UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2022-133
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 17/11/2022 in Esch-sur-Alzette
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Aditya DAMODARAN

Born on 7 March 1992 in Kannur, Kerala (India)

PROTOCOLS FOR STATEFUL ZERO-KNOWLEDGE

Dissertation defence committee

Dr Peter Y. A. RYAN, Dissertation Supervisor
Professor, Université du Luxembourg

Dr Alfredo Rial
Cryptographer, Nym Technologies SA

Dr Jean-Sébastien Coron, Chairman
Professor, Université du Luxembourg

Dr Peter B. Rønne
Researcher, Université de Lorraine, LORIA

Dr Jan Camenisch, Vice Chairman
CTO, Dfinity Foundation, Zurich

Protocols for Stateful Zero-Knowledge

Aditya Damodaran

Privacy Preserving Protocols, Zero Knowledge Proofs,
and Vector Commitments

Supervisors: Peter Y.A. Ryan and Alfredo Rial

SnT
University of Luxembourg

November 2022

Supported by the Luxembourg
National Research Fund (FNR11650748)

ABSTRACT

Privacy preserving protocols typically involve the use of Zero Knowledge (ZK) proofs, which allow a prover to prove that a certain statement holds true, to a verifier, without revealing the witness (secret information that allows one to verify whether said statement holds true) to the verifier. This mechanism allows for the participation of users in such protocols whilst preserving the privacy of sensitive personal information. In some protocols, the need arises for the reuse of the information (or witnesses) used in a proof. In other words, the witnesses used in a proof must be related to those used in previous proofs. We propose Stateful Zero Knowledge (SZK) data structures, which are primitives that allow a user to store state information related to witnesses used in proofs, and then prove subsequent facts about this information. Our primitives also decouple state information from the proofs themselves, allowing for modular protocol design. We provide formal definitions for these primitives using a composable security framework, and go on to describe constructions that securely realize these definitions.

These primitives can be used as modular building blocks to attenuate the security guarantees of existing protocols in literature, to construct privacy preserving protocols that allow for the collection of statistics about secret information, and to build protocols for other schemes that may benefit from this technique, such as those that involve access control and oblivious transfer. We describe several such protocols in this thesis. We also provide computational cost measurements for our primitives and protocols by way of implementations, in order to show that they are practical for large data structure sizes. We finally provide a notation and a compiler that takes as input a ZK proof represented by said notation and outputs a secure SZK protocol, allowing for a layer of abstraction so that practitioners may specify the security properties and the data structures they wish to use, and be presented with a ready to use implementation without needing to deal with the theoretical aspects of these primitives, essentially bridging the gap between theoretical cryptographic constructions and their implementation.

This thesis conveys the results of FNR CORE Junior project, Stateful Zero Knowledge.

RÉSUMÉ

Les protocoles de préservation de la vie privée impliquent généralement l'utilisation de preuves à divulgation nulle de connaissance, qui permettent à un prouveur de prouver à un vérificateur qu'une certaine déclaration est vraie, sans révéler le witness (informations secrètes nécessaires pour vérifier si la déclaration est vraie) au vérificateur. Ce mécanisme permet aux utilisateurs de participer à ces protocoles tout en préservant la confidentialité des informations personnelles sensibles. Dans certains protocoles, le besoin se fait sentir de la réutilisation d'informations (ou témoins) utilisées dans une preuve. En d'autres termes, les témoins utilisés dans une preuve doivent être liés à ceux utilisés dans les preuves précédentes.

Nous proposons des structures de données *Zero Knowledge*, qui sont des primitives qui permettent à un utilisateur de stocker des informations d'état, puis de prouver des faits sur ces informations. Nos primitives découplent également les informations d'état des preuves elles-mêmes, permettant une conception de protocole modulaire. Nous fournissons des définitions formelles pour ceux-ci à l'aide d'un cadre de sécurité composable, et poursuivons en décrivant des constructions qui réalisent ces définitions en toute sécurité. Ces primitives peuvent être utilisées comme blocs de construction modulaires pour atténuer les garanties de sécurité des protocoles existants dans la littérature et pour construire des protocoles de préservation de la vie privée qui permettent la collecte de statistiques sur les données et les protocoles pour d'autres schémas, tels que le contrôle d'accès. Nous fournissons des mesures de coût de calcul pour nos primitives et protocoles en les implémentant, et enfin fournissons une notation et un compilateur qui prend en entrée une preuve ZK représentée par la notation et génère un protocole SZK sécurisé pour celle-ci, permettant une couche d'abstraction telle que les développeurs peuvent spécifier les propriétés de sécurité et les structures de données qu'ils souhaitent utiliser, et se voir présenter une implémentation prête à l'emploi sans avoir à traiter des aspects théoriques de ces primitives, comblant essentiellement le fossé entre les constructions cryptographiques théoriques et leurs implémentations.

Cette thèse transmet les résultats du projet FNR CORE Junior, Stateful Zero Knowledge.

ZUSAMMENFASSUNG

Datenschutzprotokolle beinhalten typischerweise die Verwendung von Zero Knowledge Proofs, die es einem Beweiser ermöglichen, einem Prüfer zu beweisen, dass eine bestimmte Aussage wahr ist, ohne dem Prüfer die Aussage selbst zu offenbaren. Dieser Mechanismus ermöglicht es Benutzern, an solchen Protokollen teilzunehmen und gleichzeitig die Vertraulichkeit sensibler persönlicher Informationen zu wahren. In einigen Protokollen entsteht die Notwendigkeit für die Wiederverwendung von Informationen (oder Zeugen), die in einem Beweis verwendet werden. Mit anderen Worten, die in einem Beweis verwendeten Zeugen müssen mit denen verwandt sein, die in früheren Beweisen verwendet wurden.

Wir schlagen Zero-Knowledge-Datenstrukturen vor, die es einem Benutzer ermöglichen, Zustandsinformationen in Bezug auf Zeugen zu speichern und dann Fakten über diese Informationen zu beweisen. Unsere Grundelemente entkoppeln auch Zustandsinformationen von den Beweisen selbst, was ein modulares Protokolldesign ermöglicht. Wir stellen formale Definitionen dafür bereit, indem wir ein zusammensetzbares Sicherheitsframework verwenden, und somit Konstruktionen zu beschreiben, die diese Definitionen sicher realisieren. Diese Grundelemente können als modulare Bausteine verwendet werden, um die Sicherheitsgarantien bestehender Protokolle in der Literatur abzuschwächen und um die Privatsphäre wahrende Protokolle zu konstruieren, die das Sammeln von Statistiken über Daten und Protokolle für andere Schemata, wie z. B. Zugangskontrolle, ermöglichen. Wir stellen Rechenkostenmessungen für unsere Primitive und Protokolle bereit, indem wir sie implementieren. Wir stellen schließlich eine Notation und einen Compiler bereit, der einen durch die Notation repräsentierten ZK-Beweis als Eingabe nimmt und ein sicheres SZK-Protokoll dafür ausgibt, was eine solche Abstraktionsebene ermöglicht. Entwickler können die Sicherheitseigenschaften und Datenstrukturen angeben, die sie verwenden möchten, und erhalten eine gebrauchsfertige Implementierung.

Diese Diplomarbeit vermittelt die Ergebnisse des FNR CORE Junior Projekts, Stateful Zero Knowledge.

PUBLICATIONS

[1] Aditya Damodaran, Maria Dubovitskaya, and Alfredo Rial. "UC Priced Oblivious Transfer with Purchase Statistics and Dynamic Pricing." In: *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*. Vol. 11898. Lecture Notes in Computer Science. Springer, 2019, pp. 273–296. URL: http://hdl.handle.net/10993/39424.

[2] Aditya Damodaran and Alfredo Rial. "UC Updatable Databases and Applications." In: *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*. Vol. 12174. Lecture Notes in Computer Science. Springer, 2020, pp. 66–87. URL: http://hdl.handle.net/10993/42984.

[3] Aditya Damodaran and Alfredo Rial. "Unlinkable Updatable Databases and Oblivious Transfer with Access Control." In: *Information Security and Privacy - 25th Australasian Conference, ACISP 2020, Perth, WA, Australia, November 30 - December 2, 2020, Proceedings*. Vol. 12248. Lecture Notes in Computer Science. Springer, 2020, pp. 584–604. URL: http://hdl.handle.net/10993/43250.

[4] Aditya Damodaran and Alfredo Rial. "Unlinkable Updatable Hiding Databases and Privacy-Preserving Loyalty Programs." In: *Proc. Priv. Enhancing Technol.* 2021.3 (2021), pp. 95–121. URL: http://hdl.handle.net/10993/49090.

[5] Aditya Damodaran and Alfredo Rial. "A Compiler for Stateful Zero Knowledge Data Structures." To be submitted. 2022.

[6] Aditya Damodaran and Alfredo Rial. "Notations for Stateful Zero Knowledge Data Structures." To be submitted. 2022.

[7] Aditya Damodaran and Alfredo Rial. "Privacy Preserving Location Based Services." To be submitted. 2022.

[8] Aditya Damodaran and Alfredo Rial. "Sealed Bid Auctions with Bidder Statistics." To be submitted. 2022.

CODE

[1] Aditya Damodaran and Alfredo Rial. *Implementations for UC Updatable Databases and Applications*. 2020. URL: http://hdl.handle.net/10993/51275.

[2] Aditya Damodaran and Alfredo Rial. *Implementations for Unlinkable Updatable Databases and Oblivious Transfer with Access Control*. 2020. URL: http://hdl.handle.net/10993/51274.

[3] Aditya Damodaran and Alfredo Rial. *Implementations for Unlinkable Updatable Hiding Databases and Privacy-Preserving Loyalty Programs*. 2021. URL: http://hdl.handle.net/10993/49102.

[4] Aditya Damodaran and Alfredo Rial. *A Compiler for Stateful Zero Knowledge Data Structures*. 2022. URL: http://hdl.handle.net/10993/51929.

ACKNOWLEDGMENTS

I would like to express my gratitude toward my supervisors: Prof. Peter Y. A. Ryan, for his support, insight, and guidance, and for always being ready to lend a helping hand; and Dr. Alfredo Rial, who was responsible for much of the ideas and the primitives in this thesis, for his instruction and for masterfully guiding the direction of this project as its PI. It was particularly inspiring to witness how he juggled research projects and spent nights working on them all in parallel, with ease.

I would like to thank the members of my doctoral supervisory committee for their assistance: Dr. Jan Camenisch, and Prof. Jean-Sébastien Coron. Their valuable feedback and suggestions have been very helpful over the course of the past 4 years. I would also like to thank my defence jury, composed of Dr. Peter Rønne, the doctoral supervisory committee, and my supervisors. Thanks to Dr. Peter Rønne for his comments and feedback on this thesis. Thanks to my co-authors Dr. Maria Dubovitskaya, and Dr. Alfredo Rial.

I would then like to thank my cryptography professor from my masters course, Dr. Chuck Elliot, for nudging me to pursue a PhD in cryptography, and Neil Richardson and Dr. Shahrzad Zargari for introducing me to the field of computer security.

I would like to acknowledge Vera, Alpha, and Janine for holding the fort with regard to our social responsibility organization at the university during the busiest phases of the last 2 years, and thanks to Veerle Waterplas for her assistance. Many thanks to the numerous members of the Apsia research group, who are too many to name, and who've been helpful with both academic advice and with life lessons, from Marjan always being ready to offer help and research related advice, both before returning to the group, and after, to Balazs joking about health insurance whilst some of us were trying to scale a cliff in the Alps. Thanks to Hao for being an exceptionally hard-working officemate, I think the motivation rubbed off. Thanks to Jadwiga for organising the thesis defence.

Thanks to Styliani for bearing with my urgent questions of the form "How do I write this Greek letter in LaTeX because I cannot google something like this?" with patience, and Zane and Stanislav for their friendship; thanks to Alessandro and Lorenzo for being great friends, colleagues, and co-authors. I would like to thank Katinka for her support and for always being a voice of reason. I am deeply indebted to Dr. Marie-Laure Zollinger for her support during the drafting of this thesis, and for being more of a sister than a friend or colleague. Finally, I would like to thank my parents and my brother.

CONTENTS

LIST OF FIGURES

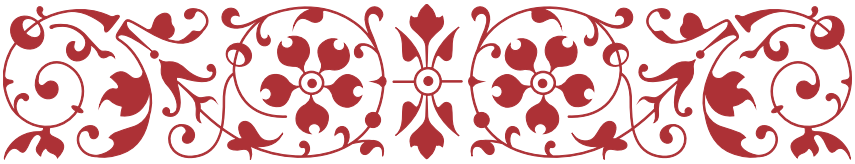
LIST OF TABLES

ACRONYMS

UC	Universal Composability
SZK	Stateful Zero Knowledge
ZK	Zero Knowledge
CD	Updatable Committed Database
UCD	Unlinkable Committed Database
NICD	Non-interactive Committed Database
UUHD	Unlinkable Updatable Hiding Database
UD	Updatable Database
UUD	Unlinkable Updatable Database
ITI	Interactive Turing Machine Instance
POT	Priced Oblivious Transfer
OT	Oblivious Transfer
OTAC	Oblivious Transfer with Access Control
LP	Loyalty Program
PPLP	Privacy Preserving Loyalty Program
PH	Purchase History
PII	Personally Identifiable Information

BP	Buyer Profile
PPB	Privacy Preserving Billing

Part I



INTRODUCTION

1 MOTIVATION

> "So this was a major breach of trust and I'm really sorry that this happened. You know we have a basic responsibility to protect people's data and if we can't do that then we don't deserve to have the opportunity to serve people."

These were Mark Zuckerberg's words [72] to a news anchor in the aftermath of the Cambridge Analytica scandal, in early 2018. The fact that an entity had been allowed to harvest data relating to several individuals, without their consent, had just come to light.

Alexander Kogan had earlier claimed to Facebook Inc., that the data he had been collecting was meant to be used for purely academic purposes.

THE CAMBRIDGE ANALYTICA INCIDENT Cambridge Analytica, an organization established in 2013 that positioned itself as a political consultancy and voter profiling firm was tasked with aiding the Ted Cruz and the Donald Trump political campaigns in 2016 [87], by building psychographic models of their electorates. The firm was of the opinion that targeted advertising campaigns tailored to the personality traits of voters, could help influence their political leanings. When the firm realized that the data required to build these models was hard to come by, they turned to Alexander Kogan, a data scientist, who went on to produce a Facebook application which sent surveys and quizzes to willing participants on a social network, in order to implicitly elicit information on their psychological traits. They also collected information on the "likes" of these users on the social network, and used it to map them to specific personality traits. While doing so, Cambridge Analytica went so far as to gather personally identifiable information from these users, including their dates of birth, cities of residence, and likes and interests, amongst other information [94].

However, much to the consternation of the world, the application leveraged the power of Facebook's Open Graph Platform to additionally gather such information from the "friends" of the application's users on the social network. This allowed Cambridge Analytica to collect personally identifiable information from about 87 million people, and in some cases, without their consent.

> Zuckerberg: "Well, clearly I wish we'd taken those steps earlier. I mean, that, that I think is probably the biggest mistake that we made here ... the feedback from the community and the world has overwhelmingly been, that, if you balance these two values of being able to take your data and some data from friends to be able to have social experiences on other apps on the one hand, this ideal of kind of data portability. And on the other hand, making sure that your data's always locked down. Guaranteeing that it never goes anywhere. You know I think we've started off a little bit on the idealistic, and maybe naive side, right, of thinking that that vision around data portability and enabling social apps was gonna be what our community preferred, and I think what we've learned over time very clearly is that the most important thing always is making sure that people's data is locked down." [72]

Facebook's aspirations, in this case, reflect those of most other "data behemoths" on the internet: using customer data to enhance user experiences and boost customer retention and sales. However, in hindsight, Zuckerberg admits to the fact that the privacy of users and sanctity of data is of sublime importance when it comes to handling such information.

CLAIRVOYANCE THROUGH PROFILING There is also an ugly side to the extent of personal information that may be gleaned from user behaviour, with their consent, producing arguably more harrowing results. Take for instance the case of a retailing giant, who assigns identification numbers to its customers in order to collect purchase information, along with a host of other demographic information [45]:

> An employee from the marketing department of the retailer, to a statistician: "If we wanted to figure out if a customer is pregnant, even if she didn't want us to know, can (sic) you do that?" [45]

The retailer in question was able to build profiles of their customers and use statistical data analysis techniques to predict changes in their lives and habits.

> "As soon as we get them buying diapers from us, they're going to start buying everything else too. If you're rushing through the store, looking for bottles, and you pass orange juice, you'll grab a carton. Oh, and there's that new DVD I want. Soon, you'll be buying cereal and paper towels from us, and keep coming back." [45]

This allowed the retailer to determine when and how targeted advertising and discounts would produce the best results.

A DAMOCLES SWORD? Companies that manage colossal amounts of data relating to their users also have greater responsibilities to protect this data, and attacks can have catastrophic results. An IT operator responsible for the collection and analysis of frequent flyer information for airline companies fell prey to a cyberattack in 2021 [47]. Though the operator claimed that this breach didn't result in the leakage of sensitive information apart from names and membership numbers, the company does hold personal customer data, passport information, and itineraries.

1.1 OUR CONTRIBUTION

1.1.1 *Privacy Preserving Protocols To The Rescue*

The cryptographic primitives and protocols laid out in this thesis aim to provide a solution to the aforementioned problems of excessive user profiling and exposure of personal information. We propose an alternative means of allowing entities to compute aggregate statistics on customer data (in order to be able to carry out customer retention activities, such as offering discounts to customers for purchasing several items belonging to a particular category from a department store, frequent flyer discounts, and the like), whilst essentially handing power and control back to the users: the users of these services are allowed to hold their own data and reveal

only the output of an aggregate statistical function against this data to the other party, whilst proving that their copy of this data is accurate. We do this by means of novel cryptographic primitives we refer to as Stateful Zero Knowledge (SZK) data structures.

The verifier learns no information on the secret held by the prover, hence the eponymous term "Zero Knowledge".

The notion of a ZK proof allows one party (commonly referred to as a *prover*) to prove that a certain statement holds true, without revealing the witnesses for this statement to another party (commonly referred to as a *verifier*). This cryptographic primitive can be used to build privacy preserving protocols, which are said to be *"privacy preserving"* in the sense that the user's privacy and data is protected when it interacts with other parties in the protocol. These primitives and protocols allow for stronger privacy guarantees, whilst still allowing organizations to carry out customer retention and incentivization practices.

Our SZK data structures allow users to store witnesses (secret information known only to the prover) that have already been used in a proof, so that they may be used to prove additional facts about this secret information. For instance, one of our protocols features a loyalty program scheme, where users may prove that they have been allotted the correct number of loyalty points for a purchase. They may then go on to compute aggregate statistics on their purchases and prove that these statistics have been computed correctly, without revealing the actual purchase history to a retailer. In existing schemes, ZK proofs and the witnesses they use are closely tied to each other, but our primitives decouple proofs from data structures that store witnesses, providing us with modular constructions.

1.1.2 *The Case For Composability*

Throughout this thesis, we employ a composability framework (namely that of the Universal Composability (UC) Framework [29]); this provides for some useful properties. In a nutshell, we gain the following:

The reader is advised to refer to Section 2.2 for a detailed account of the UC framework and related concepts.

1. Security properties hold when a protocol designed in accordance with the UC framework is executed concurrently in the presence of multiple instances of the same protocol or other protocols.

2. The protocol may be designed modularly so that modules of the protocol may be replaced by others that securely realize them. This concept is described in detail in Section 2.2.

3. Security analysis is clear-cut: the security of each module may be analysed independently rather than in situ, thanks to the "Composability Theorem", also described in Section 2.2.

1.1.3 *Bridging The Gap*

> "A colleague once told me that the world was full of bad security systems designed by people who read Applied Cryptography." - Bruce Schneier, author of the book "Applied Cryptography" [89].

The security of implementations of cryptographic software proves to be a bone of contention for cryptographers and developers alike. Though most cryptographic

libraries have been implemented true to their theoretical roots, numerous security issues do arise when developers misuse these libraries whilst implementing software [65, 96].

> "I don't see a reason to have a(sic.) x of about the same size as the p. It should be sufficient to have one about the size of q or the later used k plus a large safety margin. Decryption will be much faster with such an x." - a comment in the code for a critical GPG library. [62]

There have also been cases of security vulnerabilities in cryptographic libraries themselves, arising from the fact that the reasons behind the configuration recommendations for important cryptographic parameters and other such design choices are often lost in translation whilst programmers build libraries based on mathematical constructions. The developer of the GnuPG library saw it fit to adjust certain parameters in an implementation for an ElGamal encryption function in order to optimize the library, but these changes came at a cost: they significantly downgraded the security guarantees of the libgcrypt library [79].

Open source cryptography development teams are often understaffed and underfunded. In 2015, GnuPG was being maintained by a single developer, whilst the OpenSSL project was also suffering from similar problems when the Heartbleed vulnerability was discovered [8].

Nadi et al. [77] learnt from surveys of developers that they prefer "high level task based cryptography APIs" over low level functions [77]. With this in mind, we are of the opinion that a compiler would aid the adoption of the protocols in this thesis, and help developers build secure privacy preserving protocols without having to deal with the minute details, in the sense that we abstract away the cryptography so that developers may focus on the other aspects of building such protocols, while they may rest assured that if they are able to specify their security requirements via our notation, the compiler provides them with a secure implementation. This also does away with the need for developers to be acquainted with the state of the art with respect to the low level primitives used in our protocols. Part IV of this thesis describes our notation and compiler, which provides proof of concept implementations of our data structures, but we intend to extend this compiler with production ready libraries in the future.

1.2 ORGANIZATION

- In the following chapters of Part I, we go on to describe the technical preliminaries of this thesis.
- In Part II, we use the UC framework to describe formal definitions for various SZK data structures and the security properties they guarantee. We also depict constructions that *securely realize* the definitions laid out in Part II, and present computational cost measurements from implementations of some of these data structures.
- In Part III, we describe real world applications for our data structures, by way of privacy preserving protocols, such as the privacy preserving loyalty program protocol described earlier in this chapter. We also describe existing protocols in literature whose security properties have been attenuated by the addition of SZK data structures, and present computational cost measurements from implementations of some of these protocols.

- Part IV provides descriptions of aspects related to the implementations of our data structures and of the protocols described in Part III. We also describe a notation for SZK protocols and an SZK compiler, which takes an SZK data structure description in said notation as input, and produces an implementation for the data structure as output.

2

TECHNICAL PRELIMINARIES

2.1 INTRODUCTION

This chapter deals with a discussion of the cryptographic preliminaries which serve as the theoretical basis for our primitives and protocols described in Part II and in Part III respectively. Sections 2.3 through 2.8 also appear in public deliverables submitted to the FNR [36, 37].

2.2 UNIVERSALLY COMPOSABLE SECURITY

We employ the UC framework [29] to define our primitives and protocols, and to analyse their security. Security analysis using the framework essentially involves comparing the views of a real protocol in execution in a *real world* to that of an idealized version of the protocol, defined to be secure by definition, in an *ideal world*, when observed by a party we refer to as the *environment*.

The environment is allowed to interact with all other parties in the real and the ideal worlds.

1. We first define an *Ideal Protocol*, consisting of all parties in a protocol in addition to an *Ideal Functionality*. The Ideal Protocol is defined in such a way that it is secure by definition, and so that it guarantees the security properties we wish to achieve. The Ideal Protocol gathers inputs from all parties, securely computes their outputs on their behalf, and sends the outputs back to each party.
2. On the other hand, we also consider a *Real Protocol*, composed of all parties but with the exclusion of an Ideal Functionality. All parties use their local inputs to participate in the protocol and produce their outputs.
3. The environment interacts with the adversary in the real world, who has the power to control network communication and to corrupt parties. To account for the presence of this adversary, we introduce a *Simulator*, who is allowed to behave as the adversary does, within the ideal protocol.
4. The view of the environment $\mathcal{Z}$ (transcripts of messages from all parties) in the ideal world and that of the real world are now compared. If the environment cannot distinguish between the view of the ideal world and that of the real world, we state that the real protocol *securely realizes* the ideal protocol.

We consider only static corruptions in our models.

As defined in [29],

Definition 2.2.1 *Two binary distribution ensembles X and Y are indistinguishable if for any $c, d \in N$, there exists $k_0 \in N$, such that for all $k > k_0$ and all $a \in \cup_{k \leq k^d} \{0,1\}^k$, we have:*

$$|Pr(X(k,a) = 1) - Pr(Y(k,a) = 1)| < k^{-c}$$

Using definition 2.2.1, let $\mathsf{REAL}_{\varphi,\mathcal{A},\mathcal{Z}}(k,a)$ represent the computational output of $\mathcal{Z}$ when the adversary $\mathcal{A}$ and all other parties execute the protocol φ in the real world, and let $\mathsf{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}(k,a)$ represent that of $\mathcal{Z}$ when the simulator $\mathcal{S}$ and all dummy parties execute the ideal protocol $\mathcal{F}_\varphi$ in the ideal world, where k represents the security parameter, and a represents the input to $\mathcal{Z}$.

We state that protocol φ *securely realizes* $\mathcal{F}_\varphi$, if for all polynomial time $\mathcal{A}$ there exists a polynomial time $\mathcal{S}$ such that for all polynomial time $\mathcal{Z}$,

$$\mathsf{REAL}_{\varphi,\mathcal{A},\mathcal{Z}}(k,a) \approx \mathsf{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}(k,a) \text{ [36]}.$$

2.2.1 *The Composition Theorem*

2.2.1.1 *Protocol Emulation*

A protocol is said to UC-emulate another, if interacting with one protocol in the presence of an adversary is indistinguishable from interactions with the other and another adversary.

From [29], we have,

> **Definition 2.2.2** *Protocol π UC-emulates protocol ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$, such that for any environment $\mathcal{Z}$ and on any input, the probability that $\mathcal{Z}$ outputs 1 after interacting with $\mathcal{A}$ and parties running π differs by at most a negligible amount from the probability that $\mathcal{Z}$ outputs 1 after interacting with $\mathcal{S}$ and ϕ.*

A protocol is said to UC-realize an ideal functionality, if the protocol emulates the functionality.

2.2.1.2 *Hybrid Protocols*

The UC framework also allows for the definition of *hybrid protocols*, where parties within a protocol may make subroutine calls to ideal functionalities.

Consider a protocol π, which makes subroutine calls to another protocol ϕ. The composition theorem states that if ψ emulates ϕ, the protocol $\pi^{\psi/\phi}$, obtained by replacing calls to ϕ by calls to ψ, emulates π.

Therefore, if a protocol $\varphi^{\mathcal{G}}$ securely realizes an ideal functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model where φ is allowed to invoke the ideal functionality $\mathcal{G}$, then for any protocol ψ, which securely realizes $\mathcal{G}$, the composed protocol φ^{ψ} which is obtained by replacing every invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$ [36].

2.2.2 *A Note On Notation*

Throughout this thesis, we make use of a notation to describe aspects of our formal descriptions of UC framework models.

INTERFACES Interfaces denote points where parties may interact with ideal functionalities and send messages to or receive messages from functionalities,

depending on the tasks that must be performed with respect to the overarching protocol. We denote every message with a name consisting of three segments:

SEGMENT 1 The name of the functionality or the protocol the message corresponds to. All messages that relate to a specific functionality or protocol will share the same values in the first segment.

For instance, “pplp” for a privacy preserving loyalty program protocol.

SEGMENT 2 The name of the interface. For instance, "setup" for an interface associated with the task of setting up parameters in a protocol.

SEGMENT 3 A third field which allows for messages of different types to be sent to the same interface. We typically use 5 different entries for this field:

1. "ini" for the first message sent to an interface,
2. "sim" for messages to the simulator,
3. "rep" for responses from the simulator,
4. "req" for requests for algorithm descriptions to the simulator,
5. "alg" for messages containing algorithm descriptions in response to a previously received req message.
6. "end" for the last message denoting the end of the execution of a task.

These fields are combined to produce the name of a message. For example:

pot.setupprices.ini

refers to the first message sent to the *setupprices* interface of a *priced oblivious transfer* protocol.

This notation allows us to discern the functionality a message relates to, the task it is responsible for, and the phase within that task.

IDENTIFIERS As we model protocols in the framework around Interactive Turing Machine Instances (ITIs), the need arises for a means of distinguishing between machines that belong to the same protocol instance, and between machines that are “local” to one another, because the UC framework models communication over networks, i.e., ITIs belonging to different parties would need to resort to communicate over a network where the simulator may delay or modify messages. However, ITIs belonging to the same party may communicate directly with one another, and we refer to this mode of communication as *local*.

We assign a session identifier sid and a party identifier pid to each ITI. We use a pid to represent ITIs that belong to the same party. An sid is used to distinguish between ITIs that represent different protocol instances, and a functionality may only send or receive messages associated with the same sid.

An Ideal Functionality is considered to be "local" to all other parties in an Ideal Protocol.

- We use a session identifier sid to denote ITIs that belong to the same protocol instance.

- We use a party identifier pid to denote ITIs that belong to the same party within a protocol. Thus, ITIs with the same pid can transfer messages between each other without having to resort to sending them over a network.
- Communication over the network may be controlled by the adversary: the adversary may delay, modify or drop messages altogether.
- ITIs with differing pid values must communicate over the network (as they represent distinct parties within a protocol).

QUERY IDENTIFIERS In some cases we may wish to model the fact that an interface may be invoked more than once and the simulator may generate appropriate albeit delayed responses; we must employ a mechanism that allows for delayed responses to ensure that these responses can be correlated to the requests that generated them, even when the original order of requests is not maintained. In this case, we use a query identifier or a qid.

MESSAGE STRUCTURE Building upon the conventions described earlier in this section, every message to or from a party follows the following structure:

$$(\mathsf{pot.setupprices.ini}, sid, \mathsf{<message>})$$

Repeated messages to an interface include a qid:

$$(\mathsf{pot.setupprices.sim}, sid, qid, \mathsf{<message>})$$
$$(\mathsf{pot.setupprices.rep}, sid, qid, \mathsf{<message>})$$

ABORTS An ideal functionality may halt its execution after receiving a message from a party. In this case, the functionality sends a special abortion message to the initiating party. However, if the functionality aborts after receiving a message from the simulator, the abortion message is sent to the party which was meant to receive a response from the functionality.

To reiterate the benefits of using this framework:

COMPOSITION Thanks to the composition theorem, security properties are retained when a new protocol is composed of others defined using the framework.

MODULARITY The framework allows for the construction of "hybrid" protocols, composed of other protocols defined in the UC framework, as building blocks. This allows us to replace these building blocks by any protocol that securely realizes these constructions. This in turn provides for future-proof constructions, in the sense that building blocks may be replaced by more efficient protocols, without the need for reanalysing the security of the overarching protocol.

SECURITY ANALYSIS As hybrid protocols are composed of building blocks where security holds upon composition, this also means that the security of these protocols may be studied independently.

2.3 BILINEAR MAPS

Let $\mathbb{G}$, $\tilde{\mathbb{G}}$ and $\mathbb{G}_t$ be groups of prime order p.

A bilinear map is a map e from $\mathbb{G}$ and $\tilde{\mathbb{G}}$ to $\tilde{\mathbb{G}}$ ($e : \mathbb{G} \times \tilde{\mathbb{G}} \to \mathbb{G}_t$), where the following properties hold:

BILINEARITY $e(g^x, \tilde{g}^y) = e(g, \tilde{g})^{xy}$,

NON-DEGENERACY For all generators $g \in \mathbb{G}$ and $\tilde{g} \in \tilde{\mathbb{G}}$, $e(g, \tilde{g})$ generates $\mathbb{G}_t$,

EFFICIENCY There exists an efficient algorithm $\mathcal{G}(1^k)$ that outputs the pairing group setup $grp \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}, b \in \tilde{\mathbb{G}}$.

2.4 SECURITY ASSUMPTIONS

2.4.1 *ℓ-Diffie-Hellman Exponent*

We use the ℓ-DHE assumption, adapted to asymmetric pairing groups, as in [43].
Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$ and $\alpha \leftarrow \mathbb{Z}_p$.
Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and a tuple $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ such that $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$, for any p.p.t. adversary $\mathcal{A}$,
$\Pr[g^{(\alpha^{\ell+1})} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})] \leq \epsilon(k)$.

2.4.2 *Cube Diffie-Hellman*

Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$ and $x \leftarrow \mathbb{Z}_p$.
Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g^x, \tilde{g}^x)$, for any p.p.t. adversary $\mathcal{A}$,
$\Pr[e(g, \tilde{g})^{x^3} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g^x, \tilde{g}^x)] \leq \epsilon(k)$ [64].

2.5 SIGNATURE SCHEMES

A signature scheme consists of the algorithms KeyGen, Sign and VfSig. $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature sig on the message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, sig, m)$ outputs 1 if sig is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $\bar{m} = (m_1, \ldots, m_n)$. In this case, $\mathsf{KeyGen}(1^k, n)$ receives the maximum number n of messages as input. A signature scheme must be existentially unforgeable [52].

Definition 2.5.1 *[Existential Unforgeability] Let $\mathcal{O}_s$ be an oracle that, on input sk and a message $m \in \mathcal{M}$, outputs $\mathsf{Sign}(sk, m)$, and let S_s be a set that contains the messages sent to $\mathcal{O}_s$. For any ppt adversary $\mathcal{A}$,*

$$Pr\left[\begin{array}{l}(sk, pk) \xleftarrow{\$} \mathsf{KeyGen}(1^k);\ (m, sig) \xleftarrow{\$} \mathcal{A}(pk)^{\mathcal{O}_s(sk,\cdot)} : \\ 1 = \mathsf{VfSig}(pk, sig, m) \ \wedge\ m \in \mathcal{M} \ \wedge\ m \notin S_s\end{array}\right] \leq \epsilon(k)\ .$$

2.6 COMMITMENTS

A commitment scheme consists of algorithms CSetup, Com and VfCom. $\mathsf{CSetup}(1^k)$ generates the parameters par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment com to $x \in \mathcal{M}$ and an opening $open$. $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if com is a commitment to x with opening $open$ or 0 otherwise.

A commitment scheme must be hiding and binding. The hiding property ensures that a commitment com to x does not reveal any information about x, whereas the binding property ensures that com cannot be opened to another value x'.

Definition 2.6.1 *[Hiding Property] For any PPT adversary $\mathcal{A}$,*

$$Pr\left[\begin{array}{l} par_c \xleftarrow{\$} \mathsf{CSetup}(1^k);\ (x_0, st) \xleftarrow{\$} \mathcal{A}(par_c); \\ x_1 \xleftarrow{\$} \mathcal{M}; \\ b \xleftarrow{\$} \{0,1\};\ (com, open) \xleftarrow{\$} \mathsf{Com}(par_c, x_b); \\ b' \xleftarrow{\$} \mathcal{A}(st, com) :\ x_0 \in \mathcal{M} \ \wedge\ b = b' \end{array}\right] \leq \frac{1}{2} + \epsilon(k) \ .$$

Definition 2.6.2 *[Binding Property] For any PPT adversary $\mathcal{A}$,*

$$Pr\left[\begin{array}{l} par_c \xleftarrow{\$} \mathsf{CSetup}(1^k); \\ (com, x, open, x', open') \xleftarrow{\$} \mathcal{A}(par_c) : \\ x \in \mathcal{M} \ \wedge\ x' \in \mathcal{M} \\ \wedge\, 1 = \mathsf{VfCom}(par_c, com, x, open) \wedge \\ 1 = \mathsf{VfCom}(par_c, com, x', open') \ \wedge\ x \neq x' \end{array}\right] \leq \epsilon(k) \ .$$

We use a commitment scheme that is additively homomorphic. Given two commitments com_1 and com_2 to messages x_1 and x_2 with openings $open_1$ and $open_2$, there exists an operation $\cdot$ such that $com \leftarrow com_1 \cdot com_2$ is a commitment to $x = x_1 + x_2$ with opening $open = open_1 + open_2$. If $x_2 = 0$, then com is a rerandomization of com_1, which we denote by $com \leftarrow \mathsf{CRerand}(com_1, open_2)$.

2.7 VECTOR COMMITMENTS

2.7.1 *Hiding Vector Commitments*

Hiding vector commitments [32, 69] allow us to commit to a vector of messages and to open the commitment to one of the messages in such a way that the size of the opening (also referred to as the witness) is independent of the length of the vector. A hiding vector commitment (VC) scheme consists of the following algorithms.

$\mathsf{VC.Setup}(1^k, \ell)$. On input the security parameter 1^k and an upper bound ℓ on the size of the vector, generate the parameters of the commitment scheme par, which include a description of the message space $\mathcal{M}$ and a description of the randomness space $\mathcal{R}$.

$\mathsf{VC.Commit}(par, \mathbf{x}, r)$. On input a vector $\mathbf{x} \in \mathcal{M}^n$ ($n \leq \ell$) and $r \in \mathcal{R}$, output a commitment vc to $\mathbf{x}$.

$\mathsf{VC.Prove}(par, i, \mathbf{x}, r)$. Compute a witness w for $\mathbf{x}[i]$.

$\mathsf{VC.Verify}(par, vc, x, i, w)$. Output 1 if w is a valid witness for x being at position i and 0 otherwise.

$\mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$. On input a commitment vc with value x at position j and randomness r, output a commitment vc' with value x' at position j and randomness r'. The other positions remain unchanged.

$\mathsf{VC.VerComUpd}(par, vc, vc', w, j, x, r, x', r')$. On input commitments vc and vc', a witness w, a position j, the values x and x' and the randomness r and r', output 1 if w is a valid witness for x being at position j in the commitment vc and if vc' is an update of vc that replaces x by x' at position j and r by r'.

$\mathsf{VC.WitUpd}(par, w, i, j, x, r, x', r')$. On input a witness w for a position i valid for a commitment vc with value x at position j and randomness r, output a witness w' for position i valid for a commitment vc' with value x' at position j and randomness r'.

Definition 2.7.1 *A vector commitment scheme must be correct, hiding, and binding.*

CORRECTNESS. Correctness requires that for $par \leftarrow \mathsf{VC.Setup}(1^k, \ell)$, $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \leftarrow \mathcal{M}^n$, $r \leftarrow \mathcal{R}$, $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$, $i \leftarrow [1, n]$ and $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r)$, $\mathsf{VC.Verify}(par, vc, \mathbf{x}[i], i, w)$ outputs 1 with probability 1.

HIDING. The hiding property requires that any ppt adversary $\mathcal{A}$ has negligible advantage in the following game. $\mathcal{A}$ chooses a vector and sends it to the challenger. The challenger picks a random vector, commits to one of the vectors and sends the commitment to $\mathcal{A}$. $\mathcal{A}$ guesses which vector was used to compute the commitment. More formally, for ℓ polynomial in k we require

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\ (\mathbf{x}_0, st) \leftarrow \mathcal{A}(par);\ r \leftarrow \mathcal{R}; \\ \mathbf{x}_1 \xleftarrow{\$} \mathcal{M}^\ell;\ b \leftarrow \{0,1\};\ vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}_b, r); \\ b' \leftarrow \mathcal{A}(st, vc) :\ b = b' \ \wedge\ \mathbf{x}_0 \in \mathcal{M}^\ell \end{array}\right] = \frac{1}{2} + \epsilon(k) \ .$$

When updating a commitment, we require that $\mathcal{A}$ cannot distinguish whether a commitment vc is an update of a commitment to a vector specified by the adversary or a commitment to a random vector, i.e., that

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\ (\mathbf{x}, j, x', r, st) \leftarrow \mathcal{A}(par); \\ r' \leftarrow \mathcal{R};\ \mathbf{x}' \xleftarrow{\$} \mathcal{M}^\ell; \\ vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}', r') : \\ 1 = \mathcal{A}(st, vc) \wedge\ \mathbf{x} \in \mathcal{M}^\ell \wedge\ j \in [1, \ell] \wedge\ x' \in \mathcal{M}\ \wedge\ r \in \mathcal{R} \end{array}\right] =$$

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\ (\mathbf{x}, j, x', r, st) \leftarrow \mathcal{A}(par); \\ r' \leftarrow \mathcal{R}; \\ vc \leftarrow \mathsf{VC.ComUpd}(par, \mathsf{VC.Commit}(par, \mathbf{x}, r), j, \mathbf{x}[j], r, x', r') : \\ 1 = \mathcal{A}(st, vc) \wedge\ \mathbf{x} \in \mathcal{M}^\ell \wedge\ j \in [1, \ell] \wedge\ x' \in \mathcal{M}\ \wedge\ r \in \mathcal{R} \end{array}\right]$$

BINDING. The binding property requires that no adversary can output a vector commitment vc, a position $i \in [1, \ell]$, two values x and x' and two respective witnesses w and w' such that $\mathsf{VC.Verify}$ accepts both, i.e., for ℓ polynomial in k:

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\ (vc, i, x, x', w, w') \leftarrow \mathcal{A}(par) : \\ \mathsf{VC.Verify}(par, vc, x, i, w) = 1\ \wedge\ x \neq x' \wedge \\ \mathsf{VC.Verify}(par, vc, x', i, w') = 1\ \wedge\ i \in [1, \ell]\ \wedge x, x' \in \mathcal{M} \end{array}\right] \leq \epsilon(k)\ .$$

We use VC schemes that are additively homomorphic. Given two commitments vc_1 and vc_2 to vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ with randomness r_1 and r_2, there exists an operation $\cdot$ such that $vc \leftarrow vc_1 \cdot vc_2$ is a commitment to $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ with $r = r_1 + r_2$. (If vc_2 is a commitment to the vector $\mathbf{x}[i] = 0$ for all $i \in [1, \ell]$, then vc is a rerandomization of vc_1, which we denote by $vc = \mathsf{VC.Rerand}(vc_1, r_2)$.) In addition, if $w_{1,i}$ is a witness for $\mathbf{x}_1[i]$ being a position i in the vector $\mathbf{x}_1$ committed in vc_1, and $w_{2,i}$ is an opening for $\mathbf{x}_2[i]$ being a position i in the vector $\mathbf{x}_2$ committed in vc_2, then $w_i \leftarrow w_{1,i} \cdot w_{2,i}$ is an opening for $\mathbf{x}_1[i] + \mathbf{x}_2[i]$ being at position i in the vector $\mathbf{x}$ committed in vc.

2.7.1.1 *Construction*

We show a hiding vector commitment scheme that is secure under the Diffie-Hellman Exponent (DHE) assumption [69]. Let $k \in \mathbb{N}$ denote the security parameter.

$\mathsf{VC.Setup}(1^k, \ell)$. Generate groups $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$, pick $\alpha \leftarrow \mathbb{Z}_p$, and compute $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$, where $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$. Output the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$.

$\mathsf{VC.Commit}(par, \mathbf{x}, r)$. Let $|\mathbf{x}| = n \leq \ell$. Output

$$vc = g^r \cdot \prod_{j=1}^{n} g_{\ell+1-j}^{\mathbf{x}[j]} = g^r \cdot g_\ell^{\mathbf{x}[1]} \cdots g_{\ell+1-n}^{\mathbf{x}[n]}\ .$$

$\mathsf{VC.Prove}(par, i, \mathbf{x}, r)$. Let $|\mathbf{x}| = n \leq \ell$. Output

$$w = g_i^r \cdot \prod_{j=1, j\neq i}^{n} g_{\ell+1-j+i}^{\mathbf{x}[j]} \ .$$

$\mathsf{VC.Verify}(par, vc, x, i, w)$. Output 1 if $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$, else output 0.

$\mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$. Output the commitment

$$vc' = vc \cdot \frac{g^{r'} \cdot g_{\ell+1-j}^{x'}}{g^r \cdot g_{\ell+1-j}^{x}} = vc \cdot g^{r'-r} \cdot g_{\ell+1-j}^{x'-x} \ .$$

$\mathsf{VC.VerComUpd}(par, vc, vc', w, j, x, r, x', r')$. Run $v \leftarrow \mathsf{VC.Verify}(par, vc, x, j, w)$. If $v \leftarrow 0$, output v, else run $\bar{vc} \leftarrow \mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$. If $\bar{vc} = vc'$, output 1, else output 0.

$\mathsf{VC.WitUpd}(par, w, i, j, x, r, x', r')$. If $i = j$, output w. Otherwise output the witness

$$w' = w \cdot \frac{g_i^{r'} \cdot g_{\ell+1-j+i}^{x'}}{g_i^r \cdot g_{\ell+1-j+i}^{x}} = w \cdot g_i^{r'-r} \cdot g_{\ell+1-j+i}^{x'-x} \ .$$

Theorem 2.7.2 *This vector commitment scheme is correct, hiding, and binding as defined in 2.7.1 under the ℓ-DHE assumption.*

PROOF OF THEOREM 2.7.2. Correctness can be checked as follows.

$$\begin{aligned} e(vc, \tilde{g}_i)/e(w, \tilde{g}) &= \frac{e(g^{r+\sum_{j=1}^n \mathbf{x}[j](\alpha^{\ell+1-j})}, \tilde{g}^{(\alpha^i)})}{e(g^{(r(\alpha^i)+\sum_{j=1, j\neq i}^n \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\ &= \frac{e(g^{r(\alpha^i)+\sum_{j=1}^n \mathbf{x}[j](\alpha^{\ell+1-j+i})}, \tilde{g})}{e(g^{(r(\alpha^i)+\sum_{j=1, j\neq i}^n \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\ &= e(g, \tilde{g})^{\mathbf{x}[i](\alpha^{\ell+1})} \\ &= e(g_1, \tilde{g}_\ell)^{\mathbf{x}[i]} \ . \end{aligned}$$

This vector commitment scheme fulfils the hiding property in an information-theoretic way.

We show that this vector commitment scheme fulfils the binding property under the ℓ-DHE assumption. Given an adversary $\mathcal{A}$ that breaks the binding property with non-negligible probability ν, we construct an algorithm $\mathcal{T}$ that breaks the ℓ-DHE assumption with non-negligible probability ν. First, $\mathcal{T}$ receives an instance $(e, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, p, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ of the ℓ-DHE assumption. $\mathcal{T}$ sets $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ and sends par to $\mathcal{A}$. $\mathcal{A}$ returns (vc, i, x, x', w, w') such that $\mathsf{VC.Verify}(par, vc, x, i, w)$

$= 1$, $\mathsf{VC.Verify}(par, vc, x', i, w') = 1$, $i \in [1, \ell]$, $x, x' \in \mathcal{M}$, and $x \neq x'$. $\mathcal{T}$ computes $g_{\ell+1}$ as follows:

$$\begin{aligned} e(w, \tilde{g})e(g_1, \tilde{g}_\ell)^x &= e(w', \tilde{g})e(g_1, \tilde{g}_\ell)^{x'} \\ e(w/w', \tilde{g}) &= e(g_1, \tilde{g}_\ell)^{x'-x} \\ e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_1, \tilde{g}_\ell) \\ e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_{\ell+1}, \tilde{g}) \ . \end{aligned}$$

The last equation implies that $g_{\ell+1} = (w/w')^{1/(x'-x)}$. $\mathcal{T}$ returns $(w/w')^{1/(x'-x)}$ as a solution for the ℓ-DHE problem.

2.7.2 *Non Hiding Vector Commitments*

A non-hiding vector commitment (NHVC) scheme [32, 69] allows one to succinctly commit to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \in \mathcal{M}^n$ such that it is possible to compute an opening w to $\mathbf{x}[i]$, with the size of w independent of i and n. The scheme consists of the following algorithms.

$\mathsf{VC.Setup}(1^k, \ell)$. On input the security parameter 1^k and an upper bound ℓ on the size of the vector, generate the parameters of the vector commitment scheme par, which include a description of the message space $\mathcal{M}$.

$\mathsf{VC.Commit}(par, \mathbf{x})$. On input a vector $\mathbf{x} \in \mathcal{M}^n$ ($n \leq \ell$), output a commitment vc to $\mathbf{x}$.

$\mathsf{VC.Prove}(par, i, \mathbf{x})$. Compute an opening w for $\mathbf{x}[i]$.

$\mathsf{VC.Verify}(par, vc, x, i, w)$. Output 1 if w is a valid opening for x being at position i and 0 otherwise.

$\mathsf{VC.ComUpd}(par, vc, j, x, x')$. On input a commitment vc with value x at position j, output a commitment vc' with value x' at position j. The other positions remain unchanged.

$\mathsf{VC.WitUpd}(par, w, i, j, x, x')$. On input an opening w for position i valid for a commitment vc with value x at position j, output an opening w' for position i valid for a commitment vc' with value x' at position j.

A non-hiding VC scheme must be correct and binding [32].

CORRECTNESS. Correctness requires that for $par \leftarrow \mathsf{VC.Setup}(1^k, \ell)$, $\mathbf{x} \leftarrow (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \in \mathcal{M}^n$, $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, $i \leftarrow [1, n]$ and $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$, $\mathsf{VC.Verify}(par, vc, \mathbf{x}[i], i, w)$ outputs 1 with probability 1.

BINDING. The binding property requires that no adversary can output a vector commitment vc, a position $i \in [1, \ell]$, two values x and x' and two respective witnesses w and w' such that $\mathsf{VC.Verify}$ accepts both, i.e., for ℓ polynomial in k:

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\ (vc, i, x, x', w, w') \leftarrow \mathcal{A}(par) : \\ 1 = \mathsf{VC.Verify}(par, vc, x, i, w) \ \wedge\ x \neq x' \wedge \\ 1 = \mathsf{VC.Verify}(par, vc, x', i, w') \ \wedge\ i \in [1, \ell] \ \wedge\ x, x' \in \mathcal{M} \end{array}\right] \leq \epsilon(k) \ .$$

2.7.2.1 *Construction*

We use an NHVC scheme secure under the ℓ-DHE assumption [69].

$\mathsf{VC.Setup}(1^k, \ell)$. Generate groups $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$, pick $\alpha \leftarrow \mathbb{Z}_p$ and compute $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$, where $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$. Output $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p)$.

$\mathsf{VC.Commit}(par, \mathbf{x})$. Let $|\mathbf{x}| = n \leq \ell$. Output $vc = \prod_{j=1}^{n} g_{\ell+1-j}^{\mathbf{x}[j]}$.

$\mathsf{VC.Prove}(par, i, \mathbf{x})$. Let $|\mathbf{x}| = n \leq \ell$. Output $w = \prod_{j=1, j\neq i}^{n} g_{\ell+1-j+i}^{\mathbf{x}[j]}$.

$\mathsf{VC.Verify}(par, vc, x, i, w)$. Output 1 if $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$, else 0.

$\mathsf{VC.ComUpd}(par, vc, j, x, x')$. Output $vc' = vc \cdot g_{\ell+1-j}^{x'-x}$.

$\mathsf{VC.WitUpd}(par, w, i, j, x, x')$. If $i = j$, output w, else $w' = w \cdot g_{\ell+1-j+i}^{x'-x}$.

Theorem 2.7.3 *The NHVC scheme is correct and binding under the ℓ-DHE assumption.*

PROOF OF THEOREM 2.7.3. Correctness can be checked as follows:

$$
\begin{aligned}
& e(vc, \tilde{g}_i)/e(w, \tilde{g}) = \\
&= \frac{e(g^{\sum_{j=1}^{n} \mathbf{x}[j](\alpha^{\ell+1-j})}, \tilde{g}^{(\alpha^i)})}{e(g^{(\sum_{j=1, j\neq i}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\
&= \frac{e(g^{(\sum_{j=1}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i})}, \tilde{g})}{e(g^{(\sum_{j=1, j\neq i}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\
&= e(g, \tilde{g})^{\mathbf{x}[i](\alpha^{\ell+1})} \\
&= e(g_1, \tilde{g}_\ell)^{\mathbf{x}[i]} \ .
\end{aligned}
$$

We show that this NHVC scheme fulfils the binding property under the ℓ-DHE assumption. Given an adversary $\mathcal{A}$ that breaks the binding property with non-negligible probability ν, we construct an algorithm $\mathcal{T}$ that breaks the ℓ-DHE assumption with non-negligible probability ν. First, $\mathcal{T}$ receives an instance $(e, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, p, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ of the ℓ-DHE assumption. $\mathcal{T}$ sets $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ and sends par to $\mathcal{A}$. $\mathcal{A}$ returns (vc, i, x, x', w, w') such that $\mathsf{VC.Verify}(par, vc, x, i, w) = 1$, $\mathsf{VC.Verify}(par, vc, x', i, w') = 1$, $i \in [1, \ell]$, $x, x' \in \mathcal{M}$, and $x \neq x'$. $\mathcal{T}$ computes $g_{\ell+1}$ as follows:

$$
\begin{aligned}
e(w, \tilde{g}) e(g_1, \tilde{g}_\ell)^x &= e(w', \tilde{g}) e(g_1, \tilde{g}_\ell)^{x'} \\
e(w/w', \tilde{g}) &= e(g_1, \tilde{g}_\ell)^{x'-x} \\
e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_1, \tilde{g}_\ell) \\
e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_{\ell+1}, \tilde{g}) \ .
\end{aligned}
$$

The last equation implies that $g_{\ell+1} = (w/w')^{1/(x'-x)}$. $\mathcal{T}$ returns $(w/w')^{1/(x'-x)}$ as a solution for the ℓ-DHE problem. This concludes the proof of Theorem 2.7.3.

2.7.3 *Sub Vector Commitments*

A subvector commitment (SVC) scheme allows us to succinctly compute a commitment svc to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[\ell]) \in \mathcal{M}^\ell$. A commitment svc to $\mathbf{x}$ can be opened to a subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ is the set of indices that determine the positions of the committed vector $\mathbf{x}$ that are opened. The size of an opening w_I for $\mathbf{x}_I$ is independent of both the size of I and of the length ℓ of the committed vector. We extend the definition of SVC in [64] with algorithms to update commitments and openings.

SVC.Setup$(1^k, \ell)$. On input the security parameter 1^k and an upper bound ℓ on the size of the vector, generate the parameters par, which include a description of the message space $\mathcal{M}$.

SVC.Commit$(par, \mathbf{x})$. On input a vector $\mathbf{x} \in \mathcal{M}^\ell$, output a commitment svc to $\mathbf{x}$.

SVC.Open$(par, I, \mathbf{x})$. On input a vector $\mathbf{x}$ and a set $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$, compute an opening w_I for the subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$.

SVC.Verify$(par, svc, \mathbf{x}_I, I, w_I)$. Output 1 if w_I is a valid opening for the set of positions $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ such that $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $\mathbf{x}$ is the vector committed in svc. Otherwise output 0.

SVC.ComUpd$(par, svc, \mathbf{x}, i, x)$. On input a commitment svc to a vector $\mathbf{x}$, output a commitment svc' to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = x$ and, for all $j \in [1, \ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

SVC.OpenUpd$(par, w_I, \mathbf{x}, I, i, x)$. On input an opening w_I for a set I valid for a commitment to a vector $\mathbf{x}$, output an opening w'_I valid for a commitment to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = x$ and, for all $j \in [1, \ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

An SVC scheme must be correct and binding [64]. We recall the correctness and binding properties [64] and define correctness for the update algorithms.

CORRECTNESS. Correctness requires that for any $par \leftarrow$ SVC.Setup$(1^k, \ell)$, $\mathbf{x} \leftarrow (\mathbf{x}[1], \ldots, \mathbf{x}[\ell]) \in \mathcal{M}^\ell$, $svc \leftarrow$ SVC.Commit$(par, \mathbf{x})$, $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ and $w_I \leftarrow$ SVC.Open$(par, I, \mathbf{x})$, SVC.Verify$(par, svc, \mathbf{x}_I, I, w_I)$ outputs 1 with probability 1, where $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$.

For the update algorithms, correctness requires that, for any par, $\mathbf{x}$, svc, I and w_I computed as shown above, and for any $i \in [1, \ell]$, $x \in \mathcal{M}$, $svc' \leftarrow$ SVC.ComUpd$(par, svc, \mathbf{x}, i, x)$ and $w'_I \leftarrow$ SVC.OpenUpd$(par, w_I, \mathbf{x}, I, i, x)$, SVC.Verify$(par, svc', \mathbf{x}'_I, I, w'_I)$ outputs 1 with probability 1, where $\mathbf{x}'_I = (\mathbf{x}'[i_1], \ldots, \mathbf{x}'[i_n])$ and $\mathbf{x}'$ is such that $\mathbf{x}'[i] = x$ and for all $j \in [1, \ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

BINDING. This property requires that no adversary can output a commitment svc, two sets of positions $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ and $J = \{j_1, \ldots, j_{n'}\} \subseteq [1, \ell]$, two subvectors $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$ and $\mathbf{x}_J = (\mathbf{x}'[j_1], \ldots, \mathbf{x}'[j_{n'}])$ and two openings w_I and w_J such that SVC.Verify accepts

both, but there exists an index $i \in I \cap J$ such that $\mathbf{x}[i] \neq \mathbf{x}'[i]$, i.e., for ℓ polynomial in k:

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{SVC.Setup}(1^k, \ell);\ (svc, I, J, \mathbf{x}_I, \mathbf{x}_J, w_I, w_J) \leftarrow \mathcal{A}(par) : \\ 1 = \mathsf{SVC.Verify}(par, svc, \mathbf{x}_I, I, w_I) \wedge \\ 1 = \mathsf{SVC.Verify}(par, svc, \mathbf{x}_J, J, w_J) \wedge \\ I = \{i_1, \ldots, i_n\} \subseteq [1, \ell] \ \wedge\ J = \{j_1, \ldots, j_{n'}\} \subseteq [1, \ell] \wedge \\ \mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n]) \in \mathcal{M}^n \wedge \\ \mathbf{x}_J = (\mathbf{x}'[j_1], \ldots, \mathbf{x}'[j_{n'}]) \in \mathcal{M}^{n'} \wedge \\ \exists\, i \in I \cap J \text{ such that } \mathbf{x}[i] \neq \mathbf{x}'[i] \end{array}\right] \leq \epsilon(k) \ .$$

2.7.3.1 *Construction*

We use an SVC scheme secure under the CubeDH assumption [64], which we extend with update algorithms for commitments and openings.

$\mathsf{SVC.Setup}(1^k, \ell)$. Generate $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$. For all $i \in [1, \ell]$, pick $z_i \leftarrow \mathbb{Z}_p$ and compute $g_i \leftarrow g^{z_i}$ and $\tilde{g}_i \leftarrow \tilde{g}^{z_i}$. For all $i \in [1, \ell]$ and $i' \in [1, \ell]$ such that $i \neq i'$, compute $h_{i,i'} \leftarrow g^{z_i z_{i'}}$. Output $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \{g_i, \tilde{g}_i\}_{\forall i \in [1,\ell]}, \{h_{i,i'}\}_{\forall i, i' \in [1,\ell], i \neq i'})$.

$\mathsf{SVC.Commit}(par, \mathbf{x})$. Output $svc = \prod_{i=1}^{\ell} g_i^{\mathbf{x}[i]}$.

$\mathsf{SVC.Open}(par, I, \mathbf{x})$. Output $w_I = \prod_{i \in I} \prod_{i' \notin I} h_{i,i'}^{\mathbf{x}[i']}$.

$\mathsf{SVC.Verify}(par, svc, \mathbf{x}_I, I, w_I)$. Parse I as $\{i_1, \ldots, i_n\} \subseteq [1, \ell]$ and $\mathbf{x}_I$ as $(\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$. Output 1 if

$$e\left(\frac{svc}{\prod_{i \in I} g_i^{\mathbf{x}[i]}}, \prod_{i \in I} \tilde{g}_i\right) = e(w_I, \tilde{g})$$

$\mathsf{SVC.ComUpd}(par, svc, \mathbf{x}, i, x)$. Output $svc' = svc \cdot g_i^{x - \mathbf{x}[i]}$.

$\mathsf{SVC.OpenUpd}(par, w_I, \mathbf{x}, I, i, x)$. If $i \in I$, output w_I, else $w_I' = w_I \cdot \prod_{j \in I} h_{j,i}^{x - \mathbf{x}[i]}$.

This scheme is correct and binding under the CubeDH assumption [64].

2.8 IDEAL FUNCTIONALITIES

We use the following ideal functionalities as building blocks to construct our primitives and protocols:

2.8.1 *Common Reference String*

Our constructions use the functionality $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [29]. $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string, and consists of one interface crs.get. A party $\mathcal{P}$ uses the crs.get interface to request and receive the common reference string crs from $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. In the first invocation, $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ generates crs by running algorithm CRS.Setup. The simulator $\mathcal{S}$ also receives crs. We describe $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ below.

DESCRIPTION OF $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ is parameterized by a ppt algorithm CRS.Setup. $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string:

1. On input (crs.get.ini, sid) from any party $\mathcal{P}$:
   - If (sid, crs) is not stored, run $crs \leftarrow$ CRS.Setup and store (sid, crs).
   - Create a fresh qid and store $(qid, \mathcal{P})$.
   - Send (crs.get.sim, sid, qid, crs) to $\mathcal{S}$.

S. On input (crs.get.rep, sid, qid) from the simulator $\mathcal{S}$:
   - Abort if $(qid, \mathcal{P})$ is not stored.
   - Delete the record $(qid, \mathcal{P})$.
   - Send (crs.get.end, sid, crs) to $\mathcal{P}$.

2.8.2 *Key Registration*

We depict the ideal functionality $\mathcal{F}_{\text{REG}}$ for key registration in [29]. $\mathcal{F}_{\text{REG}}$ interacts with any party $\mathcal{T}$ that registers a message v and with any party $\mathcal{P}$ that retrieves the registered message. $\mathcal{F}_{\text{REG}}$ consists of two interfaces reg.register and reg.retrieve. $\mathcal{T}$ uses reg.register to register a message v with $\mathcal{F}_{\text{REG}}$. $\mathcal{P}$ uses reg.retrieve to retrieve v from $\mathcal{F}_{\text{REG}}$. We depict $\mathcal{F}_{\text{REG}}$ below.

DESCRIPTION OF $\mathcal{F}_{\text{REG}}$. $\mathcal{F}_{\text{REG}}$ is parameterized by a message space $\mathcal{M}$.

1. On input (reg.register.ini, sid, v) from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, sid')$, or if $v \notin \mathcal{M}$ or if there is a tuple $(sid, v', 0)$ stored.
   - Store $(sid, v, 0)$.
   - Send (reg.register.sim, sid, v) to $\mathcal{S}$.

