcal{S}}$ and proceed with the simulation of $\mathcal{P}_i$ using $\mathsf{pw}_{\mathcal{S}}$ as a password. Note that due to the simulation described in Game $\mathbf{G}_4$, $\mathcal{S}$ will ask a `TestPwd` query, and after this query the simulation described in that game is already independent of pw_i except when $\mathcal{F}$'s reply does not contain a mask. In this case, we now let $\mathcal{S}$ set the output of $\mathcal{F}_{\mathsf{liPAKE}}$ for $\mathcal{P}_i$ to be a random $K \xleftarrow{\$} \mathbb{F}_q^n$.

Regarding indistinguishability, first note that in any case the input of $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{liPAKE}}$ does not impact any values and thus we only have to argue further in case $\mathcal{S}$ is modified.

Then, it holds that $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \gamma$. Thus, $\mathcal{P}_i$'s record will get interrupted and $\mathcal{P}_i$ will obtain a uniformly random session key from $\mathcal{F}$, meaning that we only have to argue indistinguishability of E, σ_E, vk generated with either K depending on pw_i (as in the previous game) or $K \xleftarrow{\$} \mathbb{F}_q^n$ (as in the current game). Opposed to the situation in Game $\mathbf{G}_9$, note that now $\mathcal{Z}$ knows K'.

Since $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \gamma = n - t - 1$, at most t components of K' are the same as K in $\mathbf{G}_9$ with large probability $1 - \frac{n-t}{q}$, and thus w.h.p. $\mathcal{Z}$ learns at most t components of C. Therefore, the t-privacy of the Secret Sharing scheme states that nothing is leaked about U. Hence the transcript of the current and previous games are indistinguishable.

Observe that now $\mathcal{F}$ is equal to $\mathcal{F}^M_{\mathsf{fpake}}$ and $\mathcal{S}$ is equal to the simulator described in Figure 4.7. The theorem thus follows.

The simulator $\mathcal{S}$, initialized with a security parameter λ, initializes the dummy adversary $\mathcal{A}$. $\mathcal{S}$ emulates an ideal labeled iPAKE functionality $\mathcal{F}_{\mathsf{liPAKE}}$ as depicted in Figure 3.2 for all calling entities in the system[a]. Additionally, $\mathcal{S}$ interacts with an ideal functionality $\mathcal{F}^M_{\mathsf{fpake}}$ and a distinguisher, the environment $\mathcal{Z}$, via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, \mathcal{P}_i, \texttt{role})$ from $\mathcal{F}^M_{\mathsf{fpake}}$:** initialize a party $\mathcal{P}_i$ and connect it to $\mathcal{A}$.
  - If $\mathcal{P}_{1-i}$ is honest, $\mathcal{S}$ proceeds the UC protocol execution as described in Figure 4.5 using $\mathsf{pw}_{\mathcal{S}} \xleftarrow{\$} \mathbb{F}_p$ as password for $\mathcal{P}_i$ and $\mathcal{S}$'s random coins. (Cf. Game $\mathbf{G}_9$.)
  - If $\mathcal{P}_{1-i}$ is corrupted, then $\mathcal{S}$ waits until $\mathcal{P}_{1-i}$ submitted n queries to $\mathcal{F}_{\mathsf{liPAKE}}$ and then assembles $\mathsf{pw}_{\mathcal{Z}} \in \mathbb{F}_p^n$ from them. $\mathcal{S}$ sends $(\texttt{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}})$ to $\mathcal{F}^M_{\mathsf{fpake}}$. If $\mathcal{S}$ receives back a mask M, let $I := [\![1, n]\!] \setminus M$, and $\mathcal{S}$ sets up n $\mathbb{F}_q$-keys K with $K_t = K'_t \ \forall t \in I$ and $K_t \xleftarrow{\$} \mathbb{F}_p \neq K'_t \ \forall t \in M$, where K' denotes the output of $\mathcal{F}_{\mathsf{liPAKE}}$ towards $\mathcal{P}_{1-i}$. $\mathcal{S}$ now continues the simulation of $\mathcal{P}_i$ using K as outputs of $\mathcal{F}_{\mathsf{liPAKE}}$. (see Game $\mathbf{G}_4$.) If $\mathcal{S}$ does not receive a mask, it sets the output of $\mathcal{F}_{\mathsf{liPAKE}}$ for $\mathcal{P}_i$ to be $K \xleftarrow{\$} \mathbb{F}_q^n$. (Cf. Game $\mathbf{G}_{10}$.)
- **If an internally simulated party $\mathcal{P}_i$ produces an output $(\mathsf{sid}, \mathsf{k}_i)$:** Send $(\texttt{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{k}_i)$ to $\mathcal{F}^M_{\mathsf{fpake}}$.
- **If $\mathcal{Z}$ sends $(\mathsf{sid}, M_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, \mathsf{vk}_{\mathcal{Z}})$ to an honest party $\mathcal{P}_i$:** if $\mathcal{P}_{1-i}$ is honest, $\mathcal{S}$ aborts after the Vfy step in the protocol, regardless of its outcome. (Cf. Game $\mathbf{G}_5$.)

Additionally, $\mathcal{S}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any $\texttt{NewSession}$ query from $\mathcal{F}^M_{\mathsf{fpake}}$.

[a] An entity is any internally simulated ITM such as parties or the real-world adversary as well as ITMs outside $\mathcal{S}$ such as the distinguisher $\mathcal{Z}$.

Figure 4.7: The Simulator $\mathcal{S}$ for $\mathsf{FPAKE}^{\mathsf{rss}}$

□

4.2.3 Removing modeling assumptions

All modeling assumptions of our protocol come from the realization of the ideal $\mathcal{F}_{\mathsf{liPAKE}}$ functionality. e.g., the liPAKE protocol from Chapter 3 requires a random oracle, an ideal cipher and a CRS. We note that we can remove everything up to the CRS by, e.g., taking the PAKE protocol introduced in [KV11]. This protocol also securely realizes our $\mathcal{F}_{\mathsf{liPAKE}}$ functionality[3]. However, it is more costly than our liPAKE protocol since both messages each contain one non-interactive zero knowledge proof. Since fPAKE implies a regular PAKE (simply set $\delta = 0$), [CHK+05] gives strong evidence that we cannot hope to realize $\mathcal{F}_{\mathsf{fpake}}$ without a CRS.

[3] In a nutshell, their protocol is implicit-only for the same reason as the liPAKE protocol we use here: there are only two flows that do not depend on each other, so the transcript cannot reveal the outcome of a guess unless it reveals the password to anyone. Looking at the proof in [KV11], it is easily verifiable that the simulator does not make use of the answer of `TestPwd` to simulate any messages. Furthermore, the protocol already implements labels.

Chapter 5

Human authenticated key exchange

We now present another notion derived from PAKE, called *human authenticated key exchange* (HAKE). Its goal is to design a key exchange specifically tuned for human authentication, i.e. when a human is operating one of the parties, called the terminal, and wants to establish a secure key with another party, called the server.

In a regular PAKE, this would be modeled as one of the parties being both the human (that knows the *password*) and the terminal (that can actually do the computations required in the PAKE). Such a model hence implicitly requires perfect trust between human and terminal, which is not really realistic. In real life, the terminal may be compromised, which cannot be accounted for with a PAKE.

Of course, if this is the case, any key computed on this terminal will be likewise compromised. However, one could hope that keys established using PAKE with the same password but an honest terminal will stay secure. This captures both notions of forward secrecy (past sessions are not compromised, as a passive transcript does not contain enough information to deduce the key) and future or backward secrecy (future sessions are not compromised).

The latter can seem hard to achieve since the password is likely leaked in the compromising of the terminal. However, there is hope, in the form of human computation, that is requiring some computation on the part of the human so that the full secret is not leaked to the terminal.

In this chapter, we will present how to model human computation and then a HAKE itself, as well as present solutions to achieve better security.

Human-compatible notions. Here we present several notions that our definitions will use. Since it is hard to formalize human computational abilities, those are not mathematically precise. We say a message is *human-readable* if this is a short sequence of ASCII symbols, or images; *human-writable* if this is a short sequence of ASCII symbols[1]; *human-memorizable* if this is simple enough to be memorized by an average human, e.g., a simple arithmetical rule like "plus 3 modulo 10". A function is *human-computable* if an average human can evaluate it without help of additional resources other than his head, e.g., simple additions modulo 10. A set is *human-sampleable* if an average human can choose a message from the set at random according to the appropriate distribution without help of additional resources other than his head.

[1] It is also possible to incorporate mouse clicks into that, but we do not deal with it for simplicity.

5.1 Human-compatible function family

In order to make our constructions more generic, we first define a new notion, called *human-compatible function family* (HCFF), which captures the relevant security and usability properties we would like to see regarding operations made by the human.

Since this field is quite new, and that it is quite challenging to construct a primitive which uses operations simple enough to be achievable by a human, ideally in his head, and yet is complex enough to offer some security, this notion allows us to abstract our HAKE constructions from the current state of research in this area.

We will restrict ourselves to single-round computations on the part of the human, using an input given by the terminal and a secret information that only the human has access to. This can be modeled as computing a function indexed by the secret information, hence the name of human-compatible function family.

5.1.1 Syntax

A *human-compatible function family* (HCFF) is specified by the challenge space $\mathcal{C}$, the key generation algorithm $\mathcal{KG}$, which takes as input the security parameter and outputs a high-entropy key K, and the *challenge-response* function F that takes a key K and a challenge $x \in \mathcal{C}$ and returns the response $r = F_K(x)$. We require that (see above for definitions):

1. for every K output by $\mathcal{KG}$ and every $x \in \mathcal{C}$, both x and $F_K(x)$ are human-writable and human-readable;

2. $\mathcal{C}$ is human-sampleable.

We also define only-human human-compatible function family (where an additional device is excluded), which are the HCFF that also have:

1. for every K output by $\mathcal{KG}$, $F_K(\cdot)$ is human-computable;

2. every K output by $\mathcal{KG}$ is human-memorizable;

5.1.2 Security

In an authentication protocol with challenge-response pairs, intuitively, we would like that any successful authentication to a server should involve an evaluation of the function by the human user. So we expect no compromised/infected terminal to successfully authenticate to the server one more time than it interacted with the human. The security notion from the function is thus a kind of one-more unforgeability [BNPS03]. But here, any query to an $F_K(\cdot)$-oracle should help to immediately answer $F_K(x)$ to the current challenge x, since a second challenge will come from a new session that has closed the previous one, and so the previous challenge is obsolete: the adversary cannot store the $n+1$ challenges, ask n queries, and answer the $n+1$ initial challenges. In our protocols, the adversary gets a random challenge (`GetRandChal`-query), can ask any $F_K(\cdot)$-query (`GetResp`-query), but should answer that challenge (`TestResp`-query), otherwise the failure is detected. After too many failures (recorded in the unvalidated-query counter `ctr`) one may restrict oracle queries. The following security notion will formalize those restrictions to the adversary, and following potential relaxations.

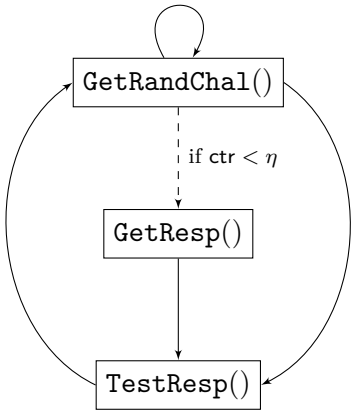

Figure 5.1: Graph of the sequential oracle calls in the η-unforgeability experiment

η-Unforgeability. As said above, we define a kind of sequential one-more unforgeability experiment, with a limit η on the unvalidated-query counter ctr, where the queries follow the graph presented on Figure 5.1. Given an HCFF F, an adversary $\mathcal{A}$, and a public parameter η, one first generates K with $\mathcal{KG}$ and initializes $\mathsf{ctr} \leftarrow 0$. Then the adversary can ask the following queries, with possible short loops on the `GetRandChal`-query and direct `TestResp`-attempt right after getting a challengei x:

1. `GetRandChal`() – It picks a new $x \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. `GetResp`(x^*) – If $\mathsf{ctr} < \eta$ and $x^* \in \mathcal{C}$, it returns $F_K(x^*)$ and increments ctr. It also marks the *fresh* x as *unfresh*. Otherwise, it outputs $\perp$;
3. `TestResp`(r) –
   - If $F_K(x) = r$ and x is *fresh*, the adversary *wins*;
   - If $F_K(x) = r$ and x is *unfresh*, it decrements ctr, marks x as *used*, and outputs 1;
   - Otherwise, it outputs 0.

Because of the sequential iterations, any `TestResp`-query relates to the previous `GetRandChal`-query. One can thus consider one memory slot to store the challenges, but one only at a time: any new challenge replaces the previous one. The dashed line from `GetRandChal` to `GetResp` emphases the restriction on the number of unvalidated queries. When $\mathsf{ctr} \geq \eta$, the adversary has no more choice than immediately trying an answer to the random challenges. The bound η represents the maximum gap that is allowed at any time between the number of `GetResp`-queries and the number of correct `TestResp`-queries. Note that a random challenge x can only be either *fresh*, *unfresh*, or *used*, and that marking it as one of those erases the other flags. Intuitively, a *fresh* challenge has not been compromised in any way, and succeeding at a `TestResp` on it would indicate the unforgeability has been breached, hence the winning status for the adversary, and the experiment stops. A challenge can switch to the *unfresh* state if the adversary asks the `GetResp`-oracle for an answer. There are only two ways for the experiment to stop: if the adversary wins with a correct `TestResp`-query on a fresh challenge; or if the adversary aborts, it then loses the game. We stress that the adversary can query the `GetResp`-oracle on any x^* of its choice, and so possibly different

from the current challenge x obtained with the previous `GetRandChal`-query. But we give it a chance to still answer correctly to the challenge x with the correct `TestResp`-query that, on an *unfresh* challenge, cancels the instrumentation of the counter ctr. This counter represents the gap between the number of `GetResp`-queries and the number of correct `TestResp`-queries on random challenges. When one limits ctr to be at most 1, any `GetResp`-query should be immediately followed by a correct `TestResp`-query (one-more unforgeability).

This definition is a weaker notion than the one-more unforgeability [BNPS03], but still allows the adversary to exploit malleability: For example, with the RSA function, for a random challenge y, the adversary can ask a `GetResp`-query on any $y' = y \cdot r^e \bmod n$, for a r of its choice, so that it can then extract a e-th root of y. But this would not help it to answer a next fresh challenge.

2-Party η-Unforgeability. Unfortunately, the above clean security notion is not enough for a HAKE application, as client-server situations and man-in-the-middle attacks intrinsically allow a more complex ordering of the queries by the adversary. We therefore present a variant of this experiment below, that is suitable for a protocol involving two parties (hence in the following $b \in \{0, 1\}$).

Given an HCFF F, an adversary $\mathcal{A}$, and a public parameter η, one first generates K with $\mathcal{KG}$ and initializes $\mathsf{ctr} \leftarrow 0$. Then the adversary can ask the following queries:

1. `GetRandChal`(b) – It picks a new $x_b \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. `GetResp`(x^*) – If $\mathsf{ctr} < \eta$ and $x^* \in \mathcal{C}$, it returns $F_K(x^*)$ and increments ctr. It also marks all *fresh* x_b as *unfresh*. Otherwise, it outputs $\perp$;
3. `TestResp`(r, b) – If x_b exists:
   - If $F_K(x_b) = r$ and x_b is *fresh*, the adversary *wins*;
   - If $F_K(x_b) = r$ and x_b is *unfresh*, it decrements ctr, marks x_b as *used* and outputs 1;
   - Otherwise, it outputs 0.

The main difference with the previous experiment is the two memory slots for challenges x_0 and x_1. But still, any `GetResp`-query must be followed by a correct `TestResp`-query to limit ctr from increasing too much.

The advantage of any adversary $\mathcal{A}$ against the unforgeability $\mathsf{Adv}_F^{\eta\text{-}\mathsf{uf}}(\mathcal{A})$ is the probability of winning in the above experiment (with a correct `TestResp`-query on a *fresh* challenge). Such a success indeed means that the adversary found the response for a new random challenge, without having asked for any `GetResp`-query.

The resources of the adversary are the polynomial running time and the numbers $q_{\mathsf{c}}, q_{\mathsf{r}}, q_{\mathsf{t}}$ of queries to `GetRandChal`, `GetResp` and `TestResp` oracles, respectively. Of course it is crucial whether there are secure instantiations of HCFF. We propose some in the next section.

Indistinguishability. For some constructions, we will simply expect the sequence of answers $(F_K(x_i))_{i=0,\ldots,T}$ to the challenges x_i (either adversarially chosen or not) to look random, or at least any new element in the sequence is not easy to predict from the previous ones.

For the sake of simplicity, we assume that there exists a global distribution $\mathcal{D}$ with large enough entropy D such that any such sequence is computationally indistinguishable from

$\mathcal{D}^{T+1}$: We denote by $\mathsf{Adv}_F^{\mathsf{dist}\text{-}c}(\mathcal{D}, \mathcal{A})$ the advantage the adversary $\mathcal{A}$ can get in distinguishing the sequence $\{y_0 = F_K(x_0), \ldots, y_{c-1} = F_K(x_{c-1})\}$ for a random K, from $(y_0, \ldots, y_{c-1}) \xleftarrow{\$} \mathcal{D} \times \ldots \times \mathcal{D}$. For the latter distribution, the probability to guess y_{c-1} from the view of $(y_0, \ldots, y_{c-2})$ is $1/2^D$.