S. On input (reg.register.rep, sid) from the simulator $\mathcal{S}$:
   - Abort if $(sid, v, 0)$ is not stored or if $(sid, v, 1)$ is already stored.
   - Store $(sid, v, 1)$ and parse sid as $(\mathcal{T}, sid')$.

- Send $(\mathsf{reg.register.end}, sid)$ to $\mathcal{T}$.

2. On input $(\mathsf{reg.retrieve.ini}, sid)$ from any party $\mathcal{P}$:
   - If $(sid, v, 1)$ is stored, set $v' \leftarrow v$; else set $v' \leftarrow \perp$.
   - Create a fresh qid and store $(qid, \mathcal{P}, v')$.
   - Send $(\mathsf{reg.retrieve.sim}, sid, qid, v')$ to $\mathcal{S}$.

S. On input $(\mathsf{reg.retrieve.rep}, sid, qid)$ from the simulator $\mathcal{S}$:
   - Abort if $(qid, \mathcal{P}, v')$ is not stored.
   - Delete the record $(qid, \mathcal{P}, v')$.
   - Send $(\mathsf{reg.retrieve.end}, sid, v')$ to $\mathcal{P}$.

2.8.3 *Authenticated Channel*

Our constructions use the functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel in [29]. $\mathcal{F}_{\mathrm{AUT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{aut.send}$. $\mathcal{T}$ uses the $\mathsf{aut.send}$ interface to send a message m to $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{F}_{\mathrm{AUT}}$ leaks m to the simulator $\mathcal{S}$ and, after receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{AUT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$. We describe $\mathcal{F}_{\mathrm{AUT}}$ below.

Figure 2.6: Description of $\mathcal{F}_{\mathrm{AUT}}$.

$\mathcal{F}_{\mathrm{AUT}}$ is parameterized by a message space $\mathcal{M}$.

1. On input $(\mathsf{aut.send.ini}, sid, m)$ from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$ or if $m \notin \mathcal{M}$.
   - Create a fresh qid and store $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{aut.send.sim}, sid, qid, m)$ to $\mathcal{S}$.

S. On input $(\mathsf{aut.send.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{R}, m)$ is not stored.
   - Delete the record $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{aut.send.end}, sid, m)$ to $\mathcal{R}$.

2.8.4 *Secure Message Transmission*

Our constructions use the functionality $\mathcal{F}_{\mathrm{SMT}}$ for secure message transmission described in [29]. $\mathcal{F}_{\mathrm{SMT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{smt.send}$. $\mathcal{T}$ uses the $\mathsf{smt.send}$ interface to send a message m to $\mathcal{F}_{\mathrm{SMT}}$. $\mathcal{F}_{\mathrm{SMT}}$ leaks $l(m)$, where $l : \mathcal{M} \to \mathbb{N}$ is a function that leaks the message length, to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{SMT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Figure 2.7: Description of $\mathcal{F}_{\text{SMT}}$.

$\mathcal{F}_{\text{SMT}}$ is parameterized by a message space $\mathcal{M}$ and by a leakage function $l : \mathcal{M} \to \mathbb{N}$, which leaks the message length.

1. On input (smt.send.ini, sid, m) from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$ or if $m \notin \mathcal{M}$.
   - Create a fresh qid and store $(qid, \mathcal{R}, m)$.
   - Send (smt.send.sim, sid, qid, $l(m)$) to $\mathcal{S}$.

S. On input (smt.send.rep, sid, qid) from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{R}, m)$ is not stored.
   - Delete the record $(qid, \mathcal{R}, m)$.
   - Send (smt.send.end, sid, m) to $\mathcal{R}$.

2.8.5 *Oblivious Transfer*

Our constructions use the ideal functionality $\mathcal{F}_{\text{OT}}$ for oblivious transfer. $\mathcal{F}_{\text{OT}}$ interacts with a sender $\mathcal{E}$ and a receiver $\mathcal{R}$, and consists of three interfaces: ot.init, ot.request and ot.transfer.

1. $\mathcal{E}$ uses the ot.init interface to send the messages $\langle m_n \rangle_{n=1}^{N}$ to $\mathcal{F}_{\text{OT}}$. $\mathcal{F}_{\text{OT}}$ stores $\langle m_n \rangle_{n=1}^{N}$ and sends the number N of messages to $\mathcal{R}$. The simulator $\mathcal{S}$ also learns N.

2. $\mathcal{R}$ uses the ot.request interface to send an index $\sigma \in [1, N]$, a commitment com_σ and an opening $open_\sigma$ to $\mathcal{F}_{\text{OT}}$. $\mathcal{F}_{\text{OT}}$ parses the commitment com_σ as $(parcom, com_\sigma, \text{COM.Verify})$ and verifies the commitment by running COM.Verify. $\mathcal{F}_{\text{OT}}$ stores $[\sigma, com_\sigma]$ and sends com_σ to $\mathcal{E}$.

3. $\mathcal{E}$ uses the ot.transfer interface to send a commitment com_σ to $\mathcal{F}_{\text{OT}}$. If a tuple $[\sigma, com_\sigma]$ is stored, $\mathcal{F}_{\text{OT}}$ sends the message m_σ to $\mathcal{R}$.

$\mathcal{F}_{\text{OT}}$ is similar to existing functionalities for OT [27], except that it receives a commitment com_σ to the index σ and an opening $open_\sigma$ for that commitment. In addition, the transfer phase is split up into two interfaces ot.request and ot.transfer, so that $\mathcal{E}$ receives com_σ in the request phase. These changes are needed to use in our POT protocol the method in [20] to ensure that, when purchasing an item, the buyer sends the same index σ to $\mathcal{F}_{\text{OT}}$ and to other functionalities. It is generally easy to modify existing UC OT protocols so that they realize our functionality $\mathcal{F}_{\text{OT}}$.

Figure 2.8: Description of $\mathcal{F}_{\text{OT}}$.

Functionality $\mathcal{F}_{\text{OT}}$ runs with a sender $\mathcal{E}$ and a receiver $\mathcal{R}$, and is parameterized with a maximum number of messages $\mathcal{N}_{max}$ and a message space $\mathcal{M}$.

1. On input (ot.init.ini, sid, $\langle m_n \rangle_{n=1}^{N}$) from $\mathcal{E}$:

a) Abort if $sid \notin (\mathcal{E}, \mathcal{R}, sid')$, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is already stored, or if $N > \mathcal{N}_{max}$.

b) Abort if for $n = 1$ to N, $m_n \notin \mathcal{M}$.

c) Store $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$.

d) Send $(\mathsf{ot.init.sim}, sid, N)$ to S.

S. On input $(\mathsf{ot.init.rep}, sid)$ from S:

a) Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.

b) Store $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ and initialize an empty table Tbl_{ot}.

c) Send $(\mathsf{ot.init.end}, sid, N)$ to $\mathcal{R}$.

2. On input $(\mathsf{ot.request.ini}, sid, \sigma, com_\sigma, open_\sigma)$ from $\mathcal{R}$:

a) Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored.

b) Abort if $\sigma \notin [1, N]$.

c) Parse com_σ as $(parcom, com_\sigma, \mathsf{COM.Verify})$.

d) Abort if $\mathsf{COM.Verify}$ is not a ppt algorithm, or if $1 \neq \mathsf{COM.Verify}(parcom, com_\sigma, open_\sigma, \sigma)$.

e) Create a fresh qid and store $(qid, \sigma, com_\sigma)$.

f) Send $(\mathsf{ot.request.sim}, sid, qid, com_\sigma)$ to S.

S. On input $(\mathsf{ot.request.rep}, sid, qid)$ from S:

a) Abort if $(qid, \sigma, com_\sigma)$ is not stored.

b) Append $[\sigma, com_\sigma]$ to Tbl_{ot}.

c) Delete the record $(qid, \sigma, com_\sigma)$.

d) Send $(\mathsf{ot.request.end}, sid, com_\sigma)$ to $\mathcal{E}$.

3. On input $(\mathsf{ot.transfer.ini}, sid, com_\sigma)$ from $\mathcal{E}$:

a) Abort if there is no entry $[\sigma, com_\sigma]$ in Tbl_{ot}.

b) Create a fresh qid and store (qid, com_σ).

c) Send $(\mathsf{ot.transfer.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{ot.transfer.rep}, sid, qid, b)$, if $\mathcal{E}$ is corrupt, or $(\mathsf{ot.transfer.rep}, sid, qid)$, if $\mathcal{E}$ is honest, from S:

a) Abort if (qid, com_σ) is not stored.

b) If $\mathcal{E}$ is corrupt and $b = 0$, set $v \leftarrow \perp$.

c) Else, set $v \leftarrow m_\sigma$.

d) Delete the record (qid, com_σ).

e) Send $(\mathsf{ot.transfer.end}, sid, v)$ to $\mathcal{R}$.

2.8.6 *Secure Pseudonymous Channel*

We depict an ideal functionality $\mathcal{F}_{\text{NYM}}$ for an idealized secure pseudonymous channel. We use $\mathcal{F}_{\text{NYM}}$ to describe some of our constructions in order to abstract away the details of real-world pseudonymous channels. $\mathcal{F}_{\text{NYM}}$ is similar to the functionality for anonymous secure message transmission in [25]. $\mathcal{F}_{\text{NYM}}$ interacts with senders $\mathcal{T}_k$ and a replier $\mathcal{R}$ and consists of two interfaces nym.send and nym.reply. $\mathcal{T}_k$ uses nym.send to send a message m and a pseudonym P to $\mathcal{R}$. $\mathcal{R}$ uses nym.reply to send a message m and a pseudonym P. $\mathcal{F}_{\text{NYM}}$ checks if there is a party $\mathcal{T}_k$ associated with pseudonym P that is awaiting a reply, and in that case sends m and P to $\mathcal{T}_k$. Therefore, $\mathcal{R}$ replies to messages from $\mathcal{T}_k$ by specifying P. We depict $\mathcal{F}_{\text{NYM}}$ below.

Figure 2.9: Description of $\mathcal{F}_{\text{NYM}}$.

$\mathcal{F}_{\text{NYM}}$ is parameterized by a message space $\mathcal{M}$, a leakage function l, which leaks the message length, and a universe of pseudonyms $\mathbb{U}_p$.

1. On input $(\mathsf{nym.send.ini}, sid, m, P)$ from $\mathcal{T}_k$:
   - Abort if $sid \neq (\mathcal{R}, sid')$, or if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}_k, m)$.
   - Send $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.send.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, P, \mathcal{T}_k, m)$ is not stored.
   - Store $(sid, P, \mathcal{T}_k)$.
   - Delete the record $(qid, P, \mathcal{T}_k, m)$.
   - Parse sid as $(\mathcal{R}, sid')$.
   - Send $(\mathsf{nym.send.end}, sid, m, P)$ to $\mathcal{R}$.

2. On input $(\mathsf{nym.reply.ini}, sid, m, P)$ from $\mathcal{R}$:
   - Abort if $sid \neq (\mathcal{R}, sid')$, or if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
   - Abort if there is not a tuple $(sid, P', \mathcal{T}_k)$ stored such that $P' = P$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}_k, m)$.
   - Delete the tuple $(sid, P, \mathcal{T}_k)$.
   - Send $(\mathsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, P, \mathcal{T}_k, m)$ is not stored.
   - Delete the record $(qid, P, \mathcal{T}_k, m)$.
   - Send $(\mathsf{nym.send.end}, sid, m, P)$ to $\mathcal{T}_k$.

2.8.7 *Public Bulletin Board*

We depict a functionality $\mathcal{F}_{\text{BB}}$ for a public bulletin board BB [97]. A BB is used to ensure that all the readers receive the same version of the database, which is needed to provide unlinkability. $\mathcal{F}_{\text{BB}}$ interacts with a writer $\mathcal{W}$ and readers $\mathcal{R}_k$. $\mathcal{W}$ uses the bb.write interface to send a message m to $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ increments a counter ct of the number of messages stored in BB and appends $[ct, m]$ to BB. $\mathcal{R}_k$ uses the bb.getbb interface on input an index i. If $i \in [1, ct]$, $\mathcal{F}_{\text{BB}}$ takes the message m stored in $[i, m]$ in BB and sends m to $\mathcal{R}_k$. We depict $\mathcal{F}_{\text{BB}}$ below.

Figure 2.10: Description of $\mathcal{F}_{\text{BB}}$.

$\mathcal{F}_{\text{BB}}$ is parameterized by a universe of messages $\mathbb{U}_m$. $\mathcal{F}_{\text{BB}}$ interacts with a writer $\mathcal{W}$ and readers $\mathcal{R}_k$.

1. On input (bb.write.ini, sid, m) from $\mathcal{W}$:
   - Abort if $sid \notin (\mathcal{W}, sid')$.
   - Abort if $m \notin \mathbb{U}_m$.
   - If (sid, BB, ct) is not stored, set BB $\leftarrow \perp$ and $ct \leftarrow 0$.
   - Increment ct, append $[ct, m]$ to BB and update ct and BB in (sid, BB, ct).
   - Create a fresh qid and store qid.
   - Send (bb.write.sim, sid, qid, m) to $\mathcal{S}$.

S. On input (bb.write.rep, sid, qid) from $\mathcal{S}$:
   - Abort if qid is not stored.
   - Delete qid.
   - Send (bb.write.end, sid) to $\mathcal{W}$.

2. On input (bb.getbb.ini, sid, i) from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k, i)$.
   - Send (bb.getbb.sim, sid, qid) to $\mathcal{S}$.

S. On input (bb.getbb.rep, sid, qid) from $\mathcal{S}$:
   - Abort if $(qid', \mathcal{R}_k, i)$ such that $qid' = qid$ is not stored.
   - If (sid, BB, ct) is stored and $i \in [1, ct]$, take $[i, m]$ from BB and set $m' \leftarrow m$, else set $m' \leftarrow \perp$.
   - Send (bb.getbb.end, sid, m') to $\mathcal{R}_k$.

2.8.8 *Interactive Zero-Knowledge Proofs of Knowledge*

We use two versions of $\mathcal{F}_{\text{ZK}}^{R}$, one for protocols that require unlinkability and another for protocols that do not require unlinkability. Both versions follow the functionality for zero-knowledge in [29].

In the version of $\mathcal{F}_{\text{ZK}}^{R}$ for protocols that require unlinkability, the functionality interacts with provers $\mathcal{P}_k$ and a prover is identified by a pseudonym P towards the verifier $\mathcal{V}$. In the version of $\mathcal{F}_{\text{ZK}}^{R}$ for protocols that do not require unlinkability, the functionality interacts with one prover $\mathcal{P}$ and the identifier $\mathcal{P}$ is revealed to $\mathcal{V}$.

Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. $\mathcal{F}_{\text{ZK}}^{R}$ consists of one interface zk.prove. A prover uses zk.prove to send a witness wit and an instance ins to $\mathcal{F}_{\text{ZK}}^{R}$. $\mathcal{F}_{\text{ZK}}^{R}$ checks if $(wit, ins) \in R$, and, in that case, sends ins to $\mathcal{V}$. We depict below the $\mathcal{F}_{\text{ZK}}^{R}$ without unlinkability.

Figure 2.11: Description of $\mathcal{F}_{\text{ZK}}^{R}$ (without unlinkability).

$\mathcal{F}_{\text{ZK}}^{R}$ is parameterized by a description of a relation R. $\mathcal{F}_{\text{ZK}}^{R}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1. On input (zk.prove.ini, sid, wit, ins) from $\mathcal{P}$:
   - Abort if $sid \neq (\mathcal{P}, \mathcal{V}, sid')$ or if $(wit, ins) \notin R$.
   - Create a fresh qid and store (qid, ins).
   - Send (zk.prove.sim, sid, qid, ins) to $\mathcal{S}$.

S. On input (zk.prove.rep, sid, qid) from $\mathcal{S}$:
   - Abort if (qid, ins) is not stored.
   - Parse sid as $(\mathcal{P}, \mathcal{V}, sid')$.
   - Delete the record (qid, ins).
   - Send (zk.prove.end, sid, ins) to $\mathcal{V}$.

In the version for protocols that require unlinkability, the prover also sends a pseudonym P, which is sent by $\mathcal{F}_{\text{ZK}}^{R}$ to $\mathcal{V}$. We depict below $\mathcal{F}_{\text{ZK}}^{R}$ with unlinkability.

Figure 2.12: Description of $\mathcal{F}_{\text{ZK}}^{R}$ (with unlinkability).

$\mathcal{F}_{\text{ZK}}^{R}$ (with unlinkability) is parameterized by a description of a relation R and by a universe of pseudonyms $\mathbb{U}_p$. $\mathcal{F}_{\text{ZK}}^{R}$ interacts with provers $\mathcal{P}_k$ and a verifier $\mathcal{V}$.

1. On input (zk.prove.ini, sid, wit, ins, P) from $\mathcal{P}_k$:
   - Abort if $sid \neq (\mathcal{V}, sid')$, or if $(wit, ins) \notin R$, or if $P \notin \mathbb{U}_p$.
   - Create a fresh qid and store (qid, ins, P).
   - Send (zk.prove.sim, sid, qid, ins) to $\mathcal{S}$.

S. On input (zk.prove.rep, sid, qid) from $\mathcal{S}$:
   - Abort if (qid, ins, P) is not stored.
   - Parse sid as $(\mathcal{V}, sid')$.
   - Delete the record (qid, ins, P).
   - Send (zk.prove.end, sid, ins, P) to the verifier $\mathcal{V}$.

2.8.8.1 *Construction*

To instantiate $\mathcal{F}_{\text{ZK}}^{R}$ (with or without unlinkability), we use the scheme in [24]. In [24], a UC ZK protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ that satisfy the formula $\phi(w_1, \ldots, w_n)$ is described as

$$\text{ℵ}\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n) \tag{2.1}$$

The formula $\phi(w_1, \ldots, w_n)$ consists of conjunctions and disjunctions of "atoms". An atom expresses *group relations*, such as $\prod_{j=1}^{k} g_j^{\mathcal{F}_j} = 1$, where the g_j's are elements of prime order groups and the $\mathcal{F}_j$'s are polynomials in the variables $(w_1, \ldots, w_n)$.

A proof system for (2.1) can be transformed into a proof system for more expressive statements about secret exponents $sexps$ and secret bases $sbases$:

$$\text{ℵ}\, sexps, sbases : \phi(sexps, bases \cup sbases) \tag{2.2}$$

The transformation adds an additional base h to the public bases. For each $g_j \in sbases$, the transformation picks a random exponent ρ_j and computes a blinded base $g'_j = g_j h^{\rho_j}$. The transformation adds g'_j to the public bases $bases$, ρ_j to the secret exponents $sexps$, and rewrites $g_j^{\mathcal{F}_j}$ into $g_j'^{\mathcal{F}_j} h^{-\mathcal{F}_j \rho_j}$.

The proof system supports pairing product equations $\prod_{j=1}^{k} e(g_j, \tilde{g}_j)^{\mathcal{F}_j} = 1$ in groups of prime order with a bilinear map e, by treating the target group $\mathbb{G}_t$ as the group of the proof system. The embedding for secret bases is unchanged, except for the case in which both bases in a pairing are secret. In this case, $e(g_j, \tilde{g}_j)^{\mathcal{F}_j}$ must be transformed into $e(g'_j, \tilde{g}'_j)^{\mathcal{F}_j} e(g'_j, \tilde{h})^{-\mathcal{F}_j \tilde{\rho}_j} e(h, \tilde{g}'_j)^{-\mathcal{F}_j \rho_j} e(h, \tilde{h})^{\mathcal{F}_j \rho_j \tilde{\rho}_j}$.

The instantiation with unlinkability uses a pseudonymous channel for communication between prover and verifier. The one without unlinkability uses an authenticated channel.

2.8.9 *Non-Interactive Zero-Knowledge*

We describe a functionality $\mathcal{F}_{\text{NIZK}}^{R}$ for non-interactive zero-knowledge proofs of knowledge. Functionality $\mathcal{F}_{\text{NIZK}}^{R}$ interacts with any parties $\mathcal{P}$ that obtain the parameters par and compute and verify proofs. The interaction between the functionality $\mathcal{F}_{\text{NIZK}}^{R}$ and these parties takes place through the following interfaces:

1. Any party $\mathcal{P}$ employs the nizk.setup interface to obtain the parameters par.
2. Any party $\mathcal{P}$ employs the nizk.prove interface to obtain a proof π on input a witness wit and an instance ins such that $(wit, ins) \in R$.
3. Any party $\mathcal{P}$ employs the nizk.verify interface to verify a proof π for an instance ins.

$\mathcal{F}_{\text{NIZK}}^{R}$ uses a table Tbl. Tbl consists of entries of the form $[ins, \pi, u]$, where ins is an instance, π is a proof, and u is a bit that indicates whether π is a valid proof for the instance ins or not. $\mathcal{F}_{\text{NIZK}}^{R}$ also uses a set $\mathbb{S}$ that contains all the parties that have obtained the parameters par.

$\mathcal{F}_{\text{NIZK}}^{R}$ is similar to the ideal functionality for non-interactive zero-knowledge in [53]. $\mathcal{F}_{\text{NIZK}}^{R}$ asks the simulator $\mathcal{S}$ to provide simulation and extraction algorithms

at setup. In [53], the functionality asks the simulator to provide simulated proofs and to extract witnesses when the nizk.prove and nizk.verify interfaces are invoked. We choose the first alternative because it hides from the simulator when the parties compute and verify proofs and the proof instances.

The functionality $\mathcal{F}_{\text{NIZK}}^{R}$ does not allow the computation and verification of proofs using parameters par that were not generated by the functionality. As can be seen, the interfaces nizk.prove and nizk.verify do not receive the parameters of the scheme as input and, in order to compute and verify proofs, the functionality employs the parameters that it stores. Therefore, a construction that realizes this functionality must ensure that the honest parties employ the same parameters. To achieve this, some form of trusted setup is required. We depict $\mathcal{F}_{\text{NIZK}}^{R}$ below.

Figure 2.13: Description of $\mathcal{F}_{\text{NIZK}}^{R}$.

NIZK.SimProve and NIZK.Extract are ppt algorithms. $\mathcal{F}_{\text{NIZK}}^{R}$ is parameterized with a description of a relation R. $\mathcal{F}_{\text{NIZK}}^{R}$ interacts with any parties $\mathcal{P}$ that obtain the parameters par and compute and verify proofs.

1. On input (nizk.setup.ini, sid) from any party $\mathcal{P}$:
   - Generate a random qid and store $(qid, \mathcal{P})$.
   - If the tuple $(sid, par, td,$ NIZK.SimProve, NIZK.Extract) is not stored, send the message (nizk.setup.req, $sid, qid, \mathcal{P}$) to $\mathcal{S}$, else send (nizk.setup.sim, $sid, qid, \mathcal{P}$) to $\mathcal{S}$.

S. On input (nizk.setup.alg, $sid, qid, par, td,$ NIZK.SimProve, NIZK.Extract) from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{P})$ is not stored.
   - If $(sid, par, td,$ NIZK.SimProve, NIZK.Extract) is not stored, do the following:
     - Store $(sid, par, td,$ NIZK.SimProve, NIZK.Extract).
     - Create an empty set $\mathbb{S}$.
   - Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{P}$.
   - Delete $(qid, \mathcal{P})$.
   - Send (nizk.setup.end, sid, par) to $\mathcal{P}$.

S. On input (nizk.setup.rep, sid, qid) from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{P})$ is not stored or if $(sid, par, td,$ NIZK.SimProve, NIZK.Extract) is not stored.
   - Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{P}$.
   - Delete $(qid, \mathcal{P})$.
   - Send (nizk.setup.end, sid, par) to $\mathcal{P}$.

2. On input (nizk.prove.ini, sid, wit, ins) from a party $\mathcal{P}$:
   - Abort if $\mathcal{P} \notin \mathbb{S}$ or if $(wit, ins) \notin R$.

- Run $\pi \leftarrow$ NIZK.SimProve(sid, par, td, ins).
- Append $[ins, \pi, 1]$ to Table Tbl.
- Send (nizk.prove.end, sid, π) to $\mathcal{P}$.

3. On input (nizk.verify.ini, sid, ins, π) from a party $\mathcal{P}$:
   - Abort if $\mathcal{P} \notin \mathbb{S}$.
   - If there is an entry $[ins, \pi, u]$ in Tbl, set $v \leftarrow u$.
   - Else, do the following:
     - Extract $wit \leftarrow$ NIZK.Extract(sid, par, td, π, ins).
     - If $(wit, ins) \notin R$, set $v \leftarrow 0$, else set $v \leftarrow 1$.
     - Append $[ins, \pi, v]$ to Tbl.
   - Send (nizk.verify.end, sid, v) to $\mathcal{P}$.

We now discuss the three interfaces of the ideal functionality $\mathcal{F}_{\text{NIZK}}^{R}$.

1. The nizk.setup.ini message is sent by any party $\mathcal{P}$. If the algorithms are not stored, $\mathcal{F}_{\text{NIZK}}^{R}$ asks the simulator $\mathcal{S}$ for the parameters par, the trapdoor td and the algorithms NIZK.SimProve and NIZK.Extract. Otherwise, $\mathcal{F}_{\text{NIZK}}^{R}$ asks the simulator to allow the setup phase to be completed for party $\mathcal{P}$. When the simulator provides the parameters, trapdoor and algorithms, the functionality stores them if they were not stored before. When the simulator provides the algorithms or when it simply prompts the completion of the setup phase for party $\mathcal{P}$, the functionality records that $\mathcal{P}$ obtains the parameters and sends the parameters to $\mathcal{P}$.

2. The nizk.prove.ini message is sent by any honest party $\mathcal{P}$ on input a witness wit and an instance ins. $\mathcal{F}_{\text{NIZK}}^{R}$ aborts if $\mathcal{P}$ did not obtain the parameters. This is required because $\mathcal{F}_{\text{NIZK}}^{R}$ enforces that the computation of a proof is a local process (note that $\mathcal{F}_{\text{NIZK}}^{R}$ does not communicate with the simulator when computing a proof), so parties in the real world must have obtained the parameters before computing a proof. $\mathcal{F}_{\text{NIZK}}^{R}$ runs the algorithm NIZK.SimProve on input par, td and ins to get a simulated proof π. NIZK.SimProve does not receive the witness wit as input, and therefore a proof of knowledge scheme that realizes this functionality must fulfill the zero-knowledge property. $\mathcal{F}_{\text{NIZK}}^{R}$ stores $[ins, \pi, 1]$ in Tbl and sends π to $\mathcal{P}$.

3. The nizk.verify.ini message is sent by any honest party $\mathcal{P}$ on input a proof π and an instance ins. $\mathcal{F}_{\text{NIZK}}^{R}$ aborts if $\mathcal{P}$ did not obtain the parameters because the verification of a proof is enforced to be a local process. If there is an entry $[ins, \pi, u]$ already stored in Tbl, then the functionality returns the bit u. Therefore, any non-interactive zero-knowledge proof of knowledge scheme that realizes this functionality must be consistent. Otherwise, the functionality runs the algorithm NIZK.Extract to get a witness wit such that $(wit, ins) \in R$. Therefore, any scheme that realizes $\mathcal{F}_{\text{NIZK}}^{R}$ must be extractable. If extraction fails, the functionality sets the verification result to 0, i.e., extraction must succeed unless the proof is incorrect. The functionality records the verification result in Tbl and returns this result.

2.8.10 *Non-Interactive Commitments*

Our constructions use the functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments in [20]. As explained in [20], $\mathcal{F}_{\text{NIC}}$ requires a trapdoor, which implies that it provides the hiding security property. $\mathcal{F}_{\text{NIC}}$ interacts with parties $\mathcal{P}_i$ and consists of the following interfaces:

1. Any party $\mathcal{P}_i$ uses the com.setup interface to set up the functionality.
2. Any party $\mathcal{P}_i$ uses the com.commit interface to send a message m and obtain a commitment com and an opening $open$. A commitment $com = (com', parcom, \mathsf{COM.Verify})$, where com' is the commitment, $parcom$ are the public parameters, and COM.Verify is the verification algorithm.
3. Any party $\mathcal{P}_i$ uses the com.validate interface to send a commitment com to the functionality in order to check that com contains the correct public parameters and verification algorithm.
4. Any party $\mathcal{P}_i$ uses the com.verify interface to send $(com, m, open)$ to the functionality in order to verify that com is a commitment to the message m with the opening $open$.

$\mathcal{F}_{\text{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as the Pedersen commitment scheme. [20].

Figure 2.14: Description of $\mathcal{F}_{\text{NIC}}$.

COM.TrapCom, COM.TrapOpen and COM.Verify are ppt algorithms.

1. On input (com.setup.ini, sid) from a party $\mathcal{P}_i$:
   - If $(sid, parcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, tdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$, and send (com.setup.end, sid, OK) as a public delayed output to $\mathcal{P}_i$.
   - Otherwise proceed to generate a random qid, store $(qid, \mathcal{P}_i)$ and send the message (com.setup.req, sid, qid) to $\mathcal{S}$.

S. On input (com.setup.alg, sid, qid, m) from $\mathcal{S}$:
   - Abort if no pair $(qid, \mathcal{P}_i)$ for some $\mathcal{P}_i$ is stored.
   - Delete record $(qid, \mathcal{P}_i)$.
   - If $(sid, parcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, tdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send (com.setup.end, sid, OK) to $\mathcal{P}_i$.
   - Otherwise proceed as follows.
     - m is $(parcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, tdcom)$.
     - Initialize both an empty table Tbl_{com} and an empty set $\mathbb{P}$, and store $(sid, parcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, tdcom)$.

  - Include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send $(\mathsf{com.setup.end}, sid, OK)$ to $\mathcal{P}_i$.

2. On input $(\mathsf{com.validate.ini}, sid, com)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$.
   - Parse com as $(com', parcom', \mathsf{COM.Verify}')$.
   - Set $v \leftarrow 1$ if $parcom' = parcom$ and $\mathsf{COM.Verify}' = \mathsf{COM.Verify}$. Otherwise, set $v \leftarrow 0$.
   - Send $(\mathsf{com.validate.end}, sid, v)$ to $\mathcal{P}_i$.
3. On input $(\mathsf{com.commit.ini}, sid, m)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $m \notin \mathcal{M}$, where $\mathcal{M}$ is defined in $parcom$.
   - Compute $(com, info) \leftarrow \mathsf{COM.TrapCom}(sid, parcom, tdcom)$.
   - Abort if there is an entry $[com, m', open', 1]$ in Tbl_{com} such that $m \neq m'$.
   - Run $open \leftarrow \mathsf{COM.TrapOpen}(sid, m, info)$.
   - Abort if $1 \neq \mathsf{COM.Verify}(sid, parcom, com, m, open)$.
   - Append $[com, m, open, 1]$ to Tbl_{com}.
   - Set $com \leftarrow (com, parcom, \mathsf{COM.Verify})$.
   - Send $(\mathsf{com.commit.end}, sid, com, open)$ to $\mathcal{P}_i$.
4. On input $(\mathsf{com.verify.ini}, sid, com, m, open)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $m \notin \mathcal{M}$ or if $open \notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in $parcom$.
   - Parse com as the tuple $(com', parcom', \mathsf{COM.Verify}')$. Abort if the parameters $parcom' \neq parcom$ or $\mathsf{COM.Verify}' \neq \mathsf{COM.Verify}$.
   - If there is an entry $[com', m, open, u]$ in Tbl_{com}, set $v \leftarrow u$.
   - Else, proceed as follows:
     - If there is an entry $[com', m', open', 1]$ in Tbl_{com} such that $m \neq m'$, set $v \leftarrow 0$.
     - Else, proceed as follows:
       * Set $v \leftarrow \mathsf{COM.Verify}(sid, parcom, com', m, open)$.
       * Append $[com', m, open, v]$ to Tbl_{com}.
   - Send $(\mathsf{com.verify.end}, sid, v)$ to $\mathcal{P}_i$.

2.9 PUTTING IT ALL TOGETHER

When we construct protocols in the hybrid model, situations arise where we must ensure that the same inputs are being sent to two or more ideal functionalities being used as building blocks in a larger protocol. For instance, if we would like to prove facts about a specific secret value, whilst also writing this information to a data structure, for security guarantees to hold we would need to ensure that the same secret information is being sent to both an ideal functionality for a zero knowledge proof, and to another for the data structure. In these cases, we use a technique described in [20]:

1. A party first uses com.commit to get a commitment com to an input value m with opening $open$,
2. The party then sends $(com, m, open)$ as input to each of the functionalities that need to receive m,
3. Each functionality runs COM.Verify to verify the commitment,
4. Other parties in the protocol receive the commitment com from each of the functionalities and use the com.validate interface to validate com.

If com received from all functionalities is the same, the binding property provided by $\mathcal{F}_{\text{NIC}}$ ensures that all functionalities have received the same input m.

Part II

STATEFUL ZERO-KNOWLEDGE DATA STRUCTURES

We introduce formal security definitions for our Stateful Zero Knowledge (SZK) data structures, in the Universal Composability (UC) framework, followed by constructions that securely realize these definitions. For each data structure, we also list the security properties they guarantee, and variants of these definitions that may find use in specific cases when it comes to using these data structures as building blocks in larger privacy preserving protocols.

3

INTRODUCTION

In Part II, we describe six data structures that may be broadly classified based on the security properties they guarantee, as follows:

1. Databases where entries are hidden from the verifier.
    a) Updatable Committed Databases (CDs)
    b) Unlinkable Committed Databases (UCDs)
    c) Non-interactive Committed Databases (NICDs)
2. Databases where entries are hidden from the verifier, but the verifier may update a prover's copy of its data structure.
    a) Unlinkable Updatable Hiding Databases (UUHDs)
3. Databases where entries are visible to both the prover and the verifier.
    a) Updatable Databases (UDs)
    b) Unlinkable Updatable Databases (UUDs)

These data structures come with their own distinct *operations*, which typically allow parties to read, write, update, or prove facts about data.

4

UPDATABLE COMMITTED DATABASES

Work by Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. This primitive was published in a paper presented at the 34th IEEE CSF Symposium in 2019 [21]. Though the author was not involved in the work done towards this paper, it forms a part of the preliminary results for our FNR project and is a component in the protocol in Chapter 11.

A CD is defined as the task of maintaining a database Tbl_{cd} consisting of entries of the form $[i, v]$, where i ranges from 0 to N_{max}, between two parties: a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. i represents a position whilst v represents the value stored at that position in Tbl_{cd}. The initial values of Tbl_{cd} are known to both parties.

4.1 OPERATIONS

WRITE. During a *write* operation, $\mathcal{P}$ updates an entry in the database, whilst hiding the entry being written to and its position from $\mathcal{V}$.

READ. During a *read* operation, $\mathcal{P}$ proves knowledge of an entry in the database, whilst hiding the entry being read and its position from $\mathcal{V}$.

4.2 SECURITY PROPERTIES

ZERO-KNOWLEDGE. Whenever $\mathcal{P}$ performs a read or a write operation against the database, the values and their positions are hidden from $\mathcal{V}$. $\mathcal{V}$ receives hiding commitments to these positions and their values, but the hiding property ensures that $\mathcal{V}$ does not learn this information.

BINDING. $\mathcal{P}$ may only read entries that have been written to the database.

Additionally, $\mathcal{F}_{\text{CD}}$ also guarantees that a value read from Tbl_{cd} at position i is equal to the value previously written to i, and that all database entries are hidden from $\mathcal{V}$.

4.3 IDEAL FUNCTIONALITY

The commitment parameters parcom and the commitment verification algorithm COM.Verify *are contained within* com_i, com_r, *and* com_w.

Figure 4.1: Ideal Functionality $\mathcal{F}_{\text{CD}}$

$\mathcal{F}_{\text{CD}}$ is parameterized by a universe of values U_v and by a maximum database size N_{max}. $\mathcal{F}_{\text{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1. On input $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$ from $\mathcal{V}$:
   - Abort if $sid \notin (\mathcal{P}, \mathcal{V}, sid')$ or if (sid, Tbl_{cd}) is already stored.
   - Abort if Tbl_{cd} does not consist of entries of the form $[i, v]$, or if the number of entries in Tbl_{cd} is not N_{max}.

- Abort if for $i = 1$ to N_{max}, $v \notin U_v$ for any entry $[i, v]$ in Tbl_{cd}.
- Initialize a counter $cv \leftarrow 0$ for the verifier and store (sid, cv) and (sid, Tbl_{cd}).
- Send $(\mathsf{cd.setup.sim}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.setup.rep}, sid)$ from $\mathcal{S}$:

- Abort if (sid, Tbl_{cd}) is not stored, or if $(sid, \mathsf{Tbl}_{cd}, cp)$ is already stored.
- Initialize a counter $cp \leftarrow 0$ for the prover and store $(sid, \mathsf{Tbl}_{cd}, cp)$.
- Send $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{P}$.

2. On input $(\mathsf{cd.read.ini}, sid, com_i, i, open_i, com_r, v_r, open_r)$ from $\mathcal{P}$:

- Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
- Abort if $i \notin [1, N_{max}]$, or if $v_r \notin U_v$, or if $[i, v_r]$ is not stored in Tbl_{cd}.
- Parse the commitment com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
- Parse the commitment com_r as $(com'_r, parcom_r, \mathsf{COM.Verify}_r)$.
- Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_r$ are not ppt algorithms.
- Abort if $1 \neq \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i)$.
- Abort if $1 \neq \mathsf{COM.Verify}_r(parcom_r, com'_r, v_r, open_r)$.
- Create a fresh qid and store (qid, com_i, com_r, cp).
- Send $(\mathsf{cd.read.sim}, sid, qid, com_i, com_r)$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.read.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if (qid, com_i, com_r, cp') is not stored.
- Abort if $cp' \neq cv$, where cv is stored in (sid, cv).
- Delete the record (qid, com_i, com_r, cp').
- Send $(\mathsf{cd.read.end}, sid, com_i, com_r)$ to $\mathcal{V}$.

3. On input $(\mathsf{cd.write.ini}, sid, com_i, i, open_i, com_w, v_w, open_w)$ from $\mathcal{P}$:

- Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
- Abort if $i \notin [1, N_{max}]$, or if $v_w \notin U_v$.
- Parse the commitment com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.

- Parse the commitment com_w as $(com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
- Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_w$ are not ppt algorithms.
- Abort if $1 \neq \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i)$.
- Abort if $1 \neq \mathsf{COM.Verify}_w(parcom_w, com'_w, v_w, open_w)$.
- Increment the counter cp in $(sid, \mathsf{Tbl}_{cd}, cp)$ and store $[i, v_w]$ in Tbl_{cd}.
- Create a fresh qid and store (qid, com_i, com_w, cp).
- Send $(\mathsf{cd.write.sim}, sid, qid, com_i, com_w)$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.write.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if (qid, com_i, com_w, cp') is not stored.
- Abort if $cp' \neq cv + 1$, where cv is stored in (sid, cv).
- Increment the counter cv in (sid, cv).
- Delete the record (qid, com_i, com_w, cp').
- Send $(\mathsf{cd.write.end}, sid, com_i, com_w)$ to $\mathcal{V}$.

$\mathcal{F}_{\text{CD}}$ features three interfaces:

1. The *setup* interface cd.setup allows $\mathcal{V}$ to initialize Tbl_{cd} to values that are known to both $\mathcal{V}$ and $\mathcal{P}$. $\mathcal{F}_{\text{CD}}$ initializes two counters cp and cv in order to count the number of write operations sent by $\mathcal{P}$ and those received by $\mathcal{V}$ respectively. This allows $\mathcal{F}_{\text{CD}}$ to enforce the fact that both parties are using the same version of the Tbl_{cd}. $\mathcal{F}_{\text{CD}}$ stores Tbl_{cd}, sends Tbl_{cd} to the simulator $\mathcal{S}$, and outputs Tbl_{cd} to $\mathcal{P}$.

2. The *read* interface cd.read allows $\mathcal{P}$ to perform a read operation. $\mathcal{P}$ sends a database position i and the value to be read at that position v_r, and commitments and openings to both values $(com_i, open_i, com_r, open_r)$ as input to $\mathcal{F}_{\text{CD}}$ via cd.read. If $[i, v_r]$ is a valid entry in the database, $\mathcal{F}_{\text{CD}}$ verifies both commitments and sends them to $\mathcal{S}$. These commitments are then passed as output to $\mathcal{V}$.

3. The *write* interface cd.write allows $\mathcal{P}$ to perform a write operation. $\mathcal{P}$ sends a database position i and the value to be written to that position v_w, and commitments and openings to both values $(com_i, open_i, com_w, open_w)$ as input to $\mathcal{F}_{\text{CD}}$ via cd.write. $\mathcal{F}_{\text{CD}}$ verifies both commitments and sends them to $\mathcal{S}$, after updating the value stored at position i_w with v_w in Tbl_{cd}. The commitments are then passed as output to $\mathcal{V}$.

The fact that $\mathcal{F}_{\text{CD}}$ accepts commitments to the positions and data read from and written to the database via the cd.read *and* cd.write *interfaces, and passes them as output to $\mathcal{V}$, facilitates modular protocol design, as explained in Section 2.9.*

4.4 CONSTRUCTION

Π_{CD} uses a hiding vector commitment (VC) scheme (see Section 2.7.1). A vector commitment vc is used to store the database Tbl_{cd}. A position in the vector com-

mitment acts as a position in Tbl_{cd}, and the value committed to in that position acts as the value stored in Tbl_{cd} in that position.

Figure 4.2: Construction Π_{CD}

Π_{CD} uses a hiding vector commitment scheme and the ideal functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_w}$. The database size N_{max} is the maximum length of a committed vector, and the universe of values U_v is given by the message space of the VC scheme.

1. On input $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$, $\mathcal{V}$ and $\mathcal{P}$ do the following:
   - $\mathcal{V}$ uses the $\mathsf{crs.get}$ interface of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ to obtain the VC parameters par.
   - $\mathcal{V}$ initializes a counter $cv \leftarrow 0$ that counts the write operations received.
   - $\mathcal{V}$ stores Tbl_{cd} in a vector $\mathbf{x}$: for $i = 1$ to N_{max}, $\mathbf{x}[i] = v$, where $[i, v] \in \mathsf{Tbl}_{cd}$.
   - $\mathcal{V}$ commits to $\mathbf{x}$: set $r \leftarrow 0$ and run $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$.
   - $\mathcal{V}$ stores (sid, cv, par, vc).
   - $\mathcal{V}$ uses the $\mathsf{aut.send}$ interface of $\mathcal{F}_{\mathrm{AUT}}$ to send Tbl_{cd} to $\mathcal{P}$.
   - $\mathcal{P}$ follows the same steps as $\mathcal{V}$ to get par, set $\mathbf{x}$ and compute vc.
   - $\mathcal{P}$ initializes a counter $cp \leftarrow 0$ that counts write operations started.
   - $\mathcal{P}$ stores $(sid, cp, par, vc, \mathbf{x}, r)$.
   - $\mathcal{P}$ outputs $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$.
2. On input $(\mathsf{cd.read.ini}, sid, com_i, i, open_i, com_r, v_r, open_r)$, $\mathcal{P}$ and $\mathcal{V}$ do the following:
   - $\mathcal{P}$ parses com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{P}$ parses com_r as $(com'_r, parcom_r, \mathsf{COM.Verify}_r)$.
   - $\mathcal{P}$ takes the stored tuple $(sid, cp, par, vc, \mathbf{x}, r)$.
   - If (sid, i, w) is not stored, $\mathcal{P}$ computes a VC witness w for position i: run $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r)$ and store (sid, i, w).
   - $\mathcal{P}$ sets $wit_r \leftarrow (w, i, open_i, v_r, open_r)$.
   - $\mathcal{P}$ sets $ins_r \leftarrow (par, vc, parcom_i, com'_i, parcom_r, com'_r, cp)$.

R_r is

$$
\begin{aligned}
&R_r = \{(wit_r, ins_r) : \\
&1 = \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i) \wedge && (4.1)\\
&1 = \mathsf{COM.Verify}_r(parcom_r, com'_r, v_r, open_r) \wedge && (4.2)\\
&1 = \mathsf{VC.Verify}(par, vc, v_r, i, w)\} && (4.3)
\end{aligned}
$$

In equation 4.1, $\mathcal{P}$ proves that com'_i is a commitment to i with opening $open_i$. Similarly, in equation 4.2, $\mathcal{P}$ proves that com'_r is a commitment to v_r with opening $open_r$. In equation 4.3, $\mathcal{P}$ proves that v_r is stored in the position i of the vector commitment vc.

- $\mathcal{P}$ uses the zk.prove interface to send wit_r and ins_r to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$.
- $\mathcal{V}$ receives $ins_r = (par', vc', parcom_i, com'_i, parcom_r, com'_r, cp)$.
- $\mathcal{V}$ takes the stored tuple (sid, cv, par, vc).
- $\mathcal{V}$ aborts if $cp \neq cv$, or if $par' \neq par$, or if $vc' \neq vc$.
- $\mathcal{V}$ sets $com_i \leftarrow (com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
- $\mathcal{V}$ sets $com_r \leftarrow (com'_r, parcom_r, \mathsf{COM.Verify}_r)$.
- $\mathcal{V}$ outputs $(\mathsf{cd.read.end}, sid, com_i, com_r)$.

3. On input $(\mathsf{cd.write.ini}, sid, com_i, i, open_i, com_w, v_w, open_w)$, $\mathcal{P}$ and $\mathcal{V}$ do the following:
   - $\mathcal{P}$ takes the stored tuple $(sid, cp, par, vc, \mathbf{x}, r)$.
   - $\mathcal{P}$ parses com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{P}$ parses com_w as $(com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
   - If (sid, i, w) is not stored, $\mathcal{P}$ computes a VC witness w for position i: run $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r)$ and store (sid, i, w).
   - $\mathcal{P}$ updates the vector commitment vc to vc': pick random $r' \leftarrow \mathcal{R}$ and run $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc, i, v_r, r, v_w, r')$, where $v_r \leftarrow \mathbf{x}[i]$.
   - $\mathcal{P}$ increments the counter of write operations started $cp' \leftarrow cp + 1$.
   - $\mathcal{P}$ sets $wit_w \leftarrow (w, i, open_i, v_r, v_w, open_w, r, r')$.
   - $\mathcal{P}$ sets $ins_w \leftarrow (par, vc, vc', parcom_i, com'_i, parcom_w, com'_w, cp')$.
   - $\mathcal{P}$ updates the vector $\mathbf{x}$ to $\mathbf{x}'$: set $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$.
   - $\mathcal{P}$ updates the stored tuple to $(sid, cp', par, vc', \mathbf{x}', r')$.
   - $\mathcal{P}$ updates the stored witnesses: for $j = 1$ to N_{max}, if (sid, j, w) is stored, run $w' \leftarrow \mathsf{VC.WitUpd}(par, w, j, i, x, r, x', r')$ and update to (sid, j, w').

R_w is

$$R_w = \{(wit_w, ins_w) :$$
$$1 = \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i) \wedge \quad (4.4)$$
$$1 = \mathsf{COM.Verify}_w(parcom_w, com'_w, v_w, open_w) \wedge \quad (4.5)$$
$$1 = \mathsf{VC.VerComUpd}(par, vc, vc', w, i, v_r, r, v_w, r')\} \quad (4.6)$$

In equation 4.4 and equation 4.5, the prover proves that com'_i and com'_w are commitments to i and v_w respectively. In equation 4.6, the prover proves that v_r is stored in the position i in the vector commitment vc, and that vc' is a vector commitment that stores the same values as vc, except that it stores v_w in the position i and that its random value is r' instead of r.

- $\mathcal{P}$ uses the zk.prove interface to send wit_w and ins_w to $\mathcal{F}_{\mathrm{ZK}}^{R_w}$.
- $\mathcal{V}$ gets $ins_w = (\hat{par}, \hat{vc}, vc', parcom_i, com'_i, parcom_w, com'_w, cp)$.
- $\mathcal{V}$ takes the stored tuple (sid, cv, par, vc).
- $\mathcal{V}$ aborts if $cp \neq cv + 1$, or if $\hat{par} \neq par$, or if $\hat{vc} \neq vc$.
- $\mathcal{V}$ sets $cv' \leftarrow cv + 1$ and updates the stored tuple to (sid, cv', par, vc').
- $\mathcal{V}$ sets $com_i \leftarrow (com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
- $\mathcal{V}$ sets $com_w \leftarrow (com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
- $\mathcal{V}$ outputs (cd.write.end, sid, com_i, com_w).

1. In the *setup* interface cd.setup, $\mathcal{P}$ and $\mathcal{V}$ use the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ to generate parameters for the VC scheme par. $\mathcal{V}$ receives as input a table Tbl_{cd} and computes a commitment vc to Tbl_{cd} with 0 randomness. $\mathcal{V}$ then sets up a counter cv and sends a copy of the initial state of Tbl_{cd} to $\mathcal{P}$ via the functionality $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{P}$ sets up a counter cp, and both parties generate a vector commitment vc to Tbl_{cd} (with randomness set to 0).

2. In the *read* interface cd.read, $\mathcal{P}$ receives a position i and the value stored at that position in Tbl_{cd} v_r with commitments and openings $(com_i, open_i)$ and $(com_r, open_r)$ to i and to $valr$ respectively, as input. $\mathcal{P}$ uses the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ to prove to $\mathcal{V}$ that com_i and com_r commit to i and vr_i such that v_r is the message committed at position i in the commitment vc. This proves that $[i, v_r] \in \mathsf{Tbl}_{cd}$.

3. In the *write* interface cd.write, $\mathcal{P}$ receives a position i and the value to be written to this position in Tbl_{cd}, v_w, with commitments and openings $(com_i, open_i)$ and $(com_w, open_w)$ to i and to v_w respectively, as input. $\mathcal{P}$ updates the commitment vc to a commitment vc' that commits to v_w at position i, while all other entries remain unchanged. $\mathcal{P}$ uses the functionality $\mathcal{F}_{\mathrm{ZK}}^{R_w}$ to prove to $\mathcal{V}$ that com_i and com_w commit to i and to v_w, and that vc' is an

update of vc where v_w is committed to, at position i in Tbl_{cd}. We note that vc' contains randomness chosen by $\mathcal{P}$ and thus, after the first execution of the write interface, the committed table will be hidden from $\mathcal{V}$.

4.5 SECURITY ANALYSIS

Theorem 4.5.1 *Π_{CD} securely realizes $\mathcal{F}_{\mathrm{CD}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_w}$-hybrid model if the VC scheme is hiding and binding.*

The proof of Theorem 4.5.1 is described in [21].

4.6 INSTANTIATION AND EFFICIENCY ANALYSIS

We refer the reader to [21] for instantiation details and a discussion on computational costs.

4.7 VARIANTS

- The setup interface cd.setup may be modified to allow $\mathcal{P}$ to initialize a database instead of $\mathcal{V}$.
- The read and write interfaces cd.read and cd.write may be modified to allow $\mathcal{P}$ to read and write multiple database entries simultaneously, by receiving tuples of the form $(i, v_i, com_i, open_i, com_{r,i}, open_{r,i})_{\forall i \in \mathbb{S}}$ $(\mathbb{S} \subseteq [1, N_{max}])$ as input. This would allow $\mathcal{P}$ to read and write more than one database entry at a time whilst reducing communication rounds.
- The database Tbl_{cd} may be modified to contain entries of the form $[i, v_{i,1}, \ldots, v_{i,m}]$, i.e., a database where a tuple of values is stored in each entry. In the cd.write and cd.read interfaces, $\mathcal{P}$ sends $(i, v_{i,1}, \ldots, v_{i,m})$ to $\mathcal{F}_{\mathrm{CD}}$, along with commitments and openings to the position and values of the entry read or written. The position $j \in [1, m]$ of each value read or written is not hidden from $\mathcal{V}$. This variant of $\mathcal{F}_{\mathrm{CD}}$ is useful for protocols where a party needs to read a tuple of values and prove that they are stored in the same entry and that each value is stored at a certain position j within the entry.

5 UNLINKABLE UPDATABLE COMMITTED DATABASES

Joint work with Alfredo Rial. This primitive was described in a public deliverable submitted to the FNR. [36].

$\mathcal{F}_{\text{UCD}}$ runs between multiple provers $\mathcal{P}_k$ and a verifier $\mathcal{V}$, and holds one database per prover DB consisting of entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$, where i ranges from 0 to N. The initial values of DB are known to all parties. $\mathcal{F}_{\text{UCD}}$ also allows provers to use a pseudonym to identify themselves, allowing for unlinkability between database operations. i represents a position whilst $(v_{i,1}, \ldots, v_{i,L})$ represents the values stored at that position in DB.

5.1 OPERATIONS

WRITE. During a *write* operation, $\mathcal{P}_k$ updates an entry in its database, whilst hiding the entry being written to and its position from $\mathcal{V}$.

READ. During a *read* operation, $\mathcal{P}_k$ proves knowledge of an entry in its database, whilst hiding the entry being read and its position from $\mathcal{V}$.

5.2 SECURITY PROPERTIES

UNLINKABILITY. $\mathcal{P}_k$ uses a unique pseudonym Y to identify itself towards $\mathcal{F}_{\text{UCD}}$ whilst interacting with the ucd.read and ucd.write interfaces;this allows for unlinkability of operations.

ZERO-KNOWLEDGE. Whenever $\mathcal{P}_k$ performs a read or a write operation against the database, the values and their positions are hidden from $\mathcal{V}$. $\mathcal{V}$ receives hiding commitments to these positions and their values, but the hiding property ensures that $\mathcal{V}$ does not learn this information.

BINDING. $\mathcal{P}_k$ may only read entries that have been written to the database.

5.3 IDEAL FUNCTIONALITY

Figure 5.1: Ideal Functionality $\mathcal{F}_{\text{UCD}}$

$\mathcal{F}_{\text{UCD}}$ is parameterized by a database size N, an entry size L, an initial database DB′, a universe of pseudonyms $\mathbb{U}_y$, and a universe of values $\mathbb{U}_v$.

1. On input $(\text{ucd.read.ini}, sid, Y, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]}))$ from $\mathcal{P}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $Y \notin \mathbb{U}_y$.

- Abort if there is a tuple $(sid,\ Y', \mathcal{P}'_k, \mathsf{flag})$ stored such that $Y' = Y$ and $\mathcal{P}'_k \neq \mathcal{P}_k$, or such that $\mathcal{P}'_k = \mathcal{P}_k$ and $\mathsf{flag} \neq 1$.
- If a tuple $(sid, \mathcal{P}_k, \mathsf{DB})$ is not stored, set $\mathsf{setup} \leftarrow 1$, take the initial database DB' and store $(sid, \mathcal{P}_k, \mathsf{DB}')$. Else, set $\mathsf{setup} \leftarrow 0$.
- Abort if $[i, v_{i,1}, \ldots, v_{i,L}] \notin \mathsf{DB}$.
- Parse the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
- For all $j \in [1, L]$:
  - Parse the commitment $com_{i,j}$ as $(com'_{i,j}, parcom, \mathsf{COM.Verify})$.
  - Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j})$.
- Set $\mathsf{flag} \leftarrow 0$ and store $(sid,\ Y, \mathcal{P}_k, \mathsf{flag})$.
- Create a fresh qid and store $(qid,\ Y, \mathsf{setup}, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$.
- Send $(\mathsf{ucd.read.sim}, sid, qid, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{S}$.

S. On input $(\mathsf{ucd.read.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid',\ Y, \mathsf{setup}, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ such that $qid = qid'$ is not stored.
- Delete the tuple $(qid',\ Y, \mathsf{setup}, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$.
- Send $(\mathsf{ucd.read.end}, sid,\ Y, \mathsf{setup}, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{V}$.

2. On input $(\mathsf{ucd.write.ini}, sid,\ Y, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j\in[1,L]}))$ from $\mathcal{P}_k$:

- Abort if $sid \notin (\mathcal{V}, sid')$, or if $Y \notin \mathbb{U}_y$.
- Abort if there is a tuple $(sid,\ Y', \mathcal{P}'_k, \mathsf{flag})$ stored such that $Y' = Y$ and $\mathcal{P}'_k \neq \mathcal{P}_k$, or such that $\mathcal{P}'_k = \mathcal{P}_k$ and $\mathsf{flag} \neq 1$.
- If a tuple $(sid, \mathcal{P}_k, \mathsf{DB})$ is not stored, set $\mathsf{setup} \leftarrow 1$, take the initial database DB' and store $(sid, \mathcal{P}_k, \mathsf{DB}')$. Else, set $\mathsf{setup} \leftarrow 0$.
- Abort if $i \notin [1, N]$, or if, for $j = 1$ to L, $v_{i,j} \notin \mathbb{U}_v$.
- Parse the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
- For all $j \in [1, L]$:

  - Parse the commitment $com_{i,j}$ as $(com'_{i,j}, parcom,$ COM.Verify).
  - Abort if $1 \neq$ COM.Verify$(parcom, com'_{i,j}, v_{i,j}, open_{i,j})$.
- Set flag $\leftarrow 0$ and store $(sid, Y, \mathcal{P}_k,$ flag$)$.
- Write $[v_{i,1}, \ldots, v_{i,L}]$ into the entry i in DB stored in $(sid, \mathcal{P}_k,$ DB$)$.
- Create a fresh qid and store $(qid, Y,$ setup$, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$.
- Send (ucd.write.sim, sid, qid, $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$) to $\mathcal{S}$.

S. On input (ucd.write.rep, sid, qid) from $\mathcal{S}$:

- Abort if $(qid', Y,$ setup$, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$ such that $qid = qid'$ is not stored.
- Delete the tuple $(qid', Y,$ setup$, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$.
- Send (ucd.write.end, sid, Y, setup$, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$ to $\mathcal{V}$.

3. On input (ucd.accept.ini, sid, Y) from $\mathcal{V}$:

- Abort if a tuple $(sid, Y', \mathcal{P}_k,$ flag$)$ such that $Y' = Y$ and flag $= 0$ is not stored.
- Update flag $\leftarrow 1$ in the tuple $(sid, Y', \mathcal{P}_k,$ flag$)$ with $Y' = Y$.
- Create a fresh qid and store $(qid, \mathcal{P}_k, Y)$.
- Send (ucd.accept.sim, sid, qid) to $\mathcal{S}$.

S. On input (ucd.accept.rep, sid, qid) from $\mathcal{S}$:

- Abort if a tuple $(qid', \mathcal{P}_k, Y)$ such that $qid = qid'$ is not stored.
- Delete the record $(qid', \mathcal{P}_k, Y)$ such that $qid = qid'$.
- Send (ucd.accept.end, sid, Y) to $\mathcal{P}_k$.

$\mathcal{F}_{\text{UCD}}$ features three interfaces:

1. The *read* interface ucd.read allows $\mathcal{P}_k$ to perform a read operation. $\mathcal{P}_k$ sends a database entry $(i, \langle v_{i,j} \rangle_{\forall j \in [1,L]})$, and commitments and openings to the database position i and the values in the entry $(com_i, open_i)$ and $(\langle com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$ respectively, as input to $\mathcal{F}_{\text{UCD}}$ via ucd.read. If $(i, \langle v_{i,j} \rangle_{\forall j \in [1,L]})$ is a valid entry in the database, $\mathcal{F}_{\text{CD}}$ verifies all commitments and sends them to $\mathcal{S}$. These commitments are then passed as output to $\mathcal{V}$. $\mathcal{P}_k$ also uses a pseudonym Y to identify itself towards $\mathcal{F}_{\text{UCD}}$.

2. The *write* interface ucd.write allows $\mathcal{P}_k$ to perform a write operation. $\mathcal{P}_k$ sends a database entry $(i, \langle v_{i,j} \rangle_{\forall j \in [1,L]})$, and commitments and openings to the database position i and the values in the entry $(com_i, open_i)$ and $(\langle com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$ respectively, as input to $\mathcal{F}_{\text{UCD}}$ via ucd.read. $\mathcal{F}_{\text{UCD}}$ verifies all commitments and sends them to $\mathcal{S}$, after updating the value stored at position i with the new database entry in DB. The commitments are then passed as output to $\mathcal{V}$. $\mathcal{P}_k$ also uses a pseudonym Y to identify itself towards $\mathcal{F}_{\text{UCD}}$.

3. The *accept* interface ucd.accept allows $\mathcal{V}$ to notify $\mathcal{P}_k$, identified by a pseudonym Y, of the fact that a read or a write operation has been registered successfully, and of that fact that $\mathcal{F}_{\text{UCD}}$ is now ready to receive another operation. Hence, every read or write operation must be followed by the execution of the ucd.accept interface.

The setup phase has been incorporated into the read and the write interfaces: when $\mathcal{P}_k$ executes one of these interfaces for the first time, the value of the variable *setup* is set to 1. $\mathcal{F}_{\text{UCD}}$ is also parameterized by an initial database DB. This allows $\mathcal{V}$ to ensure that all parties are using the right initial database.

5.4 VARIANTS

- The read and write interfaces ucd.read and ucd.write may be modified to allow $\mathcal{P}_k$ to read and write multiple database entries simultaneously, by receiving tuples of the form $(i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})_{\forall i \in \mathbb{S}}$ ($\mathbb{S} \subseteq [1, N]$) as input. This would allow $\mathcal{P}_k$ to read and write more than one database entry at a time whilst reducing communication rounds.

6 NON-INTERACTIVE COMMITTED DATABASES

Joint work with Alfredo Rial. This primitive was described in a public deliverable submitted to the FNR. [36].

A Non-Interactive Committed Database allows a prover $\mathcal{P}$ to maintain databases Tbl_{ct} consisting of entries of the form $[i, v]$, where i ranges from 0 to N_{max} and ct represents the number of times the database has been written to, and interact with multiple verifiers $\mathcal{V}$. $\mathcal{P}$ may then perform read operations, which involve proving knowledge of a position and its associated value in Tbl_{ct}, or write operations, where $\mathcal{P}$ proves that an entry in Tbl_{ct-1} has been updated to produce Tbl_{ct}. i represents a position whilst v represents the value stored at that position in Tbl_{ct}.

6.1 OPERATIONS

WRITE. During a *write* operation, $\mathcal{P}$ updates an entry in the most recent version of a database to produce a new one, whilst hiding the entry being written to and its position from $\mathcal{V}$.

READ. During a *read* operation, $\mathcal{P}$ proves knowledge of an entry in the database, whilst hiding the entry being read and its position from $\mathcal{V}$.

6.2 SECURITY PROPERTIES

ZERO-KNOWLEDGE. Whenever $\mathcal{P}$ performs a read or a write operation against a database, the values and their positions are hidden from $\mathcal{V}$. $\mathcal{V}$ receives hiding commitments to these positions and their values, but the hiding property ensures that $\mathcal{V}$ does not learn this information. The proofs computed by the algorithm $\mathsf{NICD.SimProve}$ do not receive as input the position i or the value v_1 of the database entry read, and thus they do not contain any information on $[i, v_1]$.

BINDING. $\mathcal{P}$ may only read entries that have been written to the database.

CONSISTENCY. All the verifiers verify read and write operations with respect to the same database, i.e., after ct write operations, the prover cannot produce different versions Tbl_{ct} and Tbl'_{ct} of the database for different verifiers.

6.3 IDEAL FUNCTIONALITY

The interaction between the functionality $\mathcal{F}_{\text{NICD}}$, the prover $\mathcal{P}$ and the verifiers $\mathcal{V}$ takes place through the following interfaces:

1. The prover $\mathcal{P}$ and the verifiers $\mathcal{V}$ use the $\mathsf{nicd.setup}$ interface to obtain parameters.

2. The prover $\mathcal{P}$ uses the nicd.write interface to prove that two commitments com and com_2 commit to a position i and to a value v_2 such that the entry for position i in Tbl_{ct-1} is updated to contain v_2 in Tbl_{ct}. Tbl_{ct-1} is the last database stored by $\mathcal{F}_{\text{NICD}}$ before Tbl_{ct}, i.e., the prover can only modify the last version of the database.

3. A verifier $\mathcal{V}$ uses the nicd.writevf interface to verify that, on input a counter value ct, two commitments com and com_2 commit to the position and the value that were updated in Tbl_{ct} with respect to Tbl_{ct-1}.

4. The prover $\mathcal{P}$ uses the nicd.read interface to obtain a proof π that two commitments com and com_1 commit to a position i and to a value v_1 such that there exists an entry $[i, v_1]$ in a table Tbl_{ct}.

5. A verifier $\mathcal{V}$ uses the nicd.readvf interface to verify a proof π on input a counter value ct and two commitments com and com_1 to a position and a value.

$\mathcal{F}_{\text{NICD}}$ uses a table Tbl_r. Tbl_r consists of entries of the form $[com, com_1, \pi, ct, u]$, where π is a proof, ct is a counter value, and u is a bit that indicates whether π is a valid proof that the commitments com and com_1 commit to a position i and to a value v_1 such that there exists an entry $[i, v_1]$ in Tbl_{ct}.

$\mathcal{F}_{\text{NICD}}$ uses a set $\mathbb{S}$ that contains all the parties that have obtained parameters. Additionally, $\mathcal{F}_{\text{NICD}}$ uses sets $\mathbb{S}_{ct}$ that contain all the verifiers that have verified a write operation for commitments com and com_2 and counter value ct. We depict $\mathcal{F}_{\text{NICD}}$ below.

Figure 6.1: Ideal Functionality $\mathcal{F}_{\text{NICD}}$

NICD.SimProve and NICD.Extract are ppt algorithms. $\mathcal{F}_{\text{NICD}}$ is parameterized by a universe of values U_v and by a database size N_{max}. $\mathcal{F}_{\text{NICD}}$ interacts with a prover $\mathcal{P}$ and verifiers $\mathcal{V}$.

1. On input (nicd.setup.ini, sid) from any party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{P}, sid')$, where sid is the identifier of the prover.
   - Generate a random qid and store $(qid, \mathcal{T})$.
   - If $(sid, par, td,$ NICD.SimProve, NICD.Extract$)$ is not stored, send (nicd.setup.req, $sid, qid, \mathcal{T}$) to $\mathcal{S}$, else send (nicd.setup.sim, $sid, qid, \mathcal{T}$) to $\mathcal{S}$.

S. On input (nicd.setup.alg, $sid, qid, par, td,$ NICD.SimProve, NICD.Extract) from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{T})$ is not stored.
   - If $(sid, par, td,$ NICD.SimProve, NICD.Extract$)$ is not stored, do the following.
     - Store $(sid, par, td,$ NICD.SimProve, NICD.Extract$)$.
     - Create an empty table Tbl_r.

  - Set $ct \leftarrow 0$, create a table Tbl_{ct} with entries $[i, \perp]$ for $i = 1$ to N_{max}, create a set $\mathbb{S}_{ct}$ that contains all the identities of verifiers, set $info_{ct} \leftarrow \perp$, $com \leftarrow \perp$ and $com_2 \leftarrow \perp$, and store $(sid, ct, \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com, com_2)$.
  - Create an empty set $\mathbb{S}$.
- Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{T}$.
- Delete $(qid, \mathcal{T})$.
- Send $(\mathsf{nicd.setup.end}, sid, par)$ to $\mathcal{T}$.

S. On input $(\mathsf{nicd.setup.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid, \mathcal{T})$ is not stored or if $(sid, par, td, \mathsf{NICD.SimProve}, \mathsf{NICD.Extract})$ is not stored.
- Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{T}$.
- Delete $(qid, \mathcal{T})$.
- Send $(\mathsf{nicd.setup.end}, sid, par)$ to $\mathcal{T}$.

2. On input $(\mathsf{nicd.write.ini}, sid, com, i, open, com_2, v_2, open_2)$ from the prover $\mathcal{P}$:

- Abort if $\mathcal{P} \notin \mathbb{S}$, or if $i \notin [1, N_{max}]$, or if $v_2 \notin U_v$.
- Parse the commitment com as $(com', parcom, \mathsf{COM.Verify})$ and the commitment com_2 as $(com'_2, parcom_2, \mathsf{COM.Verify}_2)$.
- Abort if $parcom \neq parcom_2$, or if $\mathsf{COM.Verify} \neq \mathsf{COM.Verify}_2$, or if $\mathsf{COM.Verify}$ is not a ppt algorithm.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com', i, open)$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_2, v_2, open_2)$.
- Take the stored tuple $(sid, ct, \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com, com_2)$ with the highest value of ct.
- Abort if there is a tuple $(sid, ct', com, i, com_2, v_2)$ such that $ct' = ct + 1$.
- Set $ct' \leftarrow ct + 1$ and store $(sid, ct', com, i, com_2, v_2)$.
- Send $(\mathsf{nicd.write.sim}, sid, ct', com, com_2)$ to $\mathcal{S}$.

S. On input $(\mathsf{nicd.write.rep}, sid, ct, info_{ct})$ from $\mathcal{S}$:

- Abort if $(sid, ct', com, i, com_2, v_2)$ such that $ct = ct'$ is not stored or if $(sid, ct, \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com, com_2)$ is already stored.
- Set $\mathsf{Tbl}_{ct} \leftarrow \mathsf{Tbl}_{ct-1}$, write $[i, v_2]$ into Tbl_{ct}, set $\mathbb{S}_{ct} \leftarrow \emptyset$ and store $(sid, ct, \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com, com_2)$.
- Send $(\mathsf{nicd.write.end}, sid, ct)$ to $\mathcal{P}$.

3. On input $(\mathsf{nicd.writevf.ini}, sid, ct, com, com_2)$ from $\mathcal{V}$:

- Abort if $\mathcal{V} \notin \mathbb{S}$ or if $\mathcal{V} \notin \mathbb{S}_{ct-1}$.
- If $(sid, ct', \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com', com'_2)$ such that $ct' = ct$ is not stored, set $v \leftarrow \perp$. Else, if $com \leftarrow com'$ and $com_2 \leftarrow com'_2$, set $v \leftarrow 1$, else set $v \leftarrow 0$.
- Generate random qid and store $(qid, ct, \mathcal{V}, v)$.
- Send $(\mathsf{nicd.writevf.sim}, sid, qid, ct)$ to $\mathcal{S}$.

S. On input $(\mathsf{nicd.writevf.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid, ct, \mathcal{V}, v)$ is not stored.
- If $v = 1$, include $\mathcal{V}$ in the set $\mathbb{S}_{ct}$ stored in the tuple $(sid, ct, \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, com, com_2)$.
- Delete record $(qid, ct, \mathcal{V}, v)$.
- Send $(\mathsf{nicd.writevf.end}, sid, ct, v)$ to $\mathcal{V}$.

4. On input $(\mathsf{nicd.read.ini}, sid, ct, com, i, open, com_1, v_1, open_1)$ from the prover $\mathcal{P}$:

- Abort if $\mathcal{P} \notin \mathbb{S}$ or if $(sid, ct', \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, \bar{com}, \bar{com}_2)$ such that $ct = ct'$ is not stored.
- Abort if $i \notin [1, N_{max}]$, or if $v_1 \notin U_v$, or if $[i, v_1]$ is not stored in Tbl_{ct}.
- Parse the commitment com as $(com', parcom, \mathsf{COM.Verify})$ and the commitment com_1 as $(com'_1, parcom_1, \mathsf{COM.Verify}_1)$.
- Abort if $parcom \neq parcom_1$, or if $\mathsf{COM.Verify} \neq \mathsf{COM.Verify}_1$, or if $\mathsf{COM.Verify}$ is not a ppt algorithm.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com', i, open)$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_1, v_1, open_1)$.
- Run $\pi \leftarrow \mathsf{NICD.SimProve}(sid, par, td, parcom, com', com'_1, info_{ct}, ct)$.
- Append $[com, com_1, \pi, ct, u]$ to Table Tbl_r.
- Send $(\mathsf{nicd.read.end}, sid, \pi)$ to $\mathcal{P}$.

5. On input $(\mathsf{nicd.readvf.ini}, sid, com, com_1, ct, \pi)$ from a verifier $\mathcal{V}$:

- Abort if $(sid, ct', \mathsf{Tbl}_{ct}, \mathbb{S}_{ct}, info_{ct}, \bar{com}, \bar{com}_2)$ is not stored or if $\mathcal{V} \notin \mathbb{S}_{ct}$.
- If there is an entry $[com, com_1, \pi, ct, u]$ in Tbl_r, set $v \leftarrow u$.
- Else, do the following:
  - Parse the commitment com as $(com', parcom, \mathsf{COM.Verify})$ and the commitment com_1 as $(com'_1, parcom_1, \mathsf{COM.Verify}_1)$.

  - Extract $(i, v_1) \leftarrow$ NICD.Extract$(sid, par, td, \pi, parcom, com', com'_1, info_{ct}, ct)$.
  - If $[i, v_1] \notin \mathsf{Tbl}_{ct}$, set $v \leftarrow 0$, else set $v \leftarrow 1$.
  - Append $[com, com_1, \pi, ct, v]$ to Tbl_r.
- Send (nicd.readvf.end, sid, v) to $\mathcal{V}$.

For the interfaces nicd.read and nicd.readvf, $\mathcal{F}_{\text{NICD}}$ follows the same approach used in $\mathcal{F}^R_{\text{NIZK}}$ in Section 2.8.9. $\mathcal{F}_{\text{NICD}}$ runs algorithms NICD.SimProve and NICD.Extract in order to compute read proofs and extract from read proofs.

However, for interfaces nicd.write and nicd.writevf, $\mathcal{F}_{\text{NICD}}$ follows a different approach. In the nicd.write interface, $\mathcal{F}_{\text{NICD}}$ receives as input two commitments com and com_2 along with a position and a value and the corresponding commitment openings from the prover. $\mathcal{F}_{\text{NICD}}$ creates Tbl_{ct} by updating Tbl_{ct-1} and stores com and com_2. In the nicd.writevf interface, $\mathcal{F}_{\text{NICD}}$ receives two commitments com' and com'_2, and if $com = com'$ and $com_2 = com'_2$, it informs $\mathcal{F}_{\text{NICD}}$ that those commitments commit to the position and value written by the prover.

In both nicd.write and nicd.writevf, $\mathcal{F}_{\text{NICD}}$ communicates with the simulator. Therefore, although the functionality does not imply communication between prover and verifiers, the computation and verification of proofs is not a local process.

In interface nicd.write, the simulator inputs database information $info_{ct}$. In the security proof for our construction for $\mathcal{F}_{\text{NICD}}$, $info_{ct}$ is the vector commitment that commits to the database. $\mathcal{F}_{\text{NICD}}$ needs $info_{ct}$ to give it as input to algorithms NICD.SimProve and NICD.Extract as part of the instance of proofs.

There are two reasons why we chose this approach for nicd.write and nicd.writevf:

- $\mathcal{F}_{\text{NICD}}$ must ensure that all the verifiers have the same version of the database, i.e., the prover should not be allowed to evolve the database differently for each of the verifiers. To ensure that all verifiers have the same database, in the nicd.write or nicd.writevf interfaces, any construction for $\mathcal{F}_{\text{NICD}}$ must implement a mechanism that allows verifiers to check that they share a common version of the database. In our construction, this implies checking that the same vector commitment is used to verify write proofs. This mechanism inevitably implies that the verification of write proofs cannot be a local process. For example, verifiers could communicate with each other, or they could retrieve the vector commitment from a trusted registration functionality, which is the approach taken in our construction. Because the process is not local, $\mathcal{F}_{\text{NICD}}$ needs to inform the simulator so that $\mathcal{F}_{\text{NICD}}$ is realizable.

- It would be possible to design $\mathcal{F}_{\text{NICD}}$ in such a way that $\mathcal{F}_{\text{NICD}}$ allows the prover to compute several write proofs for the same write operation, each proof for a different commitment pair com and com_2. However, in practice this does not provide any extra functionality or property. We remark that each write operation modifies a single position in the table, and that thus the different commitment pairs com and com_2 would have to commit to the same position and values. Additionally, giving different commitments com and com_2 to each verifier does not provide any unlinkability between prover

and verifier because, to achieve unlinkability, a number of other changes would be required.

6.4 CONSTRUCTION

We propose a construction Π_{NICD} for $\mathcal{F}_{\text{NICD}}$. Π_{NICD} uses a hiding vector commitment scheme (see Section 2.7.1). It uses the functionality $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ for a common reference string (see Section 2.8.1), which is parameterized by the setup algorithm $\mathsf{VC.Setup} = \mathsf{CRS.Setup}$ of the vector commitment scheme. $\mathsf{VC.Setup}$ outputs the parameters par. It also uses the registration functionality $\mathcal{F}_{\text{REG}}$ (see Section 2.8.2).

A vector commitment vc is used to store the database. A position in the vector commitment acts as a position in the database, and the value committed to in that position acts as the value stored in the database in that position. To compute a read proof and a write proof, the prover uses the functionalities $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$ respectively (see Section 2.8.9). The relation R_r is defined as follows.

$$\begin{aligned} R_r = \{(wit_r, ins_r) : & \\ & 1 = \mathsf{COM.Verify}(parcom, com, i, open) \wedge && (6.1)\\ & 1 = \mathsf{COM.Verify}(parcom, com_1, v_1, open_1) \wedge && (6.2)\\ & 1 = \mathsf{VC.Verify}(par, vc, v_1, i, w)\} && (6.3) \end{aligned}$$

where the witness wit_r is $(w, i, open, v_1, open_1)$ and the instance ins_r is $(par, vc, parcom, com, com_1, ct)$. The counter ct counts the number of write proofs sent by the prover. In equation 6.1, the prover proves that com is a commitment to i with opening $open$. Similarly, in equation 6.2, the prover proves that com_1 is a commitment to v_1 with opening $open_1$. In equation 6.3, the prover proves that the value v_1 is stored in the position i of the vector commitment vc.

The relation R_w is defined as follows.

$$\begin{aligned} R_w = \{(wit_w, ins_w) : & \\ & 1 = \mathsf{COM.Verify}(parcom, com, i, open) \wedge && (6.4)\\ & 1 = \mathsf{COM.Verify}(parcom, com_2, v_2, open_2) \wedge && (6.5)\\ & 1 = \mathsf{VC.VerComUpd}(par, vc, vc', w, i, v_1, r, v_2, r')\} && (6.6) \end{aligned}$$

where the witness wit_w is $(w, i, open, v_1, v_2, open_2, r, r')$ and the instance ins_w is $(par, vc, vc', parcom, com, com_2, ct)$. In equation 6.4 and equation 6.5, the prover proves that com and com_2 are commitments to i and v_2 respectively. In equation 6.6, the prover proves that v_1 is stored in the position i in the vector commitment vc, and that vc' is a vector commitment that stores the same values as vc, except that it stores v_2 in the position i and that its random value is r' instead of r.

We recall that the sizes of a vector commitment vc and of a witness w are independent of the length of the vector. Therefore, there exist efficient zero-knowledge protocols that realize $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$ whose communication complexity does not depend on the length of the vector. We depict Π_{NICD} below.

DESCRIPTION OF Π_{NICD}. Π_{NICD} uses a hiding vector commitment scheme and the ideal functionalities $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$. The constant

N_{max} denotes the size of the database, and the universe of state values U_v is given by the message space of the vector commitment scheme.

Figure 6.2: Construction Π_{NICD}

1. On input $(\mathsf{nicd.setup.ini}, sid)$, a party $\mathcal{T}$ does the following:
   - If par is not stored, $\mathcal{T}$ does the following:
     - $\mathcal{T}$ sends $(\mathsf{crs.get.ini}, sid)$ to $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ and receives $(\mathsf{crs.get.end}, sid, par)$ from the functionality $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$. To compute par, $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ runs $\mathsf{VC.Setup}(1^k, N_{max})$.
     - $\mathcal{T}$ sends $(\mathsf{nizk.setup.ini}, sid)$ to $\mathcal{F}_{\text{NIZK}}^{R_r}$ and receives $(\mathsf{nizk.setup.end}, sid, par_r)$ from $\mathcal{F}_{\text{NIZK}}^{R_r}$.
     - $\mathcal{T}$ sends $(\mathsf{nizk.setup.ini}, sid)$ to $\mathcal{F}_{\text{NIZK}}^{R_w}$ and receives $(\mathsf{nizk.setup.end}, sid, par_w)$ from $\mathcal{F}_{\text{NIZK}}^{R_w}$.
   - $\mathcal{T}$ sets $par \leftarrow (par, par_r, par_w)$, stores par and outputs $(\mathsf{nicd.setup.end}, sid, par)$.
2. On input $(\mathsf{nicd.write.ini}, sid, com, i, open, com_2, v_2, open_2)$, the prover $\mathcal{P}$ does the following:
   - $\mathcal{P}$ aborts if par is not stored.
   - $\mathcal{P}$ aborts if $i \notin [1, N_{max}]$, or if $v_2 \notin U_v$.
   - $\mathcal{P}$ parses the commitments com as $(com', parcom, \mathsf{COM.Verify})$ and com_2 as $(com'_2, parcom_2, \mathsf{COM.Verify}_2)$.
   - $\mathcal{P}$ aborts if $parcom \neq parcom_2$, or if $\mathsf{COM.Verify} \neq \mathsf{COM.Verify}_2$, or if $\mathsf{COM.Verify}$ is not a ppt algorithm.
   - $\mathcal{P}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com', i, open)$.
   - $\mathcal{P}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com'_2, v_2, open_2)$.
   - If there is no tuple $(0, vc, \mathbf{x}, r)$ stored, $\mathcal{P}$ initializes a counter $ct \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = \perp$ for $i = 1$ to N_{max}. $\mathcal{P}$ sets $r \leftarrow 0$ and runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$. (The symbol $\perp$ represents that no value is committed at a specific position and its actual value depends on the concrete implementation of the VC scheme.) $\mathcal{P}$ stores $(ct, vc, \mathbf{x}, r)$.
   - $\mathcal{P}$ takes the tuple $(ct, vc, \mathbf{x}, r)$ with the highest value of ct.
   - $\mathcal{P}$ picks random $r' \leftarrow \mathcal{R}$ and runs $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc, i, v_1, r, v_2, r')$, where $v_1 \leftarrow \mathbf{x}[i]$.
   - $\mathcal{P}$ runs $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r)$.
   - $\mathcal{P}$ sets $wit_w \leftarrow (w, i, open, v_1, v_2, open_2, r, r')$, where $v_1 \leftarrow \mathbf{x}[i]$.
   - $\mathcal{P}$ sets $ins_w \leftarrow (par, vc, vc', parcom, com', com'_2, ct + 1)$.

- $\mathcal{P}$ sends (nizk.prove.ini, sid, wit_w, ins_w) to $\mathcal{F}_{\text{NIZK}}^{R_w}$ and receives (nizk.prove.end, sid, π_w) from $\mathcal{F}_{\text{NIZK}}^{R_w}$.
- $\mathcal{P}$ sets $ct' \leftarrow ct + 1$, sets $sid_{\text{REG}} \leftarrow (sid, ct')$, sends (reg.register.ini, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$) to $\mathcal{F}_{\text{REG}}$ and receives (reg.register.end, sid_{REG}) from $\mathcal{F}_{\text{REG}}$.
- $\mathcal{P}$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_2$.
- $\mathcal{P}$ stores $(ct', vc', \mathbf{x}', r')$.
- Send (nicd.write.end, sid, ct') to $\mathcal{P}$.

3. On input (nicd.writevf.ini, sid, ct, com, com_2), a verifier $\mathcal{V}$ does the following:
   - $\mathcal{V}$ aborts if par is not stored.
   - If there is no tuple $(0, vc)$ stored, $\mathcal{V}$ initializes a counter $ct \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = \perp$ for $i = 1$ to N_{max}. $\mathcal{V}$ sets $r \leftarrow 0$ and runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$. (The symbol $\perp$ represents that no value is committed at a specific position and its actual value depends on the concrete implementation of the vector commitment scheme.) $\mathcal{V}$ stores (ct, vc).
   - Abort if $(ct - 1, vc)$ is not stored.
   - $\mathcal{V}$ sets $sid_{\text{REG}} \leftarrow (sid, ct)$, sends (reg.retrieve.ini, sid_{REG}) to $\mathcal{F}_{\text{REG}}$ and receives the message (reg.retrieve.end, sid_{REG}, v') from $\mathcal{F}_{\text{REG}}$.
   - If $v' \neq \perp$, $\mathcal{V}$ sets $\langle vc', \bar{com}, \bar{com}_2, \pi_w \rangle \leftarrow v'$, else sets $v \leftarrow \perp$ and outputs the message (nicd.writevf.end, sid, ct, v).
   - If $com \neq \bar{com}$ or if $com_2 \neq \bar{com}_2$, $\mathcal{V}$ sets $v \leftarrow 0$ and outputs (nicd.writevf.end, sid, ct, v).
   - $\mathcal{V}$ parses the commitments com as $(com', parcom, \mathsf{COM.Verify})$ and com_2 as $(com'_2, parcom_2, \mathsf{COM.Verify}_2)$.
   - $\mathcal{V}$ sets $ins_w \leftarrow (par, vc, vc', parcom, com', com'_2, ct)$.
   - $\mathcal{V}$ sends (nizk.verify.ini, sid, ins_w, π_w) to $\mathcal{F}_{\text{NIZK}}^{R_w}$ and receives (nizk.verify.end, sid, v) from $\mathcal{F}_{\text{NIZK}}^{R_w}$.
   - If $v = 1$, $\mathcal{V}$ stores (ct, vc').
   - $\mathcal{V}$ outputs (nicd.writevf.end, sid, v).

4. On input (nicd.read.ini, sid, ct, com, i, $open$, com_1, v_1, $open_1$), the prover $\mathcal{P}$ does the following:
   - $\mathcal{P}$ aborts if par is not stored, or if $(ct', vc, \mathbf{x}, r)$ such that $ct = ct'$ is not stored.
   - $\mathcal{P}$ aborts if $i \notin [1, N_{max}]$, or if $v_1 \notin U_v$, or if $\mathbf{x}[i] \neq v_1$.

- $\mathcal{P}$ parses the commitments com as $(com', parcom, \mathsf{COM.Verify})$ and com_1 as $(com'_1, parcom_1, \mathsf{COM.Verify}_1)$.
- $\mathcal{P}$ aborts if $parcom \neq parcom_1$, or if $\mathsf{COM.Verify} \neq \mathsf{COM.Verify}_1$, or if $\mathsf{COM.Verify}$ is not a ppt algorithm.
- $\mathcal{P}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com', i, open)$.
- $\mathcal{P}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com'_1, v_1, open_1)$.
- $\mathcal{P}$ runs $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r)$.
- $\mathcal{P}$ sets $wit_r \leftarrow (w, i, open, v_1, open_1)$.
- $\mathcal{P}$ sets $ins_r \leftarrow (par, vc, parcom, com', com'_1, ct)$.
- $\mathcal{P}$ sends $(\mathsf{nizk.prove.ini}, sid, wit_r, ins_r)$ to $\mathcal{F}_{\mathrm{NIZK}}^{R_r}$ and receives $(\mathsf{nizk.prove.end}, sid, \pi)$ from $\mathcal{F}_{\mathrm{NIZK}}^{R_r}$.
- $\mathcal{P}$ outputs $(\mathsf{nicd.read.end}, sid, \pi)$.

5. On input $(\mathsf{nicd.readvf.ini}, sid, com, com_1, ct, \pi)$, a verifier $\mathcal{V}$ does the following:
   - $\mathcal{V}$ aborts if par is not stored or if (ct', vc) such that $ct = ct'$ is not stored.
   - $\mathcal{V}$ parses the commitments com as $(com', parcom, \mathsf{COM.Verify})$ and com_1 as $(com'_1, parcom_1, \mathsf{COM.Verify}_1)$.
   - $\mathcal{V}$ sets $ins_r \leftarrow (par, vc, parcom, com', com'_1, ct)$.
   - $\mathcal{V}$ sends $(\mathsf{nizk.verify.ini}, sid, ins_r, \pi)$ to $\mathcal{F}_{\mathrm{NIZK}}^{R_r}$ and receives $(\mathsf{nizk.verify.end}, sid, v)$ from $\mathcal{F}_{\mathrm{NIZK}}^{R_r}$.
   - $\mathcal{V}$ outputs $(\mathsf{nicd.readvf.end}, sid, v)$.

6.5 SECURITY ANALYSIS

Theorem 6.5.1 *Π_{NICD} securely realizes $\mathcal{F}_{\mathrm{NICD}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_w}$-hybrid model if the VC scheme is hiding and binding.*

The proof of Theorem 6.5.1 is described in Section A.1.

6.6 VARIANTS

- The read and write operations of $\mathcal{F}_{\mathrm{NICD}}$ can be modified to allow $\mathcal{P}$ to read several database entries simultaneously. To this end, the $\mathsf{nicd.read}$ interface is modified so that the prover inputs a tuple $(com_i, i, open_i, com_{i,1}, v_{i,1}, open_{i,1})_{\forall i \in \mathbb{S}}$ ($\mathbb{S} \subseteq [1, N_{max}]$) of database entries to be read. The algorithm $\mathsf{NICD.SimProve}$ receives as input the tuple $(com_i, com_{i,1})_{\forall i \in \mathbb{S}}$ of commitments. The $\mathsf{nicd.readvf}$ is modified so that the verifier inputs the tuples

$(com_i, com_{i,1})_{\forall i \in \mathbb{S}}$, and algorithm NICD.Extract also receives those tuples as input. For the write operation, the nicd.write interface is modified so that the prover sends a tuple $(com_i, i, open_i, com_{i,2}, v_{i,2}, open_{i,2})_{\forall i \in \mathbb{S}}$. The nicd.writevf interface is modified so that the verifier sends a tuple $(com_i, com_{i,2})_{\forall i \in \mathbb{S}}$.

- $\mathcal{F}_{\text{NICD}}$ can be modified to store a database Tbl_{ct} of the form $[i, v_{i,1}, \ldots, v_{i,m}]$, i.e., a database where a tuple of values is stored in each entry. In the read and write operations, the prover and the verifiers would include commitments and openings to each of the values in a database entry. The position $j \in [1, m]$ of each value read or written is not hidden from $\mathcal{V}$. This variant of $\mathcal{F}_{\text{NICD}}$ is useful for protocols where a party needs to read a tuple of values and prove that they are stored in the same entry and that each value is stored at a certain position j within the entry.

7

UNLINKABLE UPDATABLE HIDING DATABASES

Joint work with Alfredo Rial. This primitive was published in a paper presented at the Privacy Enhancing Technologies Symposium in 2021 [41]. The primitive finds use as a building block in the protocol described in Chapter 13.

An Unlinkable Updatable Hiding Database is a protocol between an updater $\mathcal{U}$ and multiple readers $\mathcal{R}_k$. A UUHD consists of an update phase and a read phase. In an update phase, $\mathcal{U}$ updates the database stored by a reader $\mathcal{R}_k$ identified by a pseudonym P. We consider a simple database DB with entries of the form $[i, vr_i]$, where i is the position and vr_i the value stored at position i. $\mathcal{U}$ does not learn the contents of DB and cannot link it to previous updates. However, $\mathcal{U}$ is ensured that he is updating the last version of DB given to $\mathcal{R}_k$. In the read phase, $\mathcal{R}_k$ commits to some data from the database and proves to $\mathcal{U}$ that it is stored in DB. Later, $\mathcal{R}_k$ can use these commitments to prove in zero-knowledge other statements about the data in DB.

7.1 OPERATIONS

UPDATE. During an *update* operation, $\mathcal{U}$ updates an entry in the last version of a database DB stored by $\mathcal{R}_k$, whilst the contents of DB and the identity of $\mathcal{R}_k$ remain hidden from $\mathcal{U}$.

READ. During a *read* operation, $\mathcal{R}_k$ proves knowledge of an entry in the database, whilst hiding the entry being read and its position from $\mathcal{U}$.

7.2 SECURITY PROPERTIES

UNLINKABILITY. The read operations remain unlinkable towards $\mathcal{U}$. The functionality $\mathcal{F}_{\text{UUHD}}$ reveals to $\mathcal{U}$ a pseudonym P rather than the identifier $\mathcal{R}_k$, and $\mathcal{F}_{\text{UUHD}}$ checks that P is unique for each read operation.

HIDING. $\mathcal{U}$ does not learn the contents of any DB. $\mathcal{U}$ sets the initial values of each DB but as read operations are unlinkable, $\mathcal{U}$ cannot keep track of the updates performed on each DB and thus does not know their contents. Additionally, when reading, $\mathcal{R}_k$ sends $\mathcal{U}$ commitments to database entries, which are hiding as guaranteed by $\mathcal{F}_{\text{NIC}}$. When $\mathcal{F}_{\text{UUHD}}$ is used as building block of a protocol, $\mathcal{R}_k$ can prove in ZKs statements about the committed values rather than revealing them.

UNFORGEABILITY. $\mathcal{R}_k$ cannot modify her database DB or read from DB entries that were not stored or updated by $\mathcal{U}$. Additionally, $\mathcal{R}_k$ cannot make $\mathcal{U}$ update an old version of DB.

7.3 IDEAL FUNCTIONALITY

Figure 7.1: Ideal Functionality $\mathcal{F}_{\text{UUHD}}$

$\mathcal{F}_{\text{UUHD}}$ is parameterized by a DB size N, a universe of pseudonyms $\mathbb{U}_p$, a universe of values $\mathbb{U}_v$ and an operator $\odot : \mathbb{U}_v \times \mathbb{U}_v \to \mathbb{U}_v$.

1. On input $(\mathsf{uuhd.read.ini}, sid, P, (i, vr_i, ccom_i, copen_i, ccomr_i, copenr_i)_{i\in\mathbb{S}})$ from $\mathcal{R}_k$:
   - Abort if $sid \notin (\mathcal{U}, sid')$, or if $P \notin \mathbb{U}_p$.
   - Abort if there is a tuple $(sid, P', \mathcal{R}'_k, \mathsf{flag})$ stored such that $P' = P$, or such that $\mathcal{R}'_k = \mathcal{R}_k$ and $\mathsf{flag} \neq \perp$.
   - If a tuple $(sid, \mathcal{R}_k, \mathsf{DB})$ is not stored, set $\mathsf{flag} \leftarrow 1$, else set $\mathsf{flag} \leftarrow 0$.
   - If $\mathsf{flag} = 0$ and $\mathbb{S} \neq \emptyset$, for all $i \in \mathbb{S}$, do the following:
     - Abort if $i \notin [1, N]$, or if $vr_i \notin \mathbb{U}_v$, or if $[i, \mathbb{U}_v]$ is not stored in DB.
     - Parse the commitment $ccom_i$ as $(ccom'_i, parcom, \mathsf{COM.Verify})$.
     - Parse the commitment $ccomr_i$ as $(ccomr'_i, parcom, \mathsf{COM.Verify})$.
     - Abort if COM.Verify is not a ppt algorithm.
     - Abort if $1 \neq \mathsf{COM.Verify}(parcom, ccom'_i, i, copen_i)$.
     - Abort if $1 \neq \mathsf{COM.Verify}(parcom, ccomr'_i, vr_i, copenr_i)$.
   - Create a fresh qid and store $(qid, P, \mathcal{R}_k, \mathsf{flag}, (ccom_i, ccomr_i)_{i\in\mathbb{S}})$.
   - Send $(\mathsf{uuhd.read.sim}, sid, qid, (ccom_i, ccomr_i)_{i\in\mathbb{S}})$ to $\mathcal{S}$.

S. On input $(\mathsf{uuhd.read.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{R}_k, \mathsf{flag}, (ccom_i, ccomr_i)_{i\in\mathbb{S}})$ such that $qid = qid'$ is not stored.
   - Store the tuple $(sid, P, \mathcal{R}_k, \mathsf{flag})$.
   - Delete the tuple $(qid, P, \mathcal{R}_k, \mathsf{flag}, (ccom_i, ccomr_i)_{i\in\mathbb{S}})$.
   - Send $(\mathsf{uuhd.read.end}, sid, P, \mathsf{flag}, (ccom_i, ccomr_i)_{i\in\mathbb{S}})$ to $\mathcal{U}$.

2. On input $(\mathsf{uuhd.update.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$ from $\mathcal{U}$:
   - Abort if a tuple $(sid, P', \mathcal{R}_k, \mathsf{flag})$ such that $P' = P$ and $\mathsf{flag} \neq \perp$ is not stored.
   - Abort if, for $i \in [1, N]$, $vu_i \notin \mathbb{U}_v$.
   - If $\mathsf{flag} = 1$, set $\mathsf{DB} \leftarrow (i, vu_i)_{i\in[1,N]}$ and store a tuple $(sid, \mathcal{R}_k, \mathsf{DB})$.

- If $\mathsf{flag} = 0$, take the stored tuple $(sid, \mathcal{R}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i\in[1,N]}$ and update $\mathsf{DB} \leftarrow (i, vr_i \odot vu_i)_{i\in[1,N]}$ in the tuple $(sid, \mathcal{R}_k, \mathsf{DB})$.
- Update $\mathsf{flag} \leftarrow \perp$ in the tuple $(sid, P', \mathcal{R}_k, \mathsf{flag})$ with $P' = P$.
- Create a fresh qid and store $(qid, \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$.
- Send $(\mathsf{uuhd.update.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{uuhd.update.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if a tuple $(qid', \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$ such that $qid = qid'$ is not stored.
- Delete the record $(qid', \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$ such that $qid = qid'$.
- Send $(\mathsf{uuhd.update.end}, sid, P, (i, vu_i)_{i\in[1,N]})$ to $\mathcal{R}_k$.

7.4 CONSTRUCTION

We propose a construction Π_{UUHD} for $\mathcal{F}_{\mathrm{UUHD}}$. Π_{UUHD} uses a hiding vector commitment (VC) scheme as its main building block (see Section 2.7.1). A VC scheme allows us to compute a vector commitment vc to a vector $\mathbf{x}$ of values, where each value $\mathbf{x}[i]$ is stored at a given position i. Additionally, vc can be opened to $\mathbf{x}[i]$ with communication cost independent of the vector length. We use a VC scheme that is additively homomorphic. Π_{UUHD} also uses a pseudonymous channel that provides unlinkability in communications between any $\mathcal{R}_k$ and $\mathcal{U}$.

A VC vc is used to store the database DB by committing to a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vr_i$. In its first interaction with the updater $\mathcal{U}$, the reader $\mathcal{R}_k$ obtains a VC vc to an initial DB. $\mathcal{U}$ signs vc and gives the signature sig to $\mathcal{R}_k$. In subsequent interactions, $\mathcal{R}_k$ may want to read entries from DB, to make $\mathcal{U}$ update the database, or both. To read entries $[i, vr_i]$, $\mathcal{R}_k$ gets as input the commitments and openings $(ccom_i, copen_i, ccomr_i, copenr_i)_{i\in\mathbb{S}}$. $\mathcal{R}_k$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ to prove in ZK that $ccom_i$ and $ccomr_i$ commit to an entry $[i, vr_i]$ in DB. $\mathcal{U}$ receives a randomized version vc' of vc. To update DB with $(i, vu_i)_{i\in[1,N]}$, $\mathcal{U}$ computes a VC vc_2 to $(i, vu_i)_{i\in[1,N]}$ and then uses the homomorphic property of the VC scheme to set $vc_u \leftarrow vc' \cdot vc_2$. The VC vc_u that commits to the updated database is signed and sent to $\mathcal{R}_k$.

To prevent $\mathcal{R}_k$ from reading an old database, Π_{UUHD} uses a double-spending detection mechanism. A commitment com is signed along with every vc. com commits to a random value $s = s_1 + s_2$, where s_1 and s_2 are random values contributed by $\mathcal{R}_k$ and $\mathcal{U}$ respectively. In a read operation, $\mathcal{R}_k$ reveals a randomized version com' of com and opens com' to s. $\mathcal{U}$ checks that s has never been received before. The message space of the commitment scheme should be sufficiently large in order to avoid collisions of random values.

To provide unlinkability, in addition to rerandomizing vc' and com', $\mathcal{R}_k$ communicates with $\mathcal{U}$ through a secure pseudonymous channel modelled by $\mathcal{F}_{\mathrm{NYM}}$ (see Section 2.8.6). $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ (see Section 2.8.1) is used by $\mathcal{R}_k$ and $\mathcal{U}$ to retrieve the VC parameters and the commitment parameters. (We note that the commitments

$(ccom_i, copen_i, ccomr_i, copenr_i)_{i\in\mathbb{S}}$ received as input by $\mathcal{R}_k$ are computed outside the protocol by $\mathcal{F}_{\text{NIC}}$ and may have other parameters.) $\mathcal{F}_{\text{REG}}$ (see Section 2.8.2) is used to register the signing public key of $\mathcal{U}$.

Figure 7.2: Description of Π_{UUHD}

Π_{UUHD} is parameterized by a database size N, a universe of values $\mathbb{U}_v$, which is given by the message space of the VC scheme, and a universe of pseudonyms $\mathbb{U}_p$, which parameterizes $\mathcal{F}_{\text{NYM}}$.

1. On input (uuhd.read.ini, sid, P, $(i, vr_i, ccom_i, copen_i, ccomr_i, copenr_i)_{i\in\mathbb{S}}$), $\mathcal{R}_k$ and $\mathcal{U}$ do the following:
   - If a tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, com, s, open, sig)$ is not stored:
     - $\mathcal{R}_k$ uses the crs.get interface of $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ to obtain the VC parameters par and the commitment parameters par_c.
     - $\mathcal{R}_k$ picks a random value $s_1 \leftarrow \mathcal{M}$, computes a commitment $(com_1, open_1) \leftarrow \mathsf{Com}(par_c, s_1)$ and stores $(sid, par, par_c, com_1, s_1, open_1)$.
     - $\mathcal{R}_k$ picks a random pseudonym $P \leftarrow \mathbb{U}_p$ and uses the nym.send interface of $\mathcal{F}_{\text{NYM}}$ on input P to send com_1 to $\mathcal{U}$.
     - If P was received before, $\mathcal{U}$ aborts.
     - $\mathcal{U}$ sets a flag flag $\leftarrow 1$ (because a single commitment com_1 is received) and $(ccom_i, ccomr_i)_{i\in\mathbb{S}} \leftarrow \perp$.
     - $\mathcal{U}$ stores $(sid, P, \mathsf{flag}, com_1)$.
   - If a tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, com, s, open, sig)$ is stored:
     - $\mathcal{R}_k$ picks $r_2 \leftarrow \mathcal{R}$, sets $r' \leftarrow r + r_2$ and executes $vc' \leftarrow \mathsf{VC.Rerand}(vc, r_2)$ to rerandomize vc.
     - For all $i \in \mathbb{S}$, $\mathcal{R}_k$ does the following:
       - Abort if $\mathbf{x}[i] \neq vr_i$.
       - Compute an opening $w_i \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x}, r')$.
       - Parse the commitment $ccom_i$ as $(ccom'_i, parcom, \mathsf{COM.Verify})$.
       - Parse the commitment $ccomr_i$ as $(ccomr'_i, parcom, \mathsf{COM.Verify})$.
     - $\mathcal{R}_k$ picks random $open_2$, sets $open' \leftarrow open + open_2$ and runs the algorithm $com' \leftarrow \mathsf{CRerand}(com, open_2)$ to rerandomize com.

- $\mathcal{R}_k$ sets the witness $wit_r \leftarrow (sig, vc, com, r_2, open_2, \langle i, vr_i, w_i, copen_i, copenr_i\rangle_{i\in\mathbb{S}})$ and $ins_r \leftarrow (pk, par, par_c, vc', com', parcom, \langle ccom'_i, ccomr'_i\rangle_{i\in\mathbb{S}})$.
- $\mathcal{R}_k$ replaces vc by vc' and r by r' in the stored tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, com, s, open, sig)$.

R_r is

$$
\begin{aligned}
R_r = \{(wit_r, ins_r) : \\
& 1 = \mathsf{VfSig}(pk, sig, \langle vc, com\rangle) \wedge && (7.1)\\
& vc' = \mathsf{VC.Rerand}(vc, r_2) \wedge && (7.2)\\
& com' = \mathsf{CRerand}(com, open_2) \wedge && (7.3)\\
& \{1 = \mathsf{COM.Verify}(parcom, ccom'_i, i, copen_i) \wedge && (7.4)\\
& 1 = \mathsf{COM.Verify}(parcom, ccomr'_i, vr_i, copenr_i) \wedge && (7.5)\\
& 1 = \mathsf{VC.Verify}(par, vc', vr_i, i, w_i)\ \}_{\forall i\in\mathbb{S}}\ \} && (7.6)
\end{aligned}
$$

In equation 7.1, $\mathcal{R}_k$ proves that vc and com were signed by $\mathcal{U}$. In equation 7.2 and equation 7.3, $\mathcal{R}_k$ proves that vc' and com' are rerandomizations of vc and com. In equation 7.4 and equation 7.5, $\mathcal{R}_k$ proves that $ccom'_i$ and $ccomr'_i$ commit to i and vr_i respectively. In equation 7.6, $\mathcal{R}_k$ proves that $\mathbf{x}[i] = vr_i$, where $\mathbf{x}$ is the vector committed to in vc'.

- $\mathcal{R}_k$ randomly picks a pseudonym $P \leftarrow \mathbb{U}_p$ and uses the zk.prove interface to send wit_r, ins_r and P to $\mathcal{F}_{\text{ZK}}^{R_r}$.
- $\mathcal{U}$ receives ins_r and P. $\mathcal{U}$ aborts if P has already been received in the past. $\mathcal{U}$ checks if the signing public key, VC parameters and commitment parameters in ins_r are equal to those stored in the tuple $(sid, par, par_c, pk, sk)$. $\mathcal{U}$ sets a flag $\mathsf{flag} \leftarrow 0$ and stores $(sid, P, \mathsf{flag}, vc', com')$.
- $\mathcal{U}$ sends a message (Open com') to $\mathcal{U}$ by using the nym.reply interface of $\mathcal{F}_{\text{NYM}}$ on input P. (Here we consider that any construction for $\mathcal{F}_{\text{ZK}}^{R_r}$ uses $\mathcal{F}_{\text{NYM}}$, so $\mathcal{U}$ can reply through $\mathcal{F}_{\text{NYM}}$.)
- $\mathcal{R}_k$ picks a random value $s_1 \leftarrow \mathcal{M}$, computes a commitment $(com_1, open_1) \leftarrow \mathsf{Com}(par_c, s_1)$ and stores $(sid, com_1, s_1, open_1)$.
- $\mathcal{R}_k$ uses the nym.send interface of $\mathcal{F}_{\text{NYM}}$ on input P to send the message and the opening $(s, open')$ of com' and a new commitment com_1 to $\mathcal{U}$.
- $\mathcal{U}$ verifies the opening by running $\mathsf{VfCom}(par_c, com', s, open')$. $\mathcal{U}$ checks if s has not been received in the past, which ensures that the reader uses the last version of the database $\mathbf{x}$ in vc. $\mathcal{U}$ replaces com' by com_1 in the stored tuple $(sid, P, \mathsf{flag}, vc', com_1)$.

- $\mathcal{U}$ outputs (uuhd.read.end, sid, P, flag, ($ccom_i$, $ccomr_i$)$_{i\in\mathbb{S}}$).

2. On input (uuhd.update.ini, sid, P, $(i, vu_i)_{i\in[1,N]}$), $\mathcal{U}$ and $\mathcal{R}_k$ do the following:

   - $\mathcal{U}$ aborts if there is not a tuple (sid, P', flag, ...) such that $P = P'$.
   - If (sid, par, par_c, pk, sk) is not stored, $\mathcal{U}$ obtains par and par_c from $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$. $\mathcal{U}$ computes a signing key pair (pk, sk) $\leftarrow$ KeyGen(1^k) and uses the reg.register interface of $\mathcal{F}_{\text{REG}}$ to register pk. $\mathcal{U}$ stores (sid, par, par_c, pk, sk).
   - If flag $= 1$,
     - $\mathcal{U}$ takes the stored tuple (sid, P, flag, com_1).
     - $\mathcal{U}$ sets a vector $\mathbf{x}[i] \leftarrow vu_i$ (for all $i \in [1, N]$) and computes a vector commitment $vc \leftarrow$ VC.Commit(par, $\mathbf{x}$, 0) with 0 as randomness.
     - $\mathcal{U}$ picks a random value s_2, computes a commitment (com_2, 0) $\leftarrow$ Com(par_c, s_2) with 0 as opening, and computes $com \leftarrow com_1 \cdot com_2$.
     - $\mathcal{U}$ computes a signature $sig \leftarrow$ Sign(sk, $\langle vc, com\rangle$) on vc and com.
     - $\mathcal{U}$ uses nym.reply on input P to send ($\mathbf{x}$, s_2, sig) to $\mathcal{R}_k$.
     - $\mathcal{R}_k$ takes the stored (sid, par, par_c, com_1, s_1, $open_1$).
     - $\mathcal{R}_k$ computes $vc \leftarrow$ VC.Commit(par, $\mathbf{x}$, 0) with o as randomness.
     - $\mathcal{R}_k$ computes (com_2, 0) $\leftarrow$ Com(par_c, s_2) with o as opening and sets $com \leftarrow com_1 \cdot com_2$.
     - $\mathcal{R}_k$ uses the reg.retrieve interface of $\mathcal{F}_{\text{REG}}$ to retrieve the public key pk of $\mathcal{U}$ and verifies the signature sig by running VfSig(pk, sig, $\langle vc, com\rangle$).
     - $\mathcal{R}_k$ stores (sid, par, par_c, pk, vc, $\mathbf{x}$, $r \leftarrow 0$, com, $s \leftarrow s_1 + s_2$, $open \leftarrow open_1$, sig).
   - If flag $= 0$,
     - $\mathcal{U}$ takes the stored tuple (sid, P, flag, vc', com_1).
     - $\mathcal{U}$ sets a vector $\mathbf{x}_u[i] \leftarrow vu_i$ (for all $i \in [1, N]$), computes a commitment $vc_2 \leftarrow$ VC.Commit(par, $\mathbf{x}_u$, 0) with 0 as randomness and sets $vc_u \leftarrow vc' \cdot vc_2$.
     - $\mathcal{U}$ picks a random value s_2, computes a commitment (com_2, 0) $\leftarrow$ Com(par_c, s_2) with 0 as opening, and computes $com_u \leftarrow com_1 \cdot com_2$.
     - $\mathcal{U}$ sets a signature $sig \leftarrow$ Sign(sk, $\langle vc_u, com_u\rangle$) on vc_u and com_u.

- $\mathcal{U}$ uses nym.reply on input P to send $(\mathbf{x}_u, s_2, sig)$ to $\mathcal{R}_k$.
- $\mathcal{R}_k$ takes the stored tuple $(sid, com_1, s_1, open_1)$.
- $\mathcal{R}_k$ computes $(com_2, 0) \leftarrow \mathsf{Com}(par_c, s_2)$ with 0 as opening and sets $com_u \leftarrow com_1 \cdot com_2$.
- $\mathcal{R}_k$ takes vc from the stored tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, com, s, open, sig)$, computes $vc_2 \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, 0)$ with 0 as randomness and sets $vc_u \leftarrow vc \cdot vc_2$.
- $\mathcal{R}_k$ verifies the signature sig by running $\mathsf{VfSig}(pk, sig, \langle vc_u, com_u \rangle)$.
- $\mathcal{R}_k$ updates the tuple $(sid, par, par_c, pk, vc \leftarrow vc_u, \mathbf{x} \leftarrow \mathbf{x} + \mathbf{x}_u, r, com, s \leftarrow s_1 + s_2, open \leftarrow open_1, sig)$.

• $\mathcal{R}_k$ outputs (uuhd.update.end, $sid, P, \mathbf{x}$).

DISCUSSION. In Π_{UUHD}, an update operation always follows a read operation, and vice versa. We remark that, when only an update is required, the tuple $(i, vr_i, ccom_i, copen_i, ccomr_i, copenr_i)_{i \in \mathbb{S}}$ can be empty, and when only a read operation is needed, the tuple $(i, vu_i)_{i \in [1,N]}$ can equal $(i, 0)_{i \in [1,N]}$.

In Π_{UUHD}, $\mathcal{U}$ updates databases without knowing their content. The update operator is the sum. In our instantiation, the sum is done modulo p. We assume that p is large enough so that the sums accumulated are always smaller.

Π_{UUHD} ensures that $\mathcal{R}_k$ cannot modify a database and that $\mathcal{R}_k$ cannot use old versions of the database. However, Π_{UUHD} does not prevent a user from creating several reader identities and thus holding several databases. For applications where it is desirable that each user has only one database, Π_{UUHD} can be modified so that communication takes place through an authenticated channel when the initialization of a database is requested. Authentication allows $\mathcal{U}$ to reject those users who already have a database.

Π_{UUHD} does not prevent adversarial readers from sharing their databases. Π_{UUHD} can be extended with existing mechanisms to discourage transferability of anonymous credentials [26].

7.5 SECURITY ANALYSIS

Theorem 7.5.1 Π_{UUHD} *securely realizes* $\mathcal{F}_{\text{UUHD}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NYM}}$ *and* $\mathcal{F}_{\text{ZK}}^{R_r}$*-hybrid model if the VC scheme is hiding and binding, the commitment scheme is hiding and binding, and the signature scheme is existentially unforgeable.*

The proof of Theorem 7.5.1 is described in Section A.2.

7.6 INSTANTIATION AND EFFICIENCY ANALYSIS

We describe an instantiation of Π_{UUHD} that uses the VC scheme in Section 2.7.1.1, the Pedersen commitment scheme [80], the signature scheme in [2] and the UC ZK proof protocol in [24].

7.6.1 *Commitment Scheme*

We use the Pedersen commitment scheme [80]. $\mathsf{CSetup}(1^k)$ takes a group $\mathbb{G}$ of prime order p with generator g, picks random α, computes $h \leftarrow g^\alpha$ and sets the parameters $par_c \leftarrow (\mathbb{G}, g, h)$, which include a description of the message space $\mathcal{M} \leftarrow \mathbb{Z}_p$. $\mathsf{Com}(par_c, x)$ picks random $open \leftarrow \mathbb{Z}_p$ and outputs a commitment $com \leftarrow g^x h^{open}$ to $x \in \mathcal{M}$ and an opening $open$. $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if $com = g^x h^{open}$. This scheme is perfectly hiding and computationally binding. In [20], it is shown that Pedersen commitments realize $\mathcal{F}_{\text{NIC}}$. Therefore, we use Pedersen commitments to instantiate both the commitments computed in Π_{UUHD} and those computed by $\mathcal{F}_{\text{NIC}}$ and received as input to Π_{UUHD}.

7.6.2 *Signature Scheme*

We use the structure-preserving signature (SPS) scheme in [2]. In SPSs, the public key, the messages, and the signatures are group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, and verification must involve the evaluation of pairing product equations. We employ SPSs to sign group elements, whilst still supporting efficient ZK proofs of signature possession. In this SPS scheme, a elements in $\mathbb{G}$ and b elements in $\tilde{\mathbb{G}}$ are signed.

$\mathsf{KeyGen}(grp, a, b)$. Let $grp \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ be bilinear map parameters. Pick at random $u_1, \ldots, u_b, v, w_1, \ldots w_a, z \leftarrow \mathbb{Z}_p^*$ and compute $U_i = g^{u_i}, i \in [1..b], V = \tilde{g}^v, W_i = \tilde{g}^{w_i}, i \in [1..a]$ and $Z = \tilde{g}^z$. Return the verification key $pk \leftarrow (grp, U_1, \ldots, U_b, V, W_1, \ldots, W_a, Z)$ and the signing key $sk \leftarrow (pk, u_1, \ldots, u_b, v, w_1, \ldots, w_a, z)$.

$\mathsf{Sign}(sk, \langle m_1, \ldots, m_{a+b} \rangle)$. Pick $r \leftarrow \mathbb{Z}_p^*$, set $R \leftarrow g^r, S \leftarrow g^{z-rv} \prod_{i=1}^{a} m_i^{-w_i}$, and $T \leftarrow (\tilde{g} \prod_{i=1}^{b} m_{a+i}^{-u_i})^{1/r}$, and output the signature $sig \leftarrow (R, S, T)$.

$\mathsf{VfSig}(pk, sig, \langle m_1, \ldots, m_{a+b} \rangle)$. Output 1 if $e(R, V)e(S, \tilde{g}) \prod_{i=1}^{a} e(m_i, W_i) = e(g, Z)$ and $e(R, T) \prod_{i=1}^{b} e(U_i, m_{a+i}) = e(g, \tilde{g})$.

7.6.3 *ZK Functionality*

To instantiate $\mathcal{F}_{\text{ZK}}^{R}$, we use the scheme in [24] as described in Section 2.8.8.1.

7.6.3.1 *UC ZK Proof for the Read Relation*

To instantiate $\mathcal{F}_{\text{ZK}}^{R_r}$ with the protocol in [24], we need to instantiate R_r with our chosen VC, commitment and signature schemes. Then we need to express R_r following the notation for UC ZK proofs described above. We use a ZK proof similar to the one in [63].

In R_r, we need to prove that the position i committed to in $ccom'_i$ equals the position opened in the VC vc via the verification equation $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$. In our VC scheme, α is secret, which makes the relation between $\tilde{g}_i = \tilde{g}^{\alpha^i}$ and i not efficiently provable. To solve this problem, the updater computes SPSs that bind g^i with $\tilde{g}_i$. Given the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$, and the key pair (sk, pk), the updater, for $i \in [1, \ell]$, computes $sig_i \leftarrow \mathsf{Sign}(sk, \langle g^{sid}, \tilde{g}_i, \tilde{g}^i \rangle)$.

Let $\tilde{h} \leftarrow \tilde{\mathbb{G}}$. Let $(g, h) \in \mathbb{G}^2$ and $(\tilde{g}, \tilde{h}) \in \mathbb{G}^2$ be two sets of parameters of the Pedersen commitment scheme for the commitments $(ccom'_i, ccomr'_i)$ and com respectively. R_r involves proofs about secret bases, and we use the transformation described above for these proofs. The base h is also used to randomize secret bases in $\mathbb{G}$, and $\tilde{h}$is added to randomize bases in $\tilde{\mathbb{G}}$.

Let (U_1, U_2, V, W_1, Z) be the public key of the signature scheme for signing 1 element of $\mathbb{G}$ and 2 elements of $\tilde{\mathbb{G}}$. Let (R_i, S_i, T_i) be a signature on $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Let (R, S, T) be a signature on vc, com and $\tilde{g}^{sid}$. We describe the ZK proof for R_r as follows:

When verifying a proof, the updater can distinguish between the two types of signatures because sid is signed in different positions.

$$\begin{aligned}
&\nwarrow vc, com, R, S, T, \{i, copen_i, vr_i, copenr_i, \tilde{g}_i, w_i, R_i, S_i, T_i\}_{\forall i \in \mathbb{S}} : \\
&\quad e(R, V)e(S, \tilde{g})e(vc, W_1)e(g, Z)^{-1} = 1 \wedge && (7.7)\\
&\quad e(R, T)e(U_1, com)e(U_2, \tilde{g}^{sid})e(g, \tilde{g})^{-1} = 1 \wedge && (7.8)\\
&\quad \{ccom'_i = g^i h^{copen_i} \wedge ccomr'_i = g^{vr_i} h^{copenr_i} \wedge && (7.9)\\
&\quad e(R_i, V)e(S_i, \tilde{g})e(g^{sid}, W_1)e(g, Z)^{-1} = 1 \wedge && (7.10)\\
&\quad e(R_i, T_i)e(U_1, \tilde{g}_i)e(U_2, \tilde{g})^i e(g, \tilde{g})^{-1} = 1 \wedge && (7.11)\\
&\quad e(vc, \tilde{g}_i)^{-1} e(w_i, \tilde{g})e(g_1, \tilde{g}_\ell)^{vr_i} = 1\}_{\forall i \in \mathbb{S}} && (7.12)
\end{aligned}$$

Equation 7.7 and equation 7.8 prove knowledge of a signature (R, S, T) on vc, com and $\tilde{g}^{sid}$. Equation 7.9 proves knowledge of the openings of the Pedersen commitments $ccom'_i$ and $ccomr'_i$. Equation 7.10 and Equation 7.11 prove knowledge of a signature (R_i, S_i, T_i) on a message $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Equation 7.12 proves that the value vr_i in $ccomr'_i$ is equal to the value committed to in position i of the vector commitment vc.

To prove knowledge of the secret bases $(vc, com, R, S, T, \{\tilde{g}_i, w_i, R_i, S_i, T_i\}_{\forall i \in \mathbb{S}})$, the equations need to be modified following the transformation described above. This means that each of the bases is rerandomized and added to the instance. Therefore, a reader rerandomizes com and vc, and the fact that they are a rerandomization of the values signed in the signature (R, S, T) is proven via equation 7.7 and equation 7.8. In the case of vc, we use g instead of h as the base used for randomization.

7.6.4 *Efficiency Analysis*

7.6.4.1 *Storage Costs*

$\mathcal{R}_k$ stores the common reference string, which consists of the VC and commitment parameters. The VC parameters grow linearly with the maximum size N of the database. $\mathcal{R}_k$ also stores the public key pk, which is of constant size. Throughout protocol execution, $\mathcal{R}_k$ stores the last updated values of sig, vc and com, the committed values and their openings. In conclusion, the storage cost for $\mathcal{R}_k$ grows

linearly with N. $\mathcal{U}$ stores the common reference string and the signing key pair and thus its storage cost also grows linearly with N.

7.6.4.2 *Communication Costs*

In a read operation, $\mathcal{R}_k$ and $\mathcal{U}$ run a ZK proof for R_r where wit_r and ins_r grow linearly with the number $|\mathbb{S}|$ of positions read, but are independent of N. Therefore, the communication cost grows linearly with $|\mathbb{S}|$ but is independent of N. $\mathcal{R}_k$ sends a Pedersen commitment com_1 and an opening $(s, open')$ to $\mathcal{U}$, which have constant size. In an update, $\mathcal{U}$ sends $\mathcal{R}_k$ an update $\mathbf{x}_u$ for vc', a signature on vc' and com' and the random value s_2. The signature and s_2 are of constant size. Although the size of $\mathbf{x}_u$ could be N, in practice, $\mathbf{x}_u$ only needs to contain the values for positions that are updated. Consequently, the communication cost of read and update operations grows linearly with the number of positions read or updated and is independent of N.

7.6.4.3 *Computation Costs*

In a read operation, $\mathcal{R}_k$ needs to compute VC openings w_i for the positions read. If w_i is computed for the first time, the computation cost grows linearly with N. However, by using the homomorphic property of the VC scheme, when vc is updated, each opening w_i can be updated with cost that grows linearly with the number of updated positions but is independent of N. The remaining steps in the computation and verification of the proof for R_r are also independent of N. $\mathcal{R}_k$ computes a commitment com_1, which entails constant cost. In the first update operation, $\mathcal{U}$ computes a commitment vc to an initial database with cost linear to N. Subsequently, $\mathcal{U}$ updates vc by using the homomorphic property of the VC scheme. The computation cost grows with the number of positions updated.

In practice, if the database is initialized to a vector of zeroes, the cost is constant.

In summary, the communication cost is independent of N and the amortized computation cost is also independent of N, which makes our instantiation of Π_{UUHD} practical for large databases.

7.6.5 *Implementation and Efficiency Measurements*

We have implemented our instantiation of Π_{UUHD} in the Python programming language, using the Charm cryptographic framework [4], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve is used for the pairing group setup. To compute the UC ZK proofs for R_r, we use the compiler in [24]. The cost of a proof depends on the number of elements in the witness and on the number of equations composed by Boolean ANDs. The computation cost for the prover of a Σ-protocol for R_r involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [24] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the values in the witness, an evaluation of a Paillier encryption for each of the values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for the verifier, as well as the communication cost, also depends on the number of values in the witness and on the number of equations. Therefore, as the number of values

in the witness and of equations is independent of N in our proof for relation R_r, the computation and communication costs of our proof do not depend on N.

INTERFACE	$N = 100$	$N = 1000$
First Update	0.6731	6.1561
Computation of vc or w_i	0.0076	0.0712
1 - entry update of vc or w_i	0.0001	0.0001
1 - entry Read (1024-bit key)	1.4378	1.3991
5 - entry Read (1024-bit key)	4.7172	5.0626
1 - entry Read (2048-bit key)	6.2507	6.4053
5 - entry Read (2048-bit key)	22.3381	22.0744

Table 7.1: Π_{UUHD} execution times in seconds.