(Weak) pseudo-random functions definitely satisfy this property. But from a more practical point of view, the function implemented in the RSA SecurID device [RSA] is believed to satisfy it too, with the x_i being a time-based counter.

5.2 HCFF instantiations

We now propose some possible instantiations for the HCFF defined in the previous section.

5.2.1 Token-based HCFF

First, we introduce a simple token-based HCFF. This assumes that the human is in possession of a simple device on which it can input challenge x and get the response $r \leftarrow F_K(x)$. The device will store K and perform the computation, but the human is still responsible for the communication with the terminal.

This allows us to use strong cryptographic primitives. For instance, we could set $K \xleftarrow{\$} \{0,1\}^\lambda$ and $F_K : [\![0,9]\!]^{t'} \to [\![0,9]\!]^t$ a pseudo-random function. In the random oracle model (for modeling $\mathcal{H}$ in $F_K(x) = \mathcal{H}(K\|x)$), we have $\mathsf{Adv}_F^{\eta\text{-}\mathsf{uf}}(\mathcal{A}) \leq 10^{-t}$ for any adversary and any η, since the best strategy would be for the adversary to just guess by chance the answer to a fresh challenge. Note that this function is obviously *human-readable*, *human-writable* and *human-sampleable* as its input/output are numbers in basis 10 so it is an HCFF.

Hence this function family is a good candidate to use in our Basic HAKE protocol from Section 5.5.1 or its simplified version from Section 5.6.1.

5.2.2 Only-human HCFF

However, avoiding such devices would be much better in practice. We are thus interested in the only-human HCFF that would not require anything beyond simple human memory and brain computation power. Since such a function is necessarily weaker, we will use it in our Confirmed HAKE protocol from Section 5.5.2, which has a much tighter control over adaptive queries and therefore requires weaker security properties from the HCFF.

5.2.2.1 Construction

We present a candidate based on the construction of Blocki et al. [BBDV17], which security is based on [FPV15]: Consider a challenge space $\mathcal{C} = \mathcal{X}_l^t \subseteq [\![1,n]\!]^{lt}$, where $[\![1,n]\!] = \{1, \ldots, n\}$ is the set of n integers each representing one of the n variables and $\mathcal{X}_l$ denotes the space of ordered clauses of l variables without repetition. The parameter t indicates that each challenge consists of t independent clauses, i.e., "small" challenges. The key generation algorithm $\mathcal{KG}$ of our HCFF takes as input a parameter n, then outputs a random mapping $\sigma : [\![1,n]\!] \to \mathbb{Z}_q$ as the key K, where the integer q is a constant. Usually we set $q = 10$ because most humans are familiar with computations on digits. Let $\sigma^l : [\![1,n]\!]^l \to \mathbb{Z}_q^l = (\sigma, \cdots, \sigma)$ denote the mapping that applies σ to each element of a l-tuple. Using a public human-computable function $f : \mathbb{Z}_q^l \to \mathbb{Z}_q$ that is instantiated later, the challenge-response function F takes a key $K = \sigma$

and a challenge $x \in \mathcal{C}$ as inputs, and returns a response $r = F_K(x)$. Here $F_K : \mathcal{C} \to \mathbb{Z}_q^t$ is defined as a t-tuple $(t \geq 1)$ $(f \circ \sigma^l, \cdots, f \circ \sigma^l)$, where $\circ$ indicates the function composition.

For instance, if $n = 100$, $l = 3$, $q = 10$, $t = 2$, $x = ((1, 4, 20), (3, 36, 41))$, $\sigma(i) = (i + 3) \mod 10$ and $f = (x_1 - x_2 + x_3) \mod 10$, then $\sigma((1, 4, 20)) = (4, 7, 3)$, $\sigma((3, 36, 41)) = (6, 9, 4)$ and $F_K(x) = (0, 1)$.

Given integers $k_1, k_2 > 0$, the function f is instantiated as $f_{k_1,k_2} : \mathbb{Z}_{10}^{10+k_1+k_2} \to \mathbb{Z}_{10}$, which is defined as follows:

$$f_{k_1,k_2}(x_0, \ldots, x_{9+k_1+k_2}) = x_{\left(\sum_{i=10}^{9+k_1} x_i \mod 10\right)} + \sum_{i=10+k_1}^{9+k_1+k_2} x_i \mod 10.$$

Note that when f is instantiated as f_{k_1,k_2}, we have $l = 10 + k_1 + k_2$ and $q = 10$.

It is easy to see that such a function family is an only-human HCFF apart from the human memorization property. However, we can allow for images to represent the variables. As illustrated in [BBDV17], by using mnemonic helpers, humans are able to remember such mappings from images to digits. As evidence, the primary author of [BBDV17] was able to memorize a mapping from $n = 100$ images to digits in 2 hours.

5.2.2.2 Candidate's security

In [BBDV17], the authors proved the intractability to answer a new random challenge for the above HCFF instantiation based on the conjecture about the hardness of *random planted constraint satisfiability problem* (RP-CSP). In this section, we briefly recall one of their results, the RP-CSP conjecture, siglhtly simplified for our purpose (and thus we call it the RP-CSP * conjecture for disambiguation), and then proceed to show how to use it to achieve η-unforgeability.

The RP-CSP * conjecture. Before stating this conjecture, we introduce some notations as in [BBDV17]. Recall how $d(\alpha_1, \alpha_2) = |\{i \in [\![1, n]\!] \mid \alpha_1[i] \neq \alpha_2[i]\}|$ is the Hamming distance between two strings $\alpha_1, \alpha_2 \in \mathbb{Z}_q^n$. Then we say two mappings $\sigma_1, \sigma_2 \in \mathbb{Z}_q^n$ are *ε-correlated* if $d(\sigma_1, \sigma_2)/n \leq (q - 1)/q - \varepsilon$.

Conjecture 1 (RP-CSP *)**.** *Consider the function f_{k_1,k_2} described above, for any $\varepsilon, \varepsilon' > 0$ and any probabilistic polynomial time (in n) adversary $\mathcal{A}$, there exists an integer $N \in \mathbb{N}$, such that for all $n > N$, $m \leq n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}}$, we have $\mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) = \mathrm{negl}(n)$, where $\mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon)$ is the probability that $\mathcal{A}$ outputs a mapping σ' that is ε-correlated with the secret mapping σ given m random "small" challenge-response pairs $\{(C_i, f_{k_1,k_2}(\sigma^l(C_i)))\}_{1 \leq i \leq m}$.*

Remark 5.2.1. *The RP-CSP conjecture in [BBDV17] is a general version of the RP-CSP * Conjecture 1, where f can be instantiated as other functions. Here, for simplicity, we only state the conjecture where $f = f_{k_1,k_2}$. In [BBDV17], the authors also prove strong evidence in support of the RP-CSP conjecture: it holds for any statistical adversary and any Gaussian Elimination adversary. As observed in[FPV15], most natural algorithmic techniques have statistical analogues except the Gaussian Elimination.*

Basic η-unforgeability. To state the security theorem in [BBDV17], we need the following "basic" HCFF security notion that is a "non-malleable" version of the η-unforgeability. It

indeed assumes that asking a GetResp-query with an input different from the current random challenge should not help to answer this challenge correctly to the TestResp-query. Given an HCFF F, an adversary $\mathcal{A}$, and a public parameter η, one first generates K with $\mathcal{KG}$ and initializes $\mathsf{ctr} \leftarrow 0$. Then the adversary can ask the following queries:

1. GetRandChal() – It picks a new $x \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. GetResp(x^*) – It increments ctr if $x^* \neq x$;
   - If $\mathsf{ctr} \leq \eta$ and $x^* \in \mathcal{C}$, it outputs $F_K(x^*)$ and marks x as *unfresh*;
   - Otherwise, it outputs $\perp$;
3. TestResp(r) –
   - If $F_K(x) = r$ and x is *fresh*, the adversary *wins*;
   - If $F_K(x) = r$ and x is *unfresh*, it outputs 1;
   - Otherwise, it outputs 0.

Just like the η-unforgeability experiment, the above oracle calls are sequential (similar to Figure 5.1), starting with a GetRandChal-query. But since non-malleability is assumed, only GetResp-queries with inputs different from the current random challenges make the counter increase, and it is never decreased.

The advantage of any adversary $\mathcal{A}$ against the above unforgeability $\mathsf{Adv}_F^{\eta\text{-uf-basic}}(\mathcal{A})$ is the probability of winning in the above experiment. Such a success indeed means that the adversary found the response for a new random challenge, without having asked for a GetResp-query. The parameter η restricts the number of "adaptive" GetResp queries that $\mathcal{A}$ can make, where adaptive means "different from the current random challenge".

The resources of the adversary are the polynomial running time and the numbers $q'_{\mathsf{c}}, q'_{\mathsf{r}}, q'_{\mathsf{t}}$ of queries to the above GetRandChal, GetResp and TestResp oracles, respectively. For convenience, denote by q''_{t} the number of TestResp-queries such that the current random challenge x is *fresh*. By definition, we have $q'_{\mathsf{r}} \leq q'_{\mathsf{c}}$, $q'_{\mathsf{t}} \leq q'_{\mathsf{c}}$ and $q''_{\mathsf{t}} \leq q'_{\mathsf{c}} - q'_{\mathsf{r}}$.

HCFF security results. Under Conjecture 1, one can prove the following unforgeability result about the HCFF.

Theorem 5.2.2 (From [BBDV17]). *Given $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and $\delta > (\frac{1}{10} + \varepsilon)^t$, for any probabilistic polynomial time (in $n, q'_{\mathsf{c}}, 1/\varepsilon$) adversary $\mathcal{A}$ against the basic 0-unforgeability security of the human-compatible function family F constructed above using $f = f_{k_1,k_2}$ with*

$$q''_{\mathsf{t}} = 1, q'_{\mathsf{c}} \leq \frac{1}{t} \cdot n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}} - 1,$$

under Conjecture 1, we have $\mathsf{Adv}_F^{0\text{-uf-basic}}(\mathcal{A}) < \delta$.

Note that in the basic 0-unforgeability security game, the adversary learns nothing from GetResp(x^*) if x^* is not the current random challenge x. So if $\eta = 0$, the adversary $\mathcal{A}$ is only given random challenge-response pairs.

This result is actually not strictly good enough, even for our Confirmed HAKE. Indeed, this protocol's security stems from it's ability to detect bad behaviour (notably, adaptive querying) by one of the parties. However, if the HCFF function does not allow for at least

one adaptive query, an attacker using one might be able to break the unforgeability of the function fast enough to be able to fool the detection mechanism. Thus, we extend the RP-CSP conjecture to allow $\log n$ adaptive "small" challenge-response pairs.

Lemma 5.2.3. *Consider the function f_{k_1,k_2} described above, for any $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and any probabilistic polynomial time (in n) adversary $\mathcal{A}$, there exists an integer $N \in \mathbb{N}$, such that for all $n > N$, $m_r \leq n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}}$ and $m_a \leq t \log n$, we have $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) = \mathrm{negl}(n)$, where $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon)$ is the probability that $\mathcal{A}$ outputs a mapping σ' that is ε-correlated with the secret mapping σ given m_r random "small" challenge-response pairs and the correct responses to m_a "small" challenges adaptively chosen by $\mathcal{A}$.*

Proof. For any adversary $\mathcal{A}$ we can construct an adversary $\mathcal{B}$ such that $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) \leq 10^{t \log n} \times \mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{B}, \varepsilon)$.

$\mathcal{B}$ simulates $\mathcal{A}$'s view by providing $\mathcal{A}$ with the given m_r random "small" challenge-response pairs and randomly guessing the responses to the m_a ($\leq t \log n$) adaptive "small" challenges. The probability of correctly guessing all adaptive ones is $10^{-t \log n}$ (refer to the construction of f_{k_1,k_2}), hence the above advantage reduction. One should note that $10^{t \log n} \times \mathrm{negl}(n) = \mathrm{negl}(n)$ and $\mathcal{B}$'s running time is polynomial in n. □

Using this lemma, one can prove the following stronger unforgeability result about the HCFF, which "almost" suits our Basic HAKE (see Figure 5.3):

Theorem 5.2.4. *Given $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and $\delta > (\frac{1}{10} + \varepsilon)^t$, for any probabilistic polynomial time (in $n, q'_{\mathsf{c}}, 1/\varepsilon$) adversary $\mathcal{A}$ against the basic η-unforgeability security of the human-compatible function family F constructed above using $f = f_{k_1,k_2}$ with*

$$\eta \leq \log n, q'_{\mathsf{c}} \leq \frac{1}{t} \cdot n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}} - 1,$$

under Conjecture 1, we have $\mathsf{Adv}^{\eta\text{-uf-basic}}_{F}(\mathcal{A}) < q''_{\mathsf{t}} \cdot \delta$.

Proof. The proof is almost the same as that of Theorem 5.2.2. Informally, we need to show that any adversary $\mathcal{A}$ that breaks the basic η-unforgeability security of the HCFF can also "recover" the secret mapping σ in Conjecture 5.2.3. The reader can refer to the proof of Theorem 5 in [BBDV17], which we call the "HCP" proof below, for the details.

Nevertheless, here the theorem differs from Theorem 5.2.2 in several aspects. First, $\mathcal{A}$ can adaptively select $t \log n$ "small" challenges to get the correct responses, while in Theorem 5.2.2 only random ones are allowed. But having adaptive queries does not affect the HCP proof because it only uses $\mathcal{A}$ as a black box to predict the responses to any t "small" challenges. Second, we apply a union bound of q''_{t} queries to the final advantage. □

Remark 5.2.5. *In the above theorem $\varepsilon, \varepsilon'$ are almost 0. We can set $n = 100$, $k_1 = 1$, $k_2 = 3$ and $t = 5$, then $\eta \leq 6$, $q'_{\mathsf{c}} \leq n^2/t - 1 \approx 2000$ and $\mathsf{Adv}^{\eta\text{-uf-basic}}_{F}(\mathcal{A}) < q''_{\mathsf{t}} \cdot 10^{-t} \leq 1/50$.*

We believe a similar theorem holds for $\mathsf{Adv}^{\eta\text{-uf}}_{F}$ (by replacing the oracles with those in the 2-party η-unforgeability experiment), which our HAKE security can rely on. The intuition is as follows. With the HCFF instantiation described in this section, a $\texttt{GetResp}(x^*)$ query in the η-unforgeability experiment should not have x^* too "far" from the random challenge x output by the latest `GetRandChal` query. Otherwise, it is very unlikely for the adversary to

guess correctly in the `TestResp` query. But the adversary can modify x a little bit to guess the correct response with a smaller failure probability. This is the difference between the two unforgeability notions: the basic one does not tolerate any malleability, whereas the other can exploit malleability. Because of the size of the challenge space, that has to be quite large (it is essentially $n^{t(10+k_1+k_2)}$, and thus 2^{465}, with the above parameters), the number of challenges that are "close" to any random challenge accounts for a tiny proportion. Thus, the adversary should not get much help from such "nearly random" challenges. Besides, such queries risk increasing the counter in the `GetResp` oracle without extracting much useful information. In addition, the two memory slots will not increase much the advantage of an adversary, and so $\mathsf{Adv}_F^{\eta\text{-uf-basic}}$ and $\mathsf{Adv}_F^{\eta\text{-uf}}$ should be quite close for this specific HCFF instantiation. We leave further studies of security of the HCFF from [BBDV17] to future works.

5.3 HAKE definition

We now define HAKE as an extension of PAKE.

Protocol participants. We fix the set of participants to be $\mathsf{ID} = \{U_\ell\}_\ell \cup \{T\} \cup \{S\}$, which contains finite number of human users U_ℓ, one terminal T and one server S. And we assume that each member is uniquely described by a bit string. In the real life, each user U_ℓ can communicate with multiple servers via multiple terminals. But we justify below why considering a single terminal and a single server is sufficient

HAKE syntax. We now formally describe a HAKE protocol.

Definition 5.3.1 (HAKE Protocol)**.** *A human authenticated key exchange protocol is an interactive protocol between a human user denoted $U \in \{U_\ell\}_\ell$ and the server S, via the terminal T. It consists of two algorithms:*

- *A long-term key generation algorithm $\mathcal{LKG}$ which takes as input the security parameter and outputs a long-term key.*
- *An interactive key exchange algorithm $\mathcal{KE}$ which runs between U, T, and S. At the beginning, only U and S take as input the same long-term secret key and, at the end, T and S each outputs a session key sk_T and sk_S respectively. In case of additional explicit authentication, U and/or S may either* accept *or* reject *the connection.*

The above algorithms must satisfy the following constraints:

- *S can only communicate with T;*
- *U can only communicate with T, and*
  - *messages sent by T to U must be* human-readable*, and*
  - *messages sent by U to T must be* human-writable*;*
- *The long-term secret and the state of U, if any, must be* human-memorizable *for the duration necessary.*

The **correctness** condition requires that for every security parameter and for every long-term key output by $\mathcal{LKG}$, in any execution of $\mathcal{KE}$, U and S both accept the connection (in case of explicit authentication), T and S complete the protocol with the same session key ($\mathsf{sk}_T = \mathsf{sk}_S$).