Table 7.1 lists the execution times of the interfaces of the protocol, in seconds, evaluated against N and the key length of the Paillier encryption scheme. In a first update, the public parameters of all building blocks are computed, and the database is set up by computing vc. In the second row of Table 7.1, we show the cost of just computing vc, which is virtually the same as that of computing an opening w_i, and these costs are very small. In a 1-entry update, one database entry is modified and vc is updated, and the cost of updating an opening w_i is virtually the same. As can be seen, the cost of the first update grows linearly with N, as does the cost of setting up vc or w_i, whereas the cost of updating vc or w_i is very small and independent of N. The timings for the read interface depend greatly upon the security parameters for the Paillier encryption scheme, and also increase linearly with the number of entries read $|\mathbb{S}|$. However, the timings are independent of N.

We note that the cost of setting up vc or w_i is constant when the database is initialized to 0 in every position, which is the case in our PPLP protocol.

7.7 VARIANTS

$\mathcal{F}_{\text{UUHD}}$ can be modified to store a database DB of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$, i.e., a database where a tuple of values is stored in each entry. In the uuhd.update interface, $\mathcal{U}$ sends $(i, vu_{i,1}, \ldots, vu_{i,m})_{\forall i \in [1,N]}$, and each value $vu_{i,j}$ ($j \in [1, m]$) can be updated or not independently of other values in the same entry. In the uuhd.read interface, $\mathcal{R}_k$ sends $(i, vr_{i,1}, \ldots, vr_{i,m})_{i \in \mathbb{S}}$ along with commitments and openings to the position and values of each of the entries read, i.e., all the values in an entry are read. The position $j \in [1, m]$ of each value $vr_{i,j}$ is not hidden from $\mathcal{U}$. This variant of $\mathcal{F}_{\text{UUHD}}$ is useful for protocols where a party needs to read a tuple of values and prove that they are stored in the same entry and that each $vr_{i,j}$ is stored at a certain position j within the entry. To construct this variant, vc commits to a vector $\mathbf{x}$ of length $N \times m$ such that $\mathbf{x}[(i-1)m + j] = vr_{i,j}$ for all $i \in [1, N]$ and $j \in [1, m]$. In the update phase, each vector component can be updated independently of others regardless of whether they belong to the same database entry or not. To read the database entry i, $\mathcal{R}_k$ needs to compute openings $(w_{(i-1)m+1}, \ldots, w_{im})$ to open the positions $[(i-1)m+1, im]$ of the committed vector $\mathbf{x}$. $\mathcal{R}_k$ must also prove that those positions belong to the database entry i. To this end, the relation R_r is modified to involve a witness $wit_r \leftarrow (sig, vc, com,$

$r_2, open_2, \langle i, copen_i, \{w_{(i-1)m+j}, vr_{i,j}, or_{i,j}\}_{\forall j \in [1,m]}\rangle_{i \in \mathbb{S}})$ and $ins_r \leftarrow (pk, par, par_c, vc', com', parcom, \langle ccom'_i, \{ccomr'_j\}_{\forall j \in [1,m]}\rangle_{i \in \mathbb{S}})$.

$$
\begin{aligned}
R_r = \{ & (wit_r, ins_r) : \\
& 1 = \mathsf{VfSig}(pk, sig, \langle vc, com\rangle) \wedge \\
& vc' = \mathsf{VC.Rerand}(vc, r_2) \wedge \\
& com' = \mathsf{CRerand}(com, open_2) \wedge \\
& \{1 = \mathsf{COM.Verify}(parcom, ccom'_i, i, copen_i) \wedge \\
& \{1 = \mathsf{COM.Verify}(parcom, ccomr'_j, vr_{i,j}, or_{i,j}) \wedge \\
& 1 = \mathsf{VC.Verify}(par, vc, vr_{i,j}, (i-1)m + j, w_{(i-1)m+j})\}_{\forall j \in [1,m]} \}_{\forall i \in \mathbb{S}} \}
\end{aligned}
$$

It is worth noting that $\mathcal{F}_{\text{UUHD}}$ cannot be modified so that it authenticates readers towards the updater. If readers were authenticated, then the updater would be able to track all the changes performed on each of the databases, and thus the contents of the databases would not be hidden from the updater.

8

UPDATABLE DATABASES

Joint work with Alfredo Rial. This primitive was published in a paper presented at AFRICACRYPT in 2020 [38]. The primitive finds use as a building block in the protocol in Chapter 11. Reproduced with permission from Springer Nature.

We define UD as a two-party task between a reader $\mathcal{R}$ and an updater $\mathcal{U}$. $\mathcal{U}$ sets a database DB and may update it at any time during protocol execution. DB consists of N entries of the form $[i, vr_i]$ ($\forall i \in [1, N]$), where i is the position and vr_i is the value stored at that position. Both $\mathcal{R}$ and $\mathcal{U}$ know the content of DB. $\mathcal{R}$ reads in ZK an entry $[i, vr_i]$ from DB. $\mathcal{F}_{\text{UD}}$ ensures that it is not possible to falsely prove that $[i, vr_i]$ is stored in DB. $\mathcal{F}_{\text{UD}}$ stores a counter cr for $\mathcal{R}$ and a counter cu for $\mathcal{U}$. These counters are used to check that $\mathcal{R}$ and $\mathcal{U}$ have the same version of DB. When $\mathcal{U}$ initiates the ud.update interface, cu is incremented. When $\mathcal{F}_{\text{UD}}$ sends the update to $\mathcal{R}$, $\mathcal{F}_{\text{UD}}$ checks that $cu = cr + 1$ and then increments cr. In the ud.read interface, $\mathcal{F}_{\text{UD}}$ checks that the counters of both parties are equal, which ensures that they have the same DB.

8.1 OPERATIONS

UPDATE. During an *update* operation, $\mathcal{U}$ updates an entry in a database DB.

READ. During a *read* operation, $\mathcal{R}$ proves knowledge of an entry in the database, whilst hiding the entry being read and its position from $\mathcal{U}$.

8.2 SECURITY PROPERTIES

ZERO-KNOWLEDGE. A read operation does not reveal the database entry read to $\mathcal{U}$.

UNFORGEABILITY. $\mathcal{R}_k$ cannot read an entry if that entry was not stored in DB by $\mathcal{U}$.

8.3 IDEAL FUNCTIONALITY

Figure 8.1: Ideal Functionality $\mathcal{F}_{\text{UD}}$

$\mathcal{F}_{\text{UD}}$ is parameterized by a universe of values $\mathbb{U}_v$ and by a database size N.

1. On input $(\text{ud.update.ini}, sid, (i, vu_i)_{\forall i \in [1,N]})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{R}, \mathcal{U}, sid')$.
   - For all $i \in [1, N]$, abort if $vu_i \notin \mathbb{U}_v$.
   - If (sid, DB, cu) is not stored:

  - For all $i \in [1, N]$, abort if $vu_i = \perp$.
  - Set $\mathsf{DB} \leftarrow (i, vu_i)_{\forall i \in [1,N]}$ and $cu \leftarrow 0$ and store (sid, DB, cu).
- Else:
  - For all $i \in [1, N]$, if $vu_i \neq \perp$, update DB with $[i, vu_i]$.
  - Increment cu and update DB and cu in (sid, DB, cu).
- Create a fresh qid and store $(qid, (i, vu_i)_{\forall i \in [1,N]}, cu)$.
- Send $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i \in [1,N]})$ to $\mathcal{S}$.

S. On input $(\mathsf{ud.update.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid', (i, vu_i)_{\forall i \in [1,N]}, cu')$ such that $qid = qid'$ is not stored.
- If (sid, DB, cr) is not stored, set $\mathsf{DB} \leftarrow (i, vu_i)_{\forall i \in [1,N]}$ and $cr \leftarrow 0$ and store (sid, DB, cr).
- Else:
  - Abort if $cu' \neq cr + 1$.
  - For all $i \in [1, N]$, if $vu_i \neq \perp$, update DB with $[i, vu_i]$.
  - Increment cr and update cr and DB in (sid, DB, cr).
- Delete the record $(qid, (i, vu_i)_{\forall i \in [1,N]}, cu')$.
- Send $(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i \in [1,N]})$ to $\mathcal{R}$.

2. On input $(\mathsf{ud.read.ini}, sid, i, vr_i, com_i, open_i, comr_i, openr_i)$ from $\mathcal{R}$:

- Abort if (sid, DB, cr) is not stored.
- Abort if $i \notin [1, N]$, or if $vr_i \notin \mathbb{U}_v$, or if $[i, vr_i] \notin \mathsf{DB}$.
- Parse the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
- Parse the commitment $comr_i$ as $(comr'_i, parcom, \mathsf{COM.Verify})$.
- Abort if $\mathsf{COM.Verify}$ is not a ppt algorithm.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, comr'_i, vr_i, openr_i)$.
- Create a fresh qid and store $(qid, com_i, comr_i, cr)$.
- Send $(\mathsf{ud.read.sim}, sid, qid, com_i, comr_i)$ to $\mathcal{S}$.

S. On input $(\mathsf{ud.read.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid', com_i, comr_i, cr')$ such that $qid = qid'$ is not stored, or if $cr' \neq cu$, where cu is in (sid, DB, cu).
- Delete the record $(qid, com_i, comr_i, cr')$.
- Send $(\mathsf{ud.read.end}, sid, com_i, comr_i)$ to $\mathcal{U}$.

8.4 CONSTRUCTION

We propose a construction Π_{UD} for $\mathcal{F}_{\mathrm{UD}}$. Π_{UD} is based on non-hiding vector commitments (NHVC) [32, 69] (see Section 2.7.2). An NHVC scheme allows us to compute a commitment vc to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[N])$. To open the value $\mathbf{x}[i]$ to committed at position i, an opening w_i is computed. The size of w_i is independent of N.

Π_{UD} works as follows: $\mathcal{U}$ sends a database DB to $\mathcal{R}$, and both $\mathcal{U}$ and $\mathcal{R}$ map DB to a vector $\mathbf{x}$ and compute a commitment vc to $\mathbf{x}$. To update an entry $[i, vr_i]$ to $[i, vr_i']$, $\mathcal{U}$ sends $[i, vr_i']$ to $\mathcal{R}$, and both $\mathcal{U}$ and $\mathcal{R}$ update vc to obtain a commitment vc' to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = vr_i'$, while the other positions remain unchanged. Therefore, updates do not need any revocation mechanism. To prove in ZK that an entry $[i, vr_i]$ is in DB, a party computes an opening w_i for position i and computes a ZK proof of knowledge of (w_i, i, vr_i) that proves that $\mathbf{x}[i] = vr_i$.

We describe the interfaces of Π_{UD} in more detail:

1. In the ud.update interface, $\mathcal{U}$ uses $\mathcal{F}_{\mathrm{AUT}}$ (see Section 2.8.3) to send to $\mathcal{R}$ the update $(i, vu_i)_{\forall i \in [1,N]}$. In the first execution of this interface, $\mathcal{U}$ and $\mathcal{R}$ run VC.Commit to commit to $(i, vu_i)_{\forall i \in [1,N]}$. The functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ (see Section 2.8.1) for a common reference string is used to compute the parameters of the NHVC scheme. In the following executions, $\mathcal{U}$ and $\mathcal{R}$ update vc by using VC.ComUpd. If $\mathcal{R}$ already stores openings w_i, $\mathcal{R}$ runs VC.WitUpd to update them.

2. In the ud.read interface, $\mathcal{R}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R}$ (see Section 2.8.8) to prove that com_i and $comr_i$ commit to a position i and a value vr_i such that $\mathbf{x}[i] = vr_i$, where $\mathbf{x}$ is the vector committed in vc. The witness of R includes an opening w_i. $\mathcal{R}$ runs VC.Prove to compute it if it is not stored.

Figure 8.2: Description of Π_{UD}

N denotes the database size. The universe of values $\mathbb{U}_v$ is given by the message space of the NHVC scheme.

1. On input $(\mathsf{ud.update.ini}, sid, (i, vu_i)_{\forall i \in [1,N]})$:
   - If $(sid, par, vc, \mathbf{x}, cu)$ is not stored:
     - $\mathcal{U}$ uses crs.get to obtain the parameters par from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$. To compute par, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ runs $\mathsf{VC.Setup}(1^k, N)$.
     - $\mathcal{U}$ initializes a counter $cu \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for all $i \in [1, N]$. $\mathcal{U}$ runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$ and stores $(sid, par, vc, \mathbf{x}, cu)$.
   - Else:
     - $\mathcal{U}$ sets $cu' \leftarrow cu + 1$, $\mathbf{x}' \leftarrow \mathbf{x}$ and $vc' \leftarrow vc$. For all $i \in [1, N]$ such that $vu_i \neq \perp$, $\mathcal{U}$ computes $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc', i, \mathbf{x}'[i], vu_i)$ and $\mathbf{x}'[i] \leftarrow vu_i$.
     - $\mathcal{U}$ replaces the stored tuple $(sid, par, vc, \mathbf{x}, cu)$ by $(sid, par, vc', \mathbf{x}', cu')$.

- $\mathcal{U}$ uses aut.send to send the message $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$ to $\mathcal{R}$.
- If $(sid, par, vc, \mathbf{x}, cr)$ is stored and $cu' \neq cr + 1$, $\mathcal{R}$ aborts.
- For $j = 1$ to N, if (sid, j, w_j) is stored, $\mathcal{R}$ sets $w'_j \leftarrow w_j$ and, for all $i \in [1, N]$ such that $vu_i \neq \perp$, $w'_j \leftarrow \mathsf{VC.WitUpd}(par, w'_j, j, i, \mathbf{x}[i], vu_i)$. $\mathcal{R}$ replaces (sid, j, w_j) by (sid, j, w'_j).
- $\mathcal{R}$ performs the same operations as $\mathcal{U}$ to set or update a tuple $(sid, par, vc, \mathbf{x}, cr)$.
- $\mathcal{R}$ outputs $(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i \in [1,N]})$.

2. On input $(\mathsf{ud.read.ini}, sid, i, vr_i, com_i, open_i, comr_i, openr_i)$:
   - $\mathcal{R}$ parses com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
   - $\mathcal{R}$ parses $comr_i$ as $(comr'_i, parcom, \mathsf{COM.Verify})$.
   - $\mathcal{R}$ aborts if COM.Verify is not a ppt algorithm.
   - $\mathcal{R}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
   - $\mathcal{R}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, comr'_i, vr_i, openr_i)$.
   - $\mathcal{R}$ takes the stored tuple $(sid, par, vc, \mathbf{x}, cr)$ and aborts if $\mathbf{x}[i] \neq vr_i$.
   - If (sid, i, w_i) is not stored, $\mathcal{R}$ runs $w_i \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and stores (sid, i, w_i).

The relation R is

$$
\begin{aligned}
R = \{ (wit, ins) : \\
1 = \mathsf{COM.Verify}(parcom, com'_i, i, open_i) \wedge \\
1 = \mathsf{COM.Verify}(parcom, comr'_i, vr_i, openr_i) \wedge \\
1 = \mathsf{VC.Verify}(par, vc, vr_i, i, w_i) \}
\end{aligned}
$$

- $\mathcal{R}$ sets the witness $wit \leftarrow (w_i, i, open_i, vr_i, openr_i)$ and the instance $ins \leftarrow (par, vc, parcom, com'_i, comr'_i, cr)$. $\mathcal{R}$ uses zk.prove to send wit and ins to $\mathcal{F}_{\mathrm{ZK}}^{R}$.
- $\mathcal{U}$ receives $ins = (par', vc', parcom, com'_i, comr'_i, cr)$ from $\mathcal{F}_{\mathrm{ZK}}^{R}$.
- $\mathcal{U}$ takes the stored tuple $(sid, par, vc, \mathbf{x}, cu)$ and aborts if $cr \neq cu$, or if $par' \neq par$, or if $vc' \neq vc$.
- $\mathcal{U}$ sets the commitments $com_i \leftarrow (com'_i, parcom, \mathsf{COM.Verify})$ and $comr_i \leftarrow (comr'_i, parcom, \mathsf{COM.Verify})$. (COM.Verify is part of the description of R.)
- $\mathcal{U}$ outputs $(\mathsf{ud.read.end}, sid, com_i, comr_i)$.

8.5 SECURITY ANALYSIS

Theorem 8.5.1 *Π_{UD} securely realizes $\mathcal{F}_{\text{UD}}$ in the ($\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{ZK}}^{R}$)-hybrid model if the NHVC scheme is binding.*

The proof of Theorem 8.5.1 is described in Section A.3.

8.6 INSTANTIATION AND EFFICIENCY ANALYSIS

To instantiate Π_{UD}, we use an NHVC scheme secure under the ℓ-DHE assumption [69], as described in Section 2.7.2.1.

8.6.1 *Commitment Scheme*

We use the Pedersen commitment scheme [80] as described in Section 7.6.

8.6.2 *Signature Scheme*

We use the structure-preserving signature (SPS) scheme in [2], as described in Section 7.6.

8.6.3 *ZK Functionality*

To instantiate $\mathcal{F}_{\text{ZK}}^{R}$, we use the scheme in [24] as described in Section 2.8.8.1.

8.6.3.1 *UC ZK Proof for the Read Relation*

To instantiate $\mathcal{F}_{\text{ZK}}^{R_r}$ with the protocol in [24], we need to instantiate R_r with our chosen NHVC, commitment and signature schemes. Then we need to express R_r following the notation for UC ZK proofs described above.

In R, we need to prove that the position i committed to in com_i' equals the position opened in the NHVC vc via the verification equation $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$. In our NHVC scheme, α is secret, which makes the relation between $\tilde{g}_i = \tilde{g}^{\alpha^i}$ and i not efficiently provable. To solve this problem, the public parameters are extended with SPSs that bind g^i with $\tilde{g}_i$. Given the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$, and the key pair (sk, pk), for $i \in [1, \ell]$, $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ computes $sig_i \leftarrow \mathsf{Sign}(sk, \langle g^{sid}, \tilde{g}_i, \tilde{g}^i \rangle)$, where sid is the session identifier.

Let (U_1, U_2, V, W_1, Z) be the public key of the signature scheme. Let (R, S, T) be a signature on $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Let (g, h) be the parameters of the Pedersen commitment scheme. R involves proofs about secret bases, and we use the transformation described above for those proofs. The base h is also used to randomize secret

We note that, in many practical settings, $\mathcal{U}$ can compute the parameters and signatures. We remark that these signatures do not need to be updated when the database is updated.

bases in $\mathbb{G}$, and another base $\tilde{h} \leftarrow \tilde{\mathbb{G}}$ is added to randomize bases in $\tilde{\mathbb{G}}$. Following the notation in [24], we describe the proof as follows.

$$
\begin{aligned}
&\nwarrow i, open_i, v, openr_i, \tilde{g}_i, w, R, S, T : \\
&\quad com_i = g^i h^{open_i} \ \wedge \ comr_i = g^v h^{openr_i} \ \wedge && (8.1)\\
&\quad e(R,V)e(S,\tilde{g})e(g^{sid},W_1)e(g,Z)^{-1} = 1 \ \wedge && (8.2)\\
&\quad e(R,T)e(U_1,\tilde{g}_i)e(U_2,\tilde{g})^i e(g,\tilde{g})^{-1} = 1 \ \wedge && (8.3)\\
&\quad e(vc,\tilde{g}_i)^{-1} e(w,\tilde{g})e(g_1,\tilde{g}_\ell)^v = 1 && (8.4)
\end{aligned}
$$

Equation 8.1 proves knowledge of the openings of the Pedersen commitments com_i and $comr_i$. Equation 8.2 and Equation 8.3 prove knowledge of a signature (R, S, T) on a message $\langle g^{sid}, \tilde{g}_i, \tilde{g}^i \rangle$. Equation 8.4 proves that the value v in $comr_i$ is equal to the value committed in the position i of the vector commitment vc.

8.6.4 *Efficiency Analysis*

We analyse the storage, communication, and computation costs of our instantiation of Π_{UD}.

8.6.4.1 *Storage Costs*

$\mathcal{R}$ and $\mathcal{U}$ store a common reference string, whose size grows linearly with N. Throughout protocol execution, $\mathcal{R}$ and $\mathcal{U}$ also store the last updated values of vc and the committed vector. $\mathcal{R}$ stores the openings w_i. In conclusion, the storage cost is linear in N.

8.6.4.2 *Communication Costs*

In the ud.update interface, $\mathcal{U}$ sends $(i, vu_i)_{\forall i \in [1,N]}$ to $\mathcal{R}$. The communication cost is linear in the number of entries updated, except for the first update in which all entries must be initialized. In the ud.read interface, $\mathcal{R}$ sends an instance and a ZK proof to $\mathcal{U}$. The size of the witness and of the instance is constant and independent of N. Therefore, the communication cost of the proof is constant. In conclusion, after the first update phase, the communication cost does not depend on N.

8.6.4.3 *Computation Costs*

In the ud.update interface, $\mathcal{U}$ and $\mathcal{R}$ update vc with cost linear in the number of updates (except for the first update where all positions are initialized). $\mathcal{R}$ also updates the stored openings w_i with cost linear in the number of updates. In the ud.read interface, if w_i is not stored, $\mathcal{R}$ computes it with cost that grows linearly with N. However, if w_i is stored, the computation cost of the proof is constant and independent of N.

We note that it is possible to defer opening updates to the ud.read interface, so as to only update openings that are actually needed to compute ZK proofs. Thanks to that, the computation cost in the ud.update interface is constant. In the ud.read interface, if w_i is stored but needs to be updated, the computation cost grows linearly with the number of updates but is independent of N. The only

INTERFACE	$N = 100$	$N = 1000$
First Update	0.6844	5.9952
Computation of vc or w_i	0.0032	0.03787
1 - entry update of vc or w_i	0.0001	0.0001
Read	0.7496	0.7545

Table 8.1: Π_{UD} execution times in seconds (1024-bit key).

overhead introduced by deferring opening updates is the need to store the tuples $(i, vu_i)_{\forall i \in [1,N]}$ sent by $\mathcal{U}$.

In summary, after initializing vc and the openings w_i, the communication and computation costs are independent of N, which makes our instantiation of Π_{UD} practical for large databases.

8.6.5 *Implementation and Efficiency Measurements*

We have implemented our instantiation of Π_{UD} in the Python programming language, using the Charm cryptographic framework [4], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve was used for the pairing group setup.

To compute the UC ZK proofs for R, we use the compiler in [24]. The public parameters of the proof system contain a public key of the Paillier encryption scheme, the parameters for a multi-integer commitment scheme and the specification of a DSA group. (We refer the reader to [24] for a description of how these primitives are used in the compiler.) The cost of a proof depends on the number of elements in the witness and of the number of equations composed by Boolean ANDs. The computation cost for the prover of a Σ-protocol for R involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [24] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the values in the witness, an evaluation of a Paillier encryption for each of the values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for the verifier, as well as the communication cost, also depends on the number of values in the witness and on the number of equations. Therefore, as the number of values in the witness and of equations is independent of N in our proof for relation R, the computation and communication costs of our proof do not depend on N.

Tables 8.1 and 8.2 list the execution times of the update and read interfaces of the protocol, in seconds. The execution times of the interfaces of the protocol have been evaluated against the size N of the database, and against the security parameter of the Paillier encryption algorithm.

In a first update, the public parameters of all the building blocks are computed, and the database is set up by computing vc. In the second row of Tables 8.1 and 8.2, we show the cost of just computing vc, which is virtually the same as that of computing an opening w_i. The computation time of vc and w_i is very small. In a 1-entry update, one database entry is modified and vc is updated. The cost of updating

INTERFACE	$N = 100$	$N = 1000$
Interface	$N = 100$	$N = 1000$
First update	0.7940	6.0822
Computation of vc or w_i	0.0032	0.03787
1-entry update of vc or w_i	0.0001	0.0001
Read	3.8945	3.5911

Table 8.2: Π_{UD} execution times in seconds (2048-bit key).

an opening w_i is virtually the same. As can be seen, the cost of the first update grows linearly with the size N of the database, as does the cost of setting up vc or w_i, whereas the cost of updating vc or w_i is very small and independent of N. The execution times for the read interface depend greatly upon the security parameters for the Paillier encryption scheme. However, the execution time is independent of the database size N.

8.7 VARIANTS

- It is straightforward to modify the ud.read interface of $\mathcal{F}_{\text{UD}}$ to allow $\mathcal{R}$ to read a tuple $(i, vr_i, com_i, open_i, comr_i, openr_i)_{\forall i \in \mathbb{S}}$ ($\mathbb{S} \subseteq [1, N]$) of database entries simultaneously. This variant of $\mathcal{F}_{\text{UD}}$ allows us to reduce communication rounds when a party in a protocol that uses $\mathcal{F}_{\text{UD}}$ needs to read more than one value simultaneously, e.g. a buyer that purchases several items at once and reads the prices of those items from the database.

- $\mathcal{F}_{\text{UD}}$ can also be modified to store a database DB of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$, i.e., a database where a tuple of values is stored in each entry. In the ud.update interface, $\mathcal{U}$ sends $(i, vu_{i,1}, \ldots, vu_{i,m})_{\forall i \in [1,N]}$, and each value $vu_{i,j}$ ($j \in [1, m]$) can be updated or not independently of other values in the same entry. In the ud.read interface, $\mathcal{R}$ sends $(i, vr_{i,1}, \ldots, vr_{i,m})$ along with commitments and openings to the position and values, i.e., all the values in an entry are read. The position $j \in [1, m]$ of each value $vr_{i,j}$ is not hidden from $\mathcal{U}$. This variant of $\mathcal{F}_{\text{UD}}$ is useful for protocols where a party needs to read a tuple of values and prove that they are stored in the same entry and that each $vr_{i,j}$ is stored at a certain position j within the entry, e.g. a user that consumes some utility and reads a pricing function that is represented by a tuple of values.

- $\mathcal{F}_{\text{UD}}$ can also be modified to interact with two parties such that both of them can read and update the database, or such that a party reads and updates and the other party receives read and update operations.

8.8 APPLICATIONS

Consider the following relation R':

$$R' = \{(wit, ins) : [i, v] \in \mathsf{DB} \wedge 1 = \text{pred}_i(i) \wedge 1 = \text{pred}_v(v)\}$$

where the witness is $wit = (i, v)$ and the instance is $ins = \mathsf{DB}$. pred_i and pred_v represent predicates that i and v must fulfil, e.g., predicates that require i and v to belong to a range or set of values.

We would like to construct a ZK protocol for R' that separates each of the equations of R'. We show how this protocol is constructed by using $\mathcal{F}_{\mathrm{UD}}$ and $\mathcal{F}_{\mathrm{NIC}}$ as building blocks, along with the functionalities $\mathcal{F}_{\mathrm{ZK}}^{R_i}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_v}$.

1. On input DB, the verifier uses the ud.update interface to send DB to $\mathcal{F}_{\mathrm{UD}}$, which sends DB to the prover.

2. On input (i, v), the prover checks if $[i, v] \in \mathsf{DB}$.

3. The prover runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$. The prover uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ on input i to obtain a commitment com_i with opening $open_i$. Similarly, the prover obtains from $\mathcal{F}_{\mathrm{NIC}}$ a commitment com_v to v with opening $open_v$.

4. The prover uses ud.read to send $(i, v, com_i, open_i, com_v, open_v)$ to $\mathcal{F}_{\mathrm{UD}}$. $\mathcal{F}_{\mathrm{UD}}$ sends com_i and com_v to the verifier.

5. The verifier runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$. The verifier uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate the commitments com_i and com_v. Then the verifier stores com_i and com_v and sends a message to the prover to acknowledge receipt of the commitments.

6. The prover parses the commitment com_i as $(com_i', parcom, \mathsf{COM.Verify})$. The prover sets the witness $wit \leftarrow (i, open_i)$ and the instance $ins \leftarrow (parcom, com_i')$. The prover uses the zk.prove interface to send wit and ins to $\mathcal{F}_{\mathrm{ZK}}^{R_i}$, where R_i is

$$R_i = \{(wit, ins) : 1 = \mathsf{COM.Verify}(parcom, com_i', i, open_i) \ \wedge \ 1 = \mathrm{pred}_i(i)\}$$

7. The verifier receives ins from $\mathcal{F}_{\mathrm{ZK}}^{R_i}$. The verifier checks if the commitment in ins is equal to the stored commitment com_i. If they are equal, the binding property guaranteed by $\mathcal{F}_{\mathrm{NIC}}$ ensures that $\mathcal{F}_{\mathrm{UD}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_i}$ have also received the same position i as input.

8. The last two steps are replicated to prove that v fulfils $1 = \mathrm{pred}_v(v)$ by using $\mathcal{F}_{\mathrm{ZK}}^{R_v}$.

We believe that a modular design has two main advantages: firstly, it allows for simpler security analysis. A security proof of a protocol described in the hybrid model is much simpler than a proof that requires reductions to the security properties of different cryptographic primitives. Moreover, each of the building blocks realizes a simpler task and thus requires a simpler protocol with a less involved security analysis. Secondly, it facilitates the study in isolation of how to create efficient and secure ZK data structures. Namely, different constructions for $\mathcal{F}_{\mathrm{UD}}$ can easily be compared in terms of security and efficiency.

APPLICATIONS TO POT. The Priced Oblivious Transfer (POT) protocols in [18, 85] are based on previously proposed Oblivious Transfer (OT) protocols. However, they do not use OT as a building block. Instead, the OT protocol is modified ad-hoc to create the POT protocol, and its security has to be reanalysed when analysing the security of the POT protocol.

$\mathcal{F}_{\text{UD}}$ can be used to design a POT protocol modularly. The database DB consists of entries $[i, p_i]$, where p_i is the price to be paid for message m_i. To purchase m_i, the buyer uses the ud.read interface of $\mathcal{F}_{\text{UD}}$ to read the entry $[i, p_i]$. The provider receives the commitments c_i to i and cr_i to p_i. cr_i is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_v}$ where the buyer proves that he subtracts the price p_i from his account. c_i is used as input to a functionality for oblivious transfer (modified to receive committed inputs as described in [20]) to allow the buyer to retrieve m_i.

Therefore, $\mathcal{F}_{\text{UD}}$ allows for the design of a POT protocol that uses a functionality for OT as building block. Thanks to that, the POT protocol can be instantiated with multiple OT schemes and their security does not need to be reanalysed. Moreover, $\mathcal{F}_{\text{UD}}$ allows the provider to update prices at any time.

APPLICATIONS TO PPB. In the Privacy Preserving Billing (PPB) protocols in [83, 84], a meter reading comprises a consumption c and the time interval i of consumption. A tariff policy associates a different function $p = f_i(c)$ to each time interval (and possibly to each consumption interval). $\mathcal{F}_{\text{UD}}$ can be used to design a PPB protocol modularly, where the database DB consists of entries $[i, f_i]$. The PPB protocol works as follows:

- The meter outputs a signed meter reading (c, i).
- The user reads $[i, f_i]$ via $\mathcal{F}_{\text{UD}}$, and the provider receives commitments c_i to i and cr_i to f_i.
- c_i is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_i}$ to prove that i equals the value signed in the meter reading.
- cr_i is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_v}$ to prove that $p = f_i(c)$.

If f_i is represented by a tuple of values (e.g. the coefficients of a polynomial) the variant of $\mathcal{F}_{\text{UD}}$ for databases of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$ should be used. If the formula f_i also changes with the consumption interval, the database can also store the minimum and maximum values of the consumption interval to allow the user to prove that she uses the right formula. Using $\mathcal{F}_{\text{UD}}$ allows for the design of PPB protocols modularly and allows the provider to modify the pricing policies efficiently and at any time.

9

UNLINKABLE UPDATABLE DATABASES

Joint work with Alfredo Rial. This primitive was published in a paper presented at the 25th Australasian Conference on Information Security and Privacy in 2020 [39]. The primitive finds use as a building block in the protocol in Chapter 12. Reproduced with permission from Springer Nature.

We define UUD as a task between multiple readers $\mathcal{R}_k$ and an updater $\mathcal{U}$. $\mathcal{U}$ sets a database DB and may update it at any time during protocol execution. DB consists of N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$, where i identifies the database entry and $(v_{i,1}, \ldots, v_{i,L})$ are the values stored in this entry. Any $\mathcal{R}_k$ and $\mathcal{U}$ know the content of DB. A reader $\mathcal{R}_k$ can read DB by computing a zero-knowledge (ZK) proof of knowledge of an entry $[i, v_{i,1}, \ldots, v_{i,L}]$. $\mathcal{F}_{\text{UUD}}$ hides from $\mathcal{U}$ which entry was read but ensures that it is not possible to falsely prove that an entry is stored in DB. $\mathcal{F}_{\text{UUD}}$ allows $\mathcal{R}_k$ to remain pseudonymous and unlinkable when reading DB.

$\mathcal{F}_{\text{UUD}}$ stores counters cr_k for $\mathcal{R}_k$ and a counter cu for $\mathcal{U}$. These counters are used to check that $\mathcal{R}_k$ has the last version of DB. When $\mathcal{U}$ sends an update, cu is incremented. When $\mathcal{R}_k$ receives DB, $\mathcal{F}_{\text{UUD}}$ sets $cr_k \leftarrow cu$. When $\mathcal{R}_k$ reads DB, $\mathcal{F}_{\text{UUD}}$ checks that $cr_k = cu$, which ensures that $\mathcal{R}_k$ and $\mathcal{U}$ have the same DB.

9.1 OPERATIONS

UPDATE. During an *update* operation, $\mathcal{U}$ updates an entry in a database DB.

READ. During a *read* operation, $\mathcal{R}_k$ proves knowledge of an entry in the database, whilst hiding the entry being read, its position, and the identity of $\mathcal{R}_k$ from $\mathcal{U}$.

9.2 SECURITY PROPERTIES

UNLINKABILITY. $\mathcal{F}_{\text{UUD}}$ reveals to $\mathcal{U}$ a pseudonym P rather than the identifier $\mathcal{R}_k$. $\mathcal{R}_k$ can choose different random pseudonyms so that read operations are unlinkable.

ZERO-KNOWLEDGE. A read operation does not reveal the database entry read to $\mathcal{U}$.

UNFORGEABILITY. $\mathcal{R}_k$ cannot read an entry if that entry was not stored in DB by $\mathcal{U}$.

9.3 IDEAL FUNCTIONALITY

Figure 9.1: Ideal Functionality $\mathcal{F}_{\text{UUD}}$

$\mathcal{F}_{\text{UUD}}$ is parameterized by a universe of pseudonyms $\mathbb{U}_p$, a universe of values $\mathbb{U}_v$ and by a database size N.

1. On input $(\mathsf{uud.update.ini}, sid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{U}, sid')$.
   - For all $i \in [1, N]$ and $j \in [1, L]$, abort if $v_{i,j} \notin \mathbb{U}_v$.
   - If (sid, DB, cu) is not stored:
     - For all $i \in [1, N]$ and $j \in [1, L]$, abort if $v_{i,j} = \perp$.
     - Set $\mathsf{DB} \leftarrow (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ and $cu \leftarrow 0$ and store (sid, DB, cu).
   - Else:
     - For all $i \in [1, N]$ and $j \in [1, L]$, if $v_{i,j} \neq \perp$, update DB by storing $v_{i,j}$ at position j of entry i.
     - Increment cu and update DB and cu in (sid, DB, cu).
   - Create a fresh qid and store qid.
   - Send $(\mathsf{uud.update.sim}, sid, qid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$ to $\mathcal{S}$.

S. On input $(\mathsf{uud.update.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if qid is not stored.
   - Delete qid.
   - Send $(\mathsf{uud.update.end}, sid)$ to $\mathcal{U}$.

2. On input $(\mathsf{uud.getdb.ini}, sid)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{uud.getdb.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{uud.getdb.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
   - If (sid, DB, cu) is not stored, set $\mathsf{DB} \leftarrow \perp$.
   - Else, set $cr_k \leftarrow cu$, store $(\mathcal{R}_k, \mathsf{DB}, cr_k)$ and delete any previous tuple $(\mathcal{R}_k, \mathsf{DB}', cr'_k)$.
   - Delete $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{uud.getdb.end}, sid, \mathsf{DB})$ to $\mathcal{R}_k$.

3. On input $(\mathsf{uud.read.ini}, sid, P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]}))$ from $\mathcal{R}_k$:

- Abort if $P \notin \mathbb{U}_p$, or if $[i, v_{i,1}, \ldots, v_{i,L}] \notin \mathsf{DB}$, or if $(\mathcal{R}_k, \mathsf{DB}, cr_k)$ is not stored.
- Parse the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
- For all $j \in [1, L]$:
  - Parse the commitment $com_{i,j}$ as $(com'_{i,j}, parcom, \mathsf{COM.Verify})$.
  - Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j})$.
- Create a fresh qid and store $(qid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr_k)$.
- Send $(\mathsf{uud.read.sim}, sid, qid, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{S}$.

S. On input $(\mathsf{uud.read.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid', P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr'_k)$ such that $qid' = qid$ is not stored or if $cr'_k \neq cu$, where cu is in (sid, DB, cu).
- Delete the record $(qid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr'_k)$.
- Send $(\mathsf{uud.read.end}, sid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{U}$.

9.4 CONSTRUCTION

We describe a construction Π_{UUD} for $\mathcal{F}_{\mathrm{UUD}}$. Π_{UUD} is based on subvector commitments (SVC) [64] (see Section 2.7.3), which we extend with a UC ZK proof of knowledge of a subvector. An SVC scheme allows us to compute a commitment svc to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[N])$. svc can be opened to a subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $I = \{i_1, \ldots, i_n\} \subseteq [1, N]$. The size of the opening w_I is independent of N and of $|I|$. SVC were recently proposed as an improvement of vector commitments [32, 69], where the size of w_I is independent of N but dependent on $|I|$. In Section 2.7.3, we extend the definition of SVC in [64] to include algorithms to update commitments and openings when part of the vector is updated.

Π_{UUD} works as follows: $\mathcal{U}$ uses a bulletin board BB to publish the database DB and any $\mathcal{R}_k$ obtains DB from BB. A BB ensures that all readers obtain the same version of DB, which we need to guarantee unlinkability. Both $\mathcal{U}$ and any $\mathcal{R}_k$ map a DB with N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$ to a vector $\mathbf{x}$ of length $N \times L$ such that $\mathbf{x}[(i-1)L + j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$, and they compute a commitment svc to $\mathbf{x}$. To update a database entry, $\mathcal{U}$ updates BB, and $\mathcal{U}$ and any $\mathcal{R}_k$ update svc. Therefore, updates do not need any revocation mechanism. To prove in ZK that an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ is in DB, $\mathcal{R}_k$ computes an opening w_I for $I = \{(i-1)L+1, \ldots, (i-1)L+L\}$ and uses it to compute a

ZK proof of knowledge of the subvector $(\mathbf{x}[(i-1)L+1], \ldots, \mathbf{x}[(i-1)L+L])$. This proof guarantees that I is the correct set for index i.

We describe in more detail each of the interfaces of Π_{UD}:

1. In the uud.update interface, $\mathcal{U}$ uses $\mathcal{F}_{\mathrm{BB}}$ (see Section 2.8.7) to publish the DB and to update it. When DB is published for the first time, $\mathcal{U}$ runs SVC.Commit to commit to DB. The functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}$ (see Section 2.8.1) for a common reference string is used to compute the parameters of the SVC scheme. When DB is updated, $\mathcal{U}$ updates svc by using SVC.ComUpd.

2. In the uud.getdb interface, any $\mathcal{R}_k$ retrieves DB and its subsequent updates through $\mathcal{F}_{\mathrm{BB}}$. $\mathcal{R}_k$ follows the same steps as $\mathcal{U}$ to compute and update svc. Additionally, if $\mathcal{R}_k$ already stores openings w_i, $\mathcal{R}_k$ runs SVC.OpenUpd to update them.

3. In the uud.read interface, $\mathcal{R}_k$ uses $\mathcal{F}_{\mathrm{ZK}}^{R}$ (see Section 2.8.8) to prove that the commitments $(com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]})$ commit to an entry i and values $v_{i,1}, \ldots, v_{i,L}$ such that $\mathbf{x}[(i-1)L+j] = v_{i,j}$ for all $j \in [1, L]$, where $\mathbf{x}$ is the vector committed to in svc. R requires proving knowledge of an opening w_I for the set $I = \{(i-1)L+1, \ldots, (i-1)L+L\}$ of positions where the values for the database entry i are stored. $\mathcal{R}_k$ runs SVC.Open to compute w_I if it is not stored. R also requires a proof to associate i with I, which we denote by $I = f(i)$, where f is a function that on input i outputs the indices $I = \{(i-1)L+1, \ldots, (i-1)L+L\}$.

Figure 9.2: Construction Π_{UUD}

N denotes the database size and L the size of any entry. The function $f(i) = ((i-1)L+1, \ldots, (i-1)L+L)$ maps $i \in [1, N]$ to a set of indices where the database entry i is stored. The universe of values $\mathbb{U}_v$ is given by the message space of the SVC scheme.

1. On input $(\mathsf{uud.update.ini}, sid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i\in[1,N]})$, $\mathcal{U}$ does the following:

   - If $(sid, par, svc, \mathbf{x}, cu)$ is not stored:
     - $\mathcal{U}$ uses crs.get to obtain the parameters par from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}$. To compute par, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}$ runs $\mathsf{SVC.Setup}(1^k, N \times L)$.
     - $\mathcal{U}$ initializes a counter $cu \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[(i-1)L+j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$. $\mathcal{U}$ runs $svc \leftarrow \mathsf{SVC.Commit}(par, \mathbf{x})$ and stores $(sid, par, svc, \mathbf{x}, cu)$.
   - Else:
     - $\mathcal{U}$ sets $cu' \leftarrow cu+1, \mathbf{x}' \leftarrow \mathbf{x}$ and $svc' \leftarrow svc$. For all $i \in [1, N]$ and $j \in [1, L]$ such that $v_{i,j} \neq \perp$, $\mathcal{U}$ computes $svc' \leftarrow \mathsf{SVC.ComUpd}(par, svc', \mathbf{x}', (i-1)L+j, v_{i,j})$ and sets $\mathbf{x}'[(i-1)L+j] \leftarrow v_{i,j}$.

  - $\mathcal{U}$ replaces the stored tuple $(sid, par, svc, \mathbf{x}, cu)$ by $(sid, par, svc', \mathbf{x}', cu')$.
- $\mathcal{U}$ uses the bb.write interface to append $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ to the bulletin board.
- $\mathcal{U}$ outputs (uud.update.end, sid).

2. On input (uud.getdb.ini, sid), $\mathcal{R}_k$ does the following:
   - If $(sid, par, svc, \mathbf{x}, cr_k)$ is not stored, $\mathcal{R}_k$ obtains par from $\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}$ and initializes a counter $cr_k \leftarrow 0$.
   - $\mathcal{R}_k$ increments cr_k and uses the bb.getbb interface to read the message $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ stored at position cr_k in the bulletin board. $\mathcal{R}_k$ continues incrementing the counter and reading the bulletin board until the returned message is $\perp$.
   - $\mathcal{R}_k$ sets a tuple $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$, such that $v_{i,j}$ (for $i \in [1, N]$ and $j \in [1, L]$) is the most recent update for position j of the database entry i received from the bulletin board. If $(sid, par, svc, \mathbf{x}, cr_k)$ is not stored, $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ contains the current database to be used to set $\mathbf{x}$, otherwise it contains the update that needs to be performed on $\mathbf{x}$.
   - For $i = 1$ to N, if (sid, i, w_I) is stored, $\mathcal{R}_k$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $w_I' \leftarrow w_I$ and, for all $i \in [1, N]$ and $j \in [1, L]$ such that $v_{i,j} \neq \perp$, $w_I' \leftarrow$ SVC.OpenUpd$(par, w_I', \mathbf{x}', I, (i-1)L + j, v_{i,j})$ and $\mathbf{x}'[(i-1)L + j] = v_{i,j}$. $\mathcal{R}_k$ replaces (sid, i, w_I) by (sid, i, w_I').
   - $\mathcal{R}_k$ performs the same operations as $\mathcal{U}$ to set or update svc and $\mathbf{x}$, and stores a tuple $(sid, par, svc, \mathbf{x}, cr_k)$.
   - $\mathcal{R}$ outputs (uud.getdb.end, $sid, \mathbf{x}$).
3. On input (uud.read.ini, $sid, P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$):
   - $\mathcal{R}_k$ parses com_i as $(com_i', parcom,$ COM.Verify$)$.
   - $\mathcal{R}_k$ aborts if $1 \neq$ COM.Verify$(parcom, com_i', i, open_i)$.
   - For all $j \in [1, L]$:
     - $\mathcal{R}_k$ parses the commitment $com_{i,j}$ as $(com_{i,j}', parcom,$ COM.Verify$)$.
     - $\mathcal{R}_k$ aborts if $1 \neq$ COM.Verify$(parcom, com_{i,j}', v_{i,j}, open_{i,j})$.
   - $\mathcal{R}_k$ takes the stored tuple $(sid, par, svc, \mathbf{x}, cr_k)$ and aborts if, for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq v_{i,j}$.
   - If (sid, i, w_I) is not stored, $\mathcal{R}_k$ computes $I \leftarrow f(i)$, runs $w_I \leftarrow$ SVC.Open$(par, I, \mathbf{x})$ and stores (sid, i, w_I).

The relation R is

$$\begin{aligned} R = \{(wit, ins) : \\ & 1 = \mathsf{COM.Verify}(parcom, com'_i, i, open_i) \wedge \\ & \langle 1 = \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j})\rangle_{\forall j \in [1,L]} \wedge \\ & 1 = \mathsf{SVC.Verify}(par, svc, \langle v_{i,j}\rangle_{\forall j \in [1,L]}, I, w_I) \ \wedge \ I = f(i)\} \end{aligned}$$

- $\mathcal{R}_k$ sets the witness $wit \leftarrow (w_I, I, i, open_i, \langle v_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$ and the instance $ins \leftarrow (par, svc, parcom, com'_i, \langle com'_{i,j}\rangle_{\forall j \in [1,L]}, cr_k)$. $\mathcal{R}_k$ uses zk.prove to send wit, ins and P to $\mathcal{F}_{\mathrm{ZK}}^{R}$.
- $\mathcal{U}$ receives P and $ins = (par', svc', parcom, com'_i, \langle com'_{i,j}\rangle_{\forall j \in [1,L]}, cr_k)$ from $\mathcal{F}_{\mathrm{ZK}}^{R}$.
- $\mathcal{U}$ takes the stored tuple $(sid, par, svc, \mathbf{x}, cu)$ and aborts if $cr_k \neq cu$, or if $par' \neq par$, or if $svc' \neq svc$.
- $\mathcal{U}$ sets $com_i \leftarrow (com'_i, parcom, \mathsf{COM.Verify})$ and sets $\langle com_{i,j} \leftarrow (com'_{i,j}, parcom, \mathsf{COM.Verify})\rangle_{\forall j \in [1,L]}$. ($\mathsf{COM.Verify}$ is in the description of R.)
- $\mathcal{U}$ outputs $(\mathsf{uud.read.end}, sid, P, (com_i, \langle com_{i,j}\rangle_{\forall j \in [1,L]}))$.

9.5 SECURITY ANALYSIS

Theorem 9.5.1 Π_{UUD} *securely realizes* $\mathcal{F}_{\mathrm{UUD}}$ *in the* $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{ZK}}^{R})$*-hybrid model if the SVC scheme is binding.*

The proof of Theorem 9.5.1 is described in Section A.4.

9.6 INSTANTIATION AND EFFICIENCY ANALYSIS

We describe an instantiation of Π_{UUD} that uses the SVC scheme in Section 2.7.3, the Pedersen commitment scheme [80], the signature scheme in [2] and the UC ZK proof protocol in [24].

9.6.1 *Commitment Scheme*

We use the Pedersen commitment scheme [80] as described in Section 7.6.

9.6.2 *Signature Scheme*

We use the structure-preserving signature (SPS) scheme in [2], as described in Section 7.6.

9.6.3 *ZK Functionality*

To instantiate $\mathcal{F}_{\text{ZK}}^{R}$, we use the scheme in [24] as described in Section 2.8.8.1.

9.6.3.1 *UC ZK Proof for the Read Relation*

To instantiate $\mathcal{F}_{\text{ZK}}^{R}$ with the protocol in [24], we need to instantiate R with our chosen SVC and commitment schemes. Then we need to express R following the notation for UC ZK proofs described above.

In R, we need to prove that $I = f(i) = \{(i-1)L+1, \ldots, (i-1)L+L\}$, i.e., we need to prove that the set I of positions opened contains the positions where the database entry i is stored. To prove this statement, the public parameters of the SVC scheme are extended with SPSs that bind g^i with $(g_{(i-1)L+1}, \tilde{g}_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, \tilde{g}_{(i-1)L+L})$, i.e., i is bound with the bases of the positions in I. Given the parameters $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \{g_i, \tilde{g}_i\}_{\forall i \in [1,\ell]}, \{h_{i,i'}\}_{\forall i,i' \in [1,\ell], i \neq i'})$, we create the key pair $(sk, pk) \leftarrow \mathsf{KeyGen}(\langle p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}\rangle, L+1, L+1)$ and, for $i \in [1, \ell]$, we compute $sig_i \leftarrow \mathsf{Sign}(sk, \langle g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid}\rangle)$, where sid is the session identifier. We remark that these signatures do not need to be updated when the database is updated.

Let $(U_1, \ldots, U_{L+1}, V, W_1, \ldots, W_{L+1}, Z)$ be the public key of the signature scheme. Let (R, S, T) be a signature on $(g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid})$. Let (g, h) be the parameters of the Pedersen commitment scheme. R involves proofs about secret bases, and we use the transformation described above for those proofs. The base h is also used to randomize secret bases in $\mathbb{G}$, and another base $\tilde{h} \leftarrow \tilde{\mathbb{G}}$ is added to randomize bases in $\tilde{\mathbb{G}}$. Following the notation in [24], we describe the proof as follows.

$$
\begin{aligned}
&\aleph\, i, open_i, \langle v_{i,j}, open_{i,j}, g_{(i-1)L+j}, \tilde{g}_{(i-1)L+j}\rangle_{\forall j \in [1,L]}, w_I, R, S, T : \\
&com'_i = g^i h^{open_i} \wedge \langle com'_{i,j} = g^{v_{i,j}} h^{open_{i,j}}\rangle_{\forall j \in [1,L]} \wedge \qquad (9.1)\\
&e(R,V)e(S,\tilde{g})(\prod_{j \in [1,L]} e(g_{(i-1)L+j}, W_j))e(g, W_{L+1})^i e(g,Z)^{-1} = 1 \wedge \qquad (9.2)\\
&e(R,T)(\prod_{j \in [1,L]} e(U_j, \tilde{g}_{(i-1)L+j}))e(U_{L+1}, \tilde{g}^{sid})e(g,\tilde{g})^{-1} = 1 \wedge \qquad (9.3)\\
&e\left(\frac{vc}{\prod_{j \in [1,L]} g_{(i-1)L+j}^{v_{i,j}}}, \prod_{j \in [1,L]} \tilde{g}_{(i-1)L+j}\right) = e(w_I, \tilde{g}) \qquad (9.4)
\end{aligned}
$$

Equation 9.1 proves knowledge of the openings of the Pedersen commitments com'_i and $\langle com'_{i,j}\rangle_{\forall j \in [1,L]}$. Equation 9.2 and Equation 9.3 prove knowledge of a signature (R, S, T) on a message $(g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid})$. Equation 9.4 proves that the values $\langle v_{i,j}\rangle_{\forall j \in [1,L]}$ in $\langle com'_{i,j}\rangle_{\forall j \in [1,L]}$ are equal to the values committed in the positions $I = f(i) = \{(i-1)L+1, \ldots, (i-1)L+L\}$ of the vector commitment vc.

When a range of indices $[i_{min}, i_{max}]$ always stores (i.e., even after database updates) the same tuple $[v_{i,1}, \ldots, v_{i,L}]$, we can improve storage efficiency as fol-

lows: we compute signatures on tuples $(g_{(i'-1)L+1}, \ldots, g_{(i'-1)L+L}, g^{i_{min}}, g^{i_{max}}, \tilde{g}_{(i'-1)L+1}, \ldots, \tilde{g}_{(i'-1)L+L}, \tilde{g}^{sid})$ that bind all the indices in $[i_{min}, i_{max}]$ with the bases for the positions where the tuple is stored. (i' is used to denote the position in the SVC where the tuple is stored.) Then, in the UC ZK proof for R, we add a range proof to prove that $i \in [i_{min}, i_{max}]$, where i is committed to in com'_i, to prove that we are opening the right subvector for i.

9.6.4 *Efficiency Analysis*

We analyse the storage, communication, and computation costs of our instantiation of Π_{UUD}.

9.6.4.1 *Storage Costs*

Any $\mathcal{R}_k$ and $\mathcal{U}$ store the common reference string, whose size grows quadratically with N. Throughout protocol execution, $\mathcal{R}_k$ and $\mathcal{U}$ also store the last updated values of vc and the committed vector. $\mathcal{R}_k$ stores the openings w_I. In conclusion, the storage cost is quadratic in $N \times L$.

9.6.4.2 *Communication Costs*

In the uud.update interface, $\mathcal{U}$ sends the tuples $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$, which are retrieved by $\mathcal{R}_k$ in the uud.getdb interface. The communication cost is linear in the number of entries updated, except for the first update in which all entries must be initialized. In the uud.read interface, $\mathcal{R}_k$ sends an instance and a ZK proof to $\mathcal{U}$. The size of the witness and of the instance grows linearly with L but is independent of N. In conclusion, after the first update phase, the communication cost does not depend on N.

9.6.4.3 *Computation Costs*

In the uud.update and uud.getdb interfaces, $\mathcal{U}$ and $\mathcal{R}_k$ update vc with cost linear in the number t of updates, except for the first update where all positions are initialized. $\mathcal{R}_k$ also updates the stored openings w_I with cost linear in $t \times L$. In the uud.read interface, if w_I is not stored, $\mathcal{R}_k$ computes it with cost that grows linearly with $N \times L$. However, if w_I is stored, the computation cost of the proof grows linearly with L but is independent of N.

It is possible to defer opening updates to the uud.read interface, so as to only update openings that are actually needed to compute ZK proofs. Thanks to that, the computation cost in the uud.getdb interface is independent of N. In the uud.read interface, if w_I is stored but needs to be updated, the computation cost grows linearly with $t \times L$, but it is independent of N. The only overhead introduced by deferring opening updates is the need to store the tuples $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$.

In summary, after initialization of vc and the openings w_I, the communication and computation costs are independent of N, so in terms of communication and computation our instantiation of Π_{UUD} is practical for large databases.

INTERFACE	$N = 50$ $L = 10$	$N = 100$ $L = 5$	$N = 100$ $L = 15$	$N = 150$ $L = 10$	$N = 200$ $L = 5$	$N = 200$ $L = 15$
Setup	61.35	61.37	553.71	554.52	249.61	2205.72
Update	0.0001	0.0002	0.0001	0.0001	0.0002	0.0001
Getdb	0.0004	0.0004	0.0006	0.0004	0.0003	0.0004
Computation of vc	0.0371	0.0350	0.1093	0.1035	0.0707	0.2145
One value update of vc	1.59e-05	1.59e-05	1.71e-05	1.99e-05	1.69e-05	1.59e-05
Computation of w_I	0.3491	0.1753	1.5659	1.0485	0.3513	3.1330
One value update of w_I	0.0002	0.0001	0.0003	0.0002	0.0001	0.0003
Read proof (1024-bit key)	3.6737	2.1903	4.9621	3.6811	2.1164	5.0268
Read proof (2048-bit key)	16.6220	10.6786	25.2909	16.8730	9.8916	23.4896

Table 9.1: Π_{UUD} execution times in seconds.

9.6.5 *Implementation and Efficiency Measurements*

We have implemented our instantiation of Π_{UUD} in the Python programming language, using the Charm cryptographic framework [4], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve was used for the pairing group setup.

To compute UC ZK proofs for $\mathcal{R}_k$, we use the compiler in [24]. The public parameters of the proof system contain a public key of the Paillier encryption scheme, the parameters for a multi-integer commitment scheme, and the specification of a DSA group. (We refer the reader to [24] for a description of how these primitives are used in the compiler.) The cost of a proof depends on the number of elements in the witness and on the number of equations. The computation cost for the prover of a Σ-protocol for $\mathcal{R}_k$ involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [24] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the values in the witness, an evaluation of a Paillier encryption for each of the values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for the verifier, as well as the communication cost, also depends on the number of values in the witness, and on the number of equations. Therefore, as the number of values in the witness and the number of equations is independent of N in our proof for relation R, the computation and communication costs of our proof do not depend on N.

Table 9.1 lists the execution times of the uud.update and uud.getdb interfaces, the computation costs for read proofs, and the costs for computing and updating w_I and vc, in our implementation, in seconds. The execution times of the interfaces of the protocol have been evaluated against the size N of the database, and the size of each entry L of the database. In the setup phase, the public parameters of all the building blocks are computed, and the database is set up by computing vc. In the second and third rows of Table 9.1, we depict the execution times for the uud.update and uud.getdb interfaces for the updater $\mathcal{U}$, and a reader $\mathcal{R}_k$

respectively, after the update of a single value in an entry of the database. In the fourth row of Table 9.1, we show the cost of computing vc, and as can be seen from these values, the computation times for vc depend on the total number of values $N \times L$ in the database. However, the cost of updating vc is very small, and linear in the number t of updates, and this in turn results in small computation costs for the uud.update interface, independent of N. The cost of computation of w_I also depends on the total number of values $N \times L$ in the database, while the cost of updating w_I is linear in $t \times L$, and thus the execution times for the uud.getdb interface (which involves the updates of stored witnesses, in addition to the update of vc as in the case of the uud.update interface) are also small.

In the last two rows of Table 9.1, we show the computation costs for a read proof. These values have been evaluated against varying key lengths for the Paillier encryption scheme used in the proof system in our instantiation of Π_{UUD}. The execution times for the read interface depend greatly upon the security parameters of the Paillier encryption scheme, and increase linearly with the entry size of the database L. However, the execution times are independent of the database size N.

9.7 VARIANTS

- It is straightforward to modify the uud.read interface to allow $\mathcal{R}_k$ to read several database entries simultaneously. This variant allows us to reduce communication rounds when $\mathcal{R}_k$ needs to read more than one entry simultaneously.
- $\mathcal{F}_{\mathrm{UUD}}$ can also be modified to interact with two parties such that both of them can read and update the database, or such that a party reads and updates and the other party receives read and update operations.

10

SUMMARY

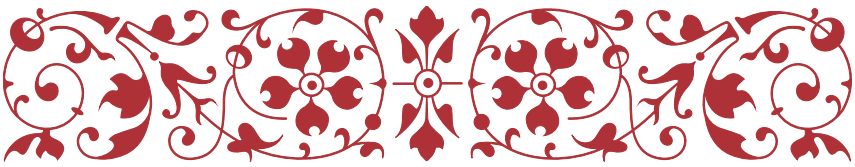
Table 10.1 provides an overview of the data structures set out in this thesis, the security properties they guarantee, and the operations they support. The notation proposed in Part IV can be used to produce specifications of these data structures and the statements to be proved during the course of execution of their operations, and the SZK compiler described in Part IV can be used to produce an implementation of the data structure and proof functionalities in question, on receiving the specification as input.

SECURITY PROPERTY	CD	UCD	NICD	UUHD	UD	UUD
Zero Knowledge	✓	✓	✓	✓	✓	✓
Binding	✓	✓	✓			
Unlinkability		✓		✓		✓
Consistency			✓			
Hiding				✓		
Unforgeability				✓	✓	✓
OPERATIONS						
Read	✓	✓	✓	✓	✓	✓
Write	✓	✓	✓			
Update				✓	✓	✓

Table 10.1: An overview of our data structures and the security properties they provide.

Part III exemplifies the possibility of using these data structures for real world applications by describing several privacy preserving protocols that use them as building blocks.

Part III



STATEFUL ZERO-KNOWLEDGE PROTOCOLS

We describe privacy preserving protocols that use the Stateful Zero Knowledge (SZK) data structures from Part II as building blocks.

11

PRICED OBLIVIOUS TRANSFER WITH STATISTICS AND DYNAMIC PRICING

Joint work with Maria Dubovitskaya and Alfredo Rial. This protocol was published in a paper presented at INDOCRYPT in 2019 [35]. This protocol makes use of the functionalities $\mathcal{F}_{\text{CD}}$ and $\mathcal{F}_{\text{UD}}$ from Chapter 4 and Chapter 8 respectively. Reproduced with permission from Springer Nature.

11.1 INTRODUCTION

A POT [3] protocol is a cryptographic protocol between a seller or vendor $\mathcal{V}$ and a buyer $\mathcal{B}$ that can be used to protect privacy in e-commerce applications. $\mathcal{V}$ sells N items (represented as messages) $\langle m_n \rangle_{n=1}^{N}$ with prices $\langle p_n \rangle_{n=1}^{N}$ assigned to them. At any transfer phase, $\mathcal{B}$ chooses an index $\sigma \in [1, N]$ and purchases the message m_σ. Security for $\mathcal{B}$ ensures that $\mathcal{V}$ does not learn the index σ or the price p_σ paid by $\mathcal{B}$. Security for $\mathcal{V}$ ensures that $\mathcal{B}$ pays the correct price p_σ for the message m_σ and that $\mathcal{B}$ does not learn any information about the messages that are not purchased.

Typically, POT schemes use a prepaid mechanism [3, 10, 18, 82, 85, 86]. $\mathcal{B}$ makes an initial deposit dep to $\mathcal{V}$, revealing the amount dep to $\mathcal{V}$. $\mathcal{B}$ and $\mathcal{V}$ can use an existing payment mechanism of their choice to carry out this transaction. After the deposit phase, when $\mathcal{B}$ purchases a message m_σ, the price p_σ is subtracted from the deposit, but the POT protocol ensures that:

1. $\mathcal{V}$ does not learn the new value $dep' = dep - p_\sigma$ of the deposit, and,
2. $dep' \geq 0$.

LACK OF MODULAR DESIGN. POT schemes [3, 10, 18, 82, 85, 86] have so far been built by extending an existing OT scheme. OT is a protocol between a sender $\mathcal{E}$ and a receiver $\mathcal{R}$. $\mathcal{E}$ inputs N messages $\langle m_n \rangle_{n=1}^{N}$. At each transfer phase, $\mathcal{R}$ obtains the message m_σ for her choice $\sigma \in [1, N]$. $\mathcal{E}$ does not learn σ, while $\mathcal{R}$ does not learn any information about other messages.

In OT schemes that have been used to build POT schemes, the interaction between $\mathcal{E}$ and $\mathcal{R}$ consists of an initialization phase followed by several transfer phases. In the initialization phase, $\mathcal{E}$ encrypts messages $\langle m_n \rangle_{n=1}^{N}$ and sends the list of ciphertexts to $\mathcal{R}$. In each transfer phase, $\mathcal{R}$, on input σ, computes a blinded request for $\mathcal{E}$. $\mathcal{E}$ sends a response that allows $\mathcal{R}$ to decrypt the ciphertext that encrypts m_σ.

Roughly speaking, to construct a POT scheme from an OT scheme, the OT scheme is typically extended as follows: first, in the initialization phase, the computation of the ciphertexts is modified in order to bind them to the prices of the encrypted messages, e.g. by using a signature scheme. Second, a deposit phase, where $\mathcal{B}$ sends an initial deposit to $\mathcal{V}$, is added. As a result of this deposit phase, $\mathcal{V}$ and $\mathcal{B}$ output a commitment or encryption to the deposit dep. Third, in each transfer phase, the request computed in the OT scheme is extended by $\mathcal{B}$ in order to send to $\mathcal{V}$ an

encryption or commitment to the new value dep' of the deposit and to prove to $\mathcal{V}$ (e.g. by using a zero-knowledge proof) that $dep' = dep - p_\sigma$ and that $dep' \geq 0$.

The main drawback of the design of existing POT schemes is a lack of modularity. Though each POT scheme is based on an underlying OT scheme, the latter is not used as a black-box building block. Instead, every OT scheme is modified and extended ad-hoc to create the POT scheme, blurring the line between the components that were present in the original OT scheme and the components which were added to create the POT scheme.

This lack of modularity results in two disadvantages: first, existing POT schemes cannot easily be modified to use another OT scheme as a building block, for example, a more efficient one. Second, every time a new POT scheme is designed, the proofs need to be built from scratch. This means that the security of the underlying OT scheme will be reanalysed instead of relying on its security guarantees. This is error-prone.

LACK OF PURCHASE STATISTICS. POT schemes [3, 10, 18, 82, 85, 86] effectively prevent $\mathcal{V}$ from learning what messages are purchased by $\mathcal{B}$. Although this is a nice privacy feature for $\mathcal{B}$, customer and sales management becomes more difficult for $\mathcal{V}$. For example, $\mathcal{V}$ is not able to learn which items are in more demand by buyers and which ones sell poorly. As another example, $\mathcal{V}$ cannot use marketing techniques like offering discounts that depend on the previous purchases of a buyer. It would be desirable that, whilst protecting the privacy of each individual purchase, $\mathcal{V}$ could obtain aggregate statistics about $\mathcal{B}$'s purchases.

LACK OF DYNAMIC PRICING. In existing POT schemes [3, 10, 18, 82, 85, 86], the price of a message is static, i.e. each message is associated with a price in the initialization phase and that price cannot change afterwards. In practical e-commerce settings, this is undesirable because sellers would like to be able to change the price of a product easily. However, modifying existing POT schemes to allow sellers to change the prices of messages at any time throughout protocol execution is not straightforward and would require rerunning the initialization phase.

11.1.1 *Our Contribution*

FUNCTIONALITY $\mathcal{F}_{\text{POTS}}$. We use the Universal Composability (UC) framework [29] to describe an ideal functionality $\mathcal{F}_{\text{POTS}}$ for priced oblivious transfer with purchase statistics and dynamic pricing. We modify a standard POT functionality to enable aggregate statistics and dynamic pricing.

Existing functionalities for POT [18, 85] consist of three interfaces: an initialization interface where $\mathcal{V}$ sends the messages $\langle m_n \rangle_{n=1}^N$ and the prices $\langle p_n \rangle_{n=1}^N$ to the functionality; a deposit interface where $\mathcal{B}$ sends a deposit dep to the functionality, which reveals dep to $\mathcal{V}$; and a transfer interface where $\mathcal{B}$ sends an index $\sigma \in [1, N]$ to the functionality and receives the message m_σ from the functionality if the current deposit is higher than the price of the message. The functionality stores the updated value of the deposit.

Our functionality $\mathcal{F}_{\text{POTS}}$ modifies the initialization interface so that $\mathcal{V}$ only inputs the messages $\langle m_n \rangle_{n=1}^N$. Additionally, this interface can be invoked multiple times to send different tuples $\langle m_n \rangle_{n=1}^N$ of messages, where each tuple is associated

with a unique epoch identifier ep. The idea is that messages of different epochs but with the same message index correspond to the same type or category of items. (This happens, e.g. when using POT to construct a conditional access system for pay-TV [10].) $\mathcal{F}_{\text{POTS}}$ also modifies the transfer interface in order to store the number of times $\mathcal{B}$ purchases items of each of the types or categories.

Moreover, $\mathcal{F}_{\text{POTS}}$ adds three new interfaces: a "setup price" interface where $\mathcal{V}$ inputs the prices, an "update price" interface where $\mathcal{V}$ modifies the price of a message, and a "reveal statistic" interface where $\mathcal{F}_{\text{POTS}}$ reveals to $\mathcal{V}$ the value of a statistic about the purchases of $\mathcal{B}$.

We propose a scheme Π_{POTS} that realizes $\mathcal{F}_{\text{POTS}}$. Π_{POTS} is designed modularly and provides purchase statistics and dynamic pricing, as described below.

MODULAR DESIGN. In the UC framework, protocols can be described modularly by using a hybrid model where parties invoke the ideal functionalities of the building blocks of a protocol. For example, consider a protocol that uses as building blocks a zero-knowledge proof of knowledge and a signature scheme. In a modular description of this protocol in the hybrid model, parties in the real world invoke the ideal functionalities for zero-knowledge proofs and for signatures.

We describe Π_{POTS} modularly in the hybrid model. Therefore, $\mathcal{V}$ and $\mathcal{B}$ in the real world invoke only ideal functionalities for the building blocks of Π_{POTS}. Interestingly, one of the building blocks used in Π_{POTS} is the ideal functionality for oblivious transfer $\mathcal{F}_{\text{OT}}$. Thanks to that, Π_{POTS} separates the task that is carried out by the underlying OT scheme from the additional tasks that are needed to create a POT scheme.

The advantages of a modular design are twofold. First, Π_{POTS} can be instantiated with any secure OT scheme, i.e., any scheme that realizes $\mathcal{F}_{\text{OT}}$. The remaining building blocks can also be instantiated with any scheme that realizes their corresponding ideal functionalities, leading to multiple possible instantiations of Π_{POTS}. Second, the security analysis in the hybrid model is simpler and does not require a reanalysis of the security of the building blocks.

One challenge when describing a UC protocol in the hybrid model is that the need arises to ensure that two or more ideal functionalities receive the same input. For example, in Π_{POTS}, it is necessary to enforce that $\mathcal{B}$ sends the same index $\sigma \in [1, N]$ to the transfer interface of $\mathcal{F}_{\text{OT}}$ and to another functionality $\mathcal{F}_{\text{UD}}$ that binds σ to the price p_σ. Otherwise, if an adversarial buyer sends different indexes σ and σ' to $\mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{UD}}$, $\mathcal{B}$ could obtain the message m_σ and pay an incorrect price $p_{\sigma'}$. To address this issue, we use the method proposed in [20] and described in Section 2.9, which uses a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments.

PURCHASE STATISTICS. In Π_{POTS}, $\mathcal{V}$ can input multiple tuples $\langle m_n \rangle_{n=1}^{N}$ of messages. We consider that messages associated with the same index σ belong to the same category.

Π_{POTS} allows $\mathcal{B}$ to reveal to $\mathcal{V}$ information related to how many purchases were made for each of the item categories. $\mathcal{B}$ stores a table Tbl_{st} of counters of how many purchases were made for each category. Tbl_{st} contains position-value entries $[\sigma, v_\sigma]$, where $\sigma \in [1, N]$ is the category index and v_σ is the counter. Any time a message m_σ is purchased, the counter for category σ is incremented in Tbl_{st}. At any time throughout the execution of Π_{POTS}, $\mathcal{B}$ can choose a statistic ST, evaluate it on input Tbl_{st} and reveal to $\mathcal{V}$ the result.

Additionally, $\mathcal{B}$ must prove to $\mathcal{V}$ that the result is correct, and thus we need a mechanism that allows $\mathcal{V}$ to keep track of the purchases made by $\mathcal{B}$, without learning them. For this purpose, Π_{POTS} uses the functionality for a committed database $\mathcal{F}_{\text{CD}}$ described in Chapter 4. $\mathcal{F}_{\text{CD}}$ stores the table Tbl_{st} of counters and allows $\mathcal{B}$ to read the counters from Tbl_{st} and to write updated counters into Tbl_{st} each time a purchase is made. $\mathcal{V}$ does not learn any information read or written but is guaranteed of the correctness of that information. $\mathcal{F}_{\text{CD}}$ allows $\mathcal{B}$ to hide from $\mathcal{V}$ not only the value of counters read or written, but also the positions where they are read or written into Tbl_{st}. This is a crucial property to construct Π_{POTS}, because the position read or written is equal to the index σ, which needs to be hidden from $\mathcal{V}$ in order to hide what message is purchased. The method in [20] is used to ensure that the index σ is the same both for the counter v_σ incremented in $\mathcal{F}_{\text{CD}}$ and for the message m_σ obtained through $\mathcal{F}_{\text{OT}}$. In [21], an efficient construction for $\mathcal{F}_{\text{CD}}$ based on vector commitments (VC) [32, 69] is provided, where a VC commits to a vector x such that $x[\sigma] = v_\sigma$ for $\sigma \in [1, N]$. In this construction, after setup, the efficiency of the read and write operations does not depend on the size of the table, which yields efficient instantiations of Π_{POTS} when N is large.

We note that $\mathcal{V}$ could be more interested in gathering aggregated statistics about multiple buyers rather than a single buyer. Interestingly, Π_{POTS} opens up that possibility. The functionalities $\mathcal{F}_{\text{CD}}$ used between $\mathcal{V}$ and each of the buyers can be used in a secure multiparty computation (MPC) protocol for the required statistic. In this MPC, each buyer uses $\mathcal{F}_{\text{CD}}$ to read and prove correctness of the information about her purchases. With the instantiation of $\mathcal{F}_{\text{CD}}$ based on a VC scheme, $\mathcal{V}$ and the buyers would run a secure MPC protocol where $\mathcal{V}$ inputs one vector commitment for each buyer and each buyer inputs the committed vector and the opening.

DYNAMIC PRICING. In existing POT schemes [3, 10, 18, 82, 85, 86], when $\mathcal{V}$ encrypts the messages $\langle m_n \rangle_{n=1}^N$ in the initialization phase, the price of the encrypted message is somehow bound to the corresponding ciphertext. This binding is done in such a way that, when $\mathcal{B}$ computes a request to purchase a message m_σ, $\mathcal{V}$ is guaranteed that the correct price p_σ is subtracted from the deposit while still not learning p_σ. In some schemes, $\mathcal{V}$ uses a signature scheme to sign the prices in the initialization phase, and $\mathcal{B}$ uses a zero-knowledge proof of signature possession to compute the request.

It would be possible to modify the initialization phase of these schemes so that ciphertexts on the messages $\langle m_n \rangle_{n=1}^N$ are computed independently of the signatures on the prices $\langle p_n \rangle_{n=1}^N$, yet enforcing that requests computed by $\mathcal{B}$ use the right price p_σ for the requested index σ. (For example, both the ciphertext and the signature could embed the index σ, and $\mathcal{B}$, as part of a request, would be required to prove in zero-knowledge that the indices in the ciphertext and in the signature are equal.) This would allow $\mathcal{V}$ to modify the prices of messages by issuing new signatures, without needing to re-encrypt the messages. However, this mechanism to update prices would also require some method to revoke previous signatures, which would heavily affect the efficiency of the Π_{POTS} protocol.

Instead, we use the functionality $\mathcal{F}_{\text{UD}}$ for an updatable database described in Chapter 8. $\mathcal{F}_{\text{UD}}$ stores a database DB with entries $[\sigma, p_\sigma]$, where $\sigma \in [1, N]$ is an index and p_σ is a price. $\mathcal{V}$ sets the initial values of DB and is also able to modify DB at any time. $\mathcal{B}$ knows the contents of DB but cannot modify them. When

purchasing a message of index σ, $\mathcal{B}$ reads from $\mathcal{F}_{\text{UD}}$ the entry $[\sigma, p_\sigma]$. $\mathcal{V}$ does not learn any information about the entry read, yet $\mathcal{V}$ is guaranteed that a valid entry is read. As in the case of $\mathcal{F}_{\text{CD}}$, we stress that $\mathcal{F}_{\text{UD}}$ reveals to $\mathcal{V}$ neither the position σ nor the value p_σ, and that the method in [20] is used to prove that the index σ received by $\mathcal{F}_{\text{UD}}$ and by $\mathcal{F}_{\text{OT}}$ are the same. In [38], an efficient construction for $\mathcal{F}_{\text{UD}}$ based on a non-hiding VC scheme [32, 69] is provided, where a non-hiding VC commits to a vector x such that $x[\sigma] = p_\sigma$ for $\sigma \in [1, N]$. In this construction, after setup, the efficiency of the read and write operations does not depend on the size of the table, which yields efficient instantiations of Π_{POTS} when N is large.

Though it might seem that dynamic pricing undermines buyer privacy in comparison to existing POT schemes, e.g. when an adversarial $\mathcal{V}$ offers different prices to each of the buyers and changes them dynamically in order to narrow down the messages that a buyer could purchase, this is not the case. In existing POT schemes, a seller is also able to offer different prices to each buyer, and to change them dynamically by restarting the protocol. The countermeasure, for both Π_{POTS} and other POT schemes, is to have lists of prices available through a secure bulletin board where buyers can check that the prices they are offered are equal to those for other buyers.

II.2 RELATED WORK

POT was initially proposed in [3]. The POT scheme in [3] is secure in a half-simulation model, where a simulation-based security definition is used to protect seller security, while an indistinguishability based security definition is used to protect buyer privacy. Later, POT schemes in the full-simulation model [18] and UC-secure schemes [85] were proposed. The scheme in [18] provides unlinkability between $\mathcal{V}$ and $\mathcal{B}$, i.e., $\mathcal{V}$ cannot link interactions with the same buyer.

We define security for POT in the UC model, like [85], and our protocol does not provide unlinkability, unlike [18] or the PIR-based scheme in [55]. Although unlinkability is important in some settings, an unlinkable POT scheme would require the use of an anonymous communication network and, in the deposit phase, it would hinder the use of widespread payment mechanisms that require authentication. Therefore, because one of the goals of this work is to facilitate sellers' deployment of POT schemes, we chose to describe a scheme that does not provide unlinkability.

The use of POT as a building block in e-commerce applications in order to protect buyer privacy has been described, e.g. in the context of buyer-seller watermarking protocols for copyright protection [82] and conditional access systems for pay-TV [10]. Our POT protocol is suitable to be used in any of the proposed settings, and it provides additional functionalities to $\mathcal{V}$. In [86], a transformation that takes a POT scheme and produces a POT scheme with optimistic fair exchange is proposed. This transformation can also be used with our POT scheme.

Oblivious Transfer with Access Control (OTAC) [17, 33] is a generalization of oblivious transfer where messages are associated with access control policies. In order to obtain a message, a receiver must prove that she fulfils the requirements described in the associated access control policy. In some schemes, access control policies are public [17, 33, 66, 100], while other schemes hide them from the receiver [16, 19].

POT could be seen as a particular case of OTAC with public access control policies. In POT, the public access control policy that $\mathcal{B}$ must fulfil to get a message is defined

as her current deposit being higher than the price of the message. However, existing OTAC schemes cannot straightforwardly be converted into a POT scheme. This is due to the fact that in adaptive POT, the fulfilment of a policy by $\mathcal{B}$ depends on the history of purchases and deposits of $\mathcal{B}$, i.e., whether the current deposit of $\mathcal{B}$ allows her to buy a message depends on how much $\mathcal{B}$ deposited and spent before. Therefore, POT schemes need to implement a mechanism that allows $\mathcal{V}$ to keep track of the current deposit of $\mathcal{B}$ without learning it, such as a commitment or an encryption of the deposit that is updated by $\mathcal{B}$ at each deposit or purchase phase. In contrast, existing OTAC schemes do not provide such a mechanism. In these schemes, a third party named an "issuer" usually certifies the attributes of a receiver, and the receiver may then use these certifications to prove that she fulfils an access control policy.

Our POT protocol uses functionality $\mathcal{F}_{\text{CD}}$ to store the deposit, in addition to the counters of purchases.

The oblivious language-based envelope (OLBE) framework in [12] generalizes POT, OTAC and similar protocols like conditional OT. However, as in the case of OTAC schemes, the instantiation of POT in the OLBE framework is only straightforward for non-adaptive POT schemes, where $\mathcal{V}$ does not need to keep track of the deposit of $\mathcal{B}$.

Privacy protection in e-commerce can also be provided by other protocols that offer anonymity/unlinkability. Here the goal is to protect the identity of $\mathcal{B}$ rather than the identity of the items purchased. Most solutions involve anonymous payment methods [22] and anonymous communication networks [42].

11.3 IDEAL FUNCTIONALITY

We depict our functionality $\mathcal{F}_{\text{POTS}}$ for POT with purchase statistics and dynamic pricing. $\mathcal{F}_{\text{POTS}}$ interacts with a seller $\mathcal{V}$ and with a buyer $\mathcal{B}$ and consists of the following interfaces:

1. $\mathcal{V}$ uses the pot.init interface to send a list of messages $\langle m_n \rangle_{n=1}^{N}$ and an epoch identifier ep to $\mathcal{F}_{\text{POTS}}$. $\mathcal{F}_{\text{POTS}}$ stores $\langle m_n \rangle_{n=1}^{N}$ and ep, and sends N and ep to $\mathcal{B}$. In the first invocation of this interface, $\mathcal{F}_{\text{POTS}}$ also initializes a deposit dep' and a table Tbl_{st} with entries of the form $[\sigma, v_\sigma]$, where $\sigma \in [1, \mathcal{N}_{max}]$ is a category and v_σ is a counter of the number of purchases made for that category.

2. $\mathcal{V}$ uses the pot.setupprices interface to send a list of prices $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ to $\mathcal{F}_{\text{POTS}}$, where $\mathcal{N}_{max}$ is the maximum number of messages in an epoch. $\mathcal{F}_{\text{POTS}}$ stores $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ and sends $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ to $\mathcal{B}$.

3. $\mathcal{V}$ uses the pot.updateprice interface to send an index n and a price p to $\mathcal{F}_{\text{POTS}}$. $\mathcal{F}_{\text{POTS}}$ updates the stored list $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ with p at position n, and sends n and p to $\mathcal{B}$.

4. $\mathcal{B}$ uses the pot.deposit interface to send a deposit dep to $\mathcal{F}_{\text{POTS}}$. $\mathcal{F}_{\text{POTS}}$ updates the stored deposit $dep' \leftarrow dep' + dep$ and sends dep to $\mathcal{V}$.

5. $\mathcal{B}$ uses the pot.transfer interface to send an epoch ep and an index σ to $\mathcal{F}_{\text{POTS}}$. If $dep' \geq p_\sigma$, $\mathcal{F}_{\text{POTS}}$ increments the counter for category σ in Tbl_{st} and sends m_σ for the epoch ep to $\mathcal{B}$.

6. $\mathcal{B}$ uses the pot.revealstatistic interface to send a function ST to $\mathcal{F}_{\text{POTS}}$. $\mathcal{F}_{\text{POTS}}$ evaluates ST on input table Tbl_{st} and sends the result v and ST to $\mathcal{V}$. ST may be any function from a universe Ψ.

In previous functionalities for POT [18, 85], $\mathcal{V}$ sends the messages and prices through the pot.init interface. In contrast, $\mathcal{F}_{\text{POTS}}$ uses the pot.setupprices and pot.updateprice interfaces to send and update the prices. This change allows for the design of a protocol where $\mathcal{V}$ can update prices without rerunning the initialization phase. We also note that, in $\mathcal{F}_{\text{POTS}}$, all the messages m_σ of the same category σ have the same price for any epoch. The idea here is that messages of the same category represent the same type of content, which is updated by $\mathcal{V}$ at each new epoch. Nevertheless, it is straightforward to modify $\mathcal{F}_{\text{POTS}}$ so that $\mathcal{V}$ can send a new list of prices for each epoch. Our construction in Section 11.4 can easily be modified to allow different prices for each epoch.

$\mathcal{F}_{\text{POTS}}$ initializes a counter ct_v and a counter ct_b in the pot.setupprices interface. ct_v is incremented each time $\mathcal{V}$ sends the update of a price, and ct_b is incremented each time $\mathcal{B}$ receives the update of a price. These counters are used by $\mathcal{F}_{\text{POTS}}$ to check that $\mathcal{V}$ and $\mathcal{B}$ have the same list of prices. We note that the simulator $\mathcal{S}$, when queried by $\mathcal{F}_{\text{POTS}}$, may not reply or may provide a delayed response, which could prevent price updates sent by $\mathcal{V}$ from being received by $\mathcal{B}$.

The session identifier sid has the structure $(\mathcal{V}, \mathcal{B}, sid')$. This allows any vendor $\mathcal{V}$ to create an instance of $\mathcal{F}_{\text{POTS}}$ with any buyer $\mathcal{B}$. After the first invocation of $\mathcal{F}_{\text{POTS}}$, $\mathcal{F}_{\text{POTS}}$ implicitly checks if the session identifier in a message is equal to the one received in the first invocation.

When invoked by $\mathcal{V}$ or $\mathcal{B}$, $\mathcal{F}_{\text{POTS}}$ first checks the correctness of the input. Concretely, $\mathcal{F}_{\text{POTS}}$ aborts if this input does not belong to the correct domain. $\mathcal{F}_{\text{POTS}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, $\mathcal{V}$ cannot invoke pot.updateprice before pot.setupprices. Similar abortion conditions are listed when $\mathcal{F}_{\text{POTS}}$ receives a message from the simulator $\mathcal{S}$.

Before $\mathcal{F}_{\text{POTS}}$ queries $\mathcal{S}$, $\mathcal{F}_{\text{POTS}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. When an interface, e.g. pot.updateprice, can be invoked more than once, $\mathcal{F}_{\text{POTS}}$ creates a query identifier qid, which allows $\mathcal{F}_{\text{POTS}}$ to match a query to $\mathcal{S}$ to a response from $\mathcal{S}$. Creating qid is not necessary if an interface, such as pot.setupprices, can be invoked only once, or if it can be invoked only once with a concrete input revealed to $\mathcal{S}$, such as pot.init, which is invoked only once per epoch.

Compared to previous functionalities for POT, $\mathcal{F}_{\text{POTS}}$ looks more complex, as we list all the conditions for abortion and because $\mathcal{F}_{\text{POTS}}$ saves state information before querying $\mathcal{S}$ and recovers it after receiving a response from $\mathcal{S}$. These operations are also required but have frequently been omitted in the description of ideal functionalities in literature. We describe $\mathcal{F}_{\text{POTS}}$ below.

Figure 11.1: Functionality $\mathcal{F}_{POTS}$

Functionality $\mathcal{F}_{\text{POTS}}$ runs with a seller $\mathcal{V}$ and a buyer $\mathcal{B}$, and is parameterized with a maximum number of messages $\mathcal{N}_{max}$, a message space $\mathcal{M}$, a maximum deposit value dep_{max}, a maximum price $\mathcal{P}_{max}$, and a universe of statistics Ψ that consists of ppt algorithms.

1. On input (pot.init.ini, sid, ep, $\langle m_n \rangle_{n=1}^{N}$) from $\mathcal{V}$:

- Abort if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
- Abort if $(sid, ep', \langle m_n \rangle_{n=1}^{N}, 0)$, where $ep' = ep$, is already stored.
- Abort if $N > \mathcal{N}_{max}$, or if for $n = 1$ to N, $m_n \notin \mathcal{M}$.
- Store $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 0)$.
- Send $(\mathsf{pot.init.sim}, sid, ep, N)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.init.rep}, sid, ep)$ from $\mathcal{S}$:

- Abort if $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
- If a tuple (sid, Tbl_{st}) is not stored, initialize $dep' \leftarrow 0$ and a table Tbl_{st} with entries $[i, 0]$ for $i = 1$ to $\mathcal{N}_{max}$, and store $(sid, dep', \mathsf{Tbl}_{st})$.
- Store $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 1)$.
- Send $(\mathsf{pot.init.end}, sid, ep, N)$ to $\mathcal{B}$.

2. On input $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ from $\mathcal{V}$:

- Abort if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$ or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is already stored.
- Abort if, for $n = 1$ to $\mathcal{N}_{max}$, $p_n \notin (0, \mathcal{P}_{max}]$.
- Initialize a counter $ct_v \leftarrow 0$ and store $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$.
- Send $(\mathsf{pot.setupprices.sim}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.setupprices.rep}, sid)$ from $\mathcal{S}$:

- Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is not stored, or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ is already stored.
- Initialize a counter $ct_b \leftarrow 0$ and store $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$.
- Send $(\mathsf{pot.setupprices.end}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ to $\mathcal{B}$.

3. On input $(\mathsf{pot.updateprice.ini}, sid, n, p)$ from $\mathcal{V}$:

- Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is not stored.
- Abort if $n \notin [1, \mathcal{N}_{max}]$, or if $p \notin (0, \mathcal{P}_{max}]$.
- Increment ct_v, set $p_n \leftarrow p$ and store them into the tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$.
- Create a fresh qid and store (qid, n, p, ct_v).
- Send $(\mathsf{pot.updateprice.sim}, sid, qid, n, p)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.updateprice.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if (qid, n, p, ct_v) is not stored, or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ is not stored, or if $ct_v \neq ct_b + 1$.

- Increment ct_b, set $p_n \leftarrow p$, and store them into the tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$.
- Delete the record (qid, n, p, ct_v).
- Send $(\mathsf{pot.updateprice.end}, sid, n, p)$ to $\mathcal{B}$.

4. On input $(\mathsf{pot.deposit.ini}, sid, dep)$ from $\mathcal{B}$:
   - Abort if $(sid, dep', \mathsf{Tbl}_{st})$ is not stored, or if $dep' + dep \notin [0, dep_{max}]$.
   - Create a fresh qid and store (qid, dep).
   - Send $(\mathsf{pot.deposit.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.deposit.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, dep) is not stored.
   - Set $dep' \leftarrow dep' + dep$ and update $(sid, dep', \mathsf{Tbl}_{st})$.
   - Delete the record (qid, dep).
   - Send $(\mathsf{pot.deposit.end}, sid, dep)$ to $\mathcal{V}$.

5. On input $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$ from $\mathcal{B}$:
   - Abort if $(sid, ep', \langle m_n \rangle_{n=1}^{N}, 1)$ for $ep' = ep$ is not stored.
   - Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ and $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ are not stored, or if $ct_b \neq ct_v$.
   - Abort if $\sigma \notin [1, N]$, or if $dep' < p_\sigma$, where dep' is stored in $(sid, dep', \mathsf{Tbl}_{st})$.
   - Create a fresh qid and store $(qid, ep, \sigma, m_\sigma)$.
   - Send $(\mathsf{pot.transfer.sim}, sid, qid, ep)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.transfer.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, ep, \sigma, m_\sigma)$ is not stored.
   - Set $dep' \leftarrow dep' - p_\sigma$, increment v_σ for the entry $[\sigma, v_\sigma]$ in Tbl_{st}, and update $(sid, dep', \mathsf{Tbl}_{st})$.
   - Delete the record $(qid, ep, \sigma, m_\sigma)$.
   - Send $(\mathsf{pot.transfer.end}, sid, m_\sigma)$ to $\mathcal{B}$.

6. On input $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$ from $\mathcal{B}$:
   - Abort if $(sid, dep', \mathsf{Tbl}_{st})$ is not stored.
   - Abort if $\mathsf{ST} \notin \Psi$.
   - Set $v \leftarrow \mathsf{ST}(\mathsf{Tbl}_{st})$.
   - Create a fresh qid and store (qid, v, ST).
   - Send $(\mathsf{pot.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.revealstatistic.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if (qid, v, ST) is not stored.
- Delete the record (qid, v, ST).
- Send $(\mathsf{pot.revealstatistic.end}, sid, v, \mathsf{ST})$ to $\mathcal{V}$.

11.4 CONSTRUCTION

INTUITION. For each epoch ep, $\mathcal{V}$ and $\mathcal{B}$ use a new instance of $\mathcal{F}_{\mathrm{OT}}$. In the pot.init interface, $\mathcal{V}$ uses ot.init to create a new instance of $\mathcal{F}_{\mathrm{OT}}$ on input the messages $\langle m_n \rangle_{n=1}^{N}$. In the pot.transfer interface, $\mathcal{B}$ uses ot.request on input an index σ and receives the message m_σ through the ot.transfer interface.

To construct a POT protocol based on $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{V}$ must set the prices, and $\mathcal{B}$ must make deposits and pay for the messages obtained. Additionally, our POT protocol allows $\mathcal{V}$ to receive aggregate statistics on purchases.

PRICES. To set prices, $\mathcal{V}$ uses $\mathcal{F}_{\mathrm{DB}}$. In the pot.setupprices interface, the seller $\mathcal{V}$ uses ud.update.ini to create an instance of $\mathcal{F}_{\mathrm{DB}}$ on input a list of prices $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$, which are stored in the table DB in $\mathcal{F}_{\mathrm{DB}}$. In the pot.updateprice interface, $\mathcal{V}$ uses ud.update.ini to update a price in the table DB.

DEPOSITS. $\mathcal{F}_{\mathrm{CD}}$ is used to store the current funds dep' of $\mathcal{B}$. In the pot.init interface, $\mathcal{V}$ uses cd.setup to create an instance of $\mathcal{F}_{\mathrm{CD}}$ on input a table Tbl_{cd} that contains a 0 at every position. The position 0 of Tbl_{cd} is used to store dep'. In the pot.deposit interface, $\mathcal{B}$ makes deposits dep to $\mathcal{V}$, which are added to the existing funds dep'. $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$ to prove in zero-knowledge to $\mathcal{V}$ that the deposit is updated correctly as $dep' \leftarrow dep' + dep$. The interfaces cd.read and cd.write are used to read dep' and to write the updated value of dep' into Tbl_{cd}.

To carry out the payment of dep, $\mathcal{V}$ and $\mathcal{B}$ use a payment mechanism outside the POT protocol.

PAYMENTS. In the pot.transfer interface, $\mathcal{B}$ must subtract the price p_σ for the purchased message m_σ from the current funds dep'. $\mathcal{B}$ uses ud.read to read the correct price p_σ from DB. Then $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$ to prove in zero-knowledge that $dep' \leftarrow dep' - p_\sigma$. The interfaces cd.read and cd.write of $\mathcal{F}_{\mathrm{CD}}$ are used to read dep' and to write the updated value of dep' into Tbl_{cd}.

STATISTICS. $\mathcal{F}_{\mathrm{CD}}$ is used to store counters on the number of purchases of each item category in the positions $[1, \mathcal{N}_{max}]$ of table Tbl_{cd}. In the pot.transfer interface, $\mathcal{B}$ uses cd.read to read the table entry $[\sigma, count_1]$ in Tbl_{cd}, where σ is the index of the message purchased m_σ and $count_1$ is the counter for that category. $\mathcal{B}$ computes $count_2 \leftarrow count_1 + 1$ and uses $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$ to prove in zero-knowledge that the counter is correctly incremented. Then $\mathcal{B}$ uses cd.write to write the entry $[\sigma, count_2]$ in Tbl_{cd}. In the pot.revealstatistic interface, $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$ to prove in zero-knowledge to $\mathcal{V}$ that a statistic v is the result of evaluating a function ST on input Tbl_{cd}. For this purpose, $\mathcal{B}$ uses cd.read to read the required table entries in Tbl_{cd}.

Figure 11.2: Construction Π_{POTS}

Π_{POTS} uses $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R}$, and $\mathcal{F}_{\text{OT}}$ from Chapter 2, and $\mathcal{F}_{\text{CD}}$ and $\mathcal{F}_{\text{UD}}$ from Chapter 4 and Chapter 8 respectively. Π_{POTS} is parameterized with a maximum number of messages $\mathcal{N}_{max}$, a message space $\mathcal{M}$, a maximum deposit value dep_{max}, a maximum price $\mathcal{P}_{max}$, and a universe of statistics Ψ consisting of ppt algorithms.

1. On input $(\mathsf{pot.init.ini}, sid, ep, \langle m_n \rangle_{n=1}^{N})$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{V}$ aborts if (sid, ep', N) is already stored for $ep' = ep$, else stores (sid, ep, N).
   - $\mathcal{V}$ aborts if $N > \mathcal{N}_{max}$, or if for $n = 1$ to N, $m_n \notin \mathcal{M}$.
   - If this is the first execution of this interface, $\mathcal{V}$ and $\mathcal{B}$ do the following:
     - $\mathcal{V}$ sets a table Tbl_{cd} of $\mathcal{N}_{max}$ entries where each entry is of the form $[i, 0]$ for $i = 0$ to $\mathcal{N}_{max}$.
     - $\mathcal{V}$ sends $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$ to a new instance of $\mathcal{F}_{\text{CD}}$.
     - $\mathcal{B}$ receives $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$ from $\mathcal{F}_{\text{CD}}$.
     - If (sid, Tbl_{cd}) is already stored or if, for $i = 0$ to $\mathcal{N}_{max}$, there exists an entry $[i, v]$ in Tbl_{cd} such that $v \neq 0$, $\mathcal{B}$ aborts, else $\mathcal{B}$ stores (sid, Tbl_{cd}).
     - $\mathcal{B}$ sets $sid_{\text{AUT}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$ and sends $(\mathsf{aut.send.ini}, sid_{\text{AUT}}, \langle \mathsf{setup} \rangle)$ to $\mathcal{F}_{\text{AUT}}$.
     - $\mathcal{V}$ receives $(\mathsf{aut.send.end}, sid_{\text{AUT}}, \langle \mathsf{setup} \rangle)$ from $\mathcal{F}_{\text{AUT}}$.
   - $\mathcal{V}$ sets $sid_{\text{OT}} \leftarrow (sid, ep)$.
   - $\mathcal{V}$ sends $(\mathsf{ot.init.ini}, sid_{\text{OT}}, \langle m_n \rangle_{n=1}^{N})$ to a new instance of $\mathcal{F}_{\text{OT}}$.
   - $\mathcal{B}$ receives $(\mathsf{ot.init.end}, sid_{\text{OT}}, N)$ from the instance of $\mathcal{F}_{\text{OT}}$.
   - $\mathcal{B}$ takes ep from sid_{OT} and stores (sid, ep, N).
   - $\mathcal{B}$ outputs $(\mathsf{pot.init.end}, sid, ep, N)$.
2. On input $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$ or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ is already stored.
   - $\mathcal{V}$ aborts if, for $n = 1$ to $\mathcal{N}_{max}$, $p_n \notin (0, \mathcal{P}_{max}]$.
   - $\mathcal{V}$ stores $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
   - For $n = 1$ to $\mathcal{N}_{max}$, $\mathcal{V}$ sets a table DB with entries $[n, p_n]$.

- $\mathcal{V}$ sends (ud.update.ini, sid, DB) to a new instance of $\mathcal{F}_{\text{UD}}$.
- $\mathcal{B}$ receives (ud.update.end, sid, DB) from $\mathcal{F}_{\text{UD}}$.
- $\mathcal{B}$ parses DB as $[n, p_n]$, for $n = 1$ to $\mathcal{N}_{max}$, and stores $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
- $\mathcal{B}$ outputs (pot.setupprices.end, sid, $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$).

3. On input (pot.updateprice.ini, sid, n, p), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ aborts if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ is not stored.
   - $\mathcal{V}$ aborts if $n \notin [1, \mathcal{N}_{max}]$, or if $p \notin (0, \mathcal{P}_{max}]$.
   - For $i = 1$ to $\mathcal{N}_{max}$, $\mathcal{V}$ sets a table DB_{temp} with entries $[i, 0]$, and $[i, p]$ where $i = n$.
   - $\mathcal{V}$ sends (ud.update.ini, sid, DB_{temp}) to $\mathcal{F}_{\text{UD}}$.
   - $\mathcal{B}$ receives (ud.update.end, sid, DB_{temp}) from $\mathcal{F}_{\text{UD}}$.
   - $\mathcal{B}$ sets $p_n \leftarrow p$) and updates the stored tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
   - $\mathcal{B}$ outputs (pot.updateprice.end, sid, n, p).

4. On input (pot.deposit.ini, sid, dep), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored for any ep.
   - $\mathcal{B}$ retrieves $[0, v]$ from Tbl_{cd} and sets $dep_1 \leftarrow v$.
   - $\mathcal{B}$ aborts if $dep_1 + dep \notin [0, dep_{max}]$.
   - If this is the first execution of the deposit interface, $\mathcal{B}$ does the following:
     - $\mathcal{B}$ sends (com.setup.ini, sid) to $\mathcal{F}_{\text{NIC}}$.
     - $\mathcal{B}$ receives (com.setup.end, sid, OK) from $\mathcal{F}_{\text{NIC}}$.
   - If $(sid, com_{dep_2}, open_{dep_2})$ is already stored, $\mathcal{B}$ sets $com_{dep_1} \leftarrow com_{dep_2}$ and $open_{dep_1} \leftarrow open_{dep_2}$. Otherwise, $\mathcal{B}$ does the following:
     - $\mathcal{B}$ sends (com.commit.ini, sid, dep_1) to $\mathcal{F}_{\text{NIC}}$.
     - $\mathcal{B}$ receives (com.commit.end, sid, com_{dep_1}, $open_{dep_1}$) from $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{B}$ sets $dep_2 \leftarrow dep_1 + dep$.
   - $\mathcal{B}$ sends (com.commit.ini, sid, dep) to $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{B}$ receives (com.commit.end, sid, com_{dep}, $open_{dep}$) from $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{B}$ sends (com.commit.ini, sid, dep_2) to $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{B}$ receives (com.commit.end, sid, com_{dep_2}, $open_{dep_2}$) from $\mathcal{F}_{\text{NIC}}$.

- $\mathcal{B}$ sets $wit_{dep} \leftarrow (dep, open_{dep}, dep_1, open_{dep_1}, dep_2, open_{dep_2})$.
- $\mathcal{B}$ parses the commitment com_{dep} as $(com'_{dep}, parcom, \mathsf{COM.Verify})$, the commitment com_{dep_1} as $(com'_{dep_1}, parcom, \mathsf{COM.Verify})$, and the commitment com_{dep_2} as $(com'_{dep_2}, parcom, \mathsf{COM.Verify})$.
- $\mathcal{B}$ sets $ins_{dep} \leftarrow (parcom, com'_{dep}, com'_{dep_1}, com'_{dep_2})$.
- $\mathcal{B}$ stores $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$.

The R_{dep} is defined as follows.

$$\begin{aligned} R_{dep} = \{(wit_{dep}, ins_{dep}) : \\ & 1 = \mathsf{COM.Verify}(parcom, com_{dep}, dep, open_{dep}) \wedge \\ & 1 = \mathsf{COM.Verify}(parcom, com_{dep_1}, dep_1, open_{dep_1}) \wedge \\ & 1 = \mathsf{COM.Verify}(parcom, com_{dep_2}, dep_2, open_{dep_2}) \wedge \\ & dep_2 = dep + dep_1 \wedge dep_2 \in [0, dep_{max}]\} \end{aligned}$$

- $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$, and sends $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{dep}, ins_{dep})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$.
- $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{dep})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$.
- If this is the first execution of the deposit interface, $\mathcal{V}$ does the following:
  - $\mathcal{V}$ sends $(\mathsf{com.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.setup.end}, sid, OK)$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ parses ins_{dep} as $(parcom, com'_{dep}, com'_{dep_1}, com'_{dep_2})$.
- $\mathcal{V}$ sets the commitment $com_{dep} \leftarrow (com'_{dep}, parcom, \mathsf{COM.Verify})$, the commitment $com_{dep_1} \leftarrow (com'_{dep_1}, parcom, \mathsf{COM.Verify})$, and the commitment $com_{dep_2} \leftarrow (com'_{dep_2}, parcom, \mathsf{COM.Verify})$.
- $\mathcal{V}$ aborts if (sid, com'_{dep_2}) is stored and $com'_{dep_2} \neq com_{dep_1}$.
- If (sid, com'_{dep_2}) is not stored, $\mathcal{V}$ does the following:
  - $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_{dep_1})$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_1})$ from $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ aborts if $b_{dep_1} \neq 1$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_{dep_2})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_{dep})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep})$ from $\mathcal{F}_{\mathrm{NIC}}$.

- $\mathcal{V}$ aborts if $b_{dep_2} = b_{dep} = 1$ does not hold.
- $\mathcal{V}$ stores (sid, ins_{dep}).
- $\mathcal{V}$ sets $sid'_{\mathrm{AUT}} \leftarrow (sid)$ and sends $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{writedeposit} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{writedeposit} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$ is not stored.
- $\mathcal{B}$ parses ins_{dep} as $(parcom, com_{dep}, com_{dep_1}, com_{dep_2})$.
- $\mathcal{B}$ parses wit_{dep} as $(dep, open_{dep}, dep_1, open_{dep_1}, dep_2, open_{dep_2})$.
- $\mathcal{B}$ deletes $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$ and stores $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$.
- If this is the first execution of this interface, $\mathcal{B}$ does the following:
  - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, 0)$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, com_0, open_0)$ from $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{B}$ stores $(sid, com_0, open_0)$.
- $\mathcal{B}$ stores $(sid, com_{dep_2}, open_{dep_2})$.
- $\mathcal{B}$ sends to $\mathcal{F}_{\mathrm{CD}}$ the message $(\mathsf{cd.write.ini}, sid, com_0, 0, open_0, com_{dep_2}, dep_2, open_{dep_2})$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, com_0, com_{dep_2})$ from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ aborts if (sid, ins_{dep}) is not stored.
- $\mathcal{V}$ aborts if the commitment com_{dep_2} in ins_{dep} is not the same as that received from $\mathcal{F}_{\mathrm{CD}}$, or if com_0 stored in (sid, com_0) is not the same as the commitment received from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \langle \text{revealdeposit} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle \text{revealdeposit} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$ is not stored.
- $\mathcal{B}$ deletes the record $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$.
- $\mathcal{B}$ updates Tbl_{cd} with $[0, dep_2]$.
- $\mathcal{B}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$.
- If this is the first execution of this interface, $\mathcal{B}$ and $\mathcal{V}$ do the following:
  - $\mathcal{B}$ sends the following message $(\mathsf{smt.send.ini}, sid_{\mathrm{SMT}}, \langle dep, open_{dep}, 0, open_0, dep_1, open_{dep_1} \rangle)$ to functionality $\mathcal{F}_{\mathrm{SMT}}$.

  - $\mathcal{V}$ receives from functionality $\mathcal{F}_{\mathrm{SMT}}$ the following message $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle dep, open_{dep}, 0, open_0, dep_1, open_{dep_1}\rangle)$.
  - $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, com_0, 0, open_0)$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v_0)$ from $\mathcal{F}_{\mathrm{NIC}}$ and aborts if $v_0 \neq 1$.
  - $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, com_{dep_1}, dep_1, open_{dep_1})$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v_{dep_1})$ from $\mathcal{F}_{\mathrm{NIC}}$ and aborts if $v_{dep_1} \neq 1$.
  - $\mathcal{V}$ aborts if $dep_1 = 0$ does not hold.
- Otherwise, $\mathcal{B}$ and $\mathcal{V}$ do the following:
  - $\mathcal{B}$ sends the message $(\mathsf{smt.send.ini}, sid_{\mathrm{SMT}}, \langle dep, open_{dep}\rangle)$ to $\mathcal{F}_{\mathrm{SMT}}$.
  - $\mathcal{V}$ receives $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle dep, open_{dep}\rangle)$ from $\mathcal{F}_{\mathrm{SMT}}$.
- $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, com_{dep}, dep, open_{dep})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v)$ from $\mathcal{F}_{\mathrm{NIC}}$ and aborts if $v \neq 1$.
- $\mathcal{V}$ aborts if $dep \notin [0, dep_{max}]$.
- $\mathcal{V}$ outputs $(\mathsf{pot.deposit.end}, sid, dep)$.

5. On input $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored, or if $(sid, \langle p_n\rangle_{n=1}^{\mathcal{N}_{max}})$ is not stored, or if $\sigma \notin [1, N]$.
   - $\mathcal{B}$ retrieves $[0, v]$ from Tbl_{cd}, and sets $dep_1 \leftarrow v$.
   - $\mathcal{B}$ aborts if $dep_1 < p_\sigma$.
   - $\mathcal{B}$ sets $com_{dep_1} \leftarrow com_{dep_2}$ and $open_{dep_1} \leftarrow open_{dep_2}$.
   - $\mathcal{B}$ sets $dep_2 \leftarrow dep_1 - p_\sigma$.
   - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, dep_2)$ to $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, com_{dep_2}, open_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, \sigma)$ to $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, com_\sigma, open_\sigma)$ from $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, p_\sigma)$ to $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, com_{p_\sigma}, open_{p_\sigma})$ from $\mathcal{F}_{\mathrm{NIC}}$.

- $\mathcal{B}$ sets $wit_{trans} \leftarrow (p_\sigma, open_{p_\sigma}, dep_1, open_{dep_1}, dep_2, open_{dep_2})$.
- $\mathcal{B}$ parses the commitment com_{p_σ} as $(com'_{p_\sigma}, parcom, \mathsf{COM.Verify})$, the commitment com_{dep_1} as $(com'_{dep_1}, parcom, \mathsf{COM.Verify})$, and the commitment com_{dep_2} as $(com'_{dep_2}, parcom, \mathsf{COM.Verify})$.
- $\mathcal{B}$ sets $ins_{trans} \leftarrow (parcom, com'_{p_\sigma}, com'_{dep_1}, com'_{dep_2})$.
- $\mathcal{B}$ stores $(sid, wit_{trans}, ins_{trans}, \text{readprice})$.

The relation R_{trans} is defined as follows:

$$\begin{aligned} R_{trans} = \{&(wit_{trans}, ins_{trans}) : \\ &1 = \mathsf{COM.Verify}(parcom, com_{p_\sigma}, p_\sigma, open_{p_\sigma}) \wedge \\ &1 = \mathsf{COM.Verify}(parcom, com_{dep_1}, dep_1, open_{dep_1}) \wedge \\ &1 = \mathsf{COM.Verify}(parcom, com_{dep_2}, dep_2, open_{dep_2}) \wedge \\ &dep_2 = dep_1 - p_\sigma \ \wedge \ dep_2 \in [0, dep_{max}]\} \end{aligned}$$

- $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$, and sends the message $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{trans}, ins_{trans})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$.
- $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{trans})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$.
- $\mathcal{V}$ parses ins_{trans} as $(parcom, com'_{p_\sigma}, com'_{dep_1}, com'_{dep_2}, pk)$.
- $\mathcal{V}$ sets the commitment $com_{p_\sigma} \leftarrow (com'_{p_\sigma}, parcom, \mathsf{COM.Verify})$, the commitment $com_{dep_1} \leftarrow (com'_{dep_1}, parcom, \mathsf{COM.Verify})$, and the commitment $com_{dep_2} \leftarrow (com'_{dep_2}, parcom, \mathsf{COM.Verify})$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_{p_\sigma})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{p_\sigma})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{p_\sigma} \neq 1$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_{dep_2})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{dep_2} \neq 1$.
- $\mathcal{V}$ aborts if com_{dep_2} stored in (sid, com_{dep_2}) is not the same as com_{dep_1} in ins_{trans}.
- $\mathcal{V}$ stores (sid, ins_{trans}).
- $\mathcal{V}$ sets $sid'_{\mathrm{AUT}} \leftarrow (sid)$ and sends to functionality $\mathcal{F}_{\mathrm{AUT}}$ the following message $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{readprice} \rangle)$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle \text{readprice} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{trans}, ins_{trans}, \text{readprice})$ is not stored.

- $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans}, \text{readprice})$ and stores $(sid, wit_{trans}, ins_{trans}, \text{commitdeposit})$.
- $\mathcal{B}$ sends to functionality $\mathcal{F}_{\text{UD}}$ the following message $(\mathsf{ud.read.ini}, sid, com_\sigma, \sigma, open_\sigma, com_{p_\sigma}, p_\sigma, open_{p_\sigma})$.
- $\mathcal{B}$ receives $(\mathsf{ud.read.end}, sid, com_\sigma, com_{p_\sigma})$ from $\mathcal{F}_{\text{UD}}$.
- $\mathcal{V}$ aborts if (sid, ins_{trans}) is not stored.
- $\mathcal{V}$ aborts if the com_{p_σ} in ins_{trans} is not the same as that received from $\mathcal{F}_{\text{UD}}$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, com_\sigma)$ to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_\sigma)$ from $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ aborts if $b_\sigma \neq 1$.
- $\mathcal{V}$ adds com_σ to ins_{trans} and stores (sid, ins_{trans}).
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\text{AUT}}, \langle \text{commitdeposit} \rangle)$ to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\text{AUT}}, \langle \text{commitdeposit} \rangle)$ from $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans}, \text{commitdeposit})$ and stores $(sid, wit_{trans}, ins_{trans}, \text{commitcounter})$.
- $\mathcal{B}$ sends to $\mathcal{F}_{\text{CD}}$ the message $(\mathsf{cd.write.ini}, sid, com_0, 0, open_0, com_{dep_2}, dep_2, open_{dep_2})$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, com_0, com_{dep_2})$ from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ aborts if (sid, ins_{trans}) is not stored.
- $\mathcal{V}$ aborts if the com_{dep_2} in ins_{trans} is not the same as that received from $\mathcal{F}_{\text{CD}}$, or if com_0 received from $\mathcal{F}_{\text{CD}}$ is not the same as com_0 stored by $\mathcal{V}$ during the first execution of the deposit interface.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\text{AUT}}, \langle \text{commitcounter} \rangle)$ to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\text{AUT}}, \langle \text{commitcounter} \rangle)$ from $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{trans}, ins_{trans}, \text{commitcounter})$ is not stored.
- $\mathcal{B}$ stores (sid, com_{dep_2}).
- $\mathcal{B}$ updates Tbl_{cd} with $[0, dep_2]$.
- $\mathcal{B}$ retrieves $[\sigma, v]$ from Tbl_{cd} and sets $count_1 \leftarrow v$.
- $\mathcal{B}$ sets $count_2 \leftarrow count_1 + 1$.
- $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, count_1)$ to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, com_{count_1}, open_{count_1})$ from $\mathcal{F}_{\text{NIC}}$.

- $\mathcal{B}$ sends (com.commit.ini, sid, $count_2$) to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{B}$ receives (com.commit.end, sid, com_{count_2}, $open_{count_2}$) from $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{B}$ sets $wit_{count} \leftarrow (\sigma, open_\sigma, count_1, open_{count_1}, count_2, open_{count_2})$.
- $\mathcal{B}$ parses the commitment com_σ as $(com'_\sigma, parcom,$ COM.Verify), the commitment com_{count_1} as $(com'_{count_1}, parcom,$ COM.Verify), and the commitment com_{count_2} as $(com'_{count_2}, parcom,$ COM.Verify).
- $\mathcal{B}$ sets $ins_{count} \leftarrow (parcom, com'_\sigma, com'_{count_1}, com'_{count_2})$.
- $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans},$ commitcounter) and stores the tuple $(sid, wit_{count}, ins_{count},$ readcounter).

The relation R_{count} is defined as follows:

$$\begin{aligned} R_{count} = \{&(wit_{count}, ins_{count}) : \\ &1 = \text{COM.Verify}(parcom, com_\sigma, \sigma, open_\sigma) \wedge \\ &1 = \text{COM.Verify}(parcom, com_{count_1}, count_1, open_{count_1}) \wedge \\ &1 = \text{COM.Verify}(parcom, com_{count_2}, count_2, open_{count_2}) \wedge \\ &count_2 = count_1 + 1\} \end{aligned}$$

- $\mathcal{B}$ sends (zk.prove.ini, sid_{ZK}, wit_{count}, ins_{count}) to a new instance of $\mathcal{F}_{\text{ZK}}^{R_{count}}$.
- $\mathcal{V}$ receives (zk.prove.end, sid_{ZK}, ins_{count}) from $\mathcal{F}_{\text{ZK}}^{R_{count}}$.
- $\mathcal{V}$ parses ins_{count} as $(parcom, com'_\sigma, com'_{count_1}, com'_{count_2})$.
- $\mathcal{V}$ sets the commitment $com_\sigma \leftarrow (com'_\sigma, parcom,$ COM.Verify), the commitment $com_{count_1} \leftarrow (com'_{count_1}, parcom,$ COM.Verify), and the commitment $com_{count_2} \leftarrow (com'_{count_2}, parcom,$ COM.Verify).
- $\mathcal{V}$ sends (com.validate.ini, sid, com_{count_1}) to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ receives (com.validate.end, sid, b_{count_1}) from $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ aborts if $b_{count_1} \neq 1$.
- $\mathcal{V}$ sends (com.validate.ini, sid, com_{count_2}) to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ receives (com.validate.end, sid, b_{count_2}) from $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ aborts if $b_{count_2} \neq 1$.
- $\mathcal{V}$ aborts if the commitment com_σ in ins_{trans} is not the same as the commitment com_σ in ins_{count}.
- $\mathcal{V}$ stores (sid, ins_{count}).
- $\mathcal{V}$ sends the message (aut.send.ini, sid'_{AUT}, ⟨readcounter⟩) to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives (aut.send.end, sid'_{AUT}, ⟨readcounter⟩) from $\mathcal{F}_{\text{AUT}}$.

- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{readcounter})$ is not stored.
- $\mathcal{B}$ parses ins_{count} as $(parcom, com_{\sigma}, com_{count_1}, com_{count_2})$.
- $\mathcal{B}$ parses wit_{count} as $(\sigma, open_{\sigma}, count_1, open_{count_1}, count_2, open_{count_2})$.
- $\mathcal{B}$ deletes the tuple $(sid, wit_{count}, ins_{count}, \text{readcounter})$ and stores the tuple $(sid, wit_{count}, ins_{count}, \text{writecounter})$.
- $\mathcal{B}$ sends the message $(\mathsf{cd.read.ini}, sid, com_{\sigma}, \sigma, open_{\sigma}, com_{count_1}, count_1, open_{count_1})$ to $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ receives $(\mathsf{cd.read.end}, sid, com_{\sigma}, com_{count_1})$ from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ aborts if ins_{count} is not stored.
- $\mathcal{V}$ aborts if the commitments in ins_{count} are not the same as those received from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\text{AUT}}, \langle\text{writecounter}\rangle)$ to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\text{AUT}}, \langle\text{writecounter}\rangle)$ from $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{writecounter})$ is not stored.
- $\mathcal{B}$ deletes the record $(sid, wit_{count}, ins_{count}, \text{writecounter})$ and stores the tuple $(sid, wit_{count}, ins_{count}, \text{transfer})$.
- $\mathcal{B}$ sends to functionality $\mathcal{F}_{\text{CD}}$the following message $(\mathsf{cd.write.ini}, sid, com_{\sigma}, \sigma, open_{\sigma}, com_{count_2}, count_2, open_{count_2})$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, com_{\sigma}, com_{count_2})$ from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ aborts if ins_{count} is not stored.
- $\mathcal{V}$ aborts if the commitments in ins_{count} are not the same as those received from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\text{AUT}}, \langle\text{transfer}\rangle)$ to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\text{AUT}}, \langle\text{transfer}\rangle)$ from $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{transfer})$ is not stored.
- $\mathcal{B}$ updates Tbl_{cd} with $[\sigma, count_2]$.
- $\mathcal{B}$ sets $sid_{\text{OT}} \leftarrow (sid, ep)$, and sends the message $(\mathsf{ot.request.ini}, sid_{\text{OT}}, \sigma, com_{\sigma}, open_{\sigma})$ to $\mathcal{F}_{\text{OT}}$.
- $\mathcal{V}$ receives $(\mathsf{ot.request.end}, sid_{\text{OT}}, com_{\sigma})$ from $\mathcal{F}_{\text{OT}}$.
- $\mathcal{V}$ aborts if com_{σ} received from $\mathcal{F}_{\text{OT}}$ is not the same as that contained in ins_{count}.
- $\mathcal{V}$ sends the message $(\mathsf{ot.transfer.ini}, sid_{\text{OT}}, com_{\sigma})$ to $\mathcal{F}_{\text{OT}}$.
- $\mathcal{B}$ receives $(\mathsf{ot.transfer.end}, sid_{\text{OT}}, m_{\sigma})$ from $\mathcal{F}_{\text{OT}}$.

- $\mathcal{B}$ outputs (pot.transfer.end, sid, m_σ).

6. On input (pot.revealstatistic.ini, sid, ST), $\mathcal{B}$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored for any ep.
   - $\mathcal{B}$ aborts if $\mathsf{ST} \notin \psi$.
   - $\mathcal{B}$ computes $result \leftarrow \mathsf{ST}(\mathsf{Tbl}_{cd})$.
   - For each entry $[i, v_i]$ in Tbl_{cd}, where v_i represents a value that was used by $\mathcal{B}$ to compute $result$, $\mathcal{B}$ and $\mathcal{V}$ do the following:
     - $\mathcal{B}$ sends the message (com.commit.ini, sid, i) to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives (com.commit.end, $sid, com_i, open_i$).
     - $\mathcal{B}$ sends the message (com.commit.ini, sid, v_i) to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives (com.commit.end, $sid, com_{v_i}, open_{v_i}$).
     - $\mathcal{B}$ stores (sid, com_i, com_{v_i}).
     - $\mathcal{B}$ sends (cd.read.ini, $sid, com_i, i, open_i, com_{v_i}, v_i, open_{v_i}$) to $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{V}$ receives (cd.read.end, sid, com_i, com_{v_i}) from $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{V}$ sends the message (com.validate.ini, sid, com_i) to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ receives (com.validate.end, sid, b_i) from $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ sends the message (com.validate.ini, sid, com_{v_i}) to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ receives (com.validate.end, sid, b_{v_i}) from $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ aborts if $b_i = b_{v_i} = 1$ does not hold.
     - $\mathcal{V}$ sets $sid_{\mathrm{AUT}} \leftarrow (sid)$ and sends (aut.send.ini, $sid_{\mathrm{AUT}}, \langle OK, com_i, com_{v_i} \rangle$) to $\mathcal{F}_{\mathrm{AUT}}$.
     - $\mathcal{B}$ receives (aut.send.end, $sid_{\mathrm{AUT}}, \langle OK, com_i, com_{v_i} \rangle$) from $\mathcal{F}_{\mathrm{AUT}}$.
     - $\mathcal{B}$ aborts if (sid, com_i, com_{v_i}) is not stored.
   - $\mathcal{B}$ sets $wit_{\mathsf{ST}} \leftarrow (\langle i, open_i, v_i, open_{v_i} \rangle_{\forall i})$.
   - $\mathcal{B}$ parses the commitment com_i as $(com'_i, parcom,$ COM.Verify) and the commitment com_{v_i} as $(com'_{v_i}, parcom,$ COM.Verify) for all i.
   - $\mathcal{B}$ sets $ins_{\mathsf{ST}} \leftarrow (result, parcom, \langle com'_i, com'_{v_i} \rangle_{\forall i})$.

The relation R_{ST} is defined as follows:

$$
\begin{aligned}
R_{\mathsf{ST}} = \{ & (wit_{\mathsf{ST}}, ins_{\mathsf{ST}}) : \\
& [\, \forall i \; 1 = \mathsf{COM.Verify}(parcom, com_i, i, open_i) \wedge \\
& 1 = \mathsf{COM.Verify}(parcom, com_{v_i}, v_i, open_{v_i}) \,] \wedge \\
& result = \mathsf{ST}(\langle i, v_i \rangle_{\forall i}) \,\}
\end{aligned}
$$

- $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$ and sends $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{\mathsf{ST}}, ins_{\mathsf{ST}})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$.
- $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{\mathsf{ST}})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$.
- $\mathcal{V}$ aborts if the commitments received from $\mathcal{F}_{\mathrm{CD}}$ are not the same as those in ins_{ST}.
- $\mathcal{V}$ outputs $(\mathsf{pot.revealstatistic.end}, sid, result, \mathsf{ST})$.

11.5 SECURITY ANALYSIS

Theorem 11.5.1 *Construction Π_{POTS} realizes functionality $\mathcal{F}_{\mathrm{POTS}}$ in the $(\mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{SMT}}, \mathcal{F}_{\mathrm{NIC}} || \mathcal{S}_{\mathrm{NIC}}, \mathcal{F}_{\mathrm{ZK}}^{R}, \mathcal{F}_{\mathrm{OT}}, \mathcal{F}_{\mathrm{CD}}, \mathcal{F}_{\mathrm{UD}})$-hybrid model.*

To prove that our construction Π_{POTS} securely realizes the ideal functionality $\mathcal{F}_{\mathrm{POTS}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{POTS}}$. The simulator thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{POTS}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, and $\mathcal{F}_{\mathrm{UD}}$. When any of the copies of these functionalities abort, $\mathcal{S}$ implicitly forwards the abortion message to the adversary if the functionality sends the abortion message to a corrupt party.

SECURITY ANALYSIS OF Π_{POTS} WHEN $\mathcal{B}$ IS CORRUPT

We first describe the simulator $\mathcal{S}$ for the case in which the buyer $\mathcal{B}$ is corrupt. $\mathcal{S}$ simulates the protocol by running the seller's side of protocol Π_{POTS} and copies of the ideal functionalities involved.