5.4 HAKE security model

In this section, we formally define a simulation-based security model for a HAKE protocol that is derived from the BPR security model from [BPR00] (see Section 2.2.3 for a gist of the basic BPR model). As already mentioned, the goal of a HAKE protocol is to ensure that a human user sharing the long-term secret with a server can help a terminal to establish a secure channel with the server, in presence of a very powerful attacker, including strong corruptions of terminals.

As usual, to model multiple and possibly concurrent (except for the human users) sessions we consider oracles π_P^j, where $j \in \mathbb{N}$ and $P \in \mathsf{ID}$. For human oracles, sessions can only be sequential, and not concurrent, meaning that humans are not allowed to run several sessions concurrently (a new session starts after the previous one ends). This is a reasonable assumption for human users. We note that since terminals do not store long-term secrets and do not preserve state between sessions, multiple terminal oracles model both multiple sessions ran from the same or different terminals.

Hence, in the following, we will consider several human users U_ℓ with different long-term secret keys, one terminal T, and one server S, with all the users' long-term secret keys. For all of them, multiple instances will model the multiple sessions (either sequential for U_ℓ, or possibly concurrent for T and S). However, while the server can concurrently run several sessions, we will also limit it to one session at a time with each user: the server will not start a new session with a user until it finishes the previous session with the same user.

Because of our specific context with a human user, there is a direct communication link between the user and the terminal, and so we can assume that the channels between instances $\pi_{U_\ell}^i$ and π_T^j are authenticated and even private (unless the terminal oracle is *compromised*, as defined below), whereas the communication between the terminal and the server is over the Internet, and so the channels between instances π_T^j and π_S^k are neither authenticated nor private.

Security experiments. We consider the following security experiments associated with a given HAKE protocol and an adversary $\mathcal{A}$, to define the two classical security notions for any authenticated key exchange: privacy (or semantic security of the session key) and authentication. In these experiments, the adversary $\mathcal{A}$ can make the following queries:

- $\texttt{Compromise}(j, \ell)$, where $j, \ell \in \mathbb{N}$ – As the result of this query, the terminal oracle π_T^j is considered to be *compromised*, and the adversary gets its internal state, i.e. the random tape, temporary variables, etc. If the terminal oracle π_T^j is not linked yet to a user, it is linked to user U_ℓ with the user oracle $\pi_{U_\ell}^i$ for a new index i, otherwise ℓ is ignored;

- $\texttt{Infect}(j)$, where $j \in \mathbb{N}$ – As the result of this query, the terminal oracle π_T^j is considered to be *infected*. wlog, we limit this query to *compromised* terminals only;

- $\texttt{SendTerm}(j, M)$, where $j \in \mathbb{N}$ and $M \in \{0,1\}^* \cup \{\texttt{Start}(\ell)\}$ – This sends message M to π_T^j. A specific $\texttt{Start}(\ell)$ message asks the terminal to initiate a session, that will be conducted with a user oracle $\pi_{U_\ell}^i$ for a new index i, unless the terminal oracle π_T^j was already linked to a user, in which cas ℓ is ignored. To compute its response to $\mathcal{A}$, π_T^j may internally talk to its linked human oracle according to the protocol. In addition,

if π_T^j is *compromised*, it will additionally give to $\mathcal{A}$ the messages exchanged with its linked human oracle[2].

- $\texttt{SendServ}(k, M)$, where $k \in \mathbb{N}$ and $M \in \{0,1\}^*$ – This sends message M to oracle π_S^k. The oracle computes the response according to the corresponding algorithm and sends the reply to $\mathcal{A}$.
- $\texttt{SendHum}(j, M)$ where $j \in \mathbb{N}$ and $M \in \{0,1\}^*$ (and human-readable) – This sends a message to the π_T^j-linked human oracle $\pi_{U_\ell}^i$ on behalf of π_T^j. This is allowed only if the terminal π_T^j is *infected* (and thus *compromised*, which implies the existence of a partnered human oracle). The oracle computes the response according to the corresponding algorithm and sends the reply to $\mathcal{A}$.
- $\texttt{Test}(j, P)$, where $j \in \mathbb{N}$ and $P \in \{T\} \cup \{S\}$ – If sk_P has been output by π_P^j, then one looks at the internal bit b (flipped once for all at the beginning of the privacy experiment, while $b = 1$ in the authentication experiment). If $b = 1$, then $\mathcal{A}$ gets the real session key sk_P, otherwise it gets a uniformly random session key. This query is only allowed if π_P^j is fresh (defined below).

In the **privacy** experiment, after having adaptively asked several of these oracle queries, the adversary $\mathcal{A}$ outputs a bit b' (a guess on the bit b involved in the `Test`-queries). The intuition is that the adversary should not be able to distinguish the real session keys from independent random strings. While in the **authentication** experiment, the goal of the adversary is to make an honest party to successfully complete the protocol execution thinking it "built a secure session" with the right party, whereas that is not the case. In order to formally define the goals and the advantages of the adversary, we present the notions of partnering and freshness, as well as the flags accept and terminate.

Flags. In order to model authentication, we follow BPR [BPR00], who defined two flags: accept essentially means that a party has all the material to compute the session key while terminate means that a party thinks that it completes the protocol execution thinking it communicates with the expected other party (a human user in our case). These two flags are initially set to `False`, and they are explicitly set to `True` in the description of the protocol. Note that in Definition 5.3.1 U and/or S *accept* if and only if in the end the terminate flag is set to `True`, otherwise, U and/or S *reject*.

Partnering. Whereas $\pi_{U_\ell}^i$ and π_T^j are declared as *linked* at the initialization of the communication because of the authenticated channels between users and the terminal, partnering between $\pi_{U_\ell}^i$ and π_S^k is *a posteriori*: they are indeed declared *partners* in the end of the protocol execution if they use the same long-term key and both accept. Then we define partnering between π_T^j and π_S^k, by saying that they are declared *partners* if π_S^k and U_ℓ^i are partners and U_ℓ^i is linked to π_T^j.

Freshness. Informally, the freshness denotes oracles that hold sessions keys that are not trivially known to the adversary.

[2]The messages to the human oracle can already be deduced by the adversary as they are a function of the oracle's random tape, but we give the adversary the whole communication for convenience.

For $P \in \{T\} \cup \{S\}$, the oracle π_P^j is *fresh*, if no `Test`-query has been asked to π_P^j nor its partner, and none of π_P^j or its partner have been *compromised* (π_T^j is fresh if it has not been compromised, and π_S^k is fresh if the terminal *linked* to the partner human user has not been compromised.).

Security notions. In the **privacy** security game, the goal of the adversary is to guess the bit b involved in the `Test`-queries. Then we measure the success of an adversary $\mathcal{A}$ that outputs a bit b', by $\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \cdot \Pr[b' = b] - 1$. This notion implies *implicit* authentication, which essentially means that no one other than the expected partners knows the session key material.

For *explicit* authentication, we define the **authentication** security game: the goal of the adversary is essentially to make a player terminate (flag $\mathsf{terminate}$ set to true) without an accepting partner (flag accept set to true). But in our case with *compromised* or even *infected* terminals, this is a bit more complex than usual. We thus split the authentication security in two parts:

- Server-authentication: a user oracle should not successfully terminate a session if there is not exactly one partner server oracle that has accepted. Then, we denote by $\mathsf{Adv}^{\mathsf{s\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A})$ the probability the adversary $\mathcal{A}$ makes such a bad event happens;
- User-authentication: a server oracle should not successfully terminate a session if there is not exactly one partner user oracle that has accepted. Then, we denote by $\mathsf{Adv}^{\mathsf{u\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A})$ the probability the adversary $\mathcal{A}$ makes such a bad event happens.

Eventually, for any adversary $\mathcal{A}, \mathcal{B}$ there exists an adversary $\mathcal{C}$ against the authentication security for which we define $\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{HAKE}}(\mathcal{C}) = \max\{\mathsf{Adv}^{\mathsf{s\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A}), \mathsf{Adv}^{\mathsf{u\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{B})\}$.

Passive sessions. We now define a new notion of *passive session*, which replaces the `Execute`-queries in the standard BPR model [BPR00] (see Section 2.2.3 for a discussion of why). Recall that `Execute`-queries allow the adversary to get full transcripts of communication between honest parties. Even though the same can be achieved via `Send`-queries, in the security analysis it is useful to count the number of observed honest sessions and the number of maliciously altered sessions separately. In addition, we will not limit to full sessions: the adversary can stop forwarding honest flows, making the session abort. Then, there can be *passive full/partial sessions*:

Definition 5.4.1 (Passive Session)**.** *A (full or partial) session between oracles π_T^j and π_S^k is called* passive*, if the messages of all queries $\mathtt{SendTerm}(j, \cdot)$ or $\mathtt{SendServ}(k, \cdot)$ are either $\mathtt{Start}(\cdot)$ or themselves an output of one of these two queries type. If flows are numbered, this also implies that the actual order of flows between T and S has not been modified. If all the outputs have been forwarded as inputs, this is a* passive full session*, otherwise this is a* passive partial session*.*

Sessions that are not passive are called *active*, since the adversary altered something in the honest execution.

This notion is stronger than the `Execute`-queries defined in the BPR security model, since the adversary does not need to decide from the beginning if all the exchanges will be passive or not. $\mathcal{A}$ can start with a passive sequence and decide at some point to stop (passive partial session) or behave differently in an adaptive way (active session).

Resources of the adversary. When doing security analyses, for every adversary and its privacy and authentication advantages, one also has to specify the adversarial resources such as the running time t, the number of oracle queries, the number of player instances, and the numbers $n_{\mathsf{passive}}/n_{\mathsf{active}}$ of (fully) passive and active sessions the adversary needs.

Discussion. We discuss a bit more about our security definitions to explain why they capture the practical threats. First, a passive network adversary is able to observe legitimate communications via `SendServ` and `SendTerm`-queries (these will satisfy the passive session definition). An active network adversary can modify legitimate messages or impersonate a terminal or a server by injecting some messages of its choice, again, via `SendServ` and `SendTerm`-queries. This models, in the standard way, possible insecurity (in terms of privacy or authentication) of the network channel between terminals and servers.

Passive-insider attacks (such as key logger and screen capture malware compromising computers or their browsers) are modeled by `Compromise`-queries followed by `SendTerm`-queries. The former gives the adversary full information about the terminal's internal state, including its random coins and registers' contents, and the latter reveals to the adversary the inputs from the human.

We consider even more powerful attackers who can take full control of the computers or some of their crucial applications such as browsers. In this case, in addition to learning the internal state and all the inputs, the adversary can impersonate the honest terminal while sending adaptively selected messages to the human. We model this by `Infect` and `SendHum`-queries.

Our model captures all the above scenarios and, moreover, it takes into account the possibility of multiple simultaneous attacks, such as colluding network and malware adversaries. One can notice that attacks involving `Infect`-queries are stronger than those with `Compromise`-queries: when an adversary infects a terminal, it takes full control on it, with knowledge of its internal state, and thus plays on its behalf, using `SendServ` and `SendHum`-queries.

Note that in any case, we are concerned with the security of a new session, in terms of privacy and authentication, over an honest terminal, that is neither compromised nor infected. Such security should be guaranteed even though the other sessions involving the same human with the same long-term secret were carried over compromised terminals, and if possible even over infected terminals. We model privacy via the `Test`-query and with the appropriate privacy advantage definition. We model authentication via the corresponding advantage definition.

We also stress that we do not consider corruption of the long-term secrets, since they are known by the users and the server only, and we do not allow to corrupt them. Would the long-term secret be leaked, we cannot guarantee any security for future sessions. The interesting open problem of dealing with such corruptions could be addressed using an asymmetric long-term secret: a verifier-based variant that would just provide an encoded version of the user's secret to the server.

5.5 Generic HAKE constructions

In this section, we propose two generic HAKE protocols. They build on a simple idea of composing a HCFF with a regular PAKE protocol used in black box. More precisely, a server chooses a random challenge x, the user U_ℓ's response is $r = F_{K_\ell}(x)$, where F is an HCFF and

Human $U_\ell(K_\ell)$		Terminal T		Server $S(K_\ell)$
	$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$		
		$(c, s) \leftarrow \mathsf{Com}(x_T)$	$\xrightarrow{\ell, c}$	
	$\xleftarrow{x}$	$x \leftarrow g(x_S + x_T)$	$\xleftarrow{x_S}$	$x_S \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$
$r \leftarrow F_{K_\ell}(x)$	$\xrightarrow{r}$		$\xrightarrow{s}$	$x_T \leftarrow \mathsf{Open}(c, s)$
				$x \leftarrow g(x_S + x_T)$
				$r \leftarrow F_{K_\ell}(x)$
			$\xleftrightarrow{\text{PAKE}(r)}$	
			$\text{PAKEsid} = (\ell, c, x_S, s)$	
		Outputs sk_T		Outputs sk_S

Figure 5.2: Basic HAKE Construction

K_ℓ is the long-term secret shared between the user and the server. And finally the terminal and the server execute the PAKE on the one-time password r, as in [PS10]. As already mentioned, whereas the server supports concurrent sessions, since the human does not, there is no sense in maintaining multiple session states for one human user.

However, a straightforward replay attack is possible. The adversary can first just eavesdrop a session by compromising a terminal, and then play on behalf of the server with the observed challenge-response pair (x, r), even when the user uses an honest terminal. The main issue is that there is no reason for the challenge to be distinct in the various sessions if we do not add a mechanism to enforce it. In [PS10]'s constructions, they assume the server is stateful to prevent it. However, we can do better.

This is the goal of our first protocol: it adds a coin-flipping protocol between the terminal and the server to avoid either party to influence the challenge x, and thus to avoid the aforementioned replay attacks. We prove it secure (in terms of privacy, which implies implicit authentication) assuming security of commitments (underlying the coin-flipping), HCFF, and PAKE. However, the concrete security depends on the bound η, which is large enough for our device-based HCFF, but the only-human HCFF construction we will propose in Section 5.2 (and its underlying hardness problem) does not tolerate a high η.

Hence, the goal of our second HAKE protocol is to add explicit authentication, which will help limiting the number of malicious challenge-response pairs the adversary can see, or at least to detect them: the user can then suspect the terminal to be infected. We still need the concrete HCFF to tolerate at least one malicious challenge, but this remains a reasonable assumption.

5.5.1 The Basic HAKE

Our first construction makes use of a commitment scheme, an HCFF, and a PAKE.

Description. Let $(\mathcal{KG}, F)$ be a human-compatible function family with challenge space $\mathcal{C}$, let $\mathcal{CS} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme, let pake be a password authenticated key exchange protocol and let $g : \mathbb{Z}_{\lvert\mathcal{C}\rvert} \to \mathcal{C}$ be a bijection. We construct the Basic HAKE $(\mathcal{LKG} = \mathcal{KG}, \mathcal{KE})$. Its interactive $\mathcal{KE}$ protocol is described on Figure 5.2, here are the descriptions.

- $\mathcal{KE}$ execution:

1. When the user invokes a terminal to establish a connection with the server, the terminal chooses its part of the challenge x_T, and commits it for the server. It also sends the user's identifier ℓ;
2. Upon receiving the commitment, the server waits until any previous session for U_ℓ finishes, then it chooses its part of the challenge x_S, and sends it in clear to the terminal;
3. The terminal then combines both parts x_T and x_S to generate the challenge $x = g(x_S + x_T)$, and asks x to the user;
4. Upon reading the challenge x, the user computes and writes down the response r for the terminal;
5. When the terminal receives the response r from the human user, it opens its commitment to the server, and can already start with the PAKE protocol execution;
6. Upon receiving the opening value of the commitment, the server opens the latter to get x_T. It can then combine both parts x_T and x_S to generate the challenge $x = g(x_S + x_T)$, and compute the response r. It can then proceed with the PAKE protocol too.

The terminal and the server both run the PAKE protocol with their (expected) common input r and a session id that is the concatenation of the transcript. At the end of the PAKE execution, they come up with two session keys, sk_T and sk_S, respectively, that will be equal if both parties used the same r in the PAKE. Since we do not consider explicit authentication, accept and terminate flags are not set.

Correctness of the Basic HAKE construction follows from correctness of the building blocks.

Security analysis. For Basic HAKE, we only assess privacy of the session key, since this protocol does not provide explicit authentication[3].

Theorem 5.5.1. *Consider the Basic HAKE protocol defined in Figure 5.2. Let $\mathcal{A}$ be an adversary against the privacy security game with static compromises, running within a time bound t and using less than n_{comp}* compromised *terminal sessions, n_{uncomp}* uncompromised *terminal sessions, n_{serv} server sessions and $n_{\mathsf{active}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ active sessions. Then there exist an adversary $\mathcal{B}_4$ attacking the 2-party n_{comp}-unforgeability of the HCFF with $q_{\mathsf{r}}, q_{\mathsf{c}}, q_{\mathsf{t}}$ queries of the corresponding type, an adversary $\mathcal{A}'$ and a distinguisher $\mathcal{B}_3$ attacking UC-security of the PAKE with a simulator $\mathcal{S}_{pake}$ as well as three adversaries $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{B}'_2$ against the commitment scheme properties, all running in time t, such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{HAKE}(\mathcal{A}) \leq\ & \mathsf{Adv}^{n_{\mathsf{comp}}\text{-}\mathsf{uf}}_{F}(\mathcal{B}_4) + 2 \times \mathsf{Adv}^{pake}_{pake}(\mathcal{S}_{pake}, \mathcal{A}', \mathcal{B}_3) \\ & + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_2)\right) ,\end{aligned}$$

where $q_{\mathsf{r}} \leq n_{\mathsf{comp}}$, $q_{\mathsf{t}} \leq n_{\mathsf{active}}$, and $q_{\mathsf{c}} \leq n_{\mathsf{uncomp}} + n_{\mathsf{comp}} + n_{\mathsf{serv}}$.

[3] There is no hope of achieving any kind of server authentication implicitly, since the human is responsible for authentication in our security model, and he does not handle keys. User authentication could be achieved, but we defer the analysis to the next protocol.

Discussion. Concrete security of the HCFF is definitely the most crucial compared to that of other building blocks, since it is hard to balance strong security and usability. This is why we emphasize this in the above theorem.

We note that the sessions which lead to `TestResp`-queries have *non-oracle-generated* flows and therefore correspond to classical on-line dictionary attacks: the adversary simply tries to impersonate the user/terminal to the server (or vice-versa), with a guess for the answer r (unless the query has been asked to the `GetResp`-oracle). Indeed, sessions with `GetResp`-queries on the exact challenges have *compromised* terminals and correspond to spyware key loggers that record random challenge-response pairs. If using such a compromised terminal can be considered rare, this remains reasonable. Eventually, the sessions which lead to `GetResp`-queries on different challenges are the most critical, but they should likely fail. And we expect them to be quite exceptional. As remarked above, such sessions likely conclude to a failure: no concrete session is established with the server the user wanted to connect to (if the HCFF is still secure after such adaptive queries). If the user can detect such a failure, he can run away from this terminal. We now propose a way for the user to detect such a dangerous terminal, and thereafter take appropriate measures. At the same time, our next proposal will achieve *explicit* authentication.

Proof. We define a series of games that aims at bounding the privacy advantage of $\mathcal{A}$. We denote $\mathcal{B}_i$ the simulator for Game $\mathbf{G}_i$, that outputs 1 if $b \leftarrow \mathcal{A}$ ($\mathcal{A}$ wins in guessing b) and 0 otherwise. We also define $\mathcal{D}_i$ the distinguisher between games $\mathbf{G}_i$ and $\mathbf{G}_{i-1}$, for $i > 1$. In general, the distinguisher $\mathcal{D}_i$ between the two successive games exactly behaves as the simulator in $\mathbf{G}_{i-1}$ with all the secrets, but just interactions with two distributions to distinguish (either on sets or on oracles). The distributions on the outputs are thus as close as the input distributions which are close enough under a computational assumption. We start from $\mathbf{G}_0$, the privacy security game, between the adversary $\mathcal{A}$ and the challenger. We rewrite it below, with explicit definitions of the queries. Then, in the last game, we explain how a simulator does without knowing anymore the long-term secret keys but using instead the oracles `GetResp`, `GetRandChal`, and `TestResp` to replace calls to F_{K_ℓ}.

We stress that the corruptions are static, which means that `Compromise`-queries must happen before any other flow in a terminal session. However, infections can still be adaptively made (but on compromised sessions only).

We also require the internal PAKE primitive (denoted by the double arrow on the Figure 5.2) to be UC-secure (see Figure 2.2), as we will need the ideal functionality $\mathcal{F}_{\mathsf{pake}}$ and the simulator $\mathcal{S}_{\mathsf{pake}}$ in the proof. In particular, we need to be able to simulate transcripts between honest players, without knowing the password, and we will need to be able to extract the password tried by the adversary. On the other hand, we will have to be able to simulate the answers to the `TestPwd` queries.

In several games, some steps will depend on whether the input message is oracle-generated, meaning it is the output of another oracle and whether the terminal (be it the source of an oracle-generated message or the local oracle) is compromised or not.

In the case of `SendTerm`-queries, we will use the terminology *compromised* or *uncompromised* to denote the fact that the local terminal instance was compromised or not, as well as *oracle-generated* or *non-oracle-generated* to denote the fact the input message was generated as output by a server oracle (from a `SendServ`-query).

In the case of `SendServ`-queries, the terminology will be *remote-compromised* or *remote-uncompromised* to denote the fact that the input message was generated as output by

a terminal oracle (from a `SendTerm`-query) that is compromised or not. However, when the input message does not come as an output of a server oracle it is called *non-oracle-generated*. Therefore there are three cases : *remote-compromised*, *remote-uncompromised*, and *non-oracle-generated*.

Game $\mathbf{G}_0$: In this game, the simulator generates the public parameters for the HC function, the commitment, and the PAKE. It also knows all the long-term secret keys of the users, which allows it to simulate every oracle π_P^j, as the latter would do in the real protocol, with a random bit b:

1. $\texttt{SendServ}(k, (\ell, c))$: generate and send $x_S \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$
2. $\texttt{SendServ}(k, s)$: open the value $x_T \leftarrow \mathsf{Open}(c, s)$, set $x \leftarrow g(x_S + x_T)$, and compute $r \leftarrow F_{K_\ell}(x)$
3. `SendServ`-PAKE queries to π_S^k: run the PAKE protocol on r
4. $\texttt{SendTerm}(j, \texttt{Start}(\ell))$:
   a) generate $x_T \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$, commit $(c, s) \leftarrow \mathsf{Com}(x_T)$, and send c
   b) If *compromised*, reveal x_T together with the random coins of the commitment
5. $\texttt{SendTerm}(j, x_S)$:
   a) Set $x \leftarrow g(x_S + x_T)$, compute $r \leftarrow F_{K_\ell}(x)$, and send the opening value s
   b) If *compromised*, reveal r
6. `SendTerm`-PAKE queries to π_T^j: run the PAKE protocol on r
7. $\texttt{SendHum}(j, x)$ (from an *infected* terminal π_T^j): Compute and return $r \leftarrow F_{K_\ell}(x)$
8. $\texttt{Compromise}(j, \ell)$: Unless a `Start`-`SendTerm`-query has already been sent to π_T^j, mark the instance π_T^j as *compromised* and reveal the random tape.
9. $\texttt{Infect}(j)$: mark the session as *infected*, allowing `SendHum`
10. $\texttt{Test}(j, P)$: according to b, and whether π_P^i is fresh or not, the real key sk_P, or a random key, or $\perp$ is returned.

By definition $\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \times \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}] - 1$.

Game $\mathbf{G}_1$: We now replace the Setup algorithm of the commitment by Setup', allowing equivocal commitments and extractability, but without any additional change: $|\Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)$, where $\mathcal{D}_1$ behaves as $\mathcal{B}_0$, with either Setup (which is $\mathbf{G}_0$) or Setup' (which is $\mathbf{G}_1$).

Game $\mathbf{G}_2$: We can now enforce random challenges, by extracting the adversary commitments and generating a commitment on a value that complements appropriately.

In order to enforce random challenges:

- When processing $\texttt{SendServ}(k, (\ell, c))$, when c is *non-oracle-generated* or *remote-compromised* (the terminal is compromised or infected), we extract x_T and generate $x_S \leftarrow g^{-1}(x) + x_T$ so that $g(x_T + x_S)$ is the expected random challenge;
- When processing $\texttt{SendTerm}(j, \texttt{Start}(\ell))$ and $\texttt{SendTerm}(j, x_S)$, if the terminal is not *compromised*, it generates a fake commitment as output of the first query and will opens it to an x_T that complements correctly with x_S in the later one.

If the enforced random challenge does not get the expected value when processing $\texttt{SendServ}(k, s)$, the simulator $\mathcal{B}'_2$ outputs (c, s) that breaks the *strong binding extractability*. Moreover, the *strong simulation indistinguishability* ensures that otherwise it is hard to distinguish this game from the previous one, by defining $\mathcal{D}_2$ exactly as the simulator $\mathcal{B}_1$, using either $\mathsf{Com}/\mathsf{Open}$ (which is $\mathbf{G}_1$) or $\mathsf{SimCom}/\mathsf{OpenCom}$ (which is $\mathbf{G}_2$).

Hence, under the strong-security of the commitment scheme, the simulation remains the same to the adversary:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_2)$$

Game $\mathbf{G}_3$: We now use $\mathcal{S}_{\mathsf{pake}}$ to emulate all the messages our simulator should send for the PAKE protocol: $|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{pake}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}', \mathcal{B}_3)$.

Since we are using a UC-secure PAKE, unless that adversary has guessed the password r, it has no information about the session key. As a consequence, unless the adversary has asked a successful $\texttt{TestPwd}$ query (event $\texttt{GuessedPwd}$), he has no advantage in breaking the privacy of our HAKE: $2 \times \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - 1 \leq \Pr_{\mathbf{G}_3}[\texttt{GuessedPwd}]$.

As a consequence,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq & \Pr_{\mathbf{G}_3}[\texttt{GuessedPwd}] + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{pake}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}', \mathcal{B}_3). \\ & + 2 \times \left(\mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_2) + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)\right)\end{aligned}$$

Game $\mathbf{G}_4$: Eventually, we can make use of oracles $\texttt{GetResp}$, $\texttt{GetRandChal}$, and $\texttt{TestResp}$ to replace calls to F_{K_ℓ}: we do not know anymore the long-term keys of the users, and we focus on a user $U = U_\ell$ (all the other calls can be made as above), but we know the trapdoor for the commitment scheme.

Thanks to the extractable and equivocal commitment scheme, we can inject random challenges and use the adversary to break the HC function on one of them. The simulator will indeed be able to set x of its choice (from the HC function random selection) for either the server oracle or the terminal oracle, while the adversary has to commit its share and cannot equivocate.

Let us now give the details of the final game: the simulator uses the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the public parameters for the PAKE, generates the public parameters for the commitment with the trapdoors (for extraction and equivocality), and uses the challenge instance of the HCFF for the parameters of F. It also sets Λ to an empty set. It will be used to keep track of the known challenge responses. Then it answers the oracle queries as follows:

1. $\texttt{SendServ}(k, (\ell, c))$:
   - If *remote-uncompromised*: store $b'_x \leftarrow 1$, generate $x_S \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$ and send it
   - If *remote-compromised* or *non-oracle-generated*:
     a) Store $b'_x \leftarrow 0$
     b) Generate $x \leftarrow \texttt{GetRandChal}(0)$.

c) Using the extraction key of the commitment scheme, open c to learn x_T

d) Compute and send $x_S \leftarrow g^{-1}(x) + x_T$

2. $\texttt{SendServ}(k, s)$: Do nothing (since the committed value x_T is already known)
3. $\texttt{SendServ}$-PAKE queries to π_S^k:
   - If $\exists r : (x, r) \in \Lambda$[4]: run the PAKE protocol on r and issue a $\texttt{TestResp}(r, b'_x)$.
   - Otherwise: use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the server flows (for an accepting PAKE transcript if the flows are *oracle-generated*). In case of a $\texttt{TestPwd}$ query on a candidate $r_{\mathcal{A}}$ run $\texttt{TestResp}(r_{\mathcal{A}}, b'_x)$
4. $\texttt{SendTerm}(j, \texttt{Start}(\ell))$: Initialize $b_x \leftarrow 0$ and
   - If *uncompromised*, generate and send an equivocal commitment c
   - If *compromised*:

     a) generate $x_T \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$, commit $(c, s) \leftarrow \mathsf{Com}(x_T)$, and send c

     b) reveal x_T together with the random coins of the commitment
5. $\texttt{SendTerm}(j, x_S)$:
   - If *compromised*, set $x^* \leftarrow g(x_S + x_T)$. Compute $r \leftarrow \texttt{GetResp}(x^*)$[5]. Then, store (x^*, r) in Λ, send the opening value s and reveal r
   - If *uncompromised*:

     a) Set $x \leftarrow \texttt{GetRandChal}(1), x_T \xleftarrow{\$} x_S + g^{-1}(x)$ and $b_x \leftarrow 1$

     b) Generate and send s such that $\mathsf{Open}(c, s) = x_T$, using the equivocation key of the commitment
6. $\texttt{SendTerm}$-PAKE queries to π_T^j:
   - If *compromised*, run the PAKE protocol on the known r
   - If *uncompromised* and *oracle-generated*, use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the terminal flow (for an accepting PAKE transcript if the flows remain oracle-generated)
   - If *uncompromised* and *non-oracle-generated*, use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the terminal flow. In case of a $\texttt{TestPwd}$ query on a candidate $r_{\mathcal{A}}$ run $\texttt{TestResp}(r_{\mathcal{A}}, b_x)$
7. $\texttt{SendHum}(j, x)$ (from an *infected* terminal):
   - If $(x, r) \in \Lambda$ for some r, output r ;
   - Otherwise, compute $r \leftarrow \texttt{GetResp}(x)$, store (x, r) in Λ and output r.
8. $\texttt{Compromise}(j, \ell)$: (unchanged) Unless a $\texttt{Start}$-$\texttt{SendTerm}$-query has already been sent to π_T^j, mark the instance π_T^j as *compromised* and reveal the random tape.
9. $\texttt{Infect}(j)$: (unchanged) mark the session as *infected*, allowing $\texttt{SendHum}$

[4] In particular, this can be the case if the $\texttt{SendServ}(k, (\ell, c))$-query was *remote-compromised* or if r was queried through the $\texttt{SendHum}$ query of an *infected* terminal session.

[5] If such a query is not allowed because too many were asked since the last $\texttt{GetRandChal}$, it first issues a $\texttt{GetRandChal}(0)$ query (and discards the result).

10. $\texttt{Test}(j, P)$: (unchanged) according to b, and whether π_P^i is fresh or not, the real key sk_P, or a random key, or $\perp$ is returned.

Note that the event GuessedPwd now means that a TestResp-answer was positive, we no previous GetResp-query. This is exactly a success in the unforgeability of the HCFF, so

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \mathsf{Adv}_F^{n_{\mathsf{comp}}\text{-}\mathsf{uf}}(\mathcal{B}_4) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{pake}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}', \mathcal{B}_3). \\ + 2 \times \left(\mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}_2') + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)\right)\end{aligned}$$

We now have to count how many queries are asked by our simulator $\mathcal{B}_4$ in Game $\mathbf{G}_4$.

One can note from this simulation that any interaction between the adversary and a safe player (*uncompromised* terminal/server) results in a $x \in \Lambda_0$ (generated by a GetRandChal-query from the simulator on behalf of the safe player). Moreover,

- in each *passive session* with an *uncompromised* terminal, there is just a GetRandChal-query;
- for each server interacting with a *compromised* terminal, the simulator makes use of a GetRandChal-query, a GetResp-query and a correct TestResp, hence ctr overall stays the same;
- in each session between a *uncompromised* terminal and an adversary trying to impersonate the server (some non-oracle-generated flows), there might be a GetRandChal-query and a TestResp-query;
- in each session between a *compromised* terminal and an adversary trying to impersonate the server (some non-oracle-generated flows) with an x of its choice, there might be a GetResp-query (and a GetRandChal in some cases), but no TestResp. Hence, ctr is incremented.
- in each session with an *infected* terminal that directly queries the user on its own challenge, there is a GetResp-query. Hence ctr is incremented.
- in each session between an honest server and an adversary playing on behalf of the user/terminal (after *infecting* or not the terminal), there are a GetRandChal-query and a TestResp-query. If one of the two previous terminal session situations occurred between those, the TestResp will decrement ctr[6] if the PAKE succeeds
  Or none occurred, in which case π_S^k is *fresh*, but a success of the PAKE means winning the unforgeability game;

As a consequence,

- a GetResp-query only appears in sessions with a compromised terminal Hence, q_{r} is at most the number of *compromised* terminal sessions, that is

$$q_{\mathsf{r}} \leq n_{\mathsf{comp}};$$

[6]Thus, counting both the terminal session that increments ctr and the server sessions that decrements it, ctr is unchanged.