Figure 11.8: Security Analysis of Π_{POTS} when $\mathcal{B}$ is corrupt

- Upon receiving a message from $\mathcal{A}$, $\mathcal{S}$ uses the first field of that message to associate the message with one of the ideal functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{UD}}$, and runs a copy of the corresponding functionality on input that message.
- When a copy of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{UD}}$ sends a message to $\mathcal{B}$ or to $\mathcal{S}$, $\mathcal{S}$ forwards the output of the functionality to $\mathcal{A}$.

- When a copy of the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, or $\mathcal{F}_{\text{UD}}$ sends a message to $\mathcal{V}$, $\mathcal{S}$ runs protocol Π_{POTS} for the seller on input that message, except when $\mathcal{F}_{\text{NIC}}$ sends a delayed output (com.setup.end, sid, OK) to $\mathcal{V}$.
- When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.init.sim, sid, ep, N) to $\mathcal{S}$, $\mathcal{S}$ sends the message (pot.init.rep, sid, ep) to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.setupprices.sim, sid, $\langle p \rangle_{n=1}^{N_{max}}$) to $\mathcal{S}$, $\mathcal{S}$ sends the message (pot.setupprices.rep, sid) to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.updateprice.sim, sid, qid, n, p) to the simulator $\mathcal{S}$, the simulator $\mathcal{S}$ sends the message (pot.updateprice.rep, sid, qid) to functionality $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message (pot.init.end, sid, ep, N), $\mathcal{S}$ runs the protocol Π_{POTS} for the seller on input (pot.init.ini, sid, ep, $\langle m_n \rangle_{n=1}^{N}$), where $\langle m_n \rangle_{n=1}^{N}$ are selected at random.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message (pot.setupprices.end, sid, $\langle p_n \rangle_{n=1}^{N_{max}}$), $\mathcal{S}$ runs protocol Π_{POTS} for the seller on input (pot.setupprices.ini, sid, $\langle p_n \rangle_{n=1}^{N_{max}}$).
- When $\mathcal{F}_{\text{POTS}}$ outputs the message (pot.updateprice.end, sid, n, p), $\mathcal{S}$ runs protocol Π_{POTS} for the seller on input (pot.updateprice.ini, sid, n, p).
- When protocol Π_{POTS} for the seller outputs the message (pot.deposit.end, sid, dep), $\mathcal{S}$ sends the message (pot.deposit.ini, sid, dep) to $\mathcal{F}_{\text{POTS}}$.
  When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.deposit.sim, sid, qid) to $\mathcal{S}$, $\mathcal{S}$ sends (pot.deposit.rep, sid, qid) to $\mathcal{F}_{\text{POTS}}$.
- When protocol Π_{POTS} outputs the message (pot.transfer.end, sid, m_σ), the simulator $\mathcal{S}$ retrieves the message (ot.request.ini, sid, σ, com_σ, $open_\sigma$) sent by the adversary $\mathcal{A}$ to the ideal functionality $\mathcal{F}_{\text{OT}}$. The simulator $\mathcal{S}$ sets $m'_\sigma \leftarrow m_\sigma$ for the copy of $\mathcal{F}_{\text{OT}}$ associated with sid'_{OT}, such that $sid'_{\text{OT}} \in (sid, ep)$. The simulator $\mathcal{S}$ sends the message (pot.transfer.ini, sid, ep, σ) to the functionality $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.transfer.sim, sid, qid, ep) to $\mathcal{S}$, $\mathcal{S}$ sends (pot.transfer.rep, sid, qid) to $\mathcal{F}_{\text{POTS}}$.
- When protocol Π_{POTS} for the seller outputs (pot.revealstatistic.end, sid, v, ST), $\mathcal{S}$ sends the message (pot.revealstatistic.ini, sid, ST) to $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message (pot.revealstatistic.sim, sid, qid) to $\mathcal{S}$, $\mathcal{S}$ sends the message (pot.revealstatistic.rep, sid, qid) to $\mathcal{F}_{\text{POTS}}$.
- $\mathcal{S}$ outputs failure if $\mathcal{A}$ produces two openings for a commitment.

Theorem 11.5.2 *When the buyer $\mathcal{B}$ is corrupt, the construction Π_{POTS} described in Section 11.5 securely realizes $\mathcal{F}_{\text{POTS}}$ in the $(\mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{NIC}}||\mathcal{S}_{\text{NIC}}, \mathcal{F}_{\text{ZK}}^{R}, \mathcal{F}_{\text{OT}}, \mathcal{F}_{\text{CD}}, \mathcal{F}_{\text{UD}})$-hybrid model.*

PROOF OF THEOREM 11.5.2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\Pi_{\text{POTS}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{POTS}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

GAME 0: This game corresponds to the execution of the real-world protocol.
Therefore, $\Pr[\textbf{Game } 0] = 0$.

GAME 1: This game proceeds as **Game** 0, except that **Game** 1 replaces the messages $\langle m \rangle_{n=1}^{N_{max}}$ that are sent as input to the ot.init interface by random messages. This change does not alter the view of the environment because, in the ot.init interface, $\mathcal{F}_{\text{OT}}$ does not send the messages to the simulator or to the buyer. Moreover, **Game** 1 copies the correct message m_σ to the copy of $\mathcal{F}_{\text{OT}}$ before the ot.transfer interface is executed, so the corrupt buyer receives the correct message. Hence, $[\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]] = 0$.

GAME 2: This game proceeds as **Game** 1, except for the fact that **Game** 2 outputs failure when the adversary produces two openings for the same commitment. However, the probability that the adversary may produce two such openings is negligible, thanks to the binding property enforced by $\mathcal{F}_{\text{NIC}}$. Therefore, $[\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]] = 0$.

$\mathcal{S}$ is indistinguishable from the real world protocol because it runs as the real-world protocol, except when it outputs failure. The probability that $\mathcal{S}$ outputs failure is negligible thanks to the security properties ensured by the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, and $\mathcal{F}_{\text{UD}}$. $\mathcal{F}_{\text{NIC}}$ serves to ensure that all commitments used in the protocol are binding, while $\mathcal{F}_{\text{ZK}}^{R}$ guarantees that the witness and instance values provided by the buyer for the deposit, transfer, and revealstatistic phases of the protocol satisfy the relations R_{dep}, R_{trans} and R_{count}, and R_{ST} respectively. R_{dep} is used to check whether the buyer is updating deposit values correctly during the deposit phase, while also ensuring that the commitments com_{dep}, com_{dep_1}, and com_{dep_2} commit to the deposit value dep, the initial deposit dep_1, and the final deposit value dep_2. The protocol relies on R_{trans} to ensure that the buyer updates deposit and counter values consistently after a purchase, while also ensuring that the commitments to the message price p_σ, initial and final deposit values dep_1 and dep_2, counter index σ, and initial and final counter values $count_1$ and $count_2$ are valid. R_{ST} is used to make sure that the result of the evaluation of function ST on Tbl_{cd} is accurate, and that the commitments to all the indices and values of the counters in Tbl_{cd} used to determine the result of $\mathsf{ST}(\mathsf{Tbl}_{cd})$ are valid. $\mathcal{F}_{\text{CD}}$ ensures that the buyer's deposit value dep and all counter values $\langle count_n \rangle_{n=1}^{N_{max}}$ read from positions in $\mathsf{Tbl}_{\mathcal{S}}$ are

equal to the values previously written to those positions. $\mathcal{F}_{\text{UD}}$ is used to ensure that the buyer can prove that she is using the right prices for the right messages (p_σ is used for m_σ). Finally, $\mathcal{F}_{\text{OT}}$ ensures sender security (the buyer does not learn any information about messages that have not been purchased).

The distribution of **Game** 2 is identical to that of our simulation. This concludes the proof of theorem II.5.2.

SECURITY ANALYSIS OF Π_{POTS} WHEN $\mathcal{V}$ IS CORRUPT In this section, we describe the simulator $\mathcal{S}$ for the case in which the seller $\mathcal{V}$ is corrupt. $\mathcal{S}$ simulates the protocol by running copies of the ideal functionalities and protocol Π_{POTS} for the buyer.

Figure II.11: Security Analysis of Π_{POTS} when $\mathcal{V}$ is corrupt

- Upon receiving a message from $\mathcal{A}$, $\mathcal{S}$ uses the first field of that message to associate the message with one of the ideal functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}^R_{\text{ZK}}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, or $\mathcal{F}_{\text{UD}}$, and runs a copy of the corresponding functionality on input that message.
- When a copy of the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}^R_{\text{ZK}}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, or $\mathcal{F}_{\text{UD}}$ sends a message to $\mathcal{V}$ or to $\mathcal{S}$, $\mathcal{S}$ forwards the output of the functionality to $\mathcal{A}$, except when $\mathcal{F}_{\text{NIC}}$ outputs a delayed message $(\mathsf{com.setup.end}, sid, OK)$.
- When a copy of the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}^R_{\text{ZK}}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, or $\mathcal{F}_{\text{UD}}$ sends a message to $\mathcal{B}$, $\mathcal{S}$ runs protocol Π_{POTS} for the buyer on input that message.
- When protocol Π_{POTS} for the buyer sends a message to any of the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}^R_{\text{ZK}}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, or $\mathcal{F}_{\text{UD}}$, $\mathcal{S}$ runs the copy of the respective functionality on input that message.
- When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.deposit.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.deposit.rep}, sid, qid)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.rep}, sid, qid)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{NIC}}$ outputs the message $(\mathsf{com.setup.req}, sid, qid)$, $\mathcal{S}$ runs a copy of $\mathcal{S}_{\text{NIC}}$ on input that message. When $\mathcal{S}_{\text{NIC}}$ replies with $(\mathsf{com.setup.alg}, sid, qid, m)$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NIC}}$ on input that message.
- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.init.end}, sid, ep, N)$, $\mathcal{S}$ sets $sid'_{\text{OT}} \leftarrow (sid, ep)$, and retrieves $(\mathsf{ot.init.ini}, sid_{\text{OT}}, \langle m_n \rangle_{n=1}^N)$ sent by $\mathcal{A}$ to $\mathcal{F}_{\text{OT}}$ such that $sid'_{\text{OT}} = sid_{\text{OT}}$, and sends the message $(\mathsf{pot.init.ini}, sid, ep, \langle m_n \rangle_{n=1}^N)$ to $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.init.sim}, sid, ep, N)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.init.rep}, sid, ep)$ to $\mathcal{F}_{\text{POTS}}$.

- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.setupprices.end}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$, $\mathcal{S}$ sends the message $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$ to the functionality $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.setupprices.sim}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.setupprices.rep}, sid)$ to $\mathcal{F}_{\text{POTS}}$.
- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.updateprice.end}, sid, n, p)$, $\mathcal{S}$ sends the message $(\mathsf{pot.updateprice.ini}, sid, n, p)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ sends $(\mathsf{pot.updateprice.sim}, sid, qid, n, p)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.updateprice.rep}, sid, qid)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message $(\mathsf{pot.deposit.end}, sid, dep)$, $\mathcal{S}$ sends the message $(\mathsf{pot.deposit.ini}, sid, dep)$ to protocol Π_{POTS} for the buyer.
- $\mathcal{S}$ selects σ such that σ is associated with the message with the lowest price p_σ and sends the message $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$ to Π_{POTS} for the buyer.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message $(\mathsf{pot.revealstatistic.end}, sid, v, \mathsf{ST})$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$ to protocol Π_{POTS} for the buyer.

Theorem 11.5.3 *When the seller $\mathcal{V}$ is corrupt, the construction Π_{POTS} described in Section 11.5 securely realizes $\mathcal{F}_{\text{POTS}}$ in the $(\mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{NIC}}||\mathcal{S}_{\text{NIC}}, \mathcal{F}_{\text{ZK}}^{R}, \mathcal{F}_{\text{OT}}, \mathcal{F}_{\text{CD}}, \mathcal{F}_{\text{UD}})$-hybrid model.*

PROOF OF THEOREM 11.5.3. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\Pi_{\text{POTS}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{POTS}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

GAME 1: This game proceeds as **Game** 0, except that in **Game** 1, the simulator simulates the buyer's side of the protocol by selecting σ associated with the message with the lowest price. This does not alter the view of the environment. Therefore, $[\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0] = 0]$.

Our simulator $\mathcal{S}$ is indistinguishable from the real world protocol because it runs as the real-world protocol, except when it outputs failure. The probability that $\mathcal{S}$ outputs failure is negligible thanks to the security properties ensured by the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, and $\mathcal{F}_{\text{UD}}$. Concretely,

$\mathcal{F}_{\text{NIC}}$ ensures that commitments are binding, and $\mathcal{A}$ does not learn any information on σ, thanks to $\mathcal{F}_{\text{OT}}$. The obliviousness property provided by $\mathcal{F}_{\text{CD}}$ also ensures that $\mathcal{A}$ does not learn any information on the counter values $count_n$ and deposit value dep stored in Tbl_{cd}.

The distribution of **Game** 1 is identical to that of our simulation. This concludes the proof of theorem 11.5.3.

INSTANTIATION AND EFFICIENCY ANALYSIS. In previous work [3, 10, 18, 82, 85, 86], the computation and communication cost of POT protocols is dominated by the cost of the underlying OT scheme. This is also the case for Π_{POTS}. However, Π_{POTS} has the advantage that it can be instantiated with any OT protocol that realizes $\mathcal{F}_{\text{OT}}$. The OT schemes used to construct UC-secure POT schemes [85], and other UC-secure OT schemes are suitable. Moreover, when new and more efficient OT schemes are available, they can also be used to instantiate Π_{POTS}.

We also note that the overhead introduced by $\mathcal{F}_{\text{NIC}}$ to allow for the modular design of Π_{POTS} is small. $\mathcal{F}_{\text{NIC}}$ can be instantiated with a perfectly hiding commitment scheme, such as Pedersen commitments [20]. Therefore, the overhead consists of computing a commitment to each of the values that need to be sent to more than one functionality, and ZK proofs of the openings of these commitments.

As discussed above, to construct POT from an OT scheme, $\mathcal{V}$ must set the prices, and $\mathcal{B}$ must make deposits and pay for the messages obtained. Additionally, our POT protocol allows $\mathcal{V}$ to receive aggregate statistics on purchases. For these tasks, Π_{POTS} uses $\mathcal{F}_{\text{UD}}$ and $\mathcal{F}_{\text{CD}}$. These functionalities can be instantiated with a non-hiding and hiding VC scheme respectively [21, 38], equipped with ZK proofs of an opening for a position of the vector. In [21, 38], concrete instantiations based on the Diffie-Hellman Exponent (DHE) assumption are provided. These instantiations involve a common reference string that grows linearly with the length of the committed vector, which in Π_{POTS} is the number N of messages. A non-hiding VC and a hiding VC commit to the tables DB and Tbl_{cd} respectively. The vector commitments, as well as openings for each position of the vector, are of size independent of N. The computation of a commitment and of an opening grows linearly with N. However, when the committed vector changes, both the vector commitment and the openings can be updated with cost independent of N. Therefore, they can be updated and reused throughout the protocol, yielding amortized cost independent of N. The ZK proofs of VC openings offer computation and communication cost independent of N. Therefore, with this instantiation, Π_{POTS} remains efficient when the number N of messages is large.

We compare below Π_{POTS} to the UC-secure scheme in [85], but we note that this comparison would be similar for other full-simulation secure POT protocols. We can conclude that Π_{POTS} provides additional functionalities such as dynamic pricing and aggregated statistics with cost similar to POT protocols that do not provide them.

PRICES. In [85], in the initialization phase, $\mathcal{V}$ encrypts the messages and for each message, $\mathcal{V}$ computes a signature that binds a ciphertext to the price of the encrypted message. This implies that one signature per message is sent from $\mathcal{V}$ to $\mathcal{B}$, and thus the cost grows linearly with N. In Π_{POTS}, $\mathcal{F}_{\text{UD}}$ is used, which can be instantiated with a non-hiding VC scheme. In this instantiation, only one vector commitment which commits to a vector that contains the

list of prices, needs to be sent from $\mathcal{V}$ to $\mathcal{B}$. Nevertheless, adding the size of the common reference string, the cost also grows linearly with N.

However, non-hiding VC schemes provide dynamic pricing at no extra cost. The vector commitment can be updated with cost independent of N. With a signature scheme, $\mathcal{V}$ could also provide a new signature on the price with cost independent of N. However, $\mathcal{V}$ needs to revoke the signature on the old price. The need for a signature revocation mechanism makes dynamic pricing costly in this case.

DEPOSIT. In [85], in the deposit phase, $\mathcal{B}$ sends a commitment to the new value of the deposit and a ZK proof that the deposit is updated. In Π_{POTS}, $\mathcal{F}_{\text{CD}}$ is used, which can be instantiated with a hiding vector commitment that stores the deposit at position 0. The size of commitments, as well as the cost of a ZK proof of deposit updated, does not depend on N in both cases. However, the common reference string of the VC scheme grows linearly with N. (We recall that $\mathcal{F}_{\text{CD}}$ not only stores the deposit but also the N counters of purchases.) By applying the UC with joint state theorem [31], it could be possible to share the common reference string for the DHE instantiations of the non-hiding and hiding VC schemes, but this affects the modularity of Π_{POTS}.

PAYMENT. In [85], $\mathcal{B}$ proves in ZK that the price of the purchased message is subtracted from her current funds. This involves a ZK proof of signature possession, to prove that the correct price is used, and a ZK proof of commitment opening, to prove that the correct value of the deposit is used. The cost of these proofs is independent of N. In Π_{POTS}, when using non-hiding and hiding VC schemes to instantiate $\mathcal{F}_{\text{UD}}$ and $\mathcal{F}_{\text{CD}}$, we need two ZK proofs of a vector commitment opening: one for the non-hiding VC scheme (for the price) and one for the hiding VC scheme (for the deposit). The amortized cost of these ZK proofs is also independent of N. The cost of a ZK proof of commitment opening for the DHE instantiation is similar to the ZK proof of signature possession in [85].

STATISTICS. Unlike [85], Π_{POTS} allows $\mathcal{V}$ to get aggregate statistics about the purchases of $\mathcal{B}$. $\mathcal{F}_{\text{CD}}$ is used to store the counters of the number of purchases for each category. With the instantiation based on a hiding VC scheme, updating the counters and reading them to compute a statistic involves ZK proofs of the opening of positions of a vector commitment, whose amortized computation and communication cost is independent of N.

AGGREGATE STATISTICS ON MULTIPLE BUYERS. Π_{POTS} allows $\mathcal{V}$ to gather statistics about the purchases of each buyer separately. Nonetheless, $\mathcal{V}$ is possibly more interested in gathering aggregate statistics about multiple buyers. This is also appealing to better protect buyer's privacy. Fortunately, Π_{POTS} enables this possibility. The functionalities $\mathcal{F}_{\text{CD}}$ used in the execution of Π_{POTS} between $\mathcal{V}$ and each of the buyers can be used to run a secure multiparty computation (MPC) protocol for the required statistic. In this protocol, each buyer reads from $\mathcal{F}_{\text{CD}}$ the counters needed for the statistic. We note that $\mathcal{F}_{\text{CD}}$ provides commitments to the counters read. These commitments can easily be plugged into existing commit-and-prove MPC protocols [30] to run an MPC between the seller and the buyers. We note that previous POT protocols do not provide this possibility because buyers

do not have any means to prove statements on their purchase histories. $\mathcal{F}_{\text{CD}}$ acts as a ZK data structure that stores information about what buyers have proven in zero-knowledge, so that this information can be reused in subsequent ZK proofs.

12 OBLIVIOUS TRANSFER WITH ACCESS CONTROL

Joint work with Alfredo Rial. This protocol was published in a paper presented at the 25th Australasian Conference on Information Security and Privacy in 2020 [39]. This protocol makes use of the functionality $\mathcal{F}_{\text{UUD}}$ from Chapter 9. Reproduced with permission from Springer Nature.

12.1 INTRODUCTION

Oblivious Transfer with Access Control (OTAC) protocols [17, 33] run between a sender $\mathcal{T}$ and receivers $\mathcal{R}_k$. $\mathcal{T}$ receives as input a tuple $(m_i, \mathsf{ACP}_i)_{\forall i \in [1,N]}$ of messages and their associated access control policies. In a transfer phase, a receiver $\mathcal{R}_k$ chooses an index $i \in [1, N]$ and obtains the message m_i if $\mathcal{R}_k$ satisfies the policy ACP_i. $\mathcal{T}$ does not learn i, whereas $\mathcal{R}_k$ does not learn any information about other messages.

In the following, we only consider OTAC schemes in which the receivers learn all policies $(\mathsf{ACP}_i)_{\forall i \in [1,N]}$, schemes that are stateless, i.e. fulfilment of a policy by $\mathcal{R}_k$ does not depend on the history of messages received by $\mathcal{R}_k$, and schemes that are adaptive, i.e. there are several transfers and $\mathcal{R}_k$ can choose i after receiving messages in previous transfers. In Section 12.2, we discuss stateful and adaptive OTAC schemes and OTAC schemes with hidden policies. Additionally, we focus on OTAC schemes that provide anonymity and unlinkability, i.e., schemes where $\mathcal{T}$ cannot link a transfer to a receiver identity $\mathcal{R}_k$ and where transfers to $\mathcal{R}_k$ are unlinkable with respect to each other.

Existing adaptive and stateless OTAC schemes follow a common pattern in their design. In the initialization phase, $\mathcal{T}$ computes N ciphertexts c_i that encrypt m_i. Some OTAC schemes [1, 17, 66] use a signature that binds ACP_i to c_i, while others [81, 98–100] use fuzzy identity-based encryption (IBE) or ciphertext-policy attribute-based encryption (CP-ABE) to encrypt m_i under ACP_i. The receivers obtain $(c_i, \mathsf{ACP}_i)_{\forall i \in [1,N]}$. To prove fulfilment of policies, $\mathcal{R}_k$ proves to an authority that she possesses some attributes and obtains a credential or a secret key for her attributes. In the transfer phase, $\mathcal{R}_k$ interacts with $\mathcal{T}$ in such a way that $\mathcal{R}_k$ can decrypt c_i for her choice i only if her certified attributes satisfy ACP_i. These OTAC schemes have several shortcomings in their design:

MODULARITY. Though some OTAC schemes are extensions of adaptive Oblivious Transfer (OT) protocols, they do not use OT protocols as building blocks. Instead, the OT scheme is modified ad-hoc to produce the OTAC scheme, blurring the line between elements which were part of the OT scheme and those which were added to provide access control. This lack of modularity results in two disadvantages: first, when the security of the OTAC scheme is analysed, the security of the underlying OT scheme needs to be reanalysed. Second, the OTAC scheme cannot be instantiated with any secure adaptive OT scheme, and consequently, whenever more efficient OT schemes are pro-

posed, the OTAC scheme cannot incorporate them and would need to be redesigned.

POLICY UPDATES. All existing OTAC schemes do not allow for policy updates, i.e., if a policy ACP_i needs to be updated, the initialization phase needs to be rerun. In practical applications of OTAC schemes (e.g. medical or financial databases), it would be desirable to be able to update policies dynamically throughout protocol execution without needing to re-encrypt messages. To enable policy updates, we would need to separate the encryptions c_i of m_i from the method used to encode policies ACP_i. As explained above, OTAC schemes use signatures schemes or CP-ABE to bind policies to ciphertexts. It would be possible to separate, for instance, a signature on the policy ACP_i from the encryption c_i of m_i, whilst still allowing $\mathcal{R}_k$ to prove the association between c_i and ACP_i in the transfer phase. However, a revocation mechanism to revoke outdated signatures would also need to be implemented, which would decrease efficiency.

STORAGE COST. All existing OTAC schemes associate each encryption c_i with a policy ACP_i. However, in practical applications, multiple database records are associated with a single policy. Therefore, if we separate the ciphertexts c_i from the method used to encode policies ACP_i, it would be possible to improve efficiency by associating a policy to multiple ciphertexts.

12.1.1 *Our Contribution*

We use UUD from Chapter 9 to construct a modular OTAC protocol which enables dynamic policy updates without the need for a revocation mechanism, and allows for the association of a policy to multiple messages.

We propose a functionality $\mathcal{F}_{\text{OTAC}}$, which follows previous OTAC functionalities [17] but introduces two main modifications. First, it splits the initialization interface into two interfaces: otac.init, in which the sender $\mathcal{T}$ receives $(m_i)_{\forall i\in[1,N]}$, and otac.policy, in which $\mathcal{T}$ receives $(\mathsf{ACP}_i)_{\forall i\in[1,N]}$. This enables $\mathcal{T}$ to make policy updates via otac.policy during protocol execution. Second, previous functionalities include an issuance phase where an issuer certifies $\mathcal{R}_k$ attributes. Instead, $\mathcal{F}_{\text{OTAC}}$ allows for more open and flexible ways of proving possession of attributes. $\mathcal{T}$ sets and updates a relation R_{ACP} that specifies what $\mathcal{R}_k$ must prove to obtain access to messages. Each policy ACP_i is an instance ins for R_{ACP} and, in the transfer phase, $\mathcal{R}_k$ must provide a witness wit such that $(wit, ins) \in R_{\mathsf{ACP}}$. wit could contain, e.g., signatures from an issuer on $\mathcal{R}_k$ attributes, but in general any data required by R_{ACP}.

We also describe a modular construction Π_{OTAC}. Π_{OTAC} uses as building block $\mathcal{F}_{\text{OT}}$, and thus Π_{OTAC} can be instantiated by any secure adaptive OT protocol. To implement access control, Π_{OTAC} uses $\mathcal{F}_{\text{UUD}}$ and $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$. $\mathcal{T}$ stores $(\mathsf{ACP}_i)_{\forall i\in[1,N]}$ in DB in $\mathcal{F}_{\text{UUD}}$. Each entry $[i, v_{i,1}, \ldots, v_{i,L}]$ stores the index i and the representation $\mathsf{ACP}_i = (v_{i,1}, \ldots, v_{i,L})$ of a policy. In a transfer phase, $\mathcal{R}_k$ uses $\mathcal{F}_{\text{UUD}}$ to read ACP_i for her choice i and then $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$ to prove fulfilment of ACP_i. One challenge when defining a hybrid protocol is to ensure that two functionalities receive the same input. For example, in the transfer interface of Π_{OTAC}, we need to ensure that the choice i sent to $\mathcal{F}_{\text{OT}}$ (to obtain m_i) and to

$\mathcal{F}_{\text{UUD}}$ (to read ACP_i) are equal. To this end, we use the method in [20] and described in Section 2.9, in which functionalities receive committed inputs produced by a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments.

Our modular design has the following advantages. First, it simplifies security analysis because security proofs in the hybrid model are simpler, and by splitting the protocol into smaller building blocks, security analysis of constructions for those building blocks are also simpler. Second, it allows for the construction of multiple instantiations by replacing each of the functionalities by any protocols that realize them.

CONSTRUCTION Π_{UUD}. In Chapter 9, we propose a construction Π_{UUD} for $\mathcal{F}_{\text{UUD}}$. Π_{UUD} is based on subvector commitments (SVC) [64], which we extend with a UC ZK proof of knowledge of a subvector. An SVC scheme allows us to compute a commitment vc to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[N])$. vc can be opened to a subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $I = \{i_1, \ldots, i_n\} \subseteq [1, N]$. The size of the opening w_I is independent of N and of $|I|$. SVC were recently proposed as an improvement of vector commitments [32, 69], where the size of w_I is independent of N but dependent on $|I|$. We extend the definition of SVC to include algorithms to update commitments and openings when part of the vector is updated.

Π_{UUD} works as follows: $\mathcal{U}$ uses a bulletin board BB to publish the database DB and any $\mathcal{R}_k$ obtains DB from BB. A BB ensures that all readers obtain the same version of DB, which we need to guarantee unlinkability. Both $\mathcal{U}$ and any $\mathcal{R}_k$ map a DB with N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$ to a vector $\mathbf{x}$ of length $N \times L$ such that $\mathbf{x}[(i-1)L+j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$, and they compute a commitment vc to $\mathbf{x}$. To update a database entry, $\mathcal{U}$ updates BB, and $\mathcal{U}$ and any $\mathcal{R}_k$ update vc. Therefore, updates do not need any revocation mechanism. To prove in ZK that an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ is in DB, $\mathcal{R}_k$ computes an opening w_I for $I = \{(i-1)L+1, \ldots, (i-1)L+L\}$ and uses it to compute a ZK proof of knowledge of the subvector $(\mathbf{x}[(i-1)L+1], \ldots, \mathbf{x}[(i-1)L+L])$. This proof guarantees that I is the correct set for index i.

We describe an efficient instantiation of Π_{UUD} in Chapter 9 that uses an SVC scheme based on the Cube Diffie-Hellman assumption [64]. In terms of efficiency, the storage cost grows quadratically with the vector length $N \times L$. However, after initializing vc and the openings w_I to the initial DB, the communication and computation costs of the update and read operations are independent of N. Therefore, our instantiation allows for an OTAC where the database of policies can be updated and read efficiently.

We describe a variant of our instantiation where each database entry is $[i_{min}, i_{max}, v_{i,1}, \ldots, v_{i,L}]$, where $[i_{min}, i_{max}] \in [1, N]$ is a range of indices. This allows for an OTAC with reduced storage cost. If the messages $(m_{i_{min}}, \ldots, m_{i_{max}})$ are associated with a single policy ACP, only one database entry is needed to store ACP. In contrast, previous OTAC that use signatures or CP-ABE need to embed a policy in every ciphertext.

Π_{UUD} can be regarded as an efficient way of implementing a ZK proof for a disjunction of statements. Namely, proving that an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ is in DB is equivalent to computing an OR proof where the prover proves that she knows at least one of the entries.

12.2 RELATED WORK

Our OTAC is adaptive, i.e., $\mathcal{R}_k$ can choose an index i after having previously received other messages. In [12], an oblivious language-based envelope protocol (OLBE) is proposed based on smooth projective hash functions. OLBE can be viewed as a non-adaptive OTAC.

Our OTAC is stateless, i.e. fulfilment of a policy by $\mathcal{R}_k$ does not depend on the history of messages accessed by $\mathcal{R}_k$. In [33], a stateful OTAC is proposed where policies are defined by a directed graph that determines the possible states of $\mathcal{R}_k$, the transitions between states and the messages that can be accessed at each stage. Priced Oblivious Transfer (POT) protocols [3, 18, 85] require the user to pay a price for each message. Typically, they involve a prepaid method, where $\mathcal{R}_k$ makes a deposit and later subtracts the prices paid from it without revealing the current funds or the prices paid. Recently, a modular POT protocol was proposed [35] based on an updatable database without unlinkability [38]. Our OTAC differs from it in that it provides unlinkability to $\mathcal{R}_k$ and in that it considers more complex policies expressed by tuples of values, while in POT the policy is simply the message price. Additionally, our OTAC can improve storage efficiency when the same policy is applied to several messages.

Our OTAC reveals the policies to $\mathcal{R}_k$. In [16, 19], OTAC with hidden policies are proposed. Our approach based on SVC cannot be followed to modularly design OTAC with hidden policies that allow for policy updates.

12.3 IDEAL FUNCTIONALITY

$\mathcal{F}_{\text{OTAC}}$ interacts with a sender $\mathcal{U}$ and receivers $\mathcal{R}_k$. $\mathcal{F}_{\text{OTAC}}$ consists of the following interfaces:

1. $\mathcal{U}$ uses the otac.init interface to send the messages $\langle m_n \rangle_{n=1}^{N}$.
2. The receiver $\mathcal{R}_k$ uses the otac.retrieve interface to retrieve N.
3. $\mathcal{U}$ uses the otac.policy interface to send (or update) the policies $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and the relation R_{ACP} to $\mathcal{F}_{\text{OTAC}}$.
4. $\mathcal{R}_k$ uses the otac.getpol interface to obtain $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and R_{ACP}.
5. $\mathcal{R}_k$ uses the otac.transfer to send a choice i and a witness wit to $\mathcal{F}_{\text{OTAC}}$. If $(wit, \mathsf{ACP}_i) \in R_{\mathsf{ACP}}$, $\mathcal{F}_{\text{OTAC}}$ sends m_i to $\mathcal{R}_k$.

The relation R_{ACP} is

$$R_{\mathsf{ACP}'} = \{(wit, ins) : \\ 1 = f(wit, \langle v_{i,j} \rangle_{\forall j \in [1,L]})\}$$

The instance otac.policy $= \langle v_{i,j} \rangle_{\forall j \in [1,L]}$ of R_{ACP} is a policy represented as a tuple of values. The function f evaluates whether the witness wit satisfies the policy. wit can contain different elements depending on f. It can, for instance, contain signatures of the attributes of $\mathcal{R}_k$ certified by an issuer.

Figure 12.1: Functionality $\mathcal{F}_{\text{OTAC}}$

$\mathcal{F}_{\text{OTAC}}$ is parameterized by universes of messages $\mathbb{U}_m$ and policies $\mathbb{U}_{pol}$.

1. On input $(\mathsf{otac.init.ini}, sid, \langle m_n \rangle_{n=1}^{N})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{U}, sid')$, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is already stored.
   - Abort if for $n = 1$ to N, $m_n \notin \mathbb{U}_m$.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$.
   - Send $(\mathsf{otac.init.sim}, sid, N)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.init.rep}, sid)$ from $\mathcal{S}$:
   - Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$.
   - Send $(\mathsf{otac.init.end}, sid)$ to $\mathcal{U}$.

2. On input $(\mathsf{otac.retrieve.ini}, sid)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.retrieve.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.retrieve.rep}, sid, qid)$ from $\mathcal{S}$.
   - Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
   - If $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored, set $N \leftarrow \perp$.
   - Else, store $(sid, \mathcal{R}_k, N)$.
   - Delete $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.retrieve.end}, sid, N)$ to $\mathcal{R}_k$.

3. On input $(\mathsf{otac.policy.ini}, sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ from $\mathcal{U}$:
   - Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored or if the number of policies is not equal to number of messages received.
   - For all $n \in [1, N]$, abort if $\mathsf{ACP}_n \notin \mathbb{U}_{pol}$.
   - If $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$ is not stored:
     - For all $n \in [1, N]$, abort if $\mathsf{ACP}_n = \perp$.
     - Abort if $R_{\mathsf{ACP}} = \perp$.
     - Store $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
   - Else:
     - For all $n \in [1, N]$, if $\mathsf{ACP}_n \neq \perp$, update ACP_n in the stored tuple $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
     - If $R_{\mathsf{ACP}} \neq \perp$, update R_{ACP} in $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.

  - Increment cu and update cu in $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
- Create a fresh qid and store qid.
- Send $(\mathsf{otac.policy.sim}, sid, qid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.policy.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if qid is not stored.
- Delete qid.
- Send $(\mathsf{otac.policy.end}, sid)$ to $\mathcal{U}$.

4. On input $(\mathsf{otac.getpol.ini}, sid)$ from $\mathcal{R}_k$:

- Create a fresh qid and store $(qid, \mathcal{R}_k)$.
- Send $(\mathsf{otac.getpol.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.getpol.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
- If $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$ is not stored, set $\langle \mathsf{ACP}_n \rangle_{n=1}^{N} \leftarrow \perp$ and $R_{\mathsf{ACP}} = \perp$.
- Else, set $cr_k \leftarrow cu$, store $(\mathcal{R}_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr_k)$ and delete any previous tuple $(\mathcal{R}_k, \langle \mathsf{ACP}'_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr'_k)$.
- Delete $(qid, \mathcal{R}_k)$.
- Send $(\mathsf{otac.getpol.end}, sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ to $\mathcal{R}_k$.

5. On input $(\mathsf{otac.transfer.ini}, sid, i, wit)$ from $\mathcal{R}_k$:

- Abort if $(sid, \mathcal{R}'_k, N)$ or $(\mathcal{R}'_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr_k)$ such that $\mathcal{R}'_k = \mathcal{R}_k$ are not stored.
- Abort if $i \notin [1, N]$, or if $(wit, \mathsf{ACP}_i) \notin R_{\mathsf{ACP}}$.
- Create a fresh qid and store (qid, m_i, cr_k), where m_i is stored in the tuple $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$.
- Send $(\mathsf{otac.transfer.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.transfer.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if (qid, m_i, cr_k) is not stored, or if $cr_k \neq cu$, where cu is stored in the tuple $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
- Delete the record (qid, m_i, cr_k).
- Send $(\mathsf{otac.transfer.end}, sid, m_i)$ to $\mathcal{R}_k$.

12.4 CONSTRUCTION

Π_{OTAC} uses an ideal functionality $\mathcal{F}_{\text{OT}}$ from Chapter 2 as building block. $\mathcal{F}_{\text{OT}}$ is used to implement the otac.init and otac.retrieve interfaces, as well as to allow $\mathcal{R}_k$ to obtain messages obliviously in the otac.transfer interface.

To implement access control, Π_{OTAC} uses the functionalities $\mathcal{F}_{\text{UUD}}$ from Chapter 9, and $\mathcal{F}_{\text{BB}}$ and $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$ from Chapter 2. In the otac.policy interface, $\mathcal{U}$ uses $\mathcal{F}_{\text{UUD}}$ to store the policies, and $\mathcal{U}$ uses $\mathcal{F}_{\text{BB}}$ to store the relation R_{ACP}. In the otac.getpol interface, $\mathcal{R}_k$ retrieves the policies and the relation from $\mathcal{F}_{\text{UUD}}$ and $\mathcal{F}_{\text{BB}}$.

In the otac.transfer interface, $\mathcal{R}_k$ reads the policy $\mathsf{ACP}_i = \langle v_{i,j}\rangle_{\forall j\in[1,L]}$ for her choice i by using $\mathcal{F}_{\text{UUD}}$. To do this, $\mathcal{R}_k$ obtains commitments com_i and $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$ to i and to the values $\langle v_{i,j}\rangle_{\forall j\in[1,L]}$ that represent the policy. $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$ are sent as input to $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$ so that $\mathcal{R}_k$ proves fulfilment of the policy. com_i is sent as input to $\mathcal{F}_{\text{OT}}$ to obtain the message m_i.

$R_{\mathsf{ACP}'}$ is a modification of R_{ACP}. In $R_{\mathsf{ACP}'}$, the instance $\langle v_{i,j}\rangle_{\forall j\in[1,L]}$ of R_{ACP} is replaced by $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$, while the witness is extended to contain $wit' \leftarrow (wit, \langle v_{i,j}, open_{i,j}\rangle_{\forall j\in[1,L]})$. I.e., the instance in $R_{\mathsf{ACP}'}$ contains commitments to the policy rather than the policy itself, which allows $\mathcal{R}_k$ to hide which instance is being used from $\mathcal{U}$. The relation $R_{\mathsf{ACP}'}$ is

$$\begin{aligned} R_{\mathsf{ACP}'} = \{&(wit', ins') : \\ &\langle 1 = \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j})\rangle_{\forall j\in[1,L]} \wedge \\ &1 = f(wit, \langle v_{i,j}\rangle_{\forall j\in[1,L]})\} \end{aligned}$$

Figure 12.2: Construction Π_{OTAC}

Π_{OTAC} is parameterized by universes of pseudonyms $\mathbb{U}_p$, messages $\mathbb{U}_m$ and policies $\mathbb{U}_{pol}$.

1. On input (otac.init.ini, sid, $\langle m_n\rangle_{n=1}^N$):
   - $\mathcal{U}$ uses the ot.init interface to send the messages $\langle m_n\rangle_{n=1}^N$ to $\mathcal{F}_{\text{OT}}$.
   - $\mathcal{U}$ outputs (otac.init.end, sid).
2. On input (otac.retrieve.ini, sid):
   - $\mathcal{R}_k$ uses the ot.retrieve interface of $\mathcal{F}_{\text{OT}}$ to retrieve the number of messages N.
   - $\mathcal{R}_k$ outputs (otac.retrieve.end, sid, N).
3. On input (otac.policy.ini, sid, $\langle \mathsf{ACP}_n\rangle_{n=1}^N$, R_{ACP}):
   - $\mathcal{U}$ parses $\langle \mathsf{ACP}_n\rangle_{n=1}^N$ as $(n, v_{n,1}, \ldots, v_{n,L})_{\forall n\in[1,N]}$. $\mathbb{U}_{pol}$ admits policies that can be represented by tuples of values.
   - $\mathcal{U}$ uses the uud.update interface to send $(n, v_{n,1}, \ldots, v_{n,L})_{\forall n\in[1,N]}$ to $\mathcal{F}_{\text{UUD}}$.
   - $\mathcal{U}$ uses the bb.write interface of $\mathcal{F}_{\text{BB}}$ to write R_{ACP} into the bulletin board.

- $\mathcal{U}$ outputs (otac.policy.end, sid).

4. On input (otac.getpol.ini, sid):
   - $\mathcal{R}_k$ uses the uud.getdb interface of $\mathcal{F}_{\text{UUD}}$ to retrieve the policies $\mathsf{DB} = (n, v_{n,1}, \ldots, v_{n,L})_{\forall n \in [1,N]} = \langle \mathsf{ACP}_n \rangle_{n=1}^{N}$.
   - If $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ is not stored, $\mathcal{R}_k$ sets $cr_k \leftarrow 0$.
   - $\mathcal{R}_k$ increments cr_k and uses the bb.getbb interface of $\mathcal{F}_{\text{BB}}$ to receive the description of a relation R_{ACP}. $\mathcal{R}_k$ continues incrementing the counter and reading the bulletin board until the returned message is $\perp$. Then $\mathcal{R}_k$ takes the previous cr_k and the last description of a relation R_{ACP} received from $\mathcal{F}_{\text{BB}}$, stores $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ and deletes any previous tuple $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$.
   - $\mathcal{R}_k$ outputs (otac.getpol.end, sid, $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$, R_{ACP}).

5. On input (otac.transfer.ini, sid, i, wit):
   - In the first execution of this interface, $\mathcal{R}_k$ runs the com.setup interface of $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{R}_k$ picks the policy $\mathsf{ACP}_i = (i, v_{i,1}, \ldots, v_{i,L})$ and R_{ACP} from the stored tuple $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$.
   - $\mathcal{R}_k$ aborts if $(wit, \mathsf{ACP}_i) \notin R_{\mathsf{ACP}}$.
   - $\mathcal{R}_k$ uses the com.commit interface of $\mathcal{F}_{\text{NIC}}$ to obtain commitments and openings $(com_i, open_i, \langle com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}$ to i and $v_{i,1}, \ldots, v_{i,L}$.
   - $\mathcal{R}_k$ picks a random pseudonym $P \leftarrow \mathbb{U}_p$.
   - $\mathcal{R}_k$ uses the uud.read interface to send $(P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}))$ to $\mathcal{F}_{\text{UUD}}$.
   - $\mathcal{U}$ receives $(P, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$ from $\mathcal{F}_{\text{UUD}}$.
   - In the first execution of this interface, $\mathcal{U}$ runs the com.setup interface of $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{U}$ uses the com.validate interface of $\mathcal{F}_{\text{NIC}}$ to validate the commitments $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$.
   - $\mathcal{U}$ uses the nym.reply interface of $\mathcal{F}_{\text{NYM}}$ to send to $\mathcal{R}_k$ a message that acknowledges receipt of the commitments. (Here, we assume that $\mathcal{F}_{\text{UUD}}$ uses the nym.send interface of $\mathcal{F}_{\text{NYM}}$ to send a message from $\mathcal{R}_k$ to $\mathcal{U}$.)
   - $\mathcal{R}_k$ sets the instance $ins' \leftarrow (\langle com_{i,j} \rangle_{\forall j \in [1,L]})$ and the witness $wit' \leftarrow (wit, (\langle v_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}))$.
   - $\mathcal{R}_k$ uses the zk.prove interface to send wit', ins' and P to $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$.

- $\mathcal{U}$ receives ins' and P from $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$. $\mathcal{U}$ aborts if the commitments in ins' or the pseudonym are not equal to the ones received from $\mathcal{F}_{\text{UUD}}$.
- $\mathcal{U}$ uses the nym.reply interface of $\mathcal{F}_{\text{NYM}}$ to send to $\mathcal{R}_k$ a message that acknowledges receipt of the ZK proof. (Here, we assume that $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ uses the nym.send interface of $\mathcal{F}_{\text{NYM}}$ to send a message from $\mathcal{R}_k$ to $\mathcal{U}$.)
- $\mathcal{R}_k$ uses the ot.request interface to send P, i, com_i, and $open_i$ to $\mathcal{F}_{\text{OT}}$.
- $\mathcal{U}$ receives P and com_i from $\mathcal{F}_{\text{OT}}$. $\mathcal{U}$ aborts if com_i or the pseudonym is not equal to the commitment received from $\mathcal{F}_{\text{UUD}}$.
- $\mathcal{U}$ uses the ot.transfer interface to send P and com_σ to $\mathcal{F}_{\text{OT}}$.
- $\mathcal{R}_k$ receives m_i from $\mathcal{F}_{\text{OT}}$.
- $\mathcal{R}_k$ outputs (otac.transfer.end, sid, m_i).

12.5 SECURITY ANALYSIS

Theorem 12.5.1 Π_{OTAC} *securely realizes* $\mathcal{F}_{\text{OTAC}}$ *in the* ($\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{UUD}}$, $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$, $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{NYM}}$)*-hybrid model.*

When $\mathcal{U}$ is corrupt, our simulator $\mathcal{S}$ proceeds by running the receiver part of Π_{OTAC}, with two modifications: first, the choice i is replaced by a random choice. This change does not alter the view of the environment because $\mathcal{F}_{\text{OT}}$ does not leak any information on i to the adversary. Second, because $\mathcal{S}$ does not know wit, $\mathcal{S}$ does not run $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$. Instead, $\mathcal{S}$ creates the message sent by $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ to the adversary. This change does not alter the view of the environment because $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ does not leak any information on wit to the adversary.

When $\mathcal{R}_k$ is corrupt, $\mathcal{S}$ proceeds by running the sender part of Π_{OTAC}, with the following modifications: first, when the honest $\mathcal{U}$ inputs the messages, $\mathcal{S}$ sends N random messages to $\mathcal{F}_{\text{OT}}$. When the adversary sends its choice i, $\mathcal{S}$ obtains m_i from $\mathcal{F}_{\text{OTAC}}$ and replaces the random message stored in $\mathcal{F}_{\text{OT}}$ by m_i before the ot.transfer interface of $\mathcal{F}_{\text{OT}}$ is run.

13

PRIVACY PRESERVING LOYALTY PROGRAMS

Joint work with Alfredo Rial. This protocol was published in a paper presented at the Privacy Enhancing Technologies Symposium in 2021 [41]. This protocol makes use of the functionality $\mathcal{F}_{\text{UUHD}}$ from Chapter 7.

13.1 INTRODUCTION

Loyalty Programs (LPs) are marketing strategies implemented by vendors ($\mathcal{V}$s) to establish lasting relationships with buyers ($\mathcal{B}$s). LPs offer accumulating benefits to $\mathcal{B}$s who purchase certain brands or buy products from certain stores [56]. LPs benefit $\mathcal{V}$s in two ways:

- They improve customer retention.
- By storing $\mathcal{B}$s' Purchase Histories (PHs) and Personally Identifiable Information (PII), and thanks to data mining techniques, LPs allow $\mathcal{V}$s to profile $\mathcal{B}$s. These Buyer Profiles (BPs) are used for market research and targeted advertising.

LPs have raised privacy concerns because BPs can reveal sensitive PII, which $\mathcal{V}$s could sell to third-party advertisers and to data brokers [58]. These privacy concerns have been shown to have an impact on participation in LPs [56].

PREVIOUS WORK. LPs can be classified depending on the type of PII and PHs collected, on whether they involve a single or multiple $\mathcal{V}$s, on the type of rewards given to $\mathcal{B}$s, on whether those rewards are transferable to other $\mathcal{B}$s, and so on. For privacy's sake, the key issue is whether $\mathcal{V}$ requires the collection of PH or not.

Some Privacy Preserving Loyalty Programs (PPLPs) do not allow $\mathcal{V}$s to collect PHs [13, 14, 46, 73]. In [46], blind signatures are used to provide unlinkability between the issuance and the redemption of loyalty points (*lpts*). In [73], $\mathcal{B}$s can anonymously download a batch of loyalty cards and use them in an unlinkable manner. In [13], an updatable anonymous credential scheme is proposed that allows $\mathcal{B}$s to get $\mathcal{V}$ to add *lpts* at each purchase without revealing the total number of *lpts* accumulated. Recently, in [14], the scheme in [13] was improved to add features such as the recoverability of failed spent *lpts* or backward unlinkability. Despite offering privacy protection to $\mathcal{B}$s, these solutions prevent any form of buyer profiling, and thus they limit the benefits that $\mathcal{V}$s obtain from LPs.

Other solutions do allow $\mathcal{V}$ to obtain some information on PHs [11, 75, 95]. In [75, 95], $\mathcal{B}$ obtains an anonymous credential at registration. At each purchase, $\mathcal{B}$ shows her anonymous credential and chooses whether her purchases are linkable or unlinkable to previous purchases. In [11], $\mathcal{B}$, at each purchase, obtains a blind signature. This signature signs purchase data along with a value chosen by $\mathcal{B}$. To obtain *lpts*, $\mathcal{B}$ can link signatures of different purchases when they sign the same $\mathcal{B}$-chosen value. $\mathcal{B}$ then reveals these signatures to $\mathcal{V}$. Therefore, in [11, 75, 95], at

each purchase, $\mathcal{B}$s need to decide whether they want the purchases to be linkable to previous purchases. The PPLPs in [11, 75, 95] have two drawbacks:

PRIVACY OF PHS. Profiling consists of running a classification algorithm over a $\mathcal{B}$'s PH. A PPLP should minimize the disclosure of PHs. However, in [11, 75, 95], when $\mathcal{B}$s choose to link their purchases in order to benefit from the LP, $\mathcal{V}$ receives more information on PHs than is actually required to create a BP. In [75, 95], all PH is revealed, while in [11] a taxonomy of products and product types is defined and $\mathcal{B}$s can reveal the exact product purchased or the product's category.

UNLINKABILITY OF PURCHASES. In [11, 75, 95], $\mathcal{B}$s need to decide whether purchases are linkable or unlinkable at the moment of purchase. It would be desirable to have a scheme where purchases are always unlinkable, and yet at a later stage $\mathcal{B}$s can be profiled based on their PH and benefit from $lpts$.

13.1.1 *Our Contribution*

We propose a PPLP that addresses these drawbacks. First, it protects $\mathcal{B}$'s privacy by avoiding the disclosure of PHs to $\mathcal{V}$. Second, it guarantees that purchases are always unlinkable. PHs and $lpts$ are stored on $\mathcal{B}$'s side and can later be used for profiling or to redeem $lpts$.

Our PPLP involves a vendor $\mathcal{V}$ and multiple buyers $\mathcal{B}_k$ and can be used in e-commerce and physical shops. $\mathcal{B}$s are equipped with an electronic device (e.g., a smartphone). The PPLP consists of the following phases.

REGISTRATION. $\mathcal{B}_k$ signs up for the PPLP.

PURCHASE. $\mathcal{B}_k$'s PH and $lpts$ are stored on $\mathcal{B}_k$'s side. Each purchase is unlinkable to previous ones. Despite unlinkability, at each purchase, $\mathcal{V}$ updates $\mathcal{B}_k$'s PH and $lpts$ without learning any of them.

REDEMPTION. $\mathcal{B}_k$ proves that she has accumulated a number of $lpts$.

PROFILING. $\mathcal{B}_k$ reveals to $\mathcal{V}$ the result of running a profiling algorithm on her PH (or, in general, any computation run on her PH) and proves that this result is correct without disclosing her PH. We describe a variant of this phase where $\mathcal{V}$ does not learn the profiling result either, but is able to use it to, for instance, give $\mathcal{B}_k$ more $lpts$.

In our PPLP, $\mathcal{V}$ (resp. $\mathcal{B}_k$) plays the role of $\mathcal{U}$ (resp. $\mathcal{R}_k$).

To describe our PPLP, we use the updatable unlinkable hiding database (UUHD) functionality described in Chapter 7 A UUHD is a protocol between an updater $\mathcal{U}$ and multiple readers $\mathcal{R}_k$. A UUHD consists of an update phase and a read phase. In an update phase, $\mathcal{U}$ updates the database stored by a reader $\mathcal{R}_k$ identified by a pseudonym P. We consider simple databases DB with entries of the form $[i, vr_i]$, where i is the position and vr_i the value stored at position i. $\mathcal{U}$ does not learn the contents of DB and cannot link it to previous updates. However, $\mathcal{U}$ is ensured that she is updating the last version of DB given to $\mathcal{R}_k$. In the read phase, $\mathcal{R}_k$ commits to some data from the database and proves to $\mathcal{U}$ that it is stored in DB. Later, $\mathcal{R}_k$ can use these commitments to prove in Zero Knowledge (ZK) other statements about the data in DB. For instance, $\mathcal{R}_k$ can prove the correctness of the result of running an algorithm on input this data. One key property of UUHD is that $\mathcal{R}_k$ is able to prove in ZK statements about both vr_i and i.

In Chapter 7, we define security for UUHD in the Universal Composability (UC) framework [29]. We define an ideal functionality $\mathcal{F}_{\text{UUHD}}$ for UUHD that is suitable for modular design, i.e., that can be used as a building block of a protocol. In the UC framework, a protocol can be described modularly in the hybrid model, where parties use ideal functionalities for the building blocks of the protocol. However, many functionalities in literature cannot be used for modular design whenever we must guarantee that two or more functionalities receive the same input. To solve this issue, we use the method in [20], described in Section 2.9, which is based on sending committed inputs to functionalities.

We propose a construction Π_{UUHD} for UUHD in Chapter 7. Π_{UUHD} uses a vector commitment (VC) scheme as its main building block. A VC scheme allows us to compute a vector commitment vc to a vector $\mathbf{x}$ of values, where each value $\mathbf{x}[i]$ is stored at a given position i. Additionally, vc can be opened to $\mathbf{x}[i]$ with communication cost independent of the vector length. We use a VC scheme that is additively homomorphic. Π_{UUHD} also uses a pseudonymous channel that provides unlinkability in communications between any $\mathcal{R}_k$ and $\mathcal{U}$.

A VC vc is used to store the database DB by committing to a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vr_i$. Initially, $\mathcal{R}_k$ obtains a signed vc to an initial database DB from $\mathcal{U}$. To read DB in an unlinkable manner, $\mathcal{R}_k$ rerandomizes vc to vc', reveals vc' to $\mathcal{U}$ and proves in Zero Knowledge (ZK) that vc' is a rerandomized version of a commitment signed by $\mathcal{U}$. Then $\mathcal{R}_k$ can prove statements about the database committed to in vc'. To update DB, $\mathcal{U}$ uses the homomorphic property of the VC scheme to update the database in vc' (without learning its contents) and produce an updated VC vc_u, which $\mathcal{U}$ signs and sends to $\mathcal{R}_k$. Π_{UUHD} implements a double-spending detection mechanism to ensure that in the next read phase, $\mathcal{R}_k$ uses vc_u and not previous commitments received from $\mathcal{U}$.

To construct a PPLP based on UUHD, we store $\mathcal{B}_k$'s PH in the form of a database DB that, for each product (or product category, depending on the detail needed to create BPs), stores the number of purchases or the amount of money spent by $\mathcal{B}_k$ on that product. Therefore, DB is a vector $\mathbf{x}$ where each position i represents a product and $\mathbf{x}[i]$ is the number of purchases or the amount of money spent on that product. An additional position is used to store the accumulated $lpts$. When $\mathcal{B}_k$ purchases some products, to update DB, $\mathcal{V}$ computes another VC to a vector $\mathbf{x}_u$ that stores the information regarding the products purchased and additional $lpts$. Then, by using the homomorphic property, $\mathcal{V}$ multiplies both vector commitments to get a VC to $\mathbf{x} + \mathbf{x}_u$ and sends the new VC to $\mathcal{B}_k$.

Our PPLP guarantees unlinkability between purchases and minimizes the PH disclosure towards $\mathcal{V}$. Thanks to double-spending detection, our PPLP ensures that $\mathcal{B}_k$ uses the last version of the PH at the next update or when reading it for profiling or redeeming $lpts$. Nevertheless, we note that $\mathcal{B}_k$ can refuse to use the PPLP for certain purchases or create more than one profile. This leads generally to the creation of better profiles but, if needed, can be discouraged.

Π_{UUHD} benefits from the efficiency properties of VCs. Particularly, updating the database is very efficient thanks to the homomorphic property. In Chapter 7, we show that the communication cost of read and update operations, as well as the computation cost of update operations, is independent of the database size N. Read operations have amortized computation cost which is also independent of N. Therefore, Π_{UUHD} is suitable for large databases.

13.2 RELATED WORK

VC SCHEMES. VC schemes [32, 69] can be based on different assumptions such as CDH, RSA and DHE. We could use VCs secure under the more standard CDH or RSA assumptions, but VCs based on DHE have efficiency advantages. A mercurial VC scheme based on DHE was proposed in [69], and subsequently DHE VC schemes were used in [57, 63, 67]. In our instantiation of Π_{UUHD}, we extend the VC scheme with signatures to enable ZK proofs of an opening w_i.

Polynomial commitments (PCs) allow us to commit to a polynomial and open the commitment to an evaluation of the polynomial. PCs can be used as VCs by committing to a polynomial that interpolates the vector to be committed. In [59], a construction of PCs from the SDH assumption is proposed. A VC based on the PC scheme from SDH has the disadvantage that the VC cannot be updated. A further generalization of VCs and PCs are functional commitments [68].

ZK PROOFS FOR LARGE DATASETS. In most ZK proofs, the computation and communication costs grow linearly with the size of the witness, which is inadequate for proofs about datasets of large size N. However, some techniques attain costs sublinear in N. Probabilistically checkable proofs [61] achieve verification cost sublinear in N, but the cost for the prover is linear in N. In succinct non-interactive arguments of knowledge [51], verification cost is independent of N, but the cost for the prover is still linear in N. ZK proofs for oblivious RAM programs [76] consist of a setup phase where the prover commits to the dataset, with cost linear in N for the prover and constant for the verifier. After setup, multiple proofs can be computed on the dataset with cost sublinear (proportional to the runtime of an ORAM program) for prover and verifier.

Our construction is somehow similar to [76], i.e. a database is committed to, and ZK proofs are computed. Storage cost is linear in N. However, the verification cost of a ZK proof is constant and independent of N. To compute a ZK proof, only the cost of computing an opening w_i is linear in N, but w_i can be reused and updated with cost independent of N. Therefore, computing a ZK proof has an amortized cost independent of N, which makes our construction practical for large databases.

UPDATABLE ANONYMOUS CREDENTIALS. In Anonymous Credential (AC) [26] schemes, issuers sign user attributes. Users show that they possess a signature from an issuer on their attributes and selectively disclose, or prove in ZK statements, about their attributes. Unlinkability ensures that uses of a credential cannot be linked to each other or to the issuance of the credential. Recently, an updatable AC scheme had been proposed in [13] and applied to LPs. An updatable AC scheme allows users to update their signed attributes blindly, i.e., without the issuer knowing the attribute values. A UUHD can thus also be used to construct an updatable AC scheme. In [13], the cost of proving and verifying statements about attributes grows with the number of attributes signed in a credential. In their LP, only the number of *lpts* is signed, and vendors do not receive any information on buyer profiles. Our construction Π_{UUHD} provides ZK proofs of cost independent of N. This allows us to design a PPLP where, in addition to *lpts*, $\mathcal{V}$ signs and updates the whole purchase history of buyers, which could consist of hundreds or thousands of products, with a computation and computation cost comparable to [13]. Buyers

can then be profiled based on the result of a profiling algorithm, without disclosing further purchase data to $\mathcal{V}$.

13.3 IDEAL FUNCTIONALITY

$\mathcal{F}_{\text{LP}}$ interacts with a vendor $\mathcal{V}$ and multiple buyers $\mathcal{B}_k$. $\mathcal{F}_{\text{LP}}$ maintains one database DB per buyer. Each DB is of size $N = M+1$ and stores the PH for M products and the *lpts* accumulated by $\mathcal{B}_k$. $\mathcal{F}_{\text{LP}}$ consists of the interfaces lp.register, lp.purchase, lp.redeem, lp.profile and lp.updatedb.

1. A buyer $\mathcal{B}_k$ sends the lp.register.ini message on input a pseudonym P. $\mathcal{F}_{\text{LP}}$ checks if P is unique and if $\mathcal{B}_k$ did not send a registration request in the past. In this case, after being prompted by the simulator $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P to $\mathcal{V}$.

2. A buyer $\mathcal{B}_k$ sends the lp.purchase.ini message on input a pseudonym P. $\mathcal{F}_{\text{LP}}$ checks if $\mathcal{B}_k$ is already registered, if P is unique and if $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that is still unfinished. In this case, after being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P to $\mathcal{V}$.

3. A buyer $\mathcal{B}_k$ sends the lp.redeem.ini message on input a pseudonym P and a number of loyalty points p. $\mathcal{F}_{\text{LP}}$ checks if $\mathcal{B}_k$ is already registered, if P is unique, if $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that is still unfinished, and if $\mathcal{B}_k$ has accumulated at least p loyalty points. In this case, after being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and p to $\mathcal{V}$.

4. A buyer $\mathcal{B}_k$ sends the lp.profile.ini message on input a pseudonym P and a profiling function f. $\mathcal{F}_{\text{LP}}$ checks if $\mathcal{B}_k$ is already registered, if P is unique, and if $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that is still unfinished. In this case, $\mathcal{F}_{\text{LP}}$ evaluates f on input the purchase history of $\mathcal{B}_k$. After being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and the profiling result res to $\mathcal{V}$.