Human $U_\ell(K_\ell)$		Terminal T		Server $S(K_\ell)$
$\mathsf{accept} \leftarrow$ `False`				$\mathsf{accept} \leftarrow$ `False`
$\mathsf{terminate} \leftarrow$ `False`				$\mathsf{terminate} \leftarrow$ `False`
	$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$ $(c, s) \leftarrow \mathsf{Com}(x_T)$	$\xrightarrow{\ell, c}$	
	$\xleftarrow{x}$	$x \leftarrow g(x_S + x_T)$	$\xleftarrow{x_S}$	$x_S \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$
$r \leftarrow F_{K_\ell}(x)$	$\xrightarrow{r}$		$\xrightarrow{s}$	$x_T \leftarrow \mathsf{Open}(c, s)$
$\mathsf{accept} \leftarrow$ `True`				$x \leftarrow g(x_S + x_T)$
				$r \leftarrow F_{K_\ell}(x)$
			$\xleftrightarrow[\text{PAKEsid} = (\ell, c, x_S, s)]{\text{PAKE}(r)}$	
		Parses key: $(k_T \lVert \mathsf{sk}_T)$		Parses key: $(k_S \lVert \mathsf{sk}_S)$
				$\mathsf{accept} \leftarrow$ `True`
$x_U \xleftarrow{\$} \mathcal{C}$	$\xrightarrow{x_U}$	$X_U \leftarrow \mathsf{Enc}[k_T](x_U)$	$\xrightarrow{X_U}$	$x_U \leftarrow \mathsf{Dec}[k_S](X_U)$
Verifies r_U	$\xleftarrow{r_U}$	$r_U \leftarrow \mathsf{Dec}[k_T](R_U)$ Outputs sk_T	$\xleftarrow{R_U}$	$R_U \leftarrow \mathsf{Enc}[k_S](F_{K_\ell}(x_U))$ Outputs sk_S
$\mathsf{terminate} \leftarrow$ `True`	$\xrightarrow{0/1}$			$\mathsf{terminate} \leftarrow$ `True`

Figure 5.3: Confirmed HAKE Construction

- a `TestResp`-query only appears in sessions between a safe player (uncompromised terminal or honest server) and the other being impersonated by the adversary. In particular, such sessions are active, hence q_t is at most the number of active sessions (n_active).

$$q_\mathsf{t} \leq n_\mathsf{active};$$

- a `GetRandChal`-query only appears in sessions with a safe player (uncompromised terminal or honest server) or, in some cases, in sessions where a `GetResp`-query was issued. Hence, q_c is at most the sum of the number of *uncompromised* terminal sessions (n_uncomp), `GetResp`-queries and the number of server sessions with a *non-oracle-generated* flow (ℓ, c) ($n_{\mathsf{nog_c}}$),

$$q_\mathsf{c} \leq n_\mathsf{uncomp} + q_\mathsf{r} + n_{\mathsf{nog_c}} \leq n_\mathsf{uncomp} + n_\mathsf{comp} + n_\mathsf{serv}.$$

- ctr is only incremented by *compromised* terminal session in which a `GetResp`-query was asked, while no concurrent server session successfully terminated. Unfortunately, such a situation is not reliably detectable by the human, though it will play an important role for the Confirmed HAKE.

$$\mathsf{ctr} \leq n_\mathsf{comp}$$

□

5.5.2 The Confirmed HAKE

We now enhance the Basic HAKE by adding two confirmations flows (see Figure 5.3) that allow the user to detect a bad behavior of the adversary, who compromised the device, and

take appropriate measures. This can happen in two different scenarios: the adversary has compromised a terminal and additionally plays on behalf of the server, which allows it to ask any query to the user through the terminal, or the adversary has infected a terminal that allows it to directly ask any query to the user.

As said above, such dangerous cases lead to no connection with the expected server. The user will thus check whether he built a secure session with the expected server, who should be able to answer a fresh random challenge. This is performed under the fresh key, established with the PAKE, using a secure authenticated encryption. As shown below, the two additional flows will not only provide *explicit* authentication, but also allow the user to detect such bad events and take measures. For this, it is important that the user does not start multiple sessions concurrently, which is anyway not realistic for a human (as already noticed above).

Description. The protocol is similar to Basic HAKE, but it uses an additional building block, an authenticated encryption scheme $\mathcal{ES} = (\mathsf{Enc}, \mathsf{Dec})$, that is used in the new last stage of the protocol. The complete description is given in Figure 5.3.

Since we now consider the authentication of the players, we additionally include accept and terminate flags in the protocol: The user U_ℓ accepts after sending the first response while the server S accepts after the PAKE. Then both terminate when they have the confirmation of the other partner. More precisely, the user terminates after sending the last bit (1 for acceptance and 0 for rejection) to the terminal (thus having verified the server's response in the last stage), and the server terminates after sending the encrypted response (thus having checked the terminal can generate a valid cipher text).

Note that if the protocol terminates, sk_T and sk_S must be equal, since our additional flows act as confirmation flows for the PAKE.

Security analysis. We now present Theorem 5.5.2 regarding the security of our Confirmed HAKE in the HAKE privacy and authenticity experiment.

While it relies on the same security properties of PAKE, authenticated encryption, commitment scheme and HCFF, a critical parameter is added, the number of human sessions that *reject* in the end.

Indeed, the explicit authentication property we achieve means that any attempt at issuing an adaptive query unrelated to the challenge will likely lead to a failure of the PAKE protocol, which can in turn be detected by the human, as he does not get the answer to x_U he looked for. This allows to use a much more restricted η in the HCFF unforgeability game (up to $\eta = 1$ for a very strict human user), which is a much more reasonable goal for an only-human HCFF such as the one derived from [BBDV17], that we will present in Section 5.2.2

Theorem 5.5.2. *Consider the Confirmed HAKE protocol defined in Figure 5.3. Let $\mathcal{A}, \mathcal{A}'$ be adversaries against the privacy and authenticity security game of HAKE within a time bound t and using less than n_{comp}* compromised *terminal sessions, n_{uncomp}* uncompromised *terminal sessions, n_{serv} server sessions, $n_{\mathsf{active}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ active sessions and n_{hr} human session that* reject *in the end. Then there exist two adversaries $\mathcal{B}_1, \mathcal{B}'_1$ attacking the 2-party $(n_{\mathsf{hr}} + 1)$-unforgeability of HCFF with $q_{\mathsf{r}}, q_{\mathsf{c}}, q_{\mathsf{t}}$ queries of the corresponding type, two adversaries $\mathcal{B}_2, \mathcal{B}'_2$ and two distinguishers $\mathcal{B}_3, \mathcal{B}'_3$ attacking UC-security of the PAKE with the simulator $\mathcal{S}_{\mathsf{pake}}$, two adversaries $\mathcal{B}_4, \mathcal{B}'_4$ against the authenticated encryption, as well as six adversaries $\mathcal{A}_1, \mathcal{A}'_1$, $\mathcal{A}_2, \mathcal{A}'_2$, and $\mathcal{A}_3, \mathcal{A}'_3$ against the commitment scheme properties, all*

running in time t, such that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathit{HAKE}}(\mathcal{A}) &\leq \mathsf{Adv}^{(n_{\mathsf{hr}}+1)-\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \times \mathsf{Adv}^{\mathit{pake}}_{\mathit{pake}}(\mathcal{S}_{\mathit{pake}}, \mathcal{B}_2, \mathcal{B}_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}_4) \\ &\quad + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}_3)\right), \\ \mathsf{Adv}^{\mathsf{auth}}_{\mathit{HAKE}}(\mathcal{A}') &\leq \mathsf{Adv}^{(n_{\mathsf{hr}}+1)-\mathsf{uf}}_{F}(\mathcal{B}'_1) + 2 \times \mathsf{Adv}^{\mathit{pake}}_{\mathit{pake}}(\mathcal{S}_{\mathit{pake}}, \mathcal{B}'_2, \mathcal{B}'_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}'_4) \\ &\quad + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}'_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}'_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}'_3)\right),\end{aligned}$$

where $q_{\mathsf{r}} \leq 2n_{\mathsf{comp}}$, $q_{\mathsf{t}} \leq n_{\mathsf{active}}$, $q_{\mathsf{c}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$.

Remark 5.5.3. *We also note that given the confirmation phase, and assuming the strong policy of resetting all credentials if the confirmation phase fails, the coin-flipping part is no longer necessary for the security proof: we could let the server choose the challenge during the first phase and the human in the second one (to avoid one player being to make replay attacks). We chose to keep it as part of the protocol because, first, this would not reduce the number of flows since the terminal always initiates such a connection, and second, without coin-flipping a network attacker could test adaptive challenges. The confirmation phase would fail, but there is no real need for the user to take severe measures and change the long-term secret in such a weak attack. Hence we prevent adaptive tests (from network attacks) with coin-flipping, which may be useful if a policy a little weaker is in use, such as resetting only if there is a suspicion of terminal infection.*

Privacy proof. The proof is very similar to the previous one, so we just introduce the new queries to add to the security games defined in Section 5.5.2, but only after the execution of the PAKE, and so the simulator (in the real game when it knows all the long-term keys) knows the output $(k_P \| \mathsf{sk}_P)$:

Game $\mathbf{G}_0$: In this game, we simply simulate every oracle according to the protocol, knowing the output of the PAKE, and all the honest players accept at the end of the PAKE.

1. Last `SendTerm`-PAKE-query (that led to the session key (k_T, sk_T)):
   a) generate $x_U \xleftarrow{\$} \mathcal{C}$ (on behalf of the user, and so the internal random coins stay secret even if π_T^j is *compromised*)
   b) send $X_U \leftarrow \mathsf{Enc}[k_T](X_U)$
2. `SendHum`(j, x) (from an *infected* terminal): Not only output r, as before, but additionally generate and send $x_U \xleftarrow{\$} \mathcal{C}$.
3. `SendServ`(k, X_U): Compute and send $R_U \leftarrow \mathsf{Enc}[k_S](F_{K_\ell}(\mathsf{Dec}[k_S](X_U)))$ (unless the decryption fails, $\mathsf{terminate}$ is set to `True`)
4. `SendTerm`(j, R_U): Compute $r'_U \leftarrow \mathsf{Dec}[k_T](R_U)$ and check whether $r'_U = F_{K_\ell}(x_U)$ or not (on behalf of the user). Unless the decryption fails, if the equality holds, the user accepts and the terminal sets $\mathsf{terminate}$ to `True`
5. `SendHum`(j, r_U) (from an *infected* terminal): Accept if $r_U = F_{K_\ell}(x_U)$.

Game $\mathbf{G}_1$: In this game, we once again make use of the underlying HC security game oracles. But now, the simulator does not know anymore the long-term keys, and did not learn the output of the PAKE when the simulator $\mathcal{S}_{\mathsf{pake}}$ was involved.

1. Last `SendTerm`-PAKE-query:
   - If *compromised* (the simulator honestly ran the PAKE on behalf of the terminal, and so the simulator knows (k_T, sk_T), but the adversary too)
     a) generate $x_U \leftarrow \texttt{GetRandChal}(1)$
     b) send $X_U \leftarrow \mathsf{Enc}[k_T](x_U)$
   - If *uncompromised*, generate a random ciphertext X_U
2. $\texttt{SendHum}(j, x)$ (from an *infected* terminal), as before, and additionally generate and send $x_U \leftarrow \texttt{GetRandChal}(1)$
3. $\texttt{SendServ}(k, X_U)$:
   - If *remote-compromised*, the simulator knows (k_S, sk_S) and x_U; it then computes $r_U \leftarrow \texttt{GetResp}(x_U)$, stores (x_U, r_U) in Λ and sends $R_U \leftarrow \mathsf{Enc}[k_0](r_U)$
   - If *remote-uncompromised*, generate and send a random ciphertext R_U
   - If *non-oracle-generated*,
     – either the simulator knows k_S because of an honest execution of the PAKE with r. Then, it can get $x'_U \leftarrow \mathsf{Dec}[k_S](X_U)$. If this decrypts correctly, it checks if (x'_U, r'_U) is in Λ for some r'_U. If it is not, get $r'_U \leftarrow \texttt{GetResp}(x'_U)$. It then sends $\mathsf{Enc}[k_S](r'U)$, and set `terminate` to `True`
     – or the simulator does not know k_S because it invoked the simulator $\mathcal{S}_{\mathsf{pake}}$, which means that the adversary should not know the session key either, and so is unable to generate a valid cipher text: the simulator makes the server abort.
4. $\texttt{SendTerm}(j, R_U)$:
   - If *compromised* and *oracle-generated*, r_U is known and must be correct so the simulator simply runs $\texttt{TestResp}(r_U, 1)$ and terminates
   - If *compromised* and *non-oracle-generated*, get $r'_U \leftarrow \mathsf{Dec}[k_T](R_U)$, and run $\texttt{TestResp}(r'_U, 1)$, to either terminate or reject (a decryption failure leads to a reject)
   - If *uncompromised* and *oracle-generated*, terminate
   - If *uncompromised* and *non-oracle-generated*, reject
5. $\texttt{SendHum}(j, r_U)$ (from an *infected* terminal): Accept if $\texttt{TestResp}(r_U, 1)$.

Note that in the added flows, all `GetRandChal` and `TestResp` occur with bit 1, which was only used previously if the terminal oracle was *uncompromised*, in which case either the adversary issued a correct `TestPwd`, and the simulator already won the HC unforgeability game, or (k_P, sk_P) is unknown and the authenticated encryption hides all flows.

From the previous simulation, in case of passive sessions with a compromised terminal, in the first part of the simulation, the simulator honestly ran the PAKE, completing it with (k_T, sk_T). It can thus continue honestly if the adversary continues to forward oracle generated flows. It then learns a new challenge-response pair. If the adversary starts to play on behalf of the server, it has to generate a forgery for the HC function (checked by the `TestResp`-query).

We will define an adversary $\mathcal{B}'_1$ against the combined security notion for authenticated encryption.

In case of passive sessions with an uncompromised terminal, k_T is unknown to the adversary, then random ciphertexts can be sent, they are indistinguishable from real ciphertexts (from the semantic security of our secure authenticated encryption scheme, in which case $\mathcal{B}_1'$ is used as a distinguisher between the encryption of the real plaintext —as in Game $\mathbf{G}_0$— or of a random plaintext —as in Game $\mathbf{G}_1$—). Eventually, if the adversary tries to impersonate the server before the PAKE, and send non-oracle-generated messages to the terminal, they should not be encrypted under the correct key k_T, unless the adversary has broken either the integrity of the authenticated encryption (in which case $\mathcal{B}_1'$ outputs the fake ciphertext message) or the PAKE (correct guess of r), which would have led to a positive answer to a `TestResp` during the simulation before: a forgery against the HC function.

As a consequence, unless the adversary has helped the simulator to break the unforgeability of the HCFF, this game $\mathbf{G}_1$ is indistinguishable from the previous one, and leads to accepted sessions by the human user for passive sessions only (with compromised terminals or not) with the expected server, which was our target.

We have already shown that the server could only agree on a key (k_P, sk_P) with a terminal linked to the expected human user, hence the global security of our protocol: privacy and authentication. In addition, the two additional flows allow the human user to detect whether the key sk_T is shared with the expected server, just leaking one more random challenge-response pair in case of compromised terminal.

With the same reasoning as in the previous proof, we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq\ & \mathsf{Adv}^{\eta-\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{pake}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}_2, \mathcal{B}_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}_4) \\ & + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}_3)\right).\end{aligned}$$

However, there are a few changes regarding the number of queries involved in the simulation $\mathcal{B}_5$, but only on *compromised* sessions, as otherwise the simulator would not have played the PAKE honestly and therefore only random-looking messages are generated:

- There may be an additional `GetRandChal` in some *compromised* sessions.
- There may be an additional `TestResp`-query in some *active* sessions, but only if the PAKE was passive, without a previous `TestResp`-query. Otherwise k_T is random, and the decryption fails.
- There may be an additional `GetResp`-query in some server session, if the PAKE succeeded, which means a *compromised* terminal session must have been used or a win was already reached.
- ctr may incremented, but only if the PAKE succeeded (meaning it was not durably increased by the previous flows). Moreover, if the human accepts, it is decreased (and overall stays the same).

Overall, it is notable that ctr is permanently increased only if the human rejects in the end or the HC unforgeability game is won. Hence:

$$\mathsf{ctr} \leq n_{\mathsf{hr}}$$

Moreover, ctr can never reach a value higher than its final value plus one. This means that $\eta = n_{\mathsf{hr}} + 1$ is sufficient for the HC unforgeability game. □

Authenticity proof. The simulated game $\mathbf{G}_1$ is indistinguishable from the real one (either the privacy security game or the authentication security game). To break the server authentication, the adversary should be able to compute the answer to a random challenge x_U with no `GetResp` allowed. Breaking the user-authentication means that the adversary succeeded in the PAKE, in order to know k_S, and to send a valid cipher text X_U and hence must also guess the answer r to a random x challenge. Since we exclude trivial attacks from the authentication security game, any winning strategy requires a successful `TestResp`-query. Hence, the advantage $\mathsf{Adv}^{\mathsf{auth}}(\mathcal{A})$ of the adversary in the authentication security game is upper-bounded the same way as $\mathsf{Adv}^{\mathsf{priv}}(\mathcal{A})$. □

5.6 Token-based HAKE constructions

In this section, we take a step back from only-human HCFF to allow the use of an additional device that will perform the computations in place of the human. In this setting, the HCFF can be quite powerful and thus resist to many adaptive queries. In our HCFF security notions, this means that it satisfies our strongest notion, *indistinguishability*.

We consider it in two scenarios: first in a similar context as the generic HAKE constructions, where one can enter a challenge onto the device to get the response; and second, a time-based token, that outputs the response every timeframe, with the time as the challenge (without having the user to enter it).

5.6.1 Simplified Basic HAKE

In the security proof of the Basic HAKE, the PAKE has to be instantiated with a UC-Secure protocol, which turns out to be quite costly. Indeed, the only really efficient scheme that achieves this security level is the encrypted key exchange protocol (EKE) [BM92]. However, the proof holds in the IC model, for a symmetric block cipher that should only output elements in the Diffie-Hellman group. In practice, the best way to do it is to iterate a large block cipher until one falls in the group. First, a large block cipher from a hash function (modeled as a random oracle) has fueled a whole line of works [CPS08; HKT11; DS16], and is nevertheless already quite costly: at the time of writing, at least 8-round Feistel network is required [DS16], with an impossibility result below 6 [CPS08]. Thereafter, additional iterations are required to build a permutation onto the group. This eventually corresponds to dozens of hash function evaluations.

Looking back at the construction, using a full PAKE seems a bit overkill in this setting, since the ephemeral secrets are only used once, and need not to be kept secret afterwards if the HCFF is strong enough.

Taking advantage of this, we present, on Figure 5.4, the Simplified Basic HAKE, that is very similar to the Basic HAKE (Figure 5.2). As said above, it does not use a full PAKE, but just commitments to mutually check the knowledge of the ephemeral secret before it is revealed which is enough in this setting.

We do not provide a proof, but it is easy to see that the hiding/binding properties of the commitment scheme and the *decisional Diffie-Hellman* (DDH) hardness replace the security properties of the PAKE in the proof of the Basic HAKE from Section 5.5.1. The only change from Theorem 5.5.1 would be that q_{r} is now only less or equal to $n_{\mathsf{total}} = n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ (the total number of sessions), which should not be an issue for a device-assisted HCFF.

Human $U_\ell(K_\ell)$		Terminal T		Server $S(K_\ell)$
	$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$		
		$(c, s) \leftarrow \mathsf{Com}(x_T)$	$\xrightarrow{\ell, c}$	
	$\xleftarrow{x}$	$x \leftarrow g(x_S \oplus x_T)$	$\xleftarrow{x_S}$	$x_S \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$
$r \leftarrow F_{K_\ell}(x)$	$\xrightarrow{r}$	$y_T \xleftarrow{\$} \mathbb{Z}_p$		
		$Y_T \leftarrow g^{y_T}$		$y_S \xleftarrow{\$} \mathbb{Z}_p, Y_S \leftarrow g^{y_S}$
		$(c_T, s_T) \leftarrow \mathsf{Com}_T(Y_T, r)$	$\xrightarrow{s, Y_T, c_T}$	$x_T \leftarrow \mathsf{Open}(c, s)$
				$x \leftarrow g(x_S \oplus x_T)$
				$r \leftarrow F_{K_\ell}(x)$
			$\xleftarrow{Y_S, c_S}$	$(c_S, s_S) \leftarrow \mathsf{Com}_S(Y_S, r)$
			$\xrightarrow{s_T}$	If $\mathsf{Open}_T(c_T, s_T) \neq (Y_T, r)$ then reject
		If $\mathsf{Open}_S(c_S, s_S) \neq (Y_S, r)$ then reject	$\xleftarrow{s_S}$	
		Outputs $\mathsf{sk}_T \leftarrow (Y_S)^{y_T}$		Outputs $\mathsf{sk}_S \leftarrow (Y_T)^{y_S}$

Figure 5.4: Simplified Basic HAKE Construction

5.6.2 Time-Based HAKE

Scenario. In this section, we focus on the particular (but quite usual) case where the physical device does not have a dedicated input but uses time instead to compute its output. More precisely, our protocol considers a device, such as the RSA-SecurId [RSA] token, that, based on an internal seed (the long-term key K_ℓ), generates a one-time password (the value $F_{K_\ell}(t)$, based on the time period t), and displays it on an LCD screen. The password is tied to an internal clock, and changes every τ (e.g. 30s). Note that such a password is already *human readable* and *human writable*, hence it satisfies our human-compatible communications.

Building on the security model presented in Section 5.4, we now consider time as a variable, that is to be segmented into timeframes (each spanning τ second). We then number those timeframes and associate to each message sent between T and S this number, representing the fact that each party can measure time and identify the timeframe in which the message was sent.

Since the one-time passwords are generated by a secure device implementing F_{K_ℓ}, we can make the assumption that, for each timeframe, the output is indistinguishable from an element sampled from the distribution $\mathcal{D}$ with entropy greater than D (which increases the advantage of an adversary $\mathcal{A}$ by at most $\mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{A})$ after T timeframes).

We rely on the requirement that any user U_ℓ can only make use of one terminal during a timeframe. That is, he may not attempt to authenticate using more than one terminal in a single time period.

Protocol. We now propose a protocol for called Time-Based HAKE that is suitable for this setting. It is presented on Figure 5.5. As in the previous Simplified Basic HAKE, it makes use

Time	Human U_ℓ		Terminal T		Server S
$\leq t$	$\mathsf{accept} \leftarrow$ `False`				$\mathsf{accept} \leftarrow$ `False`
					$\mathsf{terminate} \leftarrow$ `False`
t		$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_p, X_T \leftarrow g^{x_T}$		$x_S \xleftarrow{\$} \mathbb{Z}_p, X_S \leftarrow g^{x_S}$
t		$\xrightarrow{\mathsf{pw}_t}$	$(c_T, s_T) \leftarrow \mathsf{Com}_T(X_T, \mathsf{pw}_t)$	$\xrightarrow{\ell, X_T, c_T}$	
t	$\mathsf{accept} \leftarrow$ `True`			$\xleftarrow{X_S, c_S}$	$(c_S, s_S) \leftarrow \mathsf{Com}_S(X_S, \mathsf{pw}_t)$
$\vdots$			Wait for timeframe $> t$		Wait for timeframe $> t$
$> t$				$\xrightarrow{s_T}$	If $\mathsf{Open}_T(c_T, s_T) = (X_T, \mathsf{pw}_t)$ and $(\ell, t) \notin \Lambda$, store (ℓ, t) in Λ
$> t$					Otherwise reject
$> t$					$\mathsf{accept} \leftarrow$ `True`
$> t$			Reject if $\mathsf{Open}_S(c_S, s_S) \neq (X_S, \mathsf{pw}_t)$	$\xleftarrow{s_S}$	Outputs $(X_T)^{x_S}$
$> t$			Outputs $(X_S)^{x_T}$		$\mathsf{terminate} \leftarrow$ `True`

Figure 5.5: Time-Based HAKE Construction

of a commitment scheme on top of the unauthenticated Diffie-Hellman scheme to perform authentication.

The commitment scheme $\mathcal{CS}$ is initialized twice, with two independent setups, leading to $\mathsf{Com}_T/\mathsf{Open}_T$ and $\mathsf{Com}_S/\mathsf{Open}_S$, each of them being used for the commitments generated by the terminal and the server, respectively. We also set up a group $\mathbb{G}$ of prime order p in which the discrete logarithm problem is believed to be hard. Let g be a generator of $\mathbb{G}$.

The protocol itself is split into two parts: the *commitment* phase which must happen during a timeframe t (that we will call the *session timeframe*) and the *verification* phase, that must happen later than the session timeframe.

This delay is a clear limitation on the total speed of the protocol, which on average will take $\tau/2$. It will, however, prove necessary, as it allows $F_{K_\ell}(t)$ to be revealed without compromising the security of the scheme, therefore building on the one-time specificity of the password. To enforce a unique session in a timeframe, the server will not accept to run several sessions within the same timeframe, with the same user, as the latter should not do it anyway (see above). This would thus come from an adversary, and then allowing multiple sessions in a timeframe t can compromise other sessions in the same timeframe when $\mathbb{F}_{K_\ell}(t)$ is revealed.

It is interesting to note that this protocol uses the time period t as the HAKE challenge (the challenge is a counter) and the one-time password ($F_{K_\ell}(t)$) read from the device as the human's response. Therefore, partnering between U and S is entirely determined at the end of the session timeframe t.

Security analysis. In the security analysis, as in the previous analyses, we only consider static *compromises*. Hence `Compromise`(j) can only be the first oracle query of a session, and `Infect`(j) can only affect *compromised* sessions. Since *compromises* are known before the first flow and partnering between Human and Server is determined at the end of timeframe t, this means that *freshness* itself can be perfectly ascertained in any timeframe $> t$.

The security of our protocol heavily relies on the strong-security of the commitment scheme (see Section 2.3.1), as well as the indistinguishability of the password sequence distribution (see Section 5.1.2).

Theorem 5.6.1. *Consider the Time-Based HAKE protocol defined in Figure 5.5. Let $\mathcal{A}, \mathcal{A}'$ be adversaries against the privacy and user authentication security games with static compromises, running within time $t_{\mathcal{A}}$ and using less than n_{serv} non-passive sessions against the server oracle, n_{term} non-passive sessions against the terminal oracle, $n_{\mathsf{total}} > n_{\mathsf{term}} + n_{\mathsf{serv}}$ total sessions and $T < n_{\mathsf{total}}$ unique timeframes. Then there exist an adversary $\mathcal{D}_1$ against the indistinguishability of the password distribution $\mathcal{D}$ running in time t, an adversary $\mathcal{D}_5$ against the DDH experiment running in time $t + 8n_{\mathsf{total}}\tau_{exp}$, four adversaries $\mathcal{B}'_3$, $\mathcal{B}'_5$, $\mathcal{D}_2$, and $\mathcal{D}_3$ against the commitment scheme properties running in time t:*

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{priv}}_{HAKE}(\mathcal{A}) &\leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{ind}}_{ddh}(\mathcal{D}_5) + \mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_3) \\
&\quad + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_3) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_5) \\
\mathsf{Adv}^{\mathsf{u\text{-}auth}}_{HAKE}(\mathcal{A}') &\leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_2) \\
&\quad + n_{\mathsf{total}} \times \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_3) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_5),
\end{aligned}$$

with τ_{exp} the time necessary to exponentiate one group element, and n_{total} the global number of sessions.

Remark 5.6.2. *Note that the Time-Based HAKE only achieves user authentication in our setting, since* server authentication *requires the server identity to be approved by the human in our setting (the terminal could be* infected *so it cannot be relied on). A similar approach to the one of the Confirmed HAKE could be used to achieve a full mutual authentication.*

Proof. Once again, we will denote $\mathcal{B}_i$ the simulator for Game $\mathbf{G}_i$ that outputs 1 if $b \leftarrow \mathcal{A}$ and 0 otherwise and define $\mathcal{D}_i$ the distinguisher between games $\mathbf{G}_i$ and $\mathbf{G}_{i-1}$, for $i > 1$.

Game $\mathbf{G}_0$: This is the real game, where the adversary outputs its guess on the bit b:

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \times \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}] - 1.$$

Game $\mathbf{G}_1$: In this game the simulator will execute the real protocol, generating a password $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$ at the beginning of each timeframe t and subsequently using it whenever necessary. We will also consider a flag, `NOG-Com-OK`, which can be raised during the execution of the simulation. More precisely, in $\mathbf{G}_1$, the simulator answers each request as follows:

1. $\mathtt{SendServ}(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π^k_S's session timeframe. Generate $x_S \xleftarrow{\$} \mathbb{Z}_p$ and $X_S \leftarrow g^{x_S}$. Then set $(c_S, s_S) \leftarrow \mathsf{Com}(X_S, \mathsf{pw}_t)$ and send (X_S, c_S).
2. $\mathtt{SendServ}(k, s_T)$: Check whether $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$.
   - If so, send s_S, accept and set $\mathsf{sk}_S = (X_T)^{x_S}$. Additionally, if π^k_S is *fresh* and c_T was not *oracle-generated*, raise flag `NOG-Com-OK`.
   - If the equality is not verified reject.
3. $\mathtt{SendTerm}(j, \mathtt{Start})$: Set the current timeframe t to be π^j_T's session timeframe. Generate $x_T \xleftarrow{\$} \mathbb{Z}_p$ and $X_T \leftarrow g^{x_T}$. accept on behalf of the human and use the current password pw_t to set $(c_T, s_T) \leftarrow \mathsf{Com}(X_T, \mathsf{pw}_t)$ and send (ℓ, X_T, c_T). Additionally, if π^j_T is *compromised*, reveal the password pw_t.

4. $\texttt{SendTerm}(j, (X_S, c_S))$: Wait until the timeframe is $> t$, then send s_T.
5. $\texttt{SendTerm}(j, s_S)$: Check whether $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$.
   - If so, set $\mathsf{sk}_T = (X_S)^{x_T}$. Additionally, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag `NOG-Com-OK`.
   - If the equality is not verified, reject.
6. $\texttt{SendHum}()$: If π_T^j was marked as *infected*, reveal pw_t, where t is the *current* timeframe and accept.
7. $\texttt{Compromise}(j)$: Mark π_T^j as *compromised* and reveal the random tape.
8. $\texttt{Infect}(j)$: If π_T^j is *compromised* through a previous $\texttt{Compromise}(j)$ query, mark it as *infected*, allowing `SendHum` queries.
9. $\texttt{Test}(j, P)$: according to b and whether π_P^j is fresh or not, the real key sk_P, a random key or $\perp$ is returned.

It should be clear that this simulation performs exactly as the real game should, except for using the global distribution $\mathcal{D}$ to generate the passwords, since the additional flags are purement formal but do not affect the simulation. Hence:

$$|\Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{D}_1),$$

where $\mathcal{D}_1$ behaves as $\mathcal{B}_0$ but using either distribution $\mathcal{D}$ (which is then $\mathbf{G}_1$) or the real distribution (as in $\mathbf{G}_0$) to generate the passwords pw_t.

Game $\mathbf{G}_2$: In this game, unless the oracle is *compromised*, we will reject all openings for *non-oracle-generated* commitments.

More precisely, we change the query answers as follows:

2. $\texttt{SendServ}(k, s_T)$: If c_T was *oracle-generated* and $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$, send s_S, accept and set $\mathsf{sk}_S(X_S)^{x_T}$. Otherwise reject.
5. $\texttt{SendTerm}(j, s_S)$:
   - If *compromised*, act exactly as in $\mathbf{G}_1$
   - Otherwise: If c_S was *oracle-generated* and $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$, accept and set $\mathsf{sk}_T = (X_S)^{x_T}$. Otherwise reject.

Obviously, games $\mathbf{G}_2$ and $\mathbf{G}_1$ are not indistinguishable. However, this can only make a difference when `NOG-Com-OK` was raised. Hence:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}]$$

Game $\mathbf{G}_3$: In this game, we further straighten our requirements for openings in *non-compromised* terminals. Indeed, we will also reject an opening if it opens to a different X_S/X_T than the one it was initially generated for. More precisely:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Act as in $\mathbf{G}_1$. Then store $(c_S, X_S, s_S) \in \Upsilon_S$.
2. $\texttt{SendServ}(k, s_T)$: If $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$ and $(c_T, X_T, \cdot) \in \Upsilon_T$, send s_S, accept and set $\mathsf{sk}_S = (X_S)^{x_T}$. Otherwise reject.
3. $\texttt{SendTerm}(j, \texttt{Start})$: Act as in $\mathbf{G}_1$. Then store $(c_T, X_T, s_T) \in \Upsilon_T$.

5. $\texttt{SendTerm}(j, s_S)$:
   - If *compromised*, act exactly as in $\mathbf{G}_2$
   - Otherwise: If $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$ and $(c_S, X_S, \cdot) \in \Upsilon_S$, set $\mathsf{sk}_T = (X_S)^{x_T}$. Otherwise reject.

Game $\mathbf{G}_3$ and $\mathbf{G}_2$ are almost the same. The only difference would be if $\mathcal{A}$ somehow reuses an *oracle-generated* commitment c_P but opens it to a different key X_P^* (and the same, valid, password pw_t) than the *oracle-generated* one X_P. This would, however, break the *binding* property of $\mathcal{CS}$, as we now know two opening values (s_P, s_P^*) for two messages $((X_P, \mathsf{pw}_t), (X_P^*, \mathsf{pw}_t))$ with the same commitment c_P; we define a simulator $\mathcal{B}_3'$ that behaves as $\mathcal{B}_3$ but outputs such a tuple.

As it happens only on *non-passive* sessions, of which there is at most $n_{\mathsf{term}} + n_{\mathsf{serv}}$, we have:

$$|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{binding}}(\mathcal{B}_3').$$

Game $\mathbf{G}_4$: We now consider a CDH tuple $(A, B, C) = (g^a, g^b, g^{ab})$ and embed it into the simulation. Once again, simulations still behave as in the original Game $\mathbf{G}_1$ for *compromised* sessions. More precisely, we change the query answers from Game $\mathbf{G}_3$ as follows:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π_S^k's session timeframe. Generate $(\beta, \delta) \xleftarrow{\$} \mathbb{Z}_p^2$ and set $X_S \leftarrow B^\delta \cdot g^\beta$. Then set $(c_S, s_S) \leftarrow \mathsf{Com}(X_S, \mathsf{pw}_t)$ and send (X_S, c_S). Lastly, store $(c_S, X_S, s_S) \in \Upsilon_S$.

3. $\texttt{SendTerm}(j, \texttt{Start})$:
   - If *compromised*, act exactly as in $\mathbf{G}_3$.
   - Otherwise: Set the current timeframe t to be π_T^j's session timeframe. Generate $(\alpha, \gamma) \xleftarrow{\$} \mathbb{Z}_p^2$ and set $X_T \leftarrow A^\gamma \cdot g^\alpha$. Use the current password pw_t to set $(c_T, s_T) \leftarrow \mathsf{Com}(X_T, \mathsf{pw}_t)$ and send (ℓ, X_T, c_T). Then store $(c_T, X_T, s_T) \in \Upsilon_T$.