5. The vendor $\mathcal{V}$ sends the lp.updatedb.ini message on input a pseudonym P and a database $(i, vu_i)_{i\in[1,N]}$. $\mathcal{F}_{\text{LP}}$ checks if there is a request pending for pseudonym P, which is associated with a buyer $\mathcal{B}_k$. For a registration request, $\mathcal{F}_{\text{LP}}$ initializes the database DB of $\mathcal{B}_k$ to contain 0 at every position. For a purchase request, $\mathcal{F}_{\text{LP}}$ uses $(i, vu_i)_{i\in[1,N]}$ to update the PH and the loyalty points stored in DB. For a redeem request, $\mathcal{F}_{\text{LP}}$ updates DB by subtracting p loyalty points redeemed by $\mathcal{B}_k$. For a profile request, $\mathcal{F}_{\text{LP}}$ does not update the database. After being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and $(i, vu_i)_{i\in[1,N]}$ to $\mathcal{B}_k$.

$\mathcal{F}_{\text{LP}}$ guarantees that the requests made by buyers are *unlinkable* for $\mathcal{V}$. As can be seen, $\mathcal{F}_{\text{LP}}$ reveals to $\mathcal{V}$ a pseudonym that is unique for every request. $\mathcal{F}_{\text{LP}}$ also ensures that the PH and *lpts* of a buyer are *hidden* from $\mathcal{V}$. $\mathcal{F}_{\text{LP}}$ never reveals a buyer's database to $\mathcal{V}$ and, thanks to unlinkability, $\mathcal{V}$ is not able to link the updates of a database. $\mathcal{F}_{\text{LP}}$ also ensures *unforgeabilty* of databases, i.e. buyers are not able to modify their databases. This guarantees that *lpts* can only be redeemed if they were accumulated, and that the profiling result is correct.

Figure 13.1: Ideal Functionality $\mathcal{F}_{\text{LP}}$

$\mathcal{F}_{\text{LP}}$ is parameterized by a database size N, a universe of pseudonyms $\mathbb{U}_p$, a function family $\mathcal{F}$ and a universe of values $\mathbb{U}_v$.

1. On input $(\mathsf{lp.register.ini}, sid, P)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.register.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.register.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.register})$, delete $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.register.end}, sid, P)$ to $\mathcal{V}$.

2. On input $(\mathsf{lp.purchase.ini}, sid, P)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.purchase.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.purchase.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.purchase})$, delete $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.purchase.end}, sid, P)$ to $\mathcal{V}$.

3. On input $(\mathsf{lp.redeem.ini}, sid, P, p)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$. Take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$ and take the entry $[N, v_N] \in \mathsf{DB}$. Abort if $p \notin [0, v_N]$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k, p)$ and send $(\mathsf{lp.redeem.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.redeem.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k, p)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.redeem}, p)$, delete $(qid, P, \mathcal{B}_k, p)$ and send $(\mathsf{lp.redeem.end}, sid, P, p)$ to $\mathcal{V}$.

4. On input $(\mathsf{lp.profile.ini}, sid, P, f)$ from $\mathcal{B}_k$:

- Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$, of if $f \notin \mathcal{F}$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
- Take $(sid, \mathcal{B}_k, \mathsf{DB})$ and compute $res \leftarrow f((i, v)_{i \in \mathbb{S}})$, where $(i, v)_{i \in \mathbb{S}} \subseteq \mathsf{DB}$ and $\mathbb{S}$ is defined in f.
- Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k, res)$ and send $(\mathsf{lp.profile.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.profile.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(qid', P, \mathcal{B}_k, res)$ such that $qid = qid'$ is not stored.
- Store $(sid, P, \mathcal{B}_k, \mathsf{lp.profile})$, delete $(qid, P, \mathcal{B}_k, res)$ and send $(\mathsf{lp.profile.end}, sid, P, res)$ to $\mathcal{V}$.

5. On input $(\mathsf{lp.updatedb.ini}, sid, P, (i, vu_i)_{i \in [1,N]})$ from $\mathcal{V}$:

- Abort if $(sid, P', \mathcal{B}_k, \mathsf{lp.x}, \ldots)$ such that $P' = P$ is not stored, or if, for $i \in [1, N]$, $vu_i \notin \mathbb{U}_v$.
- If $\mathsf{lp.x} = \mathsf{lp.register}$, set $\mathsf{DB} \leftarrow (i, 0)_{i \in [1,N]}$ and store a tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
- If $\mathsf{lp.x} = \mathsf{lp.purchase}$, take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i \in [1,N]}$ and update $\mathsf{DB} \leftarrow (i, vr_i + vu_i)_{i \in [1,N]}$ in the tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
- If $\mathsf{lp.x} = \mathsf{lp.redeem}$, take p from the tuple $(sid, P', \mathcal{B}_k, \mathsf{lp.purchase}, p)$. Take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i \in [1,N]}$ and update the database entry $[N, vr_i + p]$ in the tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
- Delete the tuple $(sid, P', \mathcal{B}_k, \mathsf{lp.x}, \ldots)$. If $\mathsf{lp.x} \neq \mathsf{lp.purchase}$, set $(i, vu_i)_{i \in [1,N]} \leftarrow \perp$.
- Create fresh qid, store $(qid, \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ and send $(\mathsf{lp.updatedb.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.updatedb.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if a tuple $(qid', \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ such that $qid = qid'$ is not stored.
- In the tuple $(sid, P', \mathcal{B}_k, 0)$ such that $P' = P$, if $\mathsf{lp.x} = \mathsf{lp.register}$, replace 0 by 1, else replace o by $\perp$.
- Delete $(qid', \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ and send $(\mathsf{lp.updatedb.end}, sid, P, (i, vu_i)_{i \in [1,N]})$ to $\mathcal{B}_k$.

13.4 CONSTRUCTION

Π_{LP} is described modularly in the hybrid model, where $\mathcal{V}$ and $\mathcal{B}_k$ use the functionalities $\mathcal{F}_{\mathrm{UUHD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$. $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$ are ZK func-

tionalities used in the redemption and profiling phases respectively. This modular design allows for multiple constructions of Π_{LP}

$\mathcal{F}_{\text{UUHD}}$ is used to store the buyers' databases DB. During registration, $\mathcal{B}_k$ invokes the uuhd.read interface for the first time, and $\mathcal{V}$ invokes the uuhd.update interface on input $(i, 0)_{i\in[1,N]}$ to initialize DB. In the purchase phase, $\mathcal{B}_k$ invokes the uuhd.read interface without reading the database, and $\mathcal{V}$ invokes the uuhd.update interface on input $(i, vu_i)_{i\in[1,N]}$ to update the PH and $lpts$ stored in DB.

To redeem p $lpts$, $\mathcal{B}_k$ invokes the uuhd.read interface to read the database entry $[N, v_N]$, which stores the accumulated $lpts$. $\mathcal{B}_k$ uses $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ to prove that $p \in [0, v_N]$. Both $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ and $\mathcal{F}_{\text{UUHD}}$ receive as input commitments output by $\mathcal{F}_{\text{NIC}}$ to ensure that the value v_N read is used in the ZK proof. $\mathcal{V}$ invokes the uuhd.update interface on input $(i, 0)_{i\in[1,M]}||[N, -p]$ to subtract the redeemed loyalty points from DB.

In the profiling phase, $\mathcal{B}_k$ invokes the uuhd.read interface to read the subset $\mathbb{S}$ of database entries required by the function f. $\mathcal{B}_k$ uses $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ to prove correctness of the evaluation of f. Commitments produced by $\mathcal{F}_{\text{NIC}}$ are again used to guarantee that the entries read are used in $\mathcal{F}_{\text{ZK}}^{R_{pr}}$. $\mathcal{V}$ invokes the uuhd.update interface without updating the database to conclude this phase.

Figure 13.2: Construction Π_{LP}

Π_{LP} is parameterized by a database size N, a universe of pseudonyms $\mathbb{U}_p$, a function family $\mathcal{F}$ and a universe of values $\mathbb{U}_v$.

1. On input (lp.register.ini, sid, P), $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ sends (uuhd.read.ini, sid, P, $\perp$) to $\mathcal{F}_{\text{UUHD}}$.
   - $\mathcal{V}$ receives (uuhd.read.end, sid, P, flag, $\perp$) from $\mathcal{F}_{\text{UUHD}}$. If flag $=$ 1, $\mathcal{V}$ stores (sid, P, lp.register) and outputs (lp.register.end, sid, P).
2. On input (lp.purchase.ini, sid, P), $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ sends (uuhd.read.ini, sid, P, $\perp$) to $\mathcal{F}_{\text{UUHD}}$.
   - $\mathcal{V}$ receives (uuhd.read.end, sid, P, flag, $\perp$) from $\mathcal{F}_{\text{UUHD}}$. If flag $= 0$ and no commitments to read data are received, $\mathcal{V}$ stores (sid, P, lp.purchase) and outputs (lp.purchase.end, sid, P).
3. On input (lp.redeem.ini, sid, P, p), $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ takes the stored $(i, v_i)_{i\in[1,N]}$ and checks that $p \in [0, v_N]$.
   - $\mathcal{B}_k$ uses the com.commit interface of $\mathcal{F}_{\text{NIC}}$ on input N and v_N to get two commitments and openings (com_N, $open_N$, com_v, $open_v$) to N and v_N.

R_{rd} is

$$R_{rd} = \{(wit_{rd}, ins_{rd}) : 1 = \mathsf{COM.Verify}(parcom, com_v, v_N, open_v) \wedge p \in [0, v_N]\}$$

$\mathcal{B}_k$ proves that v_N is committed to in com_v and that $p \leq v_N$. For the latter, $\mathcal{B}_k$ uses a range proof [15].

- $\mathcal{B}_k$ sets a witness $wit_{rd} \leftarrow (v_N, open_v)$ and $ins_{rd} \leftarrow (p, com_v, com_N, N, open_N)$. $\mathcal{B}_k$ uses the zk.prove interface to send wit_{rd}, ins_{rd} and P to $\mathcal{F}_{\text{ZK}}^{R_{rd}}$.
- $\mathcal{V}$ uses the com.verify interface of $\mathcal{F}_{\text{NIC}}$ on input $(com_N, N, open_N)$ to verify that com_N commits to the position N. $\mathcal{V}$ uses the com.validate interface on input com_v to check that com_v is a commitment computed by $\mathcal{F}_{\text{NIC}}$. $\mathcal{V}$ stores the commitments (com_N, com_v).
- $\mathcal{V}$ sends a message (Read DB) to $\mathcal{B}_k$ by using the nym.reply interface of $\mathcal{F}_{\text{NYM}}$ on input P. (Here we consider that any construction for $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ uses $\mathcal{F}_{\text{NYM}}$, so $\mathcal{V}$ can reply through $\mathcal{F}_{\text{NYM}}$.)
- $\mathcal{B}_k$ stores (sid, P, p) and sends the message (uuhd.read.ini, $sid, P, (N, v_N, com_N, open_N, com_v, open_v)$) to $\mathcal{F}_{\text{UUHD}}$.
- $\mathcal{V}$ receives (uuhd.read.end, sid, P, flag, (com_N, com_v)) from $\mathcal{F}_{\text{UUHD}}$. $\mathcal{V}$ checks that the commitments (com_N, com_v) equal the ones received from $\mathcal{F}_{\text{ZK}}^{R_{rd}}$. $\mathcal{V}$ stores $(sid, P,$ lp.redeem, $p)$ and outputs (lp.redeem.end, sid, P, p).

4. On input (lp.profile.ini, sid, P, f), $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ takes her purchase history $(i, v_i)_{i\in[1,M]}$. Let $\mathbb{S} \subseteq [1, M]$ contain the indices needed to evaluate f. $\mathcal{B}_k$ commits to $(i, v_i)_{i\in\mathbb{S}}$, i.e., for all $i \in \mathbb{S}$, $\mathcal{B}_k$ uses com.commit on input i and v to get (c_i, o_i, cr_i, or_i).
   - $\mathcal{B}_k$ computes $res \leftarrow f((i, v_i)_{i\in\mathbb{S}})$.

R_{pr} is

$$\begin{aligned} R_{pr} = \{&(wit_{pr}, ins_{pr}) : \\ &\{1 = \mathsf{COM.Verify}(parcom, c_i, i, o_i) \ \wedge \ i \in \mathbb{S} \wedge \\ &1 = \mathsf{COM.Verify}(parcom, cr_i, v_i, or_i)\}_{\forall i\in\mathbb{S}} \ \wedge \ res = f((i, v_i)_{i\in\mathbb{S}})\} \end{aligned}$$

$\mathcal{B}_k$ proves that i and v_i are committed in c_i and cr_i, and that $i \in \mathbb{S}$. $\mathcal{B}_k$ also proves that res is the result of evaluating f on input $(i, v_i)_{i\in\mathbb{S}}$.

- $\mathcal{B}_k$ sets a witness $wit_{pr} \leftarrow (\langle i, o_i, v_i, or_i\rangle_{i\in\mathbb{S}})$ and $ins_{pr} \leftarrow (\langle c_i, cr_i\rangle_{i\in\mathbb{S}})$. $\mathcal{B}_k$ uses the zk.prove interface to send wit_{pr}, ins_{pr} and P to $\mathcal{F}_{\text{ZK}}^{R_{pr}}$.
- $\mathcal{V}$ uses com.validate to check that $(c_i, cr_i)_{i\in\mathbb{S}}$ are commitments computed by $\mathcal{F}_{\text{NIC}}$ and stores $(c_i, cr_i)_{i\in\mathbb{S}}$.

- $\mathcal{V}$ sends a message (Read DB) to $\mathcal{B}_k$ by using the nym.reply interface of $\mathcal{F}_{\text{NYM}}$. (Here we consider that any construction for $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ uses $\mathcal{F}_{\text{NYM}}$, so $\mathcal{V}$ can reply through $\mathcal{F}_{\text{NYM}}$.)
- $\mathcal{B}_k$ sends $\mathcal{F}_{\text{UUHD}}$ the message (uuhd.read.ini, sid, P, $(i, v_i, c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}}$).
- $\mathcal{V}$ receives (uuhd.read.end, sid, P, flag, $(c_i, cr_i)_{i\in\mathbb{S}}$) from $\mathcal{F}_{\text{UUHD}}$. $\mathcal{V}$ checks that the commitments $(c_i, cr_i)_{i\in\mathbb{S}}$ equal the ones received from $\mathcal{F}_{\text{ZK}}^{R_{rd}}$. $\mathcal{V}$ stores (sid, P, lp.profile) and outputs (lp.profile.end, sid, P, res).

5. On input (lp.updatedb.ini, sid, P, $(i, vu_i)_{i\in[1,N]}$), $\mathcal{V}$ and $\mathcal{B}_k$ do the following:
   - $\mathcal{V}$ finds the tuple (sid, P', lp.x, ...), such that $P' = P$.
   - If lp.x = lp.register, $\mathcal{V}$ initializes a database $(i, 0)_{i\in[1,N]}$ and sends (uuhd.update.ini, sid, P, $(i, 0)_{i\in[1,N]}$) to $\mathcal{F}_{\text{UUHD}}$. $\mathcal{B}_k$ receives (uuhd.update.end, sid, $(i, 0)_{i\in[1,N]}$) from $\mathcal{F}_{\text{UUHD}}$, checks that $(i, 0)_{i\in[1,N]}$ contains 0 in all positions, stores $(i, 0)_{i\in[1,N]}$ and outputs (lp.updatedb.end, sid, P, $\perp$).
   - If lp.x = lp.purchase, $\mathcal{V}$ sends (uuhd.update.ini, sid, P, $(i, vu_i)_{i\in[1,N]}$) to $\mathcal{F}_{\text{UUHD}}$. $\mathcal{B}_k$ receives (uuhd.update.end, sid, $(i, vu_i)_{i\in[1,N]}$) from $\mathcal{F}_{\text{UUHD}}$ and checks that $(i, vu_i)_{i\in[1,N]}$ is correct, i.e., the database was updated by using the correct prices paid by $\mathcal{B}_k$ for the products purchased and the correct number of $lpts$ was added. $\mathcal{B}_k$ updates her database $(i, v_i')_{i\in[1,N]}$ by setting $(i, v_i' + vu_i)_{i\in[1,N]}$ and outputs (lp.updatedb.end, sid, P, $(i, vu_i)_{i\in[1,N]}$).
   - If lp.x = lp.redeem, $\mathcal{V}$ takes p from the tuple (sid, P, lp.redeem, p) and sets a database $(i, v_i)_{i\in[1,N]}$ where $v_i = 0$ for $i \in [1, M]$ and $v_N = -p$. $\mathcal{V}$ sends (uuhd.update.ini, sid, P, $(i, v_i)_{i\in[1,N]}$) to $\mathcal{F}_{\text{UUHD}}$. $\mathcal{B}_k$ receives (uuhd.update.end, sid, $(i, v_i)_{i\in[1,N]}$) from $\mathcal{F}_{\text{UUHD}}$ and checks that $v_i = 0$ for $i \in [1, M]$ and $v_N = -p$, where p is stored in the tuple (sid, P, p). $\mathcal{B}_k$ updates her database $(i, v_i')_{i\in[1,N]}$ by setting $(N, v_N' + v_N)$ and outputs (lp.updatedb.end, sid, P, $\perp$).
   - If lp.x = lp.profile, $\mathcal{V}$ initializes a database $(i, 0)_{i\in[1,N]}$ and sends (uuhd.update.ini, sid, P, $(i, 0)_{i\in[1,N]}$) to $\mathcal{F}_{\text{UUHD}}$. $\mathcal{B}_k$ receives (uuhd.update.end, sid, $(i, 0)_{i\in[1,N]}$) from $\mathcal{F}_{\text{UUHD}}$ and checks that $(i, 0)_{i\in[1,N]}$ contains 0 in all positions. $\mathcal{B}_k$ outputs (lp.updatedb.end, sid, P, $\perp$).

Theorem 13.5.1 Π_{LP} *securely realizes* $\mathcal{F}_{\text{LP}}$ *in the* $\mathcal{F}_{\text{UUHD}}$, $\mathcal{F}_{\text{ZK}}^{R_{rd}}$, $\mathcal{F}_{\text{ZK}}^{R_{pr}}$, $\mathcal{F}_{\text{NYM}}$ *and* $\mathcal{F}_{\text{NIC}}$*-hybrid model.*

Π_{LP} provides the unlinkability and hiding properties of $\mathcal{F}_{\text{LP}}$, which protect buyers' privacy. Regarding unlinkability, $\mathcal{F}_{\text{UUHD}}$, $\mathcal{F}_{\text{ZK}}^{R_{rd}}$, $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ and $\mathcal{F}_{\text{NYM}}$ reveal to $\mathcal{V}$ a pseudonym and not the identifier $\mathcal{B}_k$. $\mathcal{B}_k$ chooses a different pseudonym for each phase. Regarding the hiding property, in the purchase phase $\mathcal{V}$ does not learn any information on the PH or on the *lpts* of $\mathcal{B}_k$. In the redemption phase, $\mathcal{V}$ only learns that $\mathcal{B}_k$ accumulated at least p points. In the profiling phase, $\mathcal{V}$ only learns the result of evaluating f on input the PH of $\mathcal{B}_k$. We recall that $\mathcal{F}_{\text{NIC}}$ ensures that commitments are hiding.

Π_{LP} prevents an adversarial $\mathcal{B}_k$ from forging her PH or her *lpts*. $\mathcal{F}_{\text{UUHD}}$ ensures that only $\mathcal{V}$ sets up and modifies databases. $\mathcal{F}_{\text{UUHD}}$ also ensures that $\mathcal{B}_k$ cannot reuse her PH or her *lpts*. Additionally, the binding property ensured by $\mathcal{F}_{\text{NIC}}$ guarantees that commitments sent to $\mathcal{F}_{\text{UUHD}}$ and to the ZK functionalities are opened to the same value. Therefore, $\mathcal{B}_k$ can only prove knowledge of the PH or *lpts* that were stored by $\mathcal{V}$.

13.6 IMPLEMENTATION AND EFFICIENCY MEASUREMENTS

We have implemented Π_{LP} in the Python programming language, using the Charm cryptographic framework [4], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. Although we have used a single core implementation of the protocol for our evaluations, we comment that the protocol may benefit from parallelization in the setup phase. The BN256 curve has been used for the pairing group setup. We use the compiler in [24] to compute UC ZK proofs for R_{rd} and R_{pr}, and the Paillier encryption scheme uses a 2048-bit key in both cases. We also employ the range proof scheme described in [15] in the redemption and profiling phases, which requires the computation and storage of a CRS of about 152 bytes. For the purposes of evaluating the profiling phase, we have used a profiling function f which checks whether the sum of the amounts paid by $\mathcal{B}_k$ for products represented by a specified set of database entries surpasses a specified threshold value. Table 13.1 depicts the average computation times for all phases of the protocol over 10 trial runs, against a database size $N = \{100, 1000\}$, which we consider adequate for shops of average size (particularly when each entry is associated with a product category rather than an individual product).[1] The timings include the database update operation performed by $\mathcal{V}$ for each of the phases. The last three rows of Table 13.1 list the average computation times for the profiling phase when a set $\mathbb{S}$ of 1, 5 and 10 database entries have been passed as input to the profiling function. The execution times for the profiling phase grow with the size of $\mathbb{S}$. However, the computation cost is independent of N. We remark that the purchase and redemption phases do not depend on $\mathbb{S}$, and that the profiling phase does not have real-time requirements. Figure 13.5 represents the relation between the execution time of the profiling phase and the size of $\mathbb{S}$.

1 The number of SKUs of a typical supermarket, which is a kind of shop that frequently uses loyalty programs, is anywhere from 15000 to 60000 (https://www.fmi.org/our-research/supermarket-facts)

PHASE	N=100	N=1000
Registration	0.6404	0.6396
Purchase	0.8776	0.8270
Redemption	2.6733	2.6656
Profiling (1-entry)	2.7701	2.7977
Profiling (5-entries)	14.0836	14.2737
Profiling (10-entries)	28.7699	28.2452

Table 13.1: PPLP phase execution times in seconds.

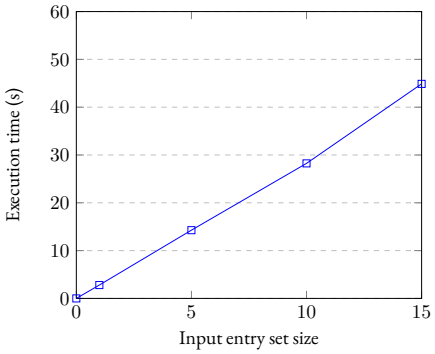


Figure 13.5: Profiling phase input entry size vs. execution time.

14

SEALED BID AUCTIONS WITH BIDDER STATISTICS

Work in progress. This protocol makes use of the functionality $\mathcal{F}_{\text{CD}}$ from Chapter 4.

14.1 INTRODUCTION

Auctions typically involve a seller (or a vendor) and multiple bidders, and can be classified into several categories. Of these, the most prominent classes are those of English auctions, where bidders make increasing bids towards one or more goods that are being sold by a seller; and Dutch auctions, where the bids decrease. These auction types are also commonly referred to as open cry dynamic auctions, as bids are publicly revealed and bidders may submit more than one bid. Other classifications include first price and second price auctions, depending on whether the winner of an auction pays the amount of the highest bid or that of the second highest bid respectively.

The term "Sealed bid auctions", on the other hand, refers to the class of auctions where bidders are allowed to cast only a single sealed bid during a bidding phase, and an auctioneer opens and examines these bids to determine the outcome of an auction; these bids are not revealed to the other bidders. Sealed bid auctions [44] are of particular interest because they require only a single round of computation, and because they make use of "sealed bids", where each bidder submits a signed encrypted bid to an authority, and all bids are revealed to the authority at the same time.

The concept of Vickrey auctions was first proposed in [93], and refers to a variant of a sealed bid auction, where the product that is being auctioned is sold to the bidder with the highest bid value, provided this bidder pays the bid value of the second-highest bid. Some Vickrey auction protocols in literature use threshold trust to provide security guarantees [49, 60]; in other words, the protocol is secure as long as a specific number of auctioneer servers are honest. [70] describes two cryptographic auction schemes which do away with the need to maintain threshold trust between the bidders and the servers responsible for computing the outcome of auctions, whilst also introducing an auction authority. The protocol described in [78] additionally introduces an auction issuer who defines the program or garbled circuit that the auctioneer must execute in order to determine the winner in an auction. For privacy to hold, the auctioneer and the auction issuer must not collude.

However, an impediment that stands in the way of the widespread use of cryptographic auctions is fraudulent bidding. [9] mentions collusion between bidders as one of the drawbacks of Vickrey auctions. For instance, the term shill bidding [91] refers to a technique by which a party makes use of fake bidders to influence the winning price in an auction. Bid rigging [88] is another issue wherein several bidders

may collude to influence the selling price of an auction. Whilst several algorithms for the detection of collusive bidders who participate in fraudulent bidding do exist for some classes of auctions, the same approach cannot be applied to sealed bid auctions where bids are encrypted. One such algorithm for online English auctions has been described in [92], where a shill score is computed for one or more bidders after studying bidding patterns over a series of auctions.

14.1.1 *Our Contribution*

We propose a protocol for sealed bid auctions where we use $\mathcal{F}_{\text{CD}}$ for a committed database to allow bidder profiling algorithms (or aggregate statistical functions) to be run by bidders on their bid histories, without revealing the histories to other parties. The bidder may then prove that they have not been engaging in fraudulent bidding activities, to the vendor (the bidder could also exchange this result with an authority for a signed certificate, for instance, to be used with other vendors).

- We propose a functionality $\mathcal{F}_{\text{SBA}}$ for Sealed Bid Auctions, modelled around the protocols described in [70]. We also modify the protocol in [70] by introducing a non-interactive zero knowledge proof, allowing the auctioneer to prove to the bidders that the outcome of an auction has been computed correctly. Our $\mathcal{F}_{\text{SBA}}$ is additionally parameterized by a function which is used to determine the outcome of the auction, allowing it to be used for various auction classes. This functionality can be used as a modular building block to build larger auction protocols.

- We propose a functionality $\mathcal{F}_{\text{SBAS}}$ for a Sealed Bid Auction with Bidder Statistics, allowing bidders to reveal the result of a running a profiling or statistical function against their bid histories to a vendor.

- We propose a hybrid protocol Π_{SBAS} for a Sealed Bid Auction with Bidder Statistics that securely realizes $\mathcal{F}_{\text{SBAS}}$, constructed using $\mathcal{F}_{\text{CD}}$ and $\mathcal{F}_{\text{SBA}}$.

Though our protocol allows for the execution of profiling and statistical functions against the bid history of a single bidder, we remark that these functionalities can also be used in MPC commit and prove protocols thanks to the fact that we have modified them to receive commitments as input, in order to detect bidder collusion between multiple bidders by running fraud detection algorithms against their histories.

14.2 IDEAL FUNCTIONALITY FOR SEALED BID AUCTIONS

Functionality $\mathcal{F}_{\text{SBA}}$ interacts with bidders $\mathcal{B}_k$, a vendor $\mathcal{V}$, and an auctioneer $\mathcal{A}$. It consists of eight interfaces: sba.setup, sba.bidrequest, sba.bid, sba.close, sba.checkclosed, sba.result, sba.bidrequest, and sba.verify.

1. The auctioneer $\mathcal{A}$ uses the sba.setup interface to initialize the auction, and sends a sba.setup.end message to the vendor $\mathcal{V}$. As in Section 2.8.9, the sba.setup interface is also used to obtain the parameters par, trapdoor td, and the algorithms SBA.SimProve and SBA.Extract from $\mathcal{S}$. An empty set $\mathbb{S}$ is created to store the identities of bidders who have received these

parameters, as the verification of proofs is a local process (implying that the bidders must receive these parameters before they may execute the sba.verify interface).

2. A bidder $\mathcal{B}_k$ uses the sba.bidrequest interface to submit a bid value bid, a commitment com_{bid} and an opening $open_{bid}$ to bid to $\mathcal{F}_{\text{SBA}}$. $\mathcal{F}_{\text{SBA}}$ parses the commitment com_{bid} as $(com'_{bid}, parcom, \text{COM.Verify})$, and verifies this commitment by running COM.Verify. $\mathcal{F}_{\text{SBA}}$ then stores bid and its commitment com_{bid}. The simulator $\mathcal{S}$ learns the identity of the bidder, but not bid.

3. The vendor $\mathcal{V}$ uses the sba.bid interface to mark a bid as "received". $\mathcal{V}$ sends a commitment com_{bid} to a previously submitted bid bid to $\mathcal{F}_{\text{SBA}}$. $\mathcal{F}_{\text{SBA}}$ then records the bid bid associated with the commitment com_{bid} as received. We split the bid phase into two interfaces in order to allow $\mathcal{V}$ to ensure that a bidder $\mathcal{B}_k$ writes the correct bid value to functionality $\mathcal{F}_{\text{CD}}$ in a hybrid protocol.

4. The vendor $\mathcal{V}$ uses the sba.close interface to trigger the bid opening phase. $\mathcal{F}_{\text{SBA}}$ will not accept any new bids from a bidder during the bid opening phase.

5. A bidder $\mathcal{B}_k$ or the auctioneer $\mathcal{A}$ uses the sba.checkclosed interface to check whether the bid opening phase has been triggered by $\mathcal{V}$. If the bid opening phase has been triggered, $\mathcal{F}_{\text{SBA}}$ sends all commitments received from executions of the sba.bid interface to $\mathcal{B}_k$ or to $\mathcal{A}$.

6. $\mathcal{A}$ uses the sba.result interface to compute and send the outcome of the auction to the vendor $\mathcal{V}$. $\mathcal{F}_{\text{SBA}}$ determines the result of the auction by means of a function F. $\mathcal{F}_{\text{SBA}}$ also computes a simulated proof π in order to prove that the result of the auction has been computed correctly. The identity of the winning bidder $\mathcal{B}_{win}$, the bid amount to be paid by the winning bidder $result$, and the proof π are sent to $\mathcal{V}$.

7. A bidder $\mathcal{B}_k$ uses the sba.bidrequest interface to retrieve the results of the auction. If the outcome of the auction has already been computed and stored, $\mathcal{F}_{\text{SBA}}$ sends the results of the auction v and a proof π to $\mathcal{B}_k$.

8. A bidder $\mathcal{B}_k$ uses the sba.verify interface to determine whether the winning bid has been determined correctly and in accordance with F by the auctioneer $\mathcal{A}$.

For a Vickrey auction, bid_x represents the second-highest bid value while $y = x + 1$.

Figure 14.1: Functionality $\mathcal{F}_{\text{SBA}}$

Functionality $\mathcal{F}_{\text{SBA}}$ interacts with bidders $\mathcal{B}_k$, a vendor $\mathcal{V}$, and an auctioneer $\mathcal{A}$. It is parameterized by a maximum permissible bid value bid_{max}, and a function F from a universe of functions Ψ used to determine the outcome of the auction. SBA.SimProve and SBA.Extract are ppt algorithms.

1. On input (sba.setup.ini, sid) from $\mathcal{A}$:
   - Abort if $sid \notin (\mathcal{A}, \mathcal{V}, sid')$.

- Abort if $(sid, par, td, \mathsf{SBA.SimProve}, \mathsf{SBA.Extract})$ is already stored.
- Abort if $(sid, \mathsf{setup}, 0)$ is already stored.
- Store $(sid, \mathsf{setup}, 0)$.
- Send $(\mathsf{sba.setup.req}, sid)$ to $\mathcal{S}$.

S. On input $(\mathsf{sba.setup.alg}, sid, par, td, \mathsf{SBA.SimProve}, \mathsf{SBA.Extract})$ from $\mathcal{S}$:

- Abort if $(sid, \mathsf{setup}, 0)$ is not stored, or if $(sid, \mathsf{setup}, 1)$ is already stored.
- Store $(sid, par, td, \mathsf{SBA.SimProve}, \mathsf{SBA.Extract})$.
- Create an empty set $\mathbb{S}$.
- Store $(sid, \mathsf{setup}, 1)$.
- Send $(\mathsf{sba.setup.end}, sid)$ to $\mathcal{V}$.

2. On input $(\mathsf{sba.bidrequest.ini}, sid, bid, com_{bid}, open_{bid})$ from $\mathcal{B}_k$:

- Abort if $(sid, \mathcal{B}_k, bid', com_{bid'}, 0)$, for any value of bid' is already stored.
- Abort if $bid \notin [0, bid_{max}]$.
- Parse the commitment com_{bid} as $(com'_{bid}, parcom, \mathsf{COM.Verify})$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_{bid}, bid, open_{bid})$.
- Store $(sid, \mathcal{B}_k, bid, com_{bid}, 0)$.
- Send $(\mathsf{sba.bidrequest.sim}, sid, com_{bid})$ to $\mathcal{S}$.

S. On input $(\mathsf{sba.bidrequest.rep}, sid, com_{bid})$ from $\mathcal{S}$:

- Abort if $(sid, \mathcal{B}_k, bid, com_{bid}, 0)$ is not stored, or if $(sid, \mathcal{B}_k, bid', com'_{bid}, \mathsf{request})$ is already stored.
- Abort if $(sid, \mathsf{setup}, 1)$ is not stored.
- Abort if $(sid, (com_{bid_i})_{\forall i}, 1)$ is already stored.
- Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{B}_k$.
- Store $(sid, \mathcal{B}_k, bid, com_{bid}, \mathsf{request})$.
- Send $(\mathsf{sba.bidrequest.end}, sid, com_{bid})$ to $\mathcal{V}$.

3. On input $(\mathsf{sba.bid.ini}, sid, com_{bid})$ from $\mathcal{V}$:

- Abort if $(sid, \mathcal{B}_k, bid', com_{bid'}, \mathsf{request})$, where $com_{bid'} = com_{bid}$ is not stored, or if $(sid, \mathcal{B}_k, bid, com_{bid}, \mathsf{sim})$ is already stored.
- Store $(sid, \mathcal{B}_k, bid, com_{bid}, \mathsf{sim})$.
- Send $(\mathsf{sba.bid.sim}, sid, com_{bid})$ to $\mathcal{S}$.

S. On input (sba.bid.rep, sid, com_{bid}) from $\mathcal{S}$:

- Abort if $(sid, \mathcal{B}_k, bid, com_{bid}, \mathsf{sim})$ is not stored, or if $(sid, \mathcal{B}_k, bid', com_{bid'}, 1)$ is already stored.
- Store $(sid, \mathcal{B}_k, bid, com_{bid}, 1)$.
- Send (sba.bid.end, sid, com_{bid}) to $\mathcal{B}_k$.

4. On input (sba.close.ini, sid) from $\mathcal{V}$:

- Abort if $(sid, \mathsf{setup}, 1)$ is not stored.
- Abort if $(sid, (com_{bid_i})_{\forall i}, 0)$ or $(sid, (com_{bid_i})_{\forall i}, 1)$ is already stored.
- Retrieve and shuffle $(com_{bid_i})_{\forall i}$ from all values of com_{bid} received from executions of the bid interface and store $(sid, (com_{bid_i})_{\forall i}, 0)$.
- Send (sba.close.sim, sid, $(com_{bid_i})_{\forall i}$) to $\mathcal{S}$.

S. On input (sba.close.rep, sid) from $\mathcal{S}$:

- Abort if $(sid, (com_{bid_i})_{\forall i}, 0)$ is not stored, or if $(sid, (com_{bid_i})_{\forall i}, 1)$ is already stored.
- Store $(sid, (com_{bid_i})_{\forall i}, 1)$.
- Send (sba.close.end, sid) to $\mathcal{V}$.

5. On input (sba.checkclosed.ini, sid) from $\mathcal{B}_k$ or $\mathcal{A}$:

- Create a fresh qid and store $(sid, qid, \mathcal{B}_k)$ or $(sid, qid, \mathcal{A})$.
- Send (sba.checkclosed.sim, sid, qid) to $\mathcal{S}$.

S. On input (sba.checkclosed.rep, sid, qid) from $\mathcal{S}$:

- Abort if $(sid, qid, \mathcal{B}_k)$ or $(sid, qid, \mathcal{A})$ is not stored.
- Delete $(sid, qid, \mathcal{B}_k)$ or $(sid, qid, \mathcal{A})$.
- If $(sid, (com_{bid_i})_{\forall i}, 1)$ is not stored,
  - Set $v \leftarrow 0$.
- Otherwise,
  - Set $v \leftarrow (com_{bid_i})_{\forall i}$.
  - Store $(sid, \mathsf{retrieved}, \mathcal{B}_k)$ or $(sid, \mathsf{retrieved}, \mathcal{A})$, if it is not already stored.
- Send (sba.checkclosed.end, sid, v) to $\mathcal{B}_k$ or to $\mathcal{A}$.

6. On input (sba.result.ini, sid) from $\mathcal{A}$:

- Abort if $(sid, result, \mathcal{B}_{win}, \pi, u, 0)$ is already stored.
- Abort if $(sid, \mathsf{retrieved}, \mathcal{A})$ is not stored.

- Compute $(result, \mathcal{B}_{win})$ such that $\langle result, \mathcal{B}_{win} \rangle = \langle bid_x, \mathcal{B}_y \rangle$ is the result of computing F against all values stored in the tuples $(sid, \mathcal{B}_i, bid_i, com_{bid_i}, 1)_{\forall i}$.
- Run $\pi \leftarrow \mathsf{SBA.SimProve}(sid, par, td, (com_{bid_i})_{\forall i})$.
- Set $u \leftarrow 1$ and store $(sid, result, \mathcal{B}_{win}, \pi, u, 0)$.
- Send $(\mathsf{sba.result.sim}, sid, result, \mathcal{B}_{win}, \pi)$ to $\mathcal{S}$.

S. On input $(\mathsf{sba.result.rep}, sid)$ from $\mathcal{S}$:

- Abort if $(sid, result, \mathcal{B}_{win}, \pi, u, 0)$ is not stored, or if $(sid, result, \mathcal{B}_{win}, \pi, u, 1)$ is already stored.
- Set $v \leftarrow (result, \mathcal{B}_{win})$.
- Store $(sid, result, \mathcal{B}_{win}, \pi, u, 1)$.
- Send $(\mathsf{sba.result.end}, sid, v, \pi)$ to $\mathcal{V}$.

7. On input $(\mathsf{sba.retrieveresult.ini}, sid)$ from $\mathcal{B}_k$:

- Abort if $(sid, \mathsf{retrieved}, \mathcal{B}_k)$ is not stored.
- Create a fresh qid and store $(sid, qid, \mathcal{B}_k)$.
- Send $(\mathsf{sba.retrieveresult.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{sba.retrieveresult.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(sid, qid, \mathcal{B}_k)$ is not stored.
- Delete $(sid, qid, \mathcal{B}_k)$.
- If $(sid, result, \mathcal{B}_{win}, \pi, u, 1)$ is not stored,
  - Set $v \leftarrow 0$.
- Otherwise,
  - Set $v \leftarrow (result, \mathcal{B}_{win}, \pi)$.
  - Store $(sid, \mathsf{resultretrieved}, \mathcal{B}_k)$ if it is not already stored.
- Send $(\mathsf{sba.retrieveresult.end}, sid, v)$ to $\mathcal{B}_k$.

8. On input $(\mathsf{sba.verify.ini}, sid, \pi)$ from $\mathcal{B}_k$:

- Abort if $(sid, \mathsf{resultretrieved}, \mathcal{B}_k)$ is not stored.
- Abort if $\mathcal{B}_k \notin \mathbb{S}$.
- If $(sid, result, \mathcal{B}_{win}, \pi, u, 1)$ is stored, set $v \leftarrow u$. Otherwise, do the following:
  - Extract $((bid_i)_{\forall i}) \leftarrow \mathsf{SBA.Extract}(sid, par, td, (com_{bid_i})_{\forall i}, \pi)$.
  - Retrieve the tuple $(sid, \mathcal{B}_k, bid, com_{bid}, 1)$.
  - If $bid \notin ((bid_i)_{\forall i})$, set $v \leftarrow 0$, otherwise set $v \leftarrow 1$.
- Send $(\mathsf{sba.verify.end}, sid, v)$ to $\mathcal{B}_k$.

14.3 IDEAL FUNCTIONALITY FOR SEALED BID AUCTIONS WITH BIDDER STATISTICS

Functionality $\mathcal{F}_{\text{SBAS}}$ interacts with bidders $\mathcal{B}_k$, a vendor $\mathcal{V}$, and an auctioneer $\mathcal{A}$. It consists of six interfaces: sbas.setup, sbas.register, sbas.bid, sbas.result, sbas.verify, and sbas.revealstatistic.

1. The auctioneer $\mathcal{A}$ uses the sbas.setup interface to initialize an auction specified by an auction identifier aid. $\mathcal{F}_{\text{SBAS}}$ ensures that an auction cannot commence until the result of all other auctions have been computed, and sends a sbas.setup.end message to the vendor $\mathcal{V}$.

2. A bidder $\mathcal{B}_k$ uses the sbas.register interface to generate a table Tbl containing N_{max} entries of the form $[i, v]$, where i represents a position in the table and v represents the value stored at that position. $\mathcal{F}_{\text{SBAS}}$ initializes Tbl to $[i, 0]$ for all entries.

3. A bidder $\mathcal{B}_k$ uses the sbas.bid interface to send a bid value bid to $\mathcal{F}_{\text{SBAS}}$, whilst also specifying an aid. $Functionality_{\text{SBAS}}$ then stores bid and records this bid in Tbl in the entry that represents aid, i.e., Tbl is updated to store $[aid, bid]$. The simulator $\mathcal{S}$ learns the identity of the bidder, but not bid.

4. The vendor $\mathcal{V}$ uses the sbas.result interface to trigger the bid opening phase, and to compute the result of the auction identified by aid. $\mathcal{F}_{\text{SBAS}}$ will not accept any new bids from a bidder during the bid opening phase. $\mathcal{F}_{\text{SBAS}}$ uses F against all received bids to compute the result of the auction $\langle result, \mathcal{B}_{win} \rangle$, and sends it to $\mathcal{V}$.

5. A bidder $\mathcal{B}_k$ uses the sbas.verify interface to retrieve the result of an auction identified by aid, and to verify this result. $\mathcal{F}_{\text{SBAS}}$ sends the result of verification v to $\mathcal{B}_k$.

6. A bidder $\mathcal{B}_k$ uses the sbas.revealstatistic interface to compute the result of an aggregate or profiling function ST against its bidding history, stored in Tbl. $\mathcal{F}_{\text{SBAS}}$ sends the result of running ST against the Tbl associated with $\mathcal{B}_k$, v, to $\mathcal{V}$.

We use auction identifiers (aid) to allow $\mathcal{F}_{\text{SBAS}}$ to execute multiple auctions after initialization, but we enforce the fact that all auctions must be completed before a new one can commence. We associate a table Tbl with every bidder after they execute the sbas.register interface; this table is used to store their bid histories: every position in Tbl represents an aid, whilst the value at the position stores the bid value cast by the bidder $\mathcal{B}_k$ in this auction. $\mathcal{F}_{\text{SBAS}}$ is parameterized by a function F which is used to determine the outcome of an auction, allowing it to be used with a wide variety of auction types. In the sbas.revealstatistic interface, $\mathcal{F}_{\text{SBAS}}$ executes an aggregate or profiling function ST against the bid history of a bidder stored in its Tbl.

When invoked by $\mathcal{V}$, $\mathcal{A}$, or $\mathcal{B}_k$, $\mathcal{F}_{\text{SBAS}}$ first checks the correctness of the input. $\mathcal{F}_{\text{SBAS}}$ aborts if this input does not belong to the correct domain. $\mathcal{F}_{\text{SBAS}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example,

$\mathcal{A}$ cannot invoke sbas.result before sbas.setup. Similar abortion conditions are listed when $\mathcal{F}_{\text{SBAS}}$ receives a message from the simulator $\mathcal{S}$.

Before $\mathcal{F}_{\text{SBAS}}$ queries $\mathcal{S}$, $\mathcal{F}_{\text{SBAS}}$ saves its state, which is recovered upon receiving a response from $\mathcal{S}$. When an interface, e.g. sbas.revealstatistic, may be invoked more than once, $\mathcal{F}_{\text{SBAS}}$ creates a query identifier qid, which allows $\mathcal{F}_{\text{SBAS}}$ to match a query to $\mathcal{S}$ to a response from $\mathcal{S}$.

Figure 14.2: Functionality $\mathcal{F}_{\text{SBAS}}$

Functionality $\mathcal{F}_{\text{SBAS}}$ runs with bidders $\mathcal{B}_k$, a vendor $\mathcal{V}$, and an auctioneer $\mathcal{A}$. It is parameterized by a maximum permissible bid value bid_{max}, a function F, and a function ST from a universe of functions Φ.

1. On input (sbas.setup.ini, sid, aid) from $\mathcal{A}$:
   - Abort if $sid \notin (\mathcal{V}, \mathcal{A}, sid')$.
   - Abort if $(sid, aid, 0)$ is already stored, or if $aid \notin [1, N_{max}]$.
   - Store $(sid, aid, 0)$.
   - Send (sbas.setup.sim, sid, aid) to $\mathcal{S}$.

S. On input (sbas.setup.rep, sid, aid) from $\mathcal{S}$:
   - Abort if $(sid, aid, 0)$ is not stored, or if $(sid, aid, 1)$ is already stored.
   - If $aid \neq 1$, do the following:
     - Abort if $(sid, aid', result, \mathcal{B}_{win}, 1)$ where $aid' = aid - 1$ is not stored.
   - Store $(sid, aid, 1)$.
   - Send (sbas.setup.end, sid, aid) to $\mathcal{V}$.

2. On input (sbas.register.ini, sid) from $\mathcal{B}_k$:
   - Abort if $(sid, \text{registered}, \mathcal{B}_k, 0)$ is already stored.
   - Store $(sid, \text{registered}, \mathcal{B}_k, 0)$.
   - Send (sbas.register.sim, sid, $\mathcal{B}_k$) to $\mathcal{S}$.

S. On input (sbas.register.rep, sid, $\mathcal{B}_k$) from $\mathcal{S}$:
   - Abort if $(sid, aid', 1)$, where $aid' = 1$ is not stored.
   - Abort if $(sid, \text{registered}, \mathcal{B}_k, 0)$ is not stored, or if $(sid, \text{registered}, \mathcal{B}_k, \text{Tbl}, 1)$ is stored.
   - Initialize a table Tbl with entries $[i, 0]$ for $i = 1$ to N_{max}.
   - Store $(sid, \text{registered}, \mathcal{B}_k, \text{Tbl}, 1)$.
   - Send (sbas.register.end, sid) to $\mathcal{B}_k$.

3. On input (sbas.bid.ini, sid, aid, bid) from $\mathcal{B}_k$:
   - Abort if $(sid, aid, \mathcal{B}_k, bid', 0)$ for any value of bid' is already stored.

- Abort if $bid \notin [0, bid_{max}]$.
- Abort if $(sid, \mathsf{registered}, \mathcal{B}_k, \mathsf{Tbl}, 1)$ is not stored.
- Store $(sid, aid, \mathcal{B}_k, bid, 0)$.
- Send $(\mathsf{sbas.bid.sim}, sid, aid, \mathcal{B}_k)$ to $\mathcal{S}$.

S. On input $(\mathsf{sbas.bid.rep}, sid, aid, \mathcal{B}_k)$ from $\mathcal{S}$:

- Abort if $(sid, aid, \mathcal{B}_k, bid, 0)$ is not stored, or if $(sid, aid, \mathcal{B}_k, bid', 1)$ is already stored.
- Abort if $(sid, aid, 1)$ is not stored.
- Abort if $(sid, aid, (bid_i)_{\forall i})$ is already stored.
- Store $(sid, aid, \mathcal{B}_k, bid, 1)$.
- Retrieve the tuple $(sid, \mathsf{registered}, \mathcal{B}_k, \mathsf{Tbl}, 1)$, and update the entry $[aid, bid]$ in Tbl.
- Send $(\mathsf{sbas.bid.end}, sid, aid)$ to $\mathcal{V}$.

4. On input $(\mathsf{sbas.result.ini}, sid, aid)$ from $\mathcal{A}$:

- Abort if $(sid, aid, result, \mathcal{B}_{win}, 0)$ is already stored.
- Store $(sid, aid, (bid_i)_{\forall i})$ from all entries of $(sid, aid, \mathcal{B}_k, bid_k, 1)$ received from executions of the bid interface.
- Retrieve $(result, \mathcal{B}_{win})$ such that $\langle result, \mathcal{B}_{win}\rangle = \langle bid_x, \mathcal{B}_y\rangle$ is the result of computing F against all values stored in the tuples $(sid, aid, \mathcal{B}_i, bid_i, 1)_{\forall i}$.
- Store $(sid, aid, result, \mathcal{B}_{win}, 0)$.
- Send $(\mathsf{sbas.result.sim}, sid, aid, result)$ to $\mathcal{S}$.

S. On input $(\mathsf{sba.result.rep}, sid, aid)$ from $\mathcal{S}$:

- Abort if $(sid, aid, result, \mathcal{B}_{win}, 0)$ is not stored, or if $(sid, aid, result, \mathcal{B}_{win}, 1)$ is already stored.
- Store $(sid, aid, result, \mathcal{B}_{win}, 1)$.
- Set $v \leftarrow (result, \mathcal{B}_{win})$.
- Send $(\mathsf{sbas.result.end}, sid, aid, v)$ to $\mathcal{V}$.

5. On input $(\mathsf{sbas.verify.ini}, sid, aid)$ from $\mathcal{B}_k$:

- Abort if $(sid, aid, result, \mathcal{B}_{win}, 1)$ is not stored.
- If $\mathsf{F}(\langle bid_i, \mathcal{B}_i\rangle_{\forall i}) \neq (result, \mathcal{B}_{win})$, set $v \leftarrow 0$. Otherwise, set $v \leftarrow 1$.
- Create a fresh qid and store $(sid, aid, qid, \mathcal{B}_k, v)$.
- Send $(\mathsf{sbas.verify.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{sbas.verify.rep}, sid, qid)$ from $\mathcal{S}$:

- Abort if $(sid, aid, qid, \mathcal{B}_k, v)$ is not stored.
- Delete the tuple $(sid, aid, qid, \mathcal{B}_k, v)$.
- Send $(\mathsf{sbas.verify.end}, sid, aid, v)$ to $\mathcal{B}_k$.

6. On input $(\mathsf{sbas.revealstatistic.ini}, sid, \mathsf{ST})$ from $\mathcal{B}_k$:
   - Abort if $(sid, \mathsf{registered}, \mathcal{B}_k, \mathsf{Tbl}, 1)$ is not stored.
   - Abort if $\mathsf{ST} \notin \Phi$.
   - Set $v \leftarrow \mathsf{ST}(\mathsf{Tbl})$ where Tbl is retrieved from the stored tuple $(sid, \mathsf{registered}, \mathcal{B}_k, \mathsf{Tbl}, 1)$.
   - Create a fresh qid and store $(sid, qid, \mathsf{ST}, \mathcal{B}_k, v)$.
   - Send $(\mathsf{sbas.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{sbas.revealstatistic.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(sid, qid, \mathsf{ST}, \mathcal{B}_k, v)$ is not stored.
   - Delete the record $(sid, qid, \mathsf{ST}, \mathcal{B}_k, v)$.
   - Send $(\mathsf{sbas.revealstatistic.end}, sid, v, \mathsf{ST})$ to $\mathcal{V}$.

14.4 CONSTRUCTION

INTUITION.

BIDDER REGISTRATION $\mathcal{A}$ initializes a new copy of $\mathcal{F}_{\text{CD}}$ for each $\mathcal{B}_k$, and uses $\mathcal{F}_{\text{CD}}$ to send an empty initial Tbl to the bidder executing the $\mathsf{sbas.register}$ interface.

BIDDING $\mathcal{A}$ initializes a new copy of $\mathcal{F}_{\text{SBA}}$ for every aid. A bidder $\mathcal{B}_k$ uses the $\mathsf{sba.checkclosed}$ interface of the $\mathcal{F}_{\text{SBA}}$ associated with aid to check whether bids are still being accepted. If the auction is still open, $\mathcal{B}_k$ uses $\mathcal{F}_{ZK}^{R_{bid}}$ to prove to $\mathcal{V}$ in zero knowledge that the bid bid lies in the range $[0, bid_{max}]$, and that com_{bid} commits to bid with opening $open_{bid}$. $\mathcal{B}_k$ then uses the $\mathsf{cd.write}$ interface of its copy of $\mathcal{F}_{\text{CD}}$ to write bid to its committed database. $\mathcal{V}$ receives commitments to both aid and to bid, and checks if com_{bid} is the same commitment as that received from $\mathcal{F}_{ZK}^{R_{bid}}$. This ensures that the $\mathcal{B}_k$ has written the right value to its committed database. To prove that it has written bid to the right position in its committed database, $\mathcal{B}_k$ opens com_{aid} to aid by revealing the opening $open_{aid}$ to $\mathcal{V}$.

AUCTION RESULTS $\mathcal{V}$ uses the $\mathsf{sba.close}$ interface of the copy of $\mathcal{F}_{\text{SBA}}$ associated with aid to close an auction before requesting $\mathcal{A}$ to determine the outcome of said auction. $\mathcal{A}$ then uses the $\mathsf{sba.checkclosed}$ interface of $\mathcal{F}_{\text{SBA}}$ to check if the auction has been closed before using $\mathsf{sba.result}$ to compute the result of the auction. The result of the auction (in accordance with F) is sent to $\mathcal{V}$ (in a construction for $\mathcal{F}_{\text{SBA}}$, this result would typically be published to a bulletin board).

BIDDER STATISTICS AND PROFILING $\mathcal{B}_k$ executes an aggregate or profiling function ST against its bid history stored in its committed database, and uses the cd.read interface of $\mathcal{F}_{\text{CD}}$ to read all entries that are required for the evaluation of ST; $\mathcal{V}$ receives commitments to these entries. $\mathcal{B}_k$ then uses $\mathcal{F}_{ZK}^{R_{stat}}$ to prove to $\mathcal{V}$ in zero knowledge that the result of ST has been computed correctly. $\mathcal{V}$ checks if the commitments received from $\mathcal{F}_{\text{CD}}$ and those from $\mathcal{F}_{ZK}^{R_{stat}}$ are the same. In this case, $\mathcal{V}$ learns no information on the bid history of $\mathcal{B}_k$, but is still able to learn the result of ST.

Figure 14.3: Description of Π_{SBAS}.

Construction Π_{SBAS} runs with bidders $\mathcal{B}_k$, a vendor $\mathcal{V}$, and an auctioneer $\mathcal{A}$. It is parameterized by a maximum permissible bid value bid_{max}, and a function ST from a universe of functions Φ. Π_{SBAS} uses the functionalities $\mathcal{F}_{\text{SBA}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{CD}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{AUT}}$, and $\mathcal{F}_{\text{ZK}}^{R}$ as building blocks.

1. On input (sbas.setup.ini, sid, aid), $\mathcal{A}$ and $\mathcal{V}$ do the following:
   - $\mathcal{A}$ aborts if $sid \notin (\mathcal{V}, \mathcal{A}, sid')$.
   - If $aid \neq 1$, $\mathcal{A}$ does the following:
     - $\mathcal{A}$ aborts if $(sid, aid', \text{result}, \mathcal{B}_{win})$ where $aid' = aid - 1$ is not stored, or if $(sid, aid, \text{result}, \mathcal{B}_{win})$ is already stored.
   - $\mathcal{A}$ sets a session identifier $sid_{aid} \leftarrow (sid, aid)$ and sends the message (sba.setup.ini, sid_{aid}) to a new instance of $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{V}$ receives the message (sba.setup.end, sid_{aid}) from $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{V}$ sends the message (com.setup.ini, sid) to a new instance of $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{V}$ receives the message (com.setup.end, sid, Ok) from $\mathcal{F}_{\text{NIC}}$.
   - $\mathcal{V}$ stores (sid, aid, setup).
   - $\mathcal{V}$ outputs (sbas.setup.end, sid, aid).
2. On input (sbas.register.ini, sid), a bidder $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ aborts if $sid \notin (\mathcal{A}, \mathcal{V}, sid')$.
   - $\mathcal{B}_k$ uses the aut.send interface of $\mathcal{F}_{\text{AUT}}$ to send the message (register) to $\mathcal{V}$.
   - $\mathcal{V}$ receives the message (register) from $\mathcal{F}_{\text{AUT}}$, and aborts if $(sid, \text{registered}, \mathcal{B}_k)$ is already stored.
   - $\mathcal{V}$ sets up a table Tbl of N_{max} entries of the form $[i, 0]$.
   - $\mathcal{V}$ stores $(sid, \text{registered}, \mathcal{B}_k)$.
   - $\mathcal{V}$ sets $sid_{\mathcal{B}_k} \leftarrow (\mathcal{V}, \mathcal{B}_k, sid)$, and sends the message (cd.setup.ini, $sid_{\mathcal{B}_k}$, Tbl) to $\mathcal{F}_{\text{CD}}$.
   - $\mathcal{B}_k$ receives the message (cd.setup.end, $sid_{\mathcal{B}_k}$, Tbl) from $\mathcal{F}_{\text{CD}}$.

- $\mathcal{B}_k$ sends the message $(\mathsf{com.setup.ini}, sid)$ to a new instance of $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}_k$ receives the message $(\mathsf{com.setup.end}, sid, \mathsf{Ok})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}_k$ stores (sid, Tbl), and outputs the message $(\mathsf{sbas.register.end}, sid)$.

3. On input $(\mathsf{sbas.bid.ini}, sid, aid, bid)$, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ aborts if $bid \notin [0, bid_{max}]$.
   - $\mathcal{B}_k$ aborts if (sid, Tbl) is not stored.
   - $\mathcal{B}_k$ aborts if $(sid, aid, (com_{bid})_{\forall i})$ is already stored.
   - $\mathcal{B}_k$ retrieves the entry $[aid, j]$ from Tbl, and aborts if $j \neq 0$.
   - $\mathcal{B}_k$ sends the message $(\mathsf{sba.checkclosed.ini}, sid_{aid})$ to $\mathcal{F}_{\mathrm{SBA}}$.
   - $\mathcal{B}_k$ receives the message $(\mathsf{sba.checkclosed.end}, sid_{aid}, v)$ from $\mathcal{F}_{\mathrm{SBA}}$.
   - If $v \neq 0$,
     - $\mathcal{B}_k$ parses v as $(com_{bid})_{\forall i}$.
     - $\mathcal{B}_k$ stores $(sid, aid, (com_{bid})_{\forall i})$.
   - Otherwise,
     - $\mathcal{B}_k$ sends the message $(\mathsf{com.commit.ini}, sid, bid)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}_k$ receives the message $(\mathsf{com.commit.end}, sid, com_{bid}, open_{bid})$ from $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}_k$ sets $wit_{bid} \leftarrow (bid, open_{bid})$.
     - $\mathcal{B}_k$ parses the commitment com_{bid} as $(com'_{bid}, parcom, \mathsf{COM.Verify})$.
     - $\mathcal{B}_k$ sets $ins_{bid} \leftarrow (parcom, com'_{bid})$.
     - $\mathcal{B}_k$ stores $(sid, wit_{bid}, ins_{bid})$.

The relation R_{bid} is defined as follows:

$$\begin{aligned} R_{bid} = \{&(wit_{bid}, ins_{bid}) : \\ &1 = \mathsf{COM.Verify}(parcom, com_{bid}, bid, open_{bid}) \wedge \\ &bid \in [0, bid_{max}]\} \end{aligned}$$

   - $\mathcal{B}_k$ sends $(\mathsf{zk.prove.ini}, sid_{\mathcal{B}_k}, wit_{bid}, ins_{bid})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{bid}}$.
   - $\mathcal{V}$ receives the message $(\mathsf{zk.prove.end}, sid_{\mathcal{B}_k}, ins_{bid})$ from $\mathcal{F}_{\mathrm{ZK}}$.
   - $\mathcal{V}$ parses ins_{bid} as $(parcom, com'_{bid})$.

- $\mathcal{V}$ sets the commitment $com_{bid} \leftarrow (com'_{bid}, parcom,$ com.verify$)$.
- $\mathcal{V}$ uses com.validate to validate com_{bid}.
- $\mathcal{V}$ aborts if (sid, com_{bid}) is already stored.
- $\mathcal{V}$ stores (sid, com_{bid}).
- $\mathcal{V}$ uses aut.send to send the message (writebid) to $\mathcal{B}_k$.
- $\mathcal{B}_k$ sends the message (com.commit.ini, aid) to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{B}_k$ receives the message (com.commit.end, com_{aid}, $open_{aid}$) from $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{B}_k$ stores $(sid, com_{aid}, open_{aid})$.
- $\mathcal{B}_k$ sends the message (cd.write.ini, $sid_{\mathcal{B}_k}$, com_{aid}, aid, $open_{aid}$, com_{bid}, bid, $open_{bid}$) to $\mathcal{F}_{\text{CD}}$, in order to write an entry $[aid, bid]$ to the table of $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ receives the message (cd.write.end, $sid_{\mathcal{B}_k}$, com_{aid}, com_{bid}) from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ aborts if the commitment com_{bid} stored in (sid, ins_{bid}) is not the same as the commitment received from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ uses aut.send to send the message (revealauctionid) to $\mathcal{B}_k$.
- $\mathcal{B}_k$ updates Tbl with $[aid, bid]$.
- $\mathcal{B}_k$ uses smt.send to send $(aid, open_{aid})$ to $\mathcal{V}$.
- $\mathcal{V}$ uses com.verify to verify $(com_{aid}, aid, open_{aid})$, and stores (sid, aid, ins_{bid}).
- $\mathcal{V}$ uses aut.send to send the message (submitbid) to $\mathcal{B}_k$.
- $\mathcal{B}_k$ sends the message (sba.bidrequest.ini, sid_{aid}, bid, com_{bid}, $open_{bid}$) to $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ receives the message (sba.bidrequest.end, sid_{aid}, com_{bid}) from $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ aborts if the commitment com_{bid} stored in (sid, aid, ins_{bid}) is not the same as the commitment received from $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ sends the message (sba.bid.ini, sid_{aid}, com_{bid}) to $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{B}_k$ receives the message (sba.bid.end, sid_{aid}, com_{bid}) from $\mathcal{F}_{\text{SBA}}$.

• $\mathcal{B}_k$ outputs the message (sbas.bid.end, sid, aid).

4. On input (sbas.result.ini, sid, aid), $\mathcal{A}$ and $\mathcal{V}$ do the following:
   • $\mathcal{V}$ sends the message (sba.close.ini, sid_{aid}) to $\mathcal{F}_{\text{SBA}}$.