9. $\texttt{Test}(j, P)$: If sk_P has been generated and π_P^j is *fresh*, we know $(\alpha, \beta, \gamma, \delta)$ that were used to construct X_T and X_S (we only compute sk_P if both are *oracle-generated*). Then, the simulator uses $\mathsf{sk}_P = C^{\gamma\delta} \cdot B^{\alpha\delta} \cdot A^{\beta\gamma} \cdot g^{\alpha\beta}$ for the real key. Otherwise, it returns $\perp$.

Regarding *compromised* sessions, either the adversary learned the password pw (from $\texttt{SendHum}$-query or by letting the *compromised* terminal ask the user) and no $\texttt{Test}$-query can be asked as neither π_T^j nor π_S^k is fresh; or the adversary did not learn the password, in which case $\texttt{Test}$-queries can still be asked to π_S^k, but the compromise did not reveal any secret.

Since we have $C = g^{ab}$, then $\mathsf{sk}_P = g^{(a\gamma+\alpha)\cdot(b\delta+\beta)}$ is exactly as the real key should be. This game is perfectly indistinguishable from the previous one:

$$\Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}] = \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}].$$

Game $\mathbf{G}_5$: We are now given a random C, independent of A and B:

Hence, using a distinguisher $\mathcal{D}_5$, that behaves as $\mathcal{B}_5$ with C either real (as in $\mathbf{G}_4$) or random (which is $\mathbf{G}_5$):

$$|\Pr_{\mathbf{G}_5}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{ddh}}(\mathcal{D}_5),$$

Note that our simulator makes 8 exponentiations to simulate any session in addition to the running time of $\mathcal{A}$, so $\mathcal{D}_5$ runs in time $t_{\mathcal{A}} + 8n_{\mathsf{total}}\tau_{\mathrm{exp}}$ with $t_{\mathcal{A}}$ the running time of $\mathcal{A}$ and τ_{exp} the time necessary to exponentiate one group element.

When C is random, then every sk_P is random and independent of all others. Hence one cannot distinguish the real key from a random key, since they are both random: $\Pr_{\mathbf{G}_5}[b \leftarrow \mathcal{A}] = 1/2$.

As a consequence,

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{ddh}}(\mathcal{D}_5) + \mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_3).$$

We now need to bound the probabilities of events `NOG-Com-OK` in Game $\mathbf{G}_1$. For this, we construct another series of games from Game $\mathbf{G}_1$, in which we will consider the probability of raising a flag instead of the regular advantage over the `Test`-queries. For the sake of simplicity, we will denote the simulators in this branch $\mathcal{C}_i$ for Game $\mathbf{G}_i$.

Game $\mathbf{G}_2$: In this game, we go back to Game $\mathbf{G}_1$ and change the simulator's behavior to make use of the equivocality property of our commitment scheme. But before any equivocation, we just change the setup procedure of the commitment:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_2),$$

where $\mathcal{D}_2$ is a distinguisher that behaves as $\mathcal{B}_1$, with either Setup (which is $\mathbf{G}_1$) or Setup' (which is $\mathbf{G}_2$).

Game $\mathbf{G}_3$: The simulator can now use its oracle access to equivocal commitments whenever it has to use commitments in order to delay the actual committed value. However, if the terminal is *compromised*, we will still use regular commitment algorithms. This is due to the fact that the adversary knows the random tape and can therefore deterministically reproduce the output of Com. The simulator runs as follows:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π^k_S's session timeframe. Generate $x_S \xleftarrow{\$} \mathbb{Z}_p$ and $X_S \leftarrow g^{x_S}$. Get $c_S \leftarrow \mathsf{GenEquivCommit}$ and send (X_S, c_S).
2. $\texttt{SendServ}(k, s_T)$: Check whether $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$.
   - If so, send $s_S \leftarrow \mathsf{OpenEquivCommit}(c_S, (X_S, \mathsf{pw}_t))$, accept and output $\mathsf{sk}_S = (X_T)^{x_S}$. Additionally, if π^k_S is *fresh* and c_T was not *oracle-generated*, raise flag `NOG-Com-OK`.
   - Otherwise output $\perp$.
3. $\texttt{SendTerm}(j, \texttt{Start})$: Set the current timeframe t to be π^j_T's session timeframe and generate $x_T \xleftarrow{\$} \mathbb{Z}_p$ and $X_T \leftarrow g^{x_T}$. Then:
   - If *compromised*: Act as in $\mathbf{G}_1$.

- If *uncompromised*: Get $c_T \leftarrow \mathsf{GenEquivCommit}$ and send (ℓ, X_T, c_T).

4. $\mathtt{SendTerm}(j, (X_S, c_S))$: Wait until the timeframe is $> t$. Then:
   - If *compromised*: Act as in $\mathbf{G}_1$.
   - If *uncompromised*: Send $s_T \leftarrow \mathsf{OpenEquivCommit}(c_T, (X_T, \mathsf{pw}_t))$.
5. $\mathtt{SendTerm}(j, s_S)$: Check whether $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$.
   - If so, generate and output $\mathsf{sk}_T = (X_S)^{x_T}$ Additionally, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag `NOG-Com-OK`.
   - Otherwise output $\perp$.

The difference is just in the use of equivocal commitments instead of real ones:

$$|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq n_{\mathsf{total}} \times \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_3).$$

where $\mathcal{D}_3$ is a distinguisher that behaves as $\mathcal{C}_2$ but uses either real (like in $\mathbf{G}_2$) or equivocal (as in $\mathbf{G}_3$) commitments.

Game $\mathbf{G}_4$: One could remark that in the previous game, unless the session is *compromised*, the simulator doesn't use pw_t in any timeframe $\leq t$. Therefore, in this game, the simulator will delay producing pw_t until the timeframe t has ended unless `Compromise` is called. This can be done using the same oracle handlers as in $\mathbf{G}_3$, simply adding the password definition:

2. $\mathtt{SendServ}(k, s_T)$: If not yet generated, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.
4. $\mathtt{SendTerm}(j, (X_S, c_S))$: After waiting for the current timeframe to be $> t$, if not yet generated, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.
7. $\mathtt{Compromise}(j)$: Generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.

This game is perfectly indistinguishable from the previous one:

$$\Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}] = \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}].$$

Game $\mathbf{G}_5$: We will now use the extractability property of our commitment scheme to check the adversary's commitment, by modifying the simulator as follows:

1. $\mathtt{SendServ}(k, (\ell, X_T, c_T))$: If c_T was not *oracle-generated*, extract $(X_T^*, \mathsf{pw}_t^*) \leftarrow \mathsf{ExtractCommit}(c_T)$. Then act as in $\mathbf{G}_4$.
2. $\mathtt{SendServ}(k, s_T)$: If c_T was *oracle-generated*, act as in $\mathbf{G}_4$. Otherwise, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}_t$. Then check whether $(X_T^*, \mathsf{pw}_t^*) = (X_T, \mathsf{pw}_t)$.
   - If so, send $s_S \leftarrow \mathsf{OpenEquivCommit}(c_S, (X_S, \mathsf{pw}_t))$, accept and output $\mathsf{sk}_S = (X_T)^{x_S}$. Additionally, if π_S^k is *fresh*, raise flag `NOG-Com-OK`.
   - Otherwise output $\perp$.
4. $\mathtt{SendTerm}(j, (X_S, c_S))$: Extract $(X_S^*, \mathsf{pw}_t^*) \leftarrow \mathsf{ExtractCommit}(c_S)$. Then act as in $\mathbf{G}_4$.
5. $\mathtt{SendTerm}(j, s_S)$: If c_S was *oracle-generated*, act as in $\mathbf{G}_4$. Otherwise, check whether $(X_S^*, \mathsf{pw}_t^*) = (X_S, \mathsf{pw}_t)$.

- If so, generate and output $\mathsf{sk}_T = (X_S)^{x_T}$ Additionally, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag `NOG-Com-OK`.
- Otherwise output $\perp$.

Note that in this game, except for compromised sessions, pw_t^* is obtained before pw_t is generated. Hence $\Pr[\mathsf{pw}_t = \mathsf{pw}_t^*] = 2^{-D}$.

The only way to distinguish Game $\mathbf{G}_5$ from Game $\mathbf{G}_4$ would be if $(\mathsf{Open}(c_P, s_P) \neq \mathsf{ExtractCommit}(c_P)$. But if so, then (c_P, s_P) breaks the *strong binding extractability*, which can be output by a simulator $\mathcal{B}_5'$ similar to $\mathcal{C}_5$. Hence:

$$|\Pr_{\mathbf{G}_5}[\texttt{NOG-Com-OK}] - \Pr_{\mathbf{G}_4}[\texttt{NOG-Com-OK}]| \leq \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}binding}}(\mathcal{B}_5').$$

Moreover, we can now bound the probability that those flags are raised in Game $\mathbf{G}_5$. Indeed, for it to happen, there must exist an oracle π_P^j such that $(X_P^*, \mathsf{pw}_t^*) = (X_P, \mathsf{pw}_t)$. In particular, we must have $\mathsf{pw}_t^* = \mathsf{pw}_t$. Moreover, this cannot happen if the session was *passive*. Hence,

$$\Pr_{\mathbf{G}_5}[\texttt{NOG-Com-OK}] \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D}.$$

And so:

$$\begin{aligned}\Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{setup\text{-}ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}eq}}(\mathcal{D}_3) \\ + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}binding}}(\mathcal{B}_5').\end{aligned}$$

In conclusion, we have various adversaries such that:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{HAKE}}^{\mathsf{priv}}(\mathcal{A}) \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{D}_1) + \mathsf{Adv}_{\mathsf{ddh}}^{\mathsf{ind}}(\mathcal{D}_5) + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{binding}}(\mathcal{B}_3') \\ + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{setup\text{-}ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}eq}}(\mathcal{D}_3) + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}binding}}(\mathcal{B}_5').\end{aligned}$$

□

Authenticity proof. The simulated Game $\mathbf{G}_1$ is indistinguishable from the real world (be it in the privacy security game or the authentication security game).
To break user authentication means that the adversary successfully opened a commitment c_T to X_T, pw_t without interacting with U_ℓ. This is exactly the situation in which flag `NOG-Com-OK` is raised. Hence:

$$\mathsf{Adv}_{\mathsf{HAKE}}^{\mathsf{u\text{-}auth}}(\mathcal{A}') \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] + \mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{D}_1)$$

Using the same results about $\Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}]$, this gives:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{HAKE}}^{\mathsf{u\text{-}auth}}(\mathcal{A}') \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{setup\text{-}ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}eq}}(\mathcal{D}_3) \\ + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}binding}}(\mathcal{B}_5') + \mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{D}_1)\end{aligned}$$

□

Scheme	Flows	Terminal		Server		Communication
		expon.	$\mathcal{H}$ eval.	expon.	$\mathcal{H}$ eval.	complexity
`1(SPAKE1)`[PS10; AP05]	4	3	1	3	1	4λ
`Time-Based HAKE`	4	2	2	2	2	10λ

Table 5.1: Performance of the Time-Based HAKE

Performances. We offer in Table 5.1 a comparison (in terms of numbers of flows, exponentiations, $\mathcal{H}$ evaluations and overall communication complexity) of the performances of our Time-Based HAKE protocol with the one-time PAKE `1(P)` construction of [PS10], instantiated with `SPAKE1` from [AP05] as a reference.

Since `SPAKE1` is also proven in the random oracle model, it is fair to use the efficient commitment scheme described in Section 2.3.1. We do not include the redundant X_P in s_P (it is transmitted at the commitment stage) for the communication complexity, and for a security parameter λ, we assume the group elements to be encoded into 2λ-long bit strings.

While our communication complexity is higher, the computational load is reduced by 30% from [PS10] with the most efficient PAKE. Relaxing the PAKE security properties allows a significant gain from the complexity point of view.

Chapter 6

Conclusion and open questions

6.1 Conclusion

In this thesis, we considered several variants of PAKE with an eye toward more usable and more realistic settings.

AKE is the basis for some of the most actually used cryptographic systems, and notably TLS. This is mostly for efficiency reasons as it allows to limit the costly asymmetric cryptographic operations to the bare minimum necessary to establish a shared key, and then rely on faster symmetric cryptographic primitives.

However, this is usually done using a PKI, with signed asymmetric keys as the basis of authentication. While this might be suitable for machine-to-machine interactions, it is not well-suited to human authentication. Indeed, actual human authentication is often performed as an afterthought, with human secrets transmitted over the symmetric channel, after the initial authentication with the PKI. Hence the whole security relies on this PKI-based authentication, which is far from flawless from a practical point of view. For instance, in TLS, it has been shown that in practice certification authorities can be corrupted. Barring that, since what is certified is seldom well understood, it is often enough to use a certificate that looks — to the actual human performing a casual validation — almost the same as the original one (e.g. with a typo in the domain's name).

PAKE is an attempt at using a symmetric authentication, where the secret known by the human is directly used in the authentication, therefore tying directly the human being into the authentication scheme. While there are conceptual difficulties with using a password as authentication — most notably the fact that it has to be provided to all parties in the first place, and their inherent small chance of being guessed — it is also easy to see that it offers less of a gap between the human's expectations and the way the protocol realizes it.

In Chapter 3, we showed how to restrict the regular PAKE UC functionality to really model for only an implicit authentication, instead of the usual explicit leakage toward the adversary. Moreover, we show that a usual PAKE construction — EKE2 — can provide it at no cost. This new primitive allows for new applications of PAKE, in protocols where the result of the PAKE itself needs to be hidden. We actually construct such a protocol in Chapter 4.

In Chapter 4, we presented a variant of PAKE, called fPAKE, that is set to tolerate some slight modifications of the password. This has multiple applications: in the usual "short sequence of alphanumerical symbols" typed by the human sense, it can account for typos and thus encourage to use longer passwords, increasing entropy overall. It can also be very

useful for biometric authentication, where the authentication means' capture is intrinsically fuzzy. We believe this extends the domain of applicability of PAKE, and may lead to more actual applications.

Finally, in Chapter 5, we extended PAKE to show how to account for misbehavior on the part of the computer one is using. More precisely, we tried to tackle the issue that a human authenticating means is likely to be used even in some insecure settings, such as using an unknown computer "just for once", which usually implies all security is forever lost shall this computer be corrupted. Though the proposed solution is far from really usable without an external device, it shows ways in which we could protect sessions even if strong corruptions are considered. It also shows how a tighter interfacing with the human can lead to better security that could ever be provided with a PKI-based AKE, as misbehavior on the server's part is handled directly by the authenticating cryptographic protocol. With this method, even a malicious server cannot learn a password he does not know already, even through the human's mistake.

6.2 Open questions

In this thesis, we approached several interesting research questions that are still open to this date.

Question 6.1. *Can fPAKE be achieved without leaking anything on a successful authentication?*

In Chapter 4, we presented several functionalities that model an ideal fPAKE, with an increasing leakage. We showed how to achieve $\mathcal{F}^{M}_{\mathsf{fpake}}$ — which leaks only correct symbols — with a security bound γ twice as large as the functionality bound δ. In [DHP+18], we also achieve $\mathcal{F}^{D}_{\mathsf{fpake}}$, this time with no security gap ($\gamma = \delta$), but this leaks the actual original password on a successful authentication.

Ideally, we would instead like to achieve $\mathcal{F}_{\mathsf{fpake}}$, which can be seen as an extension of $\mathcal{F}_{\mathsf{iPAKE}}$ and does not leak anything. i.e., even on a successful guess of the password, the adversary would not be provided feedback to learn his attempt was successful. Of course, externally, this does not mean he cannot learn it, but that is only because of a leakage of the general system, and clever usages of fPAKE could make use of this added capability.

Even a simple leakage of whether authentication succeeded, as is done in the $\mathcal{F}_{\mathsf{pake}}$ functionality, would be better than leaking the password itself, given that passwords tend to be reused by human beings, potentially in systems with different security parameters. This is especially true as part of this functionality use case is biometrics, which cannot be changed at all. Hence guaranteeing the secrecy of the authentication means can be seen as of even more import than authentication itself, since it can be reused in systems with different security.

Question 6.2. *Do practical only-human HCFF exist?*

In Section 5.2, we presented two instantiations of HCFF, that may be used in our HAKE. The simplest one, if practical, does rely on a token to perform actual computation of the response. The second one, based on the work of [BBDV17], is only-human, meaning that it really represents operations that could be conducted out of one's head. While there are some hardness results that speak in its favor, resilience to adaptive leakage of challenge-response pairs is hard to achieve and hence has a high cost for the human (in terms of the number of

image mappings to memorize and simple operations to perform) which severely impacts the practical applicability of our construction.

Hence we would like to see other, more practical, only-human HCFF functions.

Question 6.3. *Can HAKE be achieved without requiring the HCFF function to be resilient to adaptive queries.*

A related question is relative to reducing the requirements on the HCFF function. Indeed, with our latter Confirmed HAKE protocol, we are able to detect adaptive queries which can reduce this requirement to actually resisting only one adaptive query on the HCFF function, assuming a very restrictive security policy (i.e., if a potential adaptive query is detected, invalidate the password). While this is doable, it also means that some false positives, such as network timeouts in sensitive parts of the protocol, have to be treated as dangerous attacks, and a new password has to be generated and learned by the human.

Even with the Confirmed HAKE protocol, we still require the HCFF to resist one adaptive query, since without so, the adversary can spoof the detection. This already has a big impact on the HCFF parameters, and adds many constraints on the choice of this function.