- $\mathcal{V}$ receives the message (sbas.close.end, sid_{aid}) from $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ uses the aut.send interface of $\mathcal{F}_{\text{AUT}}$ to send the message (aid, computeresult) to $\mathcal{A}$.
- $\mathcal{A}$ sends the message (sba.checkclosed.ini, sid_{aid}) to $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{A}$ receives the message (sba.checkclosed.end, sid_{aid}, v) from $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{A}$ aborts if $v = 0$.
- $\mathcal{A}$ sends the message (sba.result.ini, sid_{aid}) to $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ receives the message (sba.result.end, sid_{aid}, v, π) from $\mathcal{F}_{\text{SBA}}$.
- $\mathcal{V}$ parses v as (result, $\mathcal{B}_{win}$) and stores (sid, aid, result, $\mathcal{B}_{win}$).
- $\mathcal{V}$ outputs (sbas.result.end, sid, aid).

5. On input (sbas.verify.ini, sid, aid), $\mathcal{B}_k$ does the following:
   - $\mathcal{B}_k$ sends the message (sba.retrieveresult.ini, sid_{aid}) to $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{B}_k$ receives the message (sba.retrieveresult.end, sid_{aid}, v) from $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{B}_k$ aborts if $v = 0$.
   - $\mathcal{B}_k$ parses v as ($result, \mathcal{B}_{win}, \pi$), and sends the message (sba.verify.ini, sid_{aid}, π) to $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{B}_k$ receives the message (sba.verify.end, sid_{aid}, v') from $\mathcal{F}_{\text{SBA}}$.
   - $\mathcal{B}_k$ outputs (sba.verify.end, sid, aid, v').

6. On input (sbas.revealstatistic.ini, sid, ST), $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ aborts if (sid, Tbl) is not stored.
   - $\mathcal{B}_k$ aborts if ST $\notin \Phi$.
   - $\mathcal{B}_k$ computes result $\leftarrow$ ST(Tbl) where Tbl is retrieved from the stored tuple (sid, Tbl).
   - For each entry $[i, v]$ in Tbl, where v represents a value that was used by $\mathcal{B}_k$ to compute result, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
     - $\mathcal{B}_k$ uses the com.commit interface of $\mathcal{F}_{\text{NIC}}$ to obtain commitments com_i and openings $open_i$ to i.
     - $\mathcal{B}_k$ uses the com.commit interface of $\mathcal{F}_{\text{NIC}}$ to obtain commitments com_v and openings $open_v$ to v.
     - $\mathcal{B}_k$ stores (sid, i, com_i, v, com_v).
     - $\mathcal{B}_k$ sends the message (cd.read.ini, $sid_{\mathcal{B}_k}, com_i, i,$ $open_i, com_v, v, open_v$) to $\mathcal{F}_{\text{CD}}$.

  - $\mathcal{V}$ receives the message $(\mathsf{cd.read.end}, sid_{\mathcal{B}_k}, com_i, com_v)$ from $\mathcal{F}_{\mathrm{CD}}$.
  - $\mathcal{V}$ uses the $\mathsf{com.validate}$ interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate com_i.
  - $\mathcal{V}$ uses the $\mathsf{com.validate}$ interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate com_v.
  - $\mathcal{V}$ uses the $\mathsf{aut.send}$ interface of $\mathcal{F}_{\mathrm{AUT}}$ to send the message $(\mathsf{continue}, com_i, com_v)$ to $\mathcal{B}_k$.
- $\mathcal{B}_k$ sets $wit_{stat} \leftarrow (\langle i, open_i, v, open_v \rangle \forall i)$.
- $\mathcal{B}_k$ parses com_i as $(com'_i, parcom, \mathsf{COM.Verify})$ and com_v as $(com'_v, parcom, \mathsf{COM.Verify})$ for all i.
- $\mathcal{B}_k$ sets $ins_{stat} \leftarrow (\mathsf{result}, parcom, \mathsf{ST}, \langle com'_i, com'_v \rangle_{\forall i})$.

The relation R_{stat} is defined as follows:

$$
\begin{aligned}
R_{stat} = \{ & (wit_{stat}, ins_{stat}) : \\
& [1 = \mathsf{COM.Verify}(parcom, com_i, i, open_i) \wedge \\
& 1 = \mathsf{COM.Verify}(parcom, com_v, v, open_v) \,]_{\forall i} \wedge \\
& \mathsf{result} = \mathsf{ST}(\langle i, v \rangle_{\forall i}) \}
\end{aligned}
$$

- $\mathcal{B}_k$ sends the message $(\mathsf{zk.prove.ini}, sid_{\mathcal{B}_k}, ins_{stat}, wit_{stat})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{stat}}$.
- $\mathcal{V}$ receives the message $(\mathsf{zk.prove.end}, sid_{\mathcal{B}_k}, ins_{stat})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{stat}}$.
- $\mathcal{V}$ aborts if the commitments $(com_i, com_v)_{\forall i}$ received from $\mathcal{F}_{\mathrm{CD}}$ are not the same as those in ins_{stat}.
- $\mathcal{V}$ outputs $(\mathsf{sbas.revealstatistic.end}, sid, \mathsf{result}, \mathsf{ST})$.

Part IV

IMPLEMENTATIONS

We elucidate aspects of our implementations of Stateful Zero Knowledge (SZK) data structures and protocols, introduce an extension of a notation for Zero Knowledge (ZK) proofs with support for Stateful Zero Knowledge (SZK), and describe a compiler which produces implementations based on protocols specified by means of the aforementioned notation.

15

DATA STRUCTURE AND PROTOCOL IMPLEMENTATIONS

We have implemented some primitives outlined in Part II and a protocol from Part III in order to gauge their efficiency, and to show that these schemes are practical even for large database sizes. We used the Charm Cryptographic Prototyping Framework [4] to develop proof of concept implementations for these schemes as Charm provides a Python wrapper that allows us to use Python's features (such as lists and native JSON support), whilst low level cryptographic operations are carried out in C using the Relic toolkit [7].

These implementations were developed modularly so that modules could be reused as components of the code generator in our SZK compiler. The modules include:

- An implementation of a Vector Commitment scheme [32],
- An implementation of the Pedersen Commitment scheme [80],
- An implementation of an Integer Commitment scheme [34],
- An implementation of a Structure Preserving Signature scheme [2],
- Implementations of the ideal functionalities in Chapter 2, and,
- Implementations of ZK functionalities based on the compiler in [24].

In order to facilitate the replication of our results and to make it easier for researchers to extend our code, we have published a script with the source code in [40] which can be used to reproduce our testing environment in a virtual machine, doing away with the need for the installation and configuration of the libraries required for the execution of our code.

All efficiency measurements were carried out on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve was used for the pairing group setup in all cases, unless otherwise specified. These measurements (in Section 7.6.5, Section 8.6.5, Section 9.6.5, and Section 13.6) all show that our primitives are practical for large database sizes.

16 A COMPILER FOR STATEFUL ZERO KNOWLEDGE

Joint work with Alfredo Rial. Work in progress; to be submitted in 2022.

16.1 INTRODUCTION

As discussed in Chapter 1, the fact that the implementation of the constructions in this thesis requires knowledge of several cryptographic concepts could prove to be a hindrance to the adoption of our data structures and techniques (in the construction and development of privacy preserving protocols) by developers who are not well versed with these mathematical aspects.

16.2 OUR CONTRIBUTION

We provide a notation allowing practitioners to specify the data structures they would wish to implement (after determining which of these data structures offers the security guarantees they desire), and the operations they would wish to run against these structures, whilst also specifying the proof statements they would wish to evaluate during the execution of each operation. We then go on to provide a compiler which takes this notation as input, and produces the following as output:

- UC-secure code that implements the requested data structure,
- Functions that execute the requested operations on the generated data structure, and,
- UC-secure code that implements ZK functionalities that prove statements specified in the input statement, based on the content of the generated data structures.

As this compiler was designed with our protocols in mind, we offer support for pairing product equations and bilinear maps. We also allow for the definition of task based cryptographic operations via macros in the sense that programmers can specify predefined function strings representing common cryptographic tasks such as encryption and signing, without needing to worry about the internals. The compiler converts these strings into pairing product equations, as depicted in figure Figure 16.2.

Figure 16.1: Pederson Commitment Macro Expansion

$$pedcom(par, message, opening) \Rightarrow e(g^{message}.par^{opening}, \tilde{g})$$

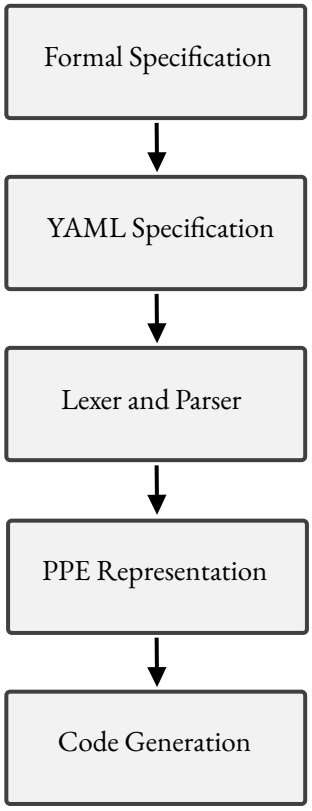


Figure 16.2: The architecture of our SZK compiler.

16.3 RELATED WORK

Notations for ZK proofs allow you to represent the relation being proved concisely. A few widely used notations can be found in [23, 28]. In [24], ZK compilers [5, 6, 48, 50, 74] take as input the specification of a ZK proof and output an executable implementation of the corresponding ZK proof protocol. Some compilers use notations for cryptographic protocols [5, 6, 24], whilst others take as input computations represented in restricted high-level languages [50, 74]. The notation in [28] is used in [5] as input to the compiler. [6] extends [5] to provide a verified compiler. [71] features a Python library for prototyping composable zero-knowledge proofs in the discrete-log setting.

Our compiler differs from these compilers by virtue of the fact that we additionally provide implementations of an SZK data structure as output. As the data structure has been decoupled from the proof functionality, this would allow us to extend an existing ZK compiler. However, as our proof statements involve pairing product equations and bilinear maps, we chose to build a compiler from scratch. Our compiler also realizes a hybrid protocol that uses ideal protocols for ZK, SZK and non-interactive commitments as building blocks which together implement the specified ZK proof.

16.4 ARCHITECTURE

The architecture of our compiler is depicted in Figure 16.2. Data structure requirements, operations, and statements to be proved are specified using our formal notation. In order to be fed as input to the compiler, these specifications must be represented in a data transfer format (YAML), before being sent to a lexer and a parser. The resulting syntax tree is then used to build pairing product equations for the proof statements, and the code generator finally produces implementations for the data structure and operations specified, and implementations for the proof statements. We describe each section in detail.

16.4.1 *Formal Notation*

We extend the notation described in [24]. The intuition behind this decision stems from the fact that this notation guarantees that a UC secure protocol for a relation exists, if it can be specified using the notation.

$$\underset{(1)}{\mathsf{CD}}[\underset{(2)}{\mathsf{purchase}} : \{\underset{(3)}{\mathsf{read}} : \underset{(4)}{\nabla\!\!\!\!\!\!\backslash\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n)}\}]$$

SEGMENT 1 To specify the data structure to be used, we use its 2-4 letter acronym. Thus, the possible options are CD, UCD, UUD, UUHD, NICD, or UD.

SEGMENT 2 To define the operations to be performed against the data structure, we use a label. This label can be derived from the nature of the task to be performed, i.e., "deposit" or "profile".

SEGMENT 3 Within each operation, we specify whether a database read, a write, or an update is being performed. The permissible options will depend on the type of database in question.

SEGMENT 4 Within each operation, we use the notation from [24] to denote the relation to be proved. We use the literals "i","v","i_n" and "v_n" to expose the database entries being read, written or updated, so that they may be included in the relation. To reiterate, in [24], a UC ZK protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ that satisfy the formula $\phi(w_1, \ldots, w_n)$ is described as $\nabla\!\!\!\!\!\!\backslash\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n)$. The formula $\phi(w_1, \ldots, w_n)$ consists of conjunctions and disjunctions of atoms. An atom expresses *group relations*, such as $\prod_{j=1}^{k} g_j^{\mathcal{F}_j} = 1$, where the g_j's are elements of prime order groups and the $\mathcal{F}_j$'s are polynomials in the variables $(w_1, \ldots, w_n)$. This proof system can be transformed into a proof system for more expressive statements about secret exponents $sexps$ and secret bases $sbases$: $\nabla\!\!\!\!\!\!\backslash\, sexps, sbases : \phi(sexps, bases \cup sbases)$. The transformation adds a base h to the public bases. For each $g_j \in sbases$, the transformation picks a random exponent ρ_j and computes a blinded base $g'_j = g_j h^{\rho_j}$. The transformation adds g'_j to the public bases $bases$, ρ_j to the secret exponents $sexps$, and rewrites $g_j^{\mathcal{F}_j}$ into $g'^{\mathcal{F}_j}_j h^{-\mathcal{F}_j \rho_j}$. The proof system supports pairing product equations $\prod_{j=1}^{k} e(g_j, \tilde{g}_j)^{\mathcal{F}_j} = 1$ in groups of prime order with

a bilinear map e, by treating the target group $\mathbb{G}_t$ as the group of the proof system. The embedding for secret bases is unchanged, except for the case in which both bases in a pairing are secret. In this case, $e(g_j, \tilde{g}_j)^{\mathcal{F}_j}$ must be transformed into $e(g'_j, \tilde{g}'_j)^{\mathcal{F}_j} e(g'_j, \tilde{h})^{-\mathcal{F}_j \tilde{\rho}_j} e(h, \tilde{g}'_j)^{-\mathcal{F}_j \rho_j} e(h, \tilde{h})^{\mathcal{F}_j \rho_j \tilde{\rho}_j}$.

16.4.2 *Specification Language*

In order to pass protocol specifications to the compiler, they must be transcribed into a format that can be read by our code. We have used the YAML markup language for data serialization, and this choice was driven by the fact that YAML gives us the following benefits:

- Most developers would already be familiar with YAML as it finds use in several programming languages.
- YAML features several enhancements over alternatives (JSON, XML). For instance, YAML allows for the use of comments.
- Modularity: YAML is platform independent, which means that these files may be consumed by other programming languages.
- The markup language is more readable compared to JSON.

Figure 16.4: A sample YAML SZK protocol description file

```
data_structure: uuhd

operations:
       - label: purchase
         db_operation: update
         witness_vars: [i,v,x,o]
         instance_vars: [com,h]
         statements: v=(2*i)+x && com=pedcom(h,x,o)
       - label: redeem
         db_operation: update
         witness_vars: [v_2,v_1,y,o]
         instance_vars: [com,h]
         statements: v_2=v_1-y && com=pedcom(h,y,o)
```

Our YAML specification files are structured as follows:

DATA STRUCTURE The data structure field contains a value specifying the SZK data structure to be used, and corresponds to segment 1 in our formal notation.

OPERATIONS The operations field must be an array containing the specifications for one or more operations.

LABEL The label field, within an operation, corresponds to segment 2 in our formal specification, and must contain a name denoting the database operation or task to be performed.

DB OPERATION This field corresponds to segment 3 in our formal specification, specifying the database operation (read, write, or update) to be invoked within an operation.

WITNESS VARS The values meant to be secret within the proof protocol must be listed in this field.

INSTANCE VARS The values meant to be publicly known within the proof protocol must be listed in this field.

STATEMENTS The statements field corresponds to segment 4 in our formal specification. It must contain the relations to be used for the ZK proof within an operation. We use double ampersand symbols (&&) to specify conjunctions of statements.

16.4.2.1 *Macros*

In order to further facilitate the high level use of our compiler, the compiler also provides support for macros: YAML files containing the description of a function for a high level cryptographic task, and the equivalent pairing product equation for the task. For instance, the following YAML file contains the description of a macro for Pedersen commitments:

Figure 16.5: A macro file for pedersen commitments

```
macro_title: "Pedersen Commitment"
macro_author: "Test"
macro_label: "pedcom"
argument_count: 3
expansion: $r = e((g^$2)*($1^$3),gt)
```

These macros serve to provide an additional layer of abstraction, if required.

16.4.3 *Lexer and Parser*

For the front end of our compiler, we use the Python programming language. We employ Lark [90], an LALR(1) parsing library for tokenization and the construction of abstract syntax trees using an EBNF grammar specification.

We enforce the fact that all exponents in PPEs must be secret values, and reserve the literals e(), g, gt, i, v, i_n, and v_n for use with PPEs.

16.4.4 *Code Generation*

The backend of our compiler generates Python code for use with the Charm cryptographic framework [4], which in turn uses the Relic toolkit [7] to execute low level cryptographic operations in C for increased efficiency. The backend additionally generates a Vagrant script: a set of instructions that may be used by the Vagrant virtual machine management tool [54] to set up a virtual machine containing a pre-configured environment where the aforementioned libraries have been installed, to save developers from having to deal with the installation and secure configuration

of these libraries. Thanks to the modularity of our SZK data structures, the code for all six of our data structures remains static and does not depend on the input to the compiler. However, the backend converts PPEs into functions. One such function is depicted in Figure 16.4.4. All secret values for each PPE must be passed as arguments to a PPE function, and the *side* argument allows us to evaluate either the LHS or the RHS of a PPE.

Passing witness values as arguments to PPE functions facilitates the evaluation of these functions with randomized witnesses during the course of execution of a ZK proof.

Figure 16.6: A PPE function in Python

```
def compute_ppe_1(self, d_1, d_2, d_3, d_4, side):
"""Pairing product equation 1."""
if side == "lhs":
    return (
        (pair(self.r_d, self.v) ** self.one)
        * (pair(self.s_d, self.gt) ** self.one)
        * (pair(self.vcomd, self.w_1) ** self.one)
        * (pair(self.comd, self.w_2) ** self.one)
        * (pair(self.g, self.z) ** -1)
    )
else:
    return (
        (pair(self.h, self.v) ** d_1)
        * (pair(self.h, self.gt) ** d_2)
        * (pair(self.g, self.w_1) ** d_3)
        * (pair(self.ped_h, self.w_2) ** d_4)
    )
```

The function in Figure 16.4.4 corresponds to a PPE function generated by the compiler based on a PPE function proving that $ccom_i$ is a commitment to i. The equivalent PPE is

$$e(ccom_i, \tilde{g}) = e(ped_g, \tilde{g})^i . e(ped_h, \tilde{g})^{copen_i}$$

Figure 16.7: A generated PPE function for a pedersen commitment in Python

```
def compute_ppe_3(self, index, i, copen_i, side):
"""Pairing product equation 3."""
if side == "lhs":
    return pair(self.ccom_i, self.gt) ** self.one
else:
    return (pair(self.ped_g, self.gt) ** i) * (
        pair(self.ped_h, self.gt) ** copen_i
    )
```

The code generator then plugs these functions in to a Σ-protocol as described in [24], and generates code for the computation and verification of integer commitments and Paillier encryptions of all witness values, and code for the Digital Signature Algorithm, as required by the framework. The output of the code generator comprises the code for a data structure, and code for the requested UC-ZK protocols, and functions for each protocol operation: these functions perform

a database read, write, or update, and then pass the values and/or commitments received as output to the ZK proof functions.

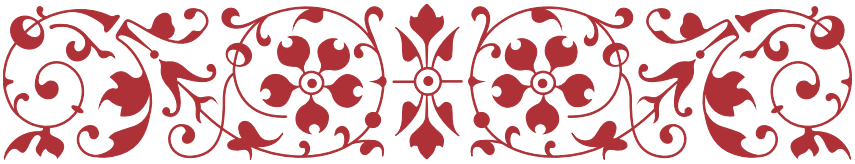
16.5 CONSIDERATIONS

- As our compiler has been designed with the intention of allowing developers to build protocols that leverage our SZK data structures, it has been implemented to allow for simple proof statements similar to those used by the protocols described in this thesis, involving range proofs, proofs that prove that a value has been updated correctly, and so on. The compiler has not been tested with complex proof statements, though thanks to its modular design this would only require tweaking the code generator.
- The compiler does not currently support multiple database operations within a protocol operation (for instance, a transfer operation in our POTS protocol requires multiple read and write operations), as this would add complexity to our formal notation, requiring a means of declaring variables and using them to store values between subsequent database operations. However, it is possible to overcome this limitation by means of a workaround: a developer would merely need to use the compiler to generate code for one protocol operation for each database operation required, and then connect these function calls by adding variables to store state between them.

16.6 FUTURE WORK

1. The Relic toolkit is not recommended for use in production environments, and is solely meant for proof of concept implementations. We intend to replace libraries in our code generator with those meant for production ready implementations.
2. The backend will be enhanced to support the declaration of aliases: variables that can be used to connect database entries between database operations. In other words, aliases would allow us to read a value from a database and then use this value in a write operation, for instance. This would allow for the generation of complex protocol operations involving multiple database operations, as discussed in Section 16.5.

Part V



CONCLUSION

16.7 CONCLUSION

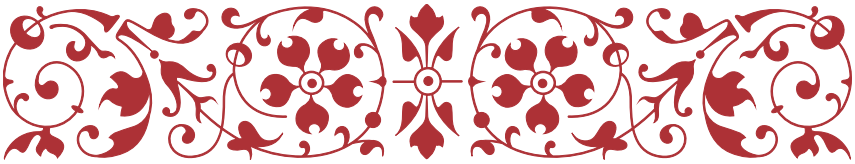
- In Part II we have provided formal definitions and constructions for our Stateful Zero Knowledge (SZK) data structures.
- In Part III we use SZK data structures as building blocks to construct privacy preserving protocols, and compare them to existing protocols. In some of our protocols, we provide additional features whilst maintaining efficiency when compared to similar protocols that do not provide such features.
- In Part II and in Part III, we also describe instantiations for some of our primitives and protocols, and implement them. The measurements from our implementations attest to the fact that these primitives and protocols remain efficient even with large input sizes.
- In Part IV, we describe a notation for specifying SZK protocols, and list the implementation details for a compiler which can be used to produce proof of concept implementations of SZK protocols.

Our protocols have been designed modularly, allowing for improvements when building blocks within these protocols are replaced by more efficient instantiations. By decoupling witnesses from proof protocols, we allow users participating in protocols to maintain control over their data whilst still allowing other entities (supermarkets, credential issuers, vendors) to compute aggregate statistics on this data, without loss of privacy. Our primitives can be used to build more such protocols with a variety of desirable security properties such as unlinkability and unforgeability. Our compiler can also be extended to produce production ready implementations. We hope that these features will be appealing to both users, and entities that wish to collect statistics on user information alike, whilst preserving privacy and that our compiler will aid in the adoption and development of such protocols.

16.8 FUTURE WORK

- We intend to construct SZK data structures modelled around other data structures, such as lists and trees.
- Our compiler will be extended to support complex proof specifications.
- Our compiler will be extended to produce production ready implementations.
- Our Privacy Preserving Loyalty Program (PPLP) program in Part III involves the use of a database maintained by a user, akin to a loyalty card. A future research direction could involve the development of mobile computing libraries that support such implementations on mobile devices.
- Another protocol involving privacy preserving location based services will also be designed.

Part VI



APPENDIX

A

SECURITY ANALYSIS

A.1 SECURITY ANALYSIS OF OUR CONSTRUCTION FOR NICD

Theorem A.1.1 *The construction* Π_{NICD} *securely realizes* $\mathcal{F}_{\text{NICD}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NIZK}}^{R_r}$ *and* $\mathcal{F}_{\text{NIZK}}^{R_w}$*-hybrid model if the vector commitment scheme is hiding and binding.*

To prove that our construction Π_{NICD} securely realizes the ideal functionality $\mathcal{F}_{\text{NICD}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\text{NICD}}$. The simulator thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\text{NICD}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to the adversary if the functionality sends the abortion message to a corrupt party.

Our simulator $\mathcal{S}$ uses simulators $\mathcal{S}_{\text{NIZK}}^{R_r}$ and $\mathcal{S}_{\text{NIZK}}^{R_w}$ for the constructions that realize $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$, respectively. We note that the simulators for all constructions that realize $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$ communicate with each of those functionalities through the same interfaces. We use $\mathcal{S}_{\text{NIZK}}^{R_r}$ and $\mathcal{S}_{\text{NIZK}}^{R_w}$ to reply the corresponding functionality when the functionality requests algorithms.

In Section A.1.1, we analyse the security of construction Π_{NICD} when (a subset of) verifiers $\mathcal{V}$ is corrupt. In Section A.1.2, we analyse the security of construction Π_{NICD} when the prover $\mathcal{P}$ and (a subset of) verifiers $\mathcal{V}$ are corrupt.

A.1.1 *Security Analysis when (a subset of) Verifiers is Corrupt*

We first describe the simulator $\mathcal{S}$ for the case in which (a subset of) verifiers is corrupt.

Figure A.2: Security Analysis of Π_{NICD} when (a subset of) Verifiers is Corrupt

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$. On input $(\mathsf{crs.get.ini}, sid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ on that input. When the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ sends $(\mathsf{crs.get.sim}, sid, qid, par)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES par. On input $(\mathsf{crs.get.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ on that input. When the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ sends $(\mathsf{crs.get.end}, sid, par)$, $\mathcal{S}$ sends $(\mathsf{crs.get.end}, sid, par)$ to $\mathcal{A}$.

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{NIZK}}^{R_r}$. On input (nizk.setup.ini, sid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NIZK}}^{R_r}$ on that input.

- When the copy of $\mathcal{F}_{\text{NIZK}}^{R_r}$ sends (nizk.setup.req, sid, qid, $\mathcal{P}$), $\mathcal{S}$ runs a copy of $\mathcal{S}_{\text{NIZK}}^{R_r}$ on input that message, and when $\mathcal{S}_{\text{NIZK}}^{R_r}$ sends the message (nizk.setup.alg, sid, qid, par_r, td_r, NIZK.SimProve$_r$, NIZK.Extract$_r$), $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_r}$ on input that message.
- When the copy of $\mathcal{F}_{\text{NIZK}}^{R_r}$ sends (nizk.setup.sim, sid, qid, $\mathcal{P}$), $\mathcal{S}$ sends that message to $\mathcal{A}$, and when $\mathcal{A}$ sends (nizk.setup.rep, sid, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_r}$ on input that message.

Finally, when $\mathcal{F}_{\text{NIZK}}^{R_r}$ sends (nizk.setup.end, sid, par_r), $\mathcal{S}$ sends $\mathcal{F}_{\text{NIZK}}^{R_r}$ to $\mathcal{A}$.

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{NIZK}}^{R_w}$. In this case $\mathcal{S}$ proceeds as in the case for $\mathcal{F}_{\text{NIZK}}^{R_r}$, but replacing $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{S}_{\text{NIZK}}^{R_r}$ by $\mathcal{F}_{\text{NIZK}}^{R_w}$ and $\mathcal{S}_{\text{NIZK}}^{R_w}$ respectively.

$\mathcal{S}$ SETS UP $\mathcal{F}_{\text{NICD}}$. When $\mathcal{A}$ has received par, par_r and par_w, $\mathcal{S}$ sends (nicd.setup.ini, sid) to $\mathcal{F}_{\text{NICD}}$.

- If $\mathcal{F}_{\text{NICD}}$ was set up before, $\mathcal{F}_{\text{NICD}}$ sends (nicd.setup.sim, sid, qid, $\mathcal{T}$) to $\mathcal{S}$ and $\mathcal{S}$ sends (nicd.setup.rep, sid, qid) to $\mathcal{F}_{\text{NICD}}$.
- If $\mathcal{F}_{\text{NICD}}$ was not set up before, $\mathcal{F}_{\text{NICD}}$ sends (nicd.setup.req, sid, qid, $\mathcal{T}$) to $\mathcal{S}$ and $\mathcal{S}$ sends (nicd.setup.alg, sid, qid, par, td, NICD.SimProve, NICD.Extract) to $\mathcal{F}_{\text{NICD}}$. The values in this last message are set as follows:
  - $par \leftarrow (par, par_r)$.
  - $td \leftarrow td_r$.
  - NICD.SimProve $\leftarrow$ NIZK.SimProve$'$.
  - NICD.Extract $\leftarrow$ NIZK.Extract$'$.

  NIZK.SimProve$'$ and NIZK.Extract$'$ are algorithms that execute NIZK.SimProve$_r$ and NIZK.Extract$_r$, but before doing so, they parse par to give as input par_r as parameters, and they add par to the instance given as input.

HONEST PARTY RUNS nicd.setup INTERFACE. ($\mathcal{F}_{\text{NICD}}$ already set up) On input from $\mathcal{F}_{\text{NICD}}$ the message (nicd.setup.sim, sid, qid, $\mathcal{T}$), $\mathcal{S}$ does the following:

- $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\text{VC.Setup}}$ on input (crs.get.ini, sid). When the copy of $\mathcal{F}_{\text{CRS}}^{\text{VC.Setup}}$ sends (crs.get.sim, sid, qid, par), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

- On input (crs.get.rep, sid, qid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\text{VC.Setup}}$ on that input.
- $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NIZK}}^{R_r}$ on input (nizk.setup.ini, sid). When the copy of $\mathcal{F}_{\text{NIZK}}^{R_r}$ sends (nizk.setup.sim, sid, qid, $\mathcal{P}$), $\mathcal{S}$ sends that message to $\mathcal{A}$.
- When $\mathcal{A}$ sends (nizk.setup.rep, sid, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_r}$ on input that message.
- $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NIZK}}^{R_w}$ on input (nizk.setup.ini, sid). When the copy of $\mathcal{F}_{\text{NIZK}}^{R_w}$ sends (nizk.setup.sim, sid, qid, $\mathcal{P}$), $\mathcal{S}$ sends that message to $\mathcal{A}$.
- When $\mathcal{A}$ sends (nizk.setup.rep, sid, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_w}$ on input that message.

After $\mathcal{A}$ has sent the three messages described above, $\mathcal{S}$ sends (nicd.setup.rep, sid, qid) to $\mathcal{F}_{\text{NICD}}$.

HONEST PARTY RUNS nicd.setup INTERFACE. ($\mathcal{F}_{\text{NICD}}$ not set up) On input from $\mathcal{F}_{\text{NICD}}$ the message (nicd.setup.req, sid, qid, $\mathcal{T}$) , $\mathcal{S}$ proceeds as above for the case in which $\mathcal{F}_{\text{NICD}}$ was already set up to communicate with $\mathcal{A}$. After that, $\mathcal{S}$ replies (nicd.setup.alg, sid, qid, par, td, NICD.SimProve, NICD.Extract) to $\mathcal{F}_{\text{NICD}}$, where the values of this message are set as described above for the case in which the setup is triggered by $\mathcal{A}$.

HONEST PROVER REGISTERS WRITE OPERATION. On input the message (nicd.write.sim, sid, ct, com, com_2) from $\mathcal{F}_{\text{NICD}}$, $\mathcal{S}$ proceeds as follows:

- Parse com as (com', $parcom$, COM.Verify).
- Parse com_2 as (com'_2, $parcom_2$, COM.Verify_2).
- If there is no tuple $(0, vc, \mathbf{x}, r)$ stored, $\mathcal{S}$ initializes a counter $ct \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = \perp$ for $i = 1$ to N_{max}. $\mathcal{S}$ sets $r \leftarrow 0$ and runs $vc \leftarrow \text{VC.Commit}(par, \mathbf{x}, r)$. (The symbol $\perp$ represents that no value is committed at a specific position and its actual value depends on the concrete implementation of the VC scheme.) $\mathcal{S}$ stores $(ct, vc, \mathbf{x}, r)$.
- Take the tuple $(ct - 1, vc, \mathbf{x}, r)$.
- Pick random vector $\mathbf{x}'$ and random r' and compute $vc' \leftarrow \text{VC.Commit}(par, \mathbf{x}', r')$.
- Set $ins_w \leftarrow (par, vc, vc', parcom, com', com'_2, ct)$.
- Run $\pi_w \leftarrow \text{NIZK.SimProve}_w(sid, par_w, td_w, ins_w)$.
- Append $[ins_w, \pi_w, 1]$ to Table Tbl of the copy of $\mathcal{F}_{\text{NIZK}}^{R_w}$.

$\mathcal{S}$ sets $sid_{\text{REG}} \leftarrow (sid, ct)$, sends (reg.register.ini, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$) to $\mathcal{F}_{\text{REG}}$. When $\mathcal{F}_{\text{REG}}$ sends (reg.register.sim, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$), $\mathcal{S}$ forwards it to $\mathcal{A}$.

$\mathcal{A}$ ALLOWS HONEST PROVER TO FINALIZE WRITE OPERATION. When $\mathcal{A}$ sends (reg.register.rep, sid_{REG}), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on that input. When $\mathcal{F}_{\text{REG}}$ outputs the message (reg.register.end, sid_{REG}), $\mathcal{S}$ stores $(ct, vc', \mathbf{x}', r')$ and sends (nicd.write.rep, sid, ct, vc') to $\mathcal{F}_{\text{NICD}}$.

HONEST VERIFIER INITIATES WRITE OPERATION RETRIEVAL. On input the message (nicd.writevf.sim, sid, qid, ct) from $\mathcal{F}_{\text{NICD}}$, $\mathcal{S}$ sets $sid_{\text{REG}} \leftarrow (sid, ct)$ and runs $\mathcal{F}_{\text{REG}}$ on input (reg.retrieve.ini, sid_{REG}). When $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.sim, sid_{REG}, qid, $\langle vc', com, com_2, \pi_w \rangle$), the functionality $\mathcal{F}_{\text{REG}}$ sends that message to $\mathcal{A}$.

HONEST VERIFIER CONCLUDES WRITE OPERATION RETRIEVAL. On input (reg.retrieve.rep, sid_{REG}, qid) from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on that input. When $\mathcal{F}_{\text{REG}}$ outputs (reg.retrieve.end, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$), $\mathcal{S}$ sends (nicd.writevf.rep, sid, qid) to $\mathcal{F}_{\text{NICD}}$.

$\mathcal{A}$ RETRIEVES WRITE OPERATION. On input (reg.retrieve.ini, sid_{REG}) from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on that input. When $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.sim, sid_{REG}, qid, v'), $\mathcal{S}$ forwards that message to $\mathcal{A}$. When $\mathcal{A}$ sends (reg.retrieve.rep, sid_{REG}, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on that input. When $\mathcal{F}_{\text{REG}}$ outputs (reg.retrieve.end, sid, v'), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ VERIFIES WRITE PROOF. On input (nizk.verify.ini, sid, ins_w, π_w) from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_w}$ on input that input. When $\mathcal{F}_{\text{NIZK}}^{R_w}$ sends (nizk.verify.end, sid, v), $\mathcal{S}$ sends that message to $\mathcal{A}$.

$\mathcal{A}$ VERIFIES READ PROOF. On input (nizk.verify.ini, sid, ins_r, π) from $\mathcal{A}$, $\mathcal{S}$ does the following:

- Parse ins_r as $(par, vc, parcom, com', com'_1, ct)$.
- Set $com \leftarrow (com', parcom, \mathsf{COM.Verify})$ and $com_1 \leftarrow (com'_1, parcom_1, \mathsf{COM.Verify}_1)$.
- Send the message (nicd.readvf.ini, sid, com, com_1, ct, π) to $\mathcal{F}_{\text{NICD}}$ and receive the message (nicd.readvf.end, sid, v) from $\mathcal{F}_{\text{NICD}}$.

$\mathcal{S}$ sends (nizk.verify.end, sid, v) to $\mathcal{A}$.

Theorem A.1.2 *When (a subset of) verifiers $\mathcal{V}$ is corrupt, Π_{NICD} securely realizes $\mathcal{F}_{\text{NICD}}$ in the $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$-hybrid model if the vector commitment scheme is hiding.*

PROOF OF THEOREM A.1.2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\text{NICD},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{NICD}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote

by Pr [**Game** i] the probability that the environment distinguishes **Game** i from the real-world protocol.

Figure A.4: Proof of Theorem A.1.2

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that Pr [**Game** 0] $= 0$.

GAME 1: This game proceeds as **Game** 0, except that in **Game** 1 the write proof computed by $\mathcal{F}_{\text{NIZK}}^{R_w}$ is replaced by a proof computed without giving the witness wit_w as input to $\mathcal{F}_{\text{NIZK}}^{R_w}$. Otherwise, the computation of the proof does not change. This change does not alter the view of the environment. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$.

GAME 2: **Game** 2 follows **Game** 1, except that **Game** 2 replaces the vector commitments computed by the honest prover by vector commitments to random vectors.

The probability that the environment can distinguish **Game** 2 from **Game** 1 is bound by the following claim.

Theorem A.1.3 *Let M be the number of write proofs sent by the honest prover. Under the hiding property of the vector commitment scheme, $|Pr[\textbf{Game } 2] - Pr[\textbf{Game } 1]| \leq M \cdot Adv_{\mathcal{A}}^{\mathsf{hid-vc}}$.*

The proof of this theorem is similar to the proof of Theorem **??**.

The distribution of **Game** 2 is identical to our simulation. This concludes the proof of Theorem A.1.2.

A.1.2 *Security Analysis when the Prover and some Verifiers are Corrupt*

We now describe the simulator $\mathcal{S}$ for the case in which the prover and (a subset of) verifiers are corrupt.

Figure A.6: Security Analysis of Π_{NICD} when the Prover and some Verifiers are Corrupt

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ RECEIVES par. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{NIZK}}^{R_r}$. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ QUERIES $\mathcal{F}_{\text{NIZK}}^{R_w}$. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{S}$ SETS UP $\mathcal{F}_{\text{NICD}}$. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

HONEST PARTY RUNS nicd.setup INTERFACE. ($\mathcal{F}_{\text{NICD}}$ already set up) The simulator $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

HONEST PARTY RUNS nicd.setup INTERFACE. ($\mathcal{F}_{\text{NICD}}$ already set up) The simulator $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ COMPUTES WRITE PROOF. On input (nizk.prove.ini, sid, wit_w, ins_w) from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_w}$ on input (nizk.prove.ini, sid, wit_w, ins_w). When $\mathcal{F}_{\text{NIZK}}^{R_w}$ sends (nizk.prove.end, sid, π_w), $\mathcal{S}$ forwards that message to $\mathcal{A}$. (If $\mathcal{F}_{\text{NIZK}}^{R_w}$ aborts, $\mathcal{S}$ forwards the abortion message to $\mathcal{A}$.)

$\mathcal{A}$ REGISTERS WRITE OPERATION. On input (reg.register.ini, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$) from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on that input. When $\mathcal{F}_{\text{REG}}$ sends (reg.register.sim, sid_{REG}, $\langle vc', com, com_2, \pi_w \rangle$), $\mathcal{S}$ forwards that message to $\mathcal{A}$. When $\mathcal{A}$ sends (reg.register.rep, sid_{REG}), $\mathcal{S}$ proceeds as follows:

- If there is no tuple $(0, vc, \mathbf{x}, r)$ stored, $\mathcal{S}$ initializes a counter $ct \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = \perp$ for $i = 1$ to N_{max}. $\mathcal{S}$ sets $r \leftarrow 0$ and runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$. (The symbol $\perp$ represents that no value is committed at a specific position and its actual value depends on the concrete implementation of the VC scheme.) $\mathcal{S}$ stores $(ct, vc, \mathbf{x}, r)$.
- $\mathcal{S}$ takes the tuple $(ct-1, vc, \mathbf{x}, r)$. (If this tuple does not exist, $\mathcal{S}$ sends (reg.register.end, sid_{REG}) to $\mathcal{A}$ and the operations described below will be carried out once the tuple is stored.)
- $\mathcal{S}$ sets $ins_w \leftarrow (par, vc, vc', parcom, com', com'_2, ct)$.
- $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_w}$ on input (nizk.verify.ini, sid, ins_w, π_w) and receives (nizk.verify.end, sid, v).
- If $v = 1$, $\mathcal{S}$ parses the witness wit_w as $(w, i, open, v_1, v_2, open_2, r, r')$. (The witness is either extracted by $\mathcal{F}_{\text{NIZK}}^{R_w}$, or was previously received by $\mathcal{S}$ from $\mathcal{A}$ when $\mathcal{A}$ computed a write proof.) If $\mathbf{x}[i] \neq v_1$, i.e. if the value $\mathbf{x}[i]$ previously stored by the simulator is not equal to the value v_1 in the witness, $\mathcal{S}$ outputs failure. Otherwise $\mathcal{S}$ sends (nicd.write.ini, sid, com, i, $open$, com_2, v_2, $open_2$) to $\mathcal{F}_{\text{NICD}}$. When $\mathcal{F}_{\text{NICD}}$ sends (nicd.write.sim, sid, ct', com, com_2), $\mathcal{S}$ sends (nicd.write.rep, sid, ct, vc') to $\mathcal{F}_{\text{NICD}}$ and receives (nicd.write.end, sid, ct) from $\mathcal{F}_{\text{NICD}}$. $\mathcal{S}$ sets $\mathbf{x}' \leftarrow \mathbf{x}$, sets $\mathbf{x}'[i] \leftarrow v_2$, and stores $(ct, vc', \mathbf{x}', r')$.

$\mathcal{S}$ sends (reg.register.end, sid_{REG}) to $\mathcal{A}$.

HONEST VERIFIER INITIATES WRITE OPERATION RETRIEVAL. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

HONEST VERIFIER CONCLUDES WRITE OPERATION RETRIEVAL. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ RETRIEVES WRITE OPERATION. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ COMPUTES READ PROOF. When $\mathcal{A}$ sends $(\mathsf{nizk.prove.ini}, sid, wit_r, ins_r)$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NIZK}}^{R_r}$ on input that message. If $\mathcal{F}_{\text{NIZK}}^{R_r}$ sends $(\mathsf{nizk.prove.end}, sid, \pi)$, i.e., if $\mathcal{F}_{\text{NIZK}}^{R_r}$ does not abort, then $\mathcal{S}$ does the following:

- $\mathcal{S}$ takes the tuple $(ct, vc, \mathbf{x}, r)$. If the tuple is not stored, or if vc does not equal the value in ins_r, $\mathcal{S}$ sends $(\mathsf{nizk.prove.end}, sid, \pi)$ to $\mathcal{A}$ and performs the operations described below when such a tuple is stored.
- $\mathcal{S}$ parses wit_r as $(w, i, open, v_1, open_1)$.
- If $\mathbf{x}[i] \neq v_1$, $\mathcal{S}$ outputs failure. Otherwise $\mathcal{S}$ sends $(\mathsf{nicd.read.ini}, sid, ct, com, i, open, com_1, v_1, open_1)$ to $\mathcal{F}_{\text{NICD}}$ and receives $(\mathsf{nicd.read.end}, sid, \pi)$. $\mathcal{S}$ sends $(\mathsf{nizk.prove.end}, sid, \pi)$ to $\mathcal{A}$.

$\mathcal{A}$ VERIFIES WRITE PROOF. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

$\mathcal{A}$ VERIFIES READ PROOF. $\mathcal{S}$ operates as in the case where only (a subset of) the verifiers are corrupt.

Theorem A.1.4 *When the prover $\mathcal{P}$ and (a subset of) verifiers $\mathcal{V}$ are corrupt, the construction Π_{NICD} securely realizes $\mathcal{F}_{\text{NICD}}$ in the $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NIZK}}^{R_r}$ and $\mathcal{F}_{\text{NIZK}}^{R_w}$-hybrid model if the vector commitment scheme is binding.*

PROOF OF THEOREM A.1.4. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\text{NICD},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{NICD}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Figure A.8: Proof of Theorem A.1.4

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that $\Pr[\textbf{Game } 0] = 0$.

GAME 1: **Game** 1 follows **Game** 0, except that **Game** 1 outputs failure when the adversary sends a witness-instance pair to compute a write or a read proof that fulfil the corresponding relation R_r or R_w, but such that, for the position i in the witness, the value v_1 in the witness

is not equal to the value $\mathbf{x}[i]$ that the simulator had recorded as stored in the position i.

Theorem A.1.5 *Under the binding property of the vector commitment scheme, we have that* $|Pr\,[\textbf{\textit{Game}}\ 1] - Pr\,[\textbf{\textit{Game}}\ 0]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-vc}}$.

The proof of this theorem is similar to the proof of Theorem **??**.

The distribution of **Game** 1 is identical to our simulation. This concludes the proof of Theorem A.1.4.

A.2 SECURITY ANALYSIS OF OUR CONSTRUCTION FOR UUHD

Theorem A.2.1 Π_{UUHD} *securely realizes* $\mathcal{F}_{\mathrm{UUHD}}$ *in the* $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{NYM}}$ *and* $\mathcal{F}_{\mathrm{ZK}}^{R_r}$*-hybrid model if the VC scheme is hiding and binding, the commitment scheme is hiding and binding, and the signature scheme is existentially unforgeable.*

To prove that Π_{UUHD} securely realizes $\mathcal{F}_{\mathrm{UUHD}}$, we must show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{UUHD}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{UUHD}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_r}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to $\mathcal{A}$ if the functionality sends the abortion message to a corrupt party.

In Section A.2.1, we analyse the security of Π_{UUHD} when (a subset of) readers $\mathcal{R}_k$ are corrupt. In Section A.2.2, we analyse the security of Π_{UUHD} when the updater $\mathcal{U}$ is corrupt. We do not analyse in detail the security of Π_{UUHD} when $\mathcal{U}$ and (a subset of) readers $\mathcal{R}_k$ are corrupt. We note that, in Π_{UUHD}, honest readers communicate with $\mathcal{U}$ but not with other readers. Therefore, for this case the simulator and the security proof are very similar to the case where only $\mathcal{U}$ is corrupt.

The use of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ to generate the parameters guarantees that they are generated honestly and that $\mathcal{R}_k$ and $\mathcal{U}$ do not know any trapdoor. When $\mathcal{R}_k$ is corrupt, the binding property of the VC scheme guarantees that $\mathcal{R}_k$ is not able to open a VC vc to a value vr_i at position i if $vr_i \neq \mathbf{x}[i]$, where $\mathbf{x}$ is the vector committed by the honest $\mathcal{U}$ in vc. The binding property of the commitment scheme guarantees that the commitment com cannot be opened to a random value s different from the one committed, which ensures that $\mathcal{R}_k$ cannot open the same commitment twice without being detected. The unforgeability property of the signature scheme ensures that $\mathcal{R}_k$ can only use vc and com that were signed by $\mathcal{U}$.

When $\mathcal{U}$ is corrupt, the hiding property of the VC scheme guarantees that the database in vc remains hidden from $\mathcal{U}$. The use of $\mathcal{F}_{\mathrm{NYM}}$ guarantees unlinkability between read operations, so that $\mathcal{U}$ cannot keep track of the updates for a particular reader. The hiding property of the commitment scheme hides the random value s

committed in com during the update phase, which is needed to provide unlinkability between the update phase and the next read phase. The use of $\mathcal{F}_{\text{REG}}$ guarantees that $\mathcal{U}$ does not set up a different public key for each reader, which would also break unlinkability.

A.2.1 *Security Analysis when the Readers are Corrupt*

We describe $\mathcal{S}$ for the case in which (a subset of) $\mathcal{R}_k$ are corrupt.

Figure A.11: Security Analysis of Π_{UUHD} when the Readers are Corrupt

$\mathcal{A}$ REQUESTS PARAMETERS. On input (crs.get.ini, sid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ on that input. When the copy of $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ sends (crs.get.sim, sid, qid, $\langle par, par_c \rangle$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES PARAMETERS. On input (crs.get.rep, sid, qid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ on that input. When the copy of $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ sends (crs.get.end, sid, $\langle par, par_c \rangle$), $\mathcal{S}$ sends (crs.get.end, sid, $\langle par, par_c \rangle$) to $\mathcal{A}$.

$\mathcal{A}$ SENDS A ZK PROOF. On input the message (zk.prove.ini, sid, wit_r, ins_r, P) from $\mathcal{A}$, $\mathcal{S}$ stores (wit_r, ins_r, P) and runs $\mathcal{F}_{\text{ZK}}^{R_r}$ on that input. When $\mathcal{F}_{\text{ZK}}^{R_r}$ sends the message (zk.prove.sim, sid, qid, ins_r), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

HONEST $\mathcal{U}$ RECEIVES ZK PROOF. On input (zk.prove.rep, sid, qid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{ZK}}^{R_r}$ on that input. When the copy of $\mathcal{F}_{\text{ZK}}^{R_r}$ sends (zk.prove.end, sid, ins_r, P), $\mathcal{S}$ parses the stored wit_r as (sig, vc, com, r_2, $open_2$, $\langle i, vr_i, w_i, o_i, or_i \rangle_{i \in \mathbb{S}}$) and does the following:

- If the pseudonym P was received before, $\mathcal{S}$ aborts. (We recall that the honest updater also aborts if a pseudonym is reused.)
- Else, if $\mathcal{A}$ did not receive a signature sig on (vc, com), $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ finds the stored tuple (vc, $\mathbf{x}$, r, com, s, $open$) that contains vc and com. If, for any $i \in \mathbb{S}$, $\mathbf{x}[i] \neq vr_i$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ takes com' from ins_r and runs $\mathcal{F}_{\text{NYM}}$ on input (nym.reply.ini, sid, (Open com'), P). When $\mathcal{F}_{\text{NYM}}$ sends (nym.reply.sim, sid, qid, l(Open com')), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES (Open com') MESSAGE. When $\mathcal{A}$ sends (nym.reply.rep, sid, qid), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends (nym.send.end, sid, (Open com'), P), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ SENDS COMMITMENT (AND OPENING). On input $(\mathsf{nym.send.ini}, sid, m, P)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

HONEST $\mathcal{U}$ RECEIVES COMMITMENT (AND OPENING). On input from $\mathcal{A}$ the message $(\mathsf{nym.send.rep}, sid, qid)$, the simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.end}, sid, m, P)$, $\mathcal{S}$ parses m as either com_1 or $(com_1, s, open')$ and proceeds as follows.

- If $m = com_1$, $\mathcal{S}$ aborts if P was already received, else stores $(wit_r \leftarrow \perp, ins_r \leftarrow \perp, P)$ and sends $(\mathsf{uuhd.read.ini}, sid, P, \perp)$ to $\mathcal{F}_{\text{UUHD}}$.
- If $m = (com_1, s, open')$, the simulator $\mathcal{S}$ finds the stored tuple (wit_r, ins_r, P) with the same pseudonym previously received when $\mathcal{A}$ sends a ZK proof. If no such tuple exists, $\mathcal{S}$ aborts. Otherwise $\mathcal{S}$ parses ins_r as $(pk, par, par_c, vc', com', parcom, \langle c_i', cr_i' \rangle_{i \in \mathbb{S}})$ and aborts if $(s, open')$ is not a valid opening for com'. If the opening is valid, $\mathcal{S}$ parses wit_r as $(sig, vc, com, r_2, open_2, \langle i, vr_i, w_i, o_i, or_i \rangle_{i \in \mathbb{S}})$ and finds the stored tuple $(vc, \mathbf{x}, r, com, s', open)$ that contains vc and com.
  - If $s = s'$, $\mathcal{S}$ aborts (in this case, $\mathcal{A}$ double-spent a VC).
  - If $s \neq s'$, $\mathcal{S}$ outputs failure.
  - If $s' = \perp$, $\mathcal{S}$ stores $(s, open' - open_2)$ in that tuple and sends $(\mathsf{uuhd.read.ini}, sid, P, (i, vr_i, c_i, o_i, cr_i, or_i)_{i \in \mathbb{S}})$ to $\mathcal{F}_{\text{UUHD}}$. When $\mathcal{F}_{\text{UUHD}}$ sends the message $(\mathsf{uuhd.read.sim}, sid, qid, (c_i, cr_i)_{i \in \mathbb{S}})$, $\mathcal{S}$ sends $(\mathsf{uuhd.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUHD}}$.

HONEST $\mathcal{U}$ SENDS UPDATE. When $\mathcal{F}_{\text{UUHD}}$ sends the message $(\mathsf{uuhd.update.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends $(\mathsf{uuhd.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUHD}}$. When $\mathcal{F}_{\text{UUHD}}$ sends $(\mathsf{uuhd.update.end}, sid, P, (i, vu_i)_{i \in [1,N]})$, $\mathcal{S}$ sets $\mathbf{x}_u[i] \leftarrow vu_i$ (for all $i \in [1, N]$) and picks a random value s_2. $\mathcal{S}$ picks the stored tuple (wit_r, ins_r, P) and the tuple $(vc, \mathbf{x}, r, com, s, open)$ that contains vc and com in wit_r. $\mathcal{S}$ follows Π_{UUHD} to compute the new values of vc, $\mathbf{x}$, and com and updates the tuple $(vc, \mathbf{x}, r, com, s, open)$ accordingly (r is updated by using wit_r). If a signing key pair (pk, sk) is not stored, $\mathcal{S}$ computes and stores (pk, sk) and registers pk with the copy of $\mathcal{F}_{\text{REG}}$. $\mathcal{S}$ computes a signature sig on vc and com and runs $\mathcal{F}_{\text{NYM}}$ on input $(\mathsf{nym.reply.ini}, sid, \langle \mathbf{x}_u, s_2, sig \rangle, P)$. When $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(\langle \mathbf{x}_u, s_2, sig \rangle))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES UPDATE. When $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NYM}}$ on that input. When $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.reply.end}, sid, \langle \mathbf{x}_u, s_2, sig \rangle, P)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ REQUESTS PUBLIC KEY. On input $(\mathsf{reg.retrieve.ini}, sid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{REG}}$ on that input. When the copy of $\mathcal{F}_{\text{REG}}$ sends $(\mathsf{reg.retrieve.sim}, sid, qid, pk)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES PUBLIC KEY. On input $(\mathsf{reg.retrieve.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{REG}}$ on that input. When the copy of $\mathcal{F}_{\text{REG}}$ sends $(\mathsf{reg.retrieve.end}, sid, pk)$, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, sid, pk)$ to $\mathcal{A}$.

Theorem A.2.2 *When (a subset of) $\mathcal{R}_k$ are corrupt, Π_{UUHD} securely realizes $\mathcal{F}_{\text{UUHD}}$ in the $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{NYM}}$ and $\mathcal{F}_{\text{ZK}}^{R_r}$-hybrid model if the VC scheme is binding, the commitment scheme is binding, and the signature scheme is existentially unforgeable.*

PROOF OF THEOREM A.2.2. We show by means of a series of hybrid games that $\mathcal{Z}$ cannot distinguish between the real-world protocol and our simulation with non-negligible probability. Pr [**Game** i] is the probability that $\mathcal{Z}$ distinguishes **Game** i from the real-world protocol.

Figure A.13: Proof of Theorem A.2.2

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that Pr [**Game** 0] $= 0$.

GAME 1: **Game** 1 follows **Game** 0, except that **Game** 1 outputs failure when $\mathcal{A}$ sends a signature sig on vc and com and $\mathcal{A}$ did not receive a signature on those values before.

The probability that **Game** 1 outputs failure is bound by theorem A.2.3.

Theorem A.2.3 *Under the existential unforgeability property of the signature scheme, we have that $|Pr[\textbf{Game } 1] - Pr[\textbf{Game } 0]| \leq Adv_{\mathcal{A}}^{\mathsf{unf-sig}}$.*

We omit a formal proof of theorem A.2.3. In a nutshell, we can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 1 output failure with non-negligible probability, wins the existential unforgeability game with that probability. B receives the public key from the challenger and registers it with $\mathcal{F}_{\text{REG}}$. To compute signatures on vc and com, B uses the signing oracle. When $\mathcal{A}$ sends a signature sig on vc and com as part of the witness wit_r sent to $\mathcal{F}_{\text{ZK}}^{R_r}$ such that B did not compute before a signature on those values, B submits sig along with (vc, com) in order to win the existential unforgeability game.

GAME 2: **Game** 2 follows **Game** 1, except that **Game** 2 outputs failure when $\mathcal{A}$ sends a witness $(sig, vc, com, r_2, open_2, \langle i, vr_i, w_i, o_i, or_i\rangle_{i\in\mathbb{S}})$ and an instance $(pk, par, par_c, vc', com', parcom, \langle c'_i, cr'_i\rangle_{i\in\mathbb{S}})$ such that w_i opens the commitment vc' to a value $vr_i \neq \mathbf{x}[i]$ at position i, where $\mathbf{x}$ is the vector committed in vc (vc' is a rerandomisation of vc).

The probability that **Game** 2 outputs failure is bound by theorem A.2.4.

Theorem A.2.4 *Under the binding property of the VC scheme, we have that* $|Pr\,[\textbf{\textit{Game}}\ 2] - Pr\,[\textbf{\textit{Game}}\ 1]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-vc}}$.

We omit a formal proof of theorem A.2.4. We can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 2 output failure with non-negligible probability, wins the binding game of the VC scheme with that probability. B receives the parameters of the VC scheme from the challenger and stores them in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. When $\mathcal{A}$ sends an opening w_i that opens vc' to vr_i as described above, B computes another opening w'_i that opens vc' to $\mathbf{x}[i]$. We note that wit_r contains r_2, so B knows the randomness change of vc' with respect to vc and is able to compute w'_i. B sends vc' along with i, w_i, vr_i, w'_i, and $\mathbf{x}[i]$ to win the binding game of the VC scheme.

GAME 3: **Game** 3 follows **Game** 2, except that **Game** 3 outputs failure when $\mathcal{A}$ sends an opening $(s, open')$ for a commitment com' such that a previous opening $(\hat{s}, \hat{open'})$ was received for a commitment $\hat{com'}$ and both com' and $\hat{com'}$ are rerandomisations of com.

The probability that **Game** 3 outputs failure is bound by theorem A.2.5.

Theorem A.2.5 *Under the binding property of the commitment scheme, we have the following:* $|Pr\,[\textbf{\textit{Game}}\ 3] - Pr\,[\textbf{\textit{Game}}\ 2]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-com}}$.

We omit a formal proof of theorem A.2.5. We can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 3 output failure with non-negligible probability, wins the binding game of the commitment scheme with that probability. B receives the parameters of the commitment scheme from the challenger and stores them in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. When $\mathcal{A}$ sends an opening $(s, open')$ for a commitment com' such that a previous opening $(\hat{s}, \hat{open'})$ was received for a commitment $\hat{com'}$ and both com' and $\hat{com'}$ are both rerandomisations of a commitment com, B computes $(s, open' - open_2)$ and $(\hat{s}, \hat{open'} - \hat{open}_2)$. The values $open_2$ and $\hat{open}_2$ are the randomness used to rerandomize com into com' and $\hat{com'}$ respectively, which B obtains through the witnesses wit_r sent to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$. B sends com along with $(s, open' - open_2)$ and $(\hat{s}, \hat{open'} - \hat{open}_2)$ to win the binding game of the commitment scheme.

The distribution of **Game** 3 is identical to our simulation. This concludes the proof of Theorem A.2.2.

A.2.2 *Security Analysis when the Updater is Corrupt*

We describe $\mathcal{S}$ for the case in which $\mathcal{U}$ is corrupt.

Figure A.19: Security Analysis of Π_{UUHD} when the Updater is Corrupt

$\mathcal{A}$ REQUESTS OR RECEIVES PARAMETERS. $\mathcal{S}$ runs as in the case where (a subset of) $\mathcal{R}_k$ are corrupt.

$\mathcal{A}$ REGISTERS PUBLIC KEY. On input $(\mathsf{reg.register.ini}, sid, pk)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{REG}}$ on that input. When the copy of $\mathcal{F}_{\text{REG}}$ sends $(\mathsf{reg.register.sim}, sid, pk)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ ENDS REGISTRATION OF PUBLIC KEY. On input $(\mathsf{reg.register.rep}, sid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{REG}}$ on that input. When the copy of $\mathcal{F}_{\text{REG}}$ sends $(\mathsf{reg.register.end}, sid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

HONEST $\mathcal{R}_k$ STARTS ZK PROOF. On input $(\mathsf{uuhd.read.sim}, sid, qid, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\text{UUHD}}$, $\mathcal{S}$ sends $(\mathsf{uuhd.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUHD}}$ and receives $(\mathsf{uuhd.read.end}, sid, P, \mathsf{flag}, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\text{UUHD}}$. If $\mathsf{flag} = 0$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ computes a VC vc' to a random vector and a commitment com' to a random value s with opening $open'$.
- Parse the commitment c_i as $(c_i', parcom, \mathsf{COM.Verify})$.
- Parse the commitment cr_i as $(cr_i', parcom, \mathsf{COM.Verify})$.
- $\mathcal{S}$ sets the instance as $ins_r \leftarrow (pk, par, par_c, vc', com', parcom, \langle c_i', cr_i'\rangle_{i\in\mathbb{S}})$ and stores (qid, ins_r, P) and $(sid, P, ins_r, s, open')$.
- $\mathcal{S}$ sends $(\mathsf{zk.prove.sim}, sid, qid, ins_r)$ to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES ZK PROOF. On input $(\mathsf{zk.prove.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ sends an abortion message to $\mathcal{A}$ if (qid, ins_r, P) is not stored.
- $\mathcal{S}$ deletes the record (qid, ins_r, P).
- $\mathcal{S}$ sends $(\mathsf{zk.prove.end}, sid, ins_r, P)$ to $\mathcal{A}$.

$\mathcal{A}$ REQUESTS OPENING. On input $(\mathsf{nym.reply.ini}, sid, (\text{Open } com'), P)$ from $\mathcal{A}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NYM}}$ on input $(\mathsf{nym.reply.ini}, sid, (\text{Open } com'), P)$. When $\mathcal{F}_{\text{NYM}}$ sends the message $(\mathsf{nym.reply.sim}, sid, qid, l(\text{Open } com'))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

HONEST $\mathcal{R}_k$ RECEIVES REQUEST. On input $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.end}, sid, (\text{Open } com'), P)$, $\mathcal{S}$ continues with the case "Honest $\mathcal{R}_k$ sends commitment (and opening)".

HONEST $\mathcal{R}_k$ SENDS COMMITMENT (AND OPENING). On input the message $(\mathsf{uuhd.read.sim}, sid, qid, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\text{UUHD}}$, $\mathcal{S}$ sends $(\mathsf{uuhd.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUHD}}$ and receives $(\mathsf{uuhd.read.end}, sid, P, \mathsf{flag