Hence producing a HAKE with less stringent requirements on the HCFF would be a very valuable contribution.

Abbreviations

AE authenticated encryption. 70

AKE authenticated key exchange. 2, 8, 58, 85, 86

CDH computationnal Diffie-Hellman. 10, 24, 28–30, 32, 79

CRS common reference string. 18, 24, 26, 48

CS commitment scheme. 12, 14, 62, 63, 70, 77, 83

DDH decisional Diffie-Hellman. 10, 74, 77

ECC error-correcting code. 3

EUF-CMA existencially unforgeable under adaptive chosen message attack. 40

fPAKE fuzzy password authenticated key exchange. 3–5, 35–38, 48, 85, 86, 91

HAKE human authenticated key exchange. x, 4, 11, 49, 50, 52, 53, 55–59, 61–63, 65, 67, 69–71, 73–77, 79, 81, 83, 86, 87, 91

HCFF human-compatible function family. x, 50–57, 61–64, 66, 68, 70, 73, 74, 86, 87

IC ideal cipher. 10, 18, 24, 26, 48, 74

IND-CPA indistinguishability under chosen plaintext attacks. 15

iPAKE implicit-only password authenticated key exchange. 3, 5, 8, 21, 22, 36, 41, 47

liPAKE labeled implicit-only password authenticated key exchange. 5, 22, 23, 39, 42, 48

MDS maximum distance separable. 15–17, 40

PAKE password authenticated key exchange. ix, 2–5, 8–12, 21–24, 26, 28, 30, 32, 34, 35, 37, 38, 40–42, 48, 49, 57, 61–68, 70–74, 83, 85, 86, 91

PKI public-key infrastructure. 2, 85, 86

RO random oracle. 18, 24, 26, 48

RP-CSP random planted constraint satisfiability problem. 54, 56

RSS robust secret sharing. 3–5, 15–17, 38–41, 46

UC universal composability. ix, 3, 4, 7–12, 14, 18, 21, 23–25, 34, 35, 38, 41–43, 47, 63, 64, 66, 70, 74, 85, 91

List of Illustrations

Figures

2.1 Ideal Functionality $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$ for PAKE (recalled from [CHK+05]) 9
2.2 Ideal Functionality $\mathcal{F}_{\mathsf{pake}}$ for PAKE (simplified from $\mathcal{F}^{\mathtt{H}}_{\mathsf{pake}}$) 10
2.3 Functionality $\mathcal{F}_{\mathsf{CRS}}$ 18
2.4 Functionality $\mathcal{F}_{\mathsf{RO}}$ 19
2.5 Functionality $\mathcal{F}_{\mathsf{IC}}$ 19

3.1 Functionality $\mathcal{F}_{\mathsf{iPAKE}}$ 22
3.2 Functionality $\mathcal{F}_{\mathsf{liPAKE}}$ 23
3.3 Protocol EKE2, in a group $\mathbb{G} = \langle g \rangle$ of prime order q, with a hash function $H : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \mathbb{G} \to \{0,1\}^\lambda$ and a symmetric cipher $\mathcal{E} : \mathbb{G} \to \{0,1\}^\kappa, \mathcal{D} : \{0,1\}^* \to \mathbb{G}$ for keys in $\mathbb{P} \times \mathcal{L}$. 24
3.4 An UC Execution of EKE2 25
3.5 Transition from Game $\mathbf{G}_0$ (left) to Game $\mathbf{G}_1$ (right), showing a setting where $\mathcal{P}_{1-i}$ is corrupted. 26
3.6 The Simulator $\mathcal{S}$ for the EKE2 Protocol indistinguishability from $\mathcal{F}_{\mathsf{liPAKE}}$ . . 34

4.1 Ideal Functionality $\mathcal{F}_{\mathsf{fpake}}$ for fPAKE 36
4.2 A Modified `TestPwd` Interface to Allow for Different Leakage 37
4.3 A First Natural Construction (with code-offset fuzzy sketch and PAKE) . . . 38
4.4 $\mathsf{FPAKE}^{\mathsf{rss}}$ Protocol 39
4.5 A UC Execution of $\mathsf{FPAKE}^{\mathsf{rss}}$ 43
4.6 Transition from Game $\mathbf{G}_0$ (left) to Game $\mathbf{G}_1$ (right), showing a setting where both parties are honest. 44
4.7 The Simulator $\mathcal{S}$ for $\mathsf{FPAKE}^{\mathsf{rss}}$ 47

5.1 Graph of the sequential oracle calls in the η-unforgeability experiment 51
5.2 Basic HAKE Construction 62
5.3 Confirmed HAKE Construction 69
5.4 Simplified Basic HAKE Construction 75
5.5 Time-Based HAKE Construction 76

Tables

5.1 Performance of the Time-Based HAKE 83

Bibliography

[ABB+13] Michel Abdalla, Fabrice Benhamouda, Olivier Blazy, Céline Chevalier, and David Pointcheval. "SPHF-Friendly Non-interactive Commitments". In: *ASIACRYPT 2013, Part I*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8269. LNCS. Springer, Heidelberg, Dec. 2013, pp. 214–234. DOI: 10.1007/978-3-642-42033-7_12 (cit. on pp. 13, 14).

[ABDP15] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. "Simple Functional Encryption Schemes for Inner Products". In: *PKC 2015*. Ed. by Jonathan Katz. Vol. 9020. LNCS. Springer, Heidelberg, Mar. 2015, pp. 733–751. DOI: 10.1007/978-3-662-46447-2_33 (cit. on p. 4).

[ACCP08] Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. "Efficient Two-Party Password-Based Key Exchange Protocols in the UC Framework". In: *CT-RSA 2008*. Ed. by Tal Malkin. Vol. 4964. LNCS. Springer, Heidelberg, Apr. 2008, pp. 335–351 (cit. on pp. 10, 24).

[ACP09] Michel Abdalla, Céline Chevalier, and David Pointcheval. "Smooth Projective Hashing for Conditionally Extractable Commitments". In: *CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. LNCS. Springer, Heidelberg, Aug. 2009, pp. 671–689 (cit. on p. 10).

[AP05] Michel Abdalla and David Pointcheval. "Simple Password-Based Encrypted Key Exchange Protocols". In: *CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. LNCS. Springer, Heidelberg, Feb. 2005, pp. 191–208 (cit. on p. 83).

[BBDV17] Jeremiah Blocki, Manuel Blum, Anupam Datta, and Santosh Vempala. "Towards Human Computable Passwords". In: *ITCS 2017*. Ed. by Christos H. Papadimitriou. Vol. 4266. 67: LIPIcs, Jan. 2017, 10:1–10:47 (cit. on pp. 4, 53–57, 70, 86).

[BCDP17] Alexandra Boldyreva, Shan Chen, Pierre-Alain Dupont, and David Pointcheval. "Human Computing for Handling Strong Corruptions in Authenticated Key Exchange". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Aug. 2017, pp. 159–175. DOI: 10.1109/CSF.2017.31 (cit. on p. 4).

[BCL+05] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. "Secure Computation Without Authentication". In: *CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 361–377 (cit. on p. 41).

[BDK+05] Xavier Boyen, Yevgeniy Dodis, Jonathan Katz, Rafail Ostrovsky, and Adam Smith. "Secure Remote Authentication Using Biometric Data". In: *EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 147–163 (cit. on p. 38).

[BM92] Steven M. Bellovin and Michael Merritt. "Encrypted Key Exchange: Password-Based Protocols Secure against Dictionary Attacks". In: *1992 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 1992, pp. 72–84. DOI: 10.1109/RISP.1992.213269 (cit. on pp. 2, 3, 10, 23, 74).

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. "Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 156–171 (cit. on p. 2).

[BN00] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm". In: *ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. LNCS. Springer, Heidelberg, Dec. 2000, pp. 531–545 (cit. on pp. 14, 15).

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. "The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme". In: *Journal of Cryptology* 16.3 (June 2003), pp. 185–215 (cit. on pp. 50, 52).

[Boy04] Xavier Boyen. "Reusable Cryptographic Fuzzy Extractors". In: *ACM CCS 04*. Ed. by Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel. ACM Press, Oct. 2004, pp. 82–91 (cit. on pp. 3, 4, 38).

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. "Authenticated Key Exchange Secure against Dictionary Attacks". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 139–155 (cit. on pp. 2, 3, 8, 11, 18, 21, 23, 58–60).

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93*. Ed. by V. Ashby. ACM Press, Nov. 1993, pp. 62–73 (cit. on p. 14).

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145 (cit. on pp. 3, 14, 24, 35, 42).

[Can07] Ran Canetti. "Obtaining Universally Compoable Security: Towards the Bare Bones of Trust (Invited Talk)". In: *ASIACRYPT 2007*. Ed. by Kaoru Kurosawa. Vol. 4833. LNCS. Springer, Heidelberg, Dec. 2007, pp. 88–112 (cit. on p. 18).

[CDD+15] Ronald Cramer, Ivan Bjerre Damgård, Nico Döttling, Serge Fehr, and Gabriele Spini. "Linear Secret Sharing Schemes from Error Correcting Codes and Universal Hash Functions". In: *EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. LNCS. Springer, Heidelberg, Apr. 2015, pp. 313–336. DOI: 10.1007/978-3-662-46803-6_11 (cit. on p. 16).

[CF01] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. LNCS. Springer, Heidelberg, Aug. 2001, pp. 19–40 (cit. on p. 14).

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. "Universally Composable Password-Based Key Exchange". In: *EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 404–421 (cit. on pp. 3, 8–11, 21, 35, 41, 48).

[CPS08] Jean-Sébastien Coron, Jacques Patarin, and Yannick Seurin. "The Random Oracle Model and the Ideal Cipher Model Are Equivalent". In: *CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 1–20 (cit. on p. 74).

[Cra05] Ronald Cramer, ed. *EUROCRYPT 2005*. Vol. 3494. LNCS. Springer, Heidelberg, May 2005.

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 23).

[DHP+18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. "Fuzzy Password-Authenticated Key Exchange". In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Heidelberg, Apr. 2018, pp. 393–424. DOI: 10.1007/978-3-319-78372-7_13 (cit. on pp. 4, 38, 86).

[DP17] Pierre-Alain Dupont and David Pointcheval. "Functional Encryption with Oblivious Helper". In: *ASIACCS 17*. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. ACM Press, Apr. 2017, pp. 205–214 (cit. on p. 4).

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, May 2004, pp. 523–540 (cit. on pp. 3, 4, 38, 40).

[DS16] Yuanxi Dai and John P. Steinberger. "Indifferentiability of 8-Round Feistel Networks". In: *CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. LNCS. Springer, Heidelberg, Aug. 2016, pp. 95–120. DOI: 10.1007/978-3-662-53018-4_4 (cit. on p. 74).

[FHH14] Eduarda S. V. Freire, Julia Hesse, and Dennis Hofheinz. "Universally Composable Non-Interactive Key Exchange". In: *SCN 14*. Ed. by Michel Abdalla and Roberto De Prisco. Vol. 8642. LNCS. Springer, Heidelberg, Sept. 2014, pp. 1–20. DOI: 10.1007/978-3-319-10879-7_1 (cit. on p. 42).

[FLM11] Marc Fischlin, Benoît Libert, and Mark Manulis. "Non-interactive and Reusable Universally Composable String Commitments with Adaptive Security". In: *ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. LNCS. Springer, Heidelberg, Dec. 2011, pp. 468–485 (cit. on p. 14).

[FPV15] Vitaly Feldman, Will Perkins, and Santosh Vempala. "On the Complexity of Random Satisfiability Problems with Planted Solutions". In: *47th ACM STOC*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. ACM Press, June 2015, pp. 77–86 (cit. on pp. 4, 53, 54).

[HKT11] Thomas Holenstein, Robin Künzler, and Stefano Tessaro. "The equivalence of the random oracle model and the ideal cipher model, revisited". In: *43rd ACM STOC*. Ed. by Lance Fortnow and Salil P. Vadhan. ACM Press, June 2011, pp. 89–98 (cit. on p. 74).

[HM04] Dennis Hofheinz and Jörn Müller-Quade. "Universally Composable Commitments Using Random Oracles". In: *TCC 2004*. Ed. by Moni Naor. Vol. 2951. LNCS. Springer, Heidelberg, Feb. 2004, pp. 58–76 (cit. on p. 18).

[JW99] Ari Juels and Martin Wattenberg. "A Fuzzy Commitment Scheme". In: *ACM CCS 99*. ACM Press, Nov. 1999, pp. 28–36 (cit. on pp. 38, 40).

[KV11] Jonathan Katz and Vinod Vaikuntanathan. "Round-Optimal Password-Based Authenticated Key Exchange". In: *TCC 2011*. Ed. by Yuval Ishai. Vol. 6597. LNCS. Springer, Heidelberg, Mar. 2011, pp. 293–310 (cit. on p. 48).

[MS81] Robert J. McEliece and Dilip V. Sarwate. "On Sharing Secrets and Reed-Solomon Codes". In: *Communications of the ACM* 24.9 (1981), pp. 583–584. DOI: 10.1145/358746.358762. URL: http://doi.acm.org/10.1145/358746.358762 (cit. on p. 18).

[Pre00] Bart Preneel, ed. *EUROCRYPT 2000*. Vol. 1807. LNCS. Springer, Heidelberg, May 2000.

[PS10] Kenneth G. Paterson and Douglas Stebila. "One-Time-Password-Authenticated Key Exchange". In: *ACISP 10*. Ed. by Ron Steinfeld and Philip Hawkes. Vol. 6168. LNCS. Springer, Heidelberg, July 2010, pp. 264–281 (cit. on pp. 62, 83).

[RBBK01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption". In: *ACM CCS 01*. ACM Press, Nov. 2001, pp. 196–205 (cit. on p. 15).

[Rot06] Ron M. Roth. *Introduction to coding theory*. Cambridge University Press, 2006. ISBN: 978-0-521-84504-5 (cit. on p. 15).

[RSA] *RSA SecurId Hardware Tokens*. RSA Security. https://www.rsa.com/en-us/products-services/identity-access-management/securid/hardware-tokens (cit. on pp. 53, 75).

[Sho01] Victor Shoup. *A Proposal for an ISO Standard for Public Key Encryption*. Cryptology ePrint Archive, Report 2001/112. http://eprint.iacr.org/2001/112. 2001 (cit. on p. 22).

Résumé

L'échange de clef authentifié est probablement la primitive asymétrique la plus utilisée, notamment du fait de son inclusion dans le protocole TLS. Pour autant, son cousin, l'échange de clef authentifié par mot de passe, où l'authentification s'effectue par comparaison de mot de passe, l'est bien moins, bien qu'ayant déjà fait l'objet d'études considérables. C'est pourtant une primitive finalement bien plus proche d'une authentification réelle, dès lors qu'une des parties est humaine.

Dans cette thèse, nous considérons des primitives avancées fondées sur l'échange de clef authentifié par mot de passe, en gardant à l'œil ses applications pratiques. Spécifiquement, nous introduisons une nouvelle primitive, l'échange de clef authentifié par mot de passe approximatif, où la condition de succès de l'authentification est désormais d'avoir une distance suffisamment faible entre les deux mots de passe, et plus nécessairement l'égalité parfaite. Nous fournissons un modèle de sécurité dans le cadre du modèle de composabilité universelle (UC) ainsi qu'une construction reposant sur un partage de secret robuste et des échanges de clefs authentifiés par mot de passe exact.

Dans une seconde partie, nous considérons le problème pratique de la perte du mot de passe dès lors qu'une session est conduite sur un terminal compromis. Étant donné qu'il s'agit d'un problème intrinsèque à l'authentification par mot de passe, nous étendons le modèle BPR habituel pour prendre en compte, en lieu et place du mot de passe, des questions-réponses, toujours de faible entropie. Nous fournissons plusieurs protocoles dans ce modèle, dont certains reposent sur des familles de fonctions compatibles avec les humains, dans lesquelles les opérations requises pour dériver la réponse depuis la question sont suffisamment simples pour être faites de tête, permettant donc à l'humain de s'identifier directement.

Mots Clés

authentification, clef-publique, cryptographie, échange de clef, mot de passe, mot de passe approximatif

Abstract

Authenticated key exchange is probably the most widely deployed asymmetric cryptographic primitive, notably because of its inclusion in the TLS protocol. Its cousin, password-authenticated key exchange — where the authentication is done using a low-entropy password — while having been studied extensively as well has been much less used in practice. It is, however, a primitive much closer to actual authentication when at least one party is human.

In this thesis, we consider advanced primitives based on password-authenticated key exchange, with an eye toward practical applications. Specifically, we introduce fuzzy password-authenticated key exchange, where the authentication succeeds as long as the two passwords are close enough, and not necessarily equal. We provide a security model in the UC framework, as well as a construction based on regular password-authenticated key exchanges and robust secret-sharing schemes.

Secondly, we consider the practical problem of password leakage when taking into account sessions conducted on a corrupted device. As there is intrinsically no hope with regular password authentication, we extend the BPR security model to consider low-entropy challenge responses instead. We then provide several instantiations, some based on human-compatible function families, where the operation required to answer the challenge are simple enough to be conducted in one's head, allowing the actual authentication to be directly performed by the human being.

Keywords

authentication, cryptography, fuzzy password, key exchange, password, public-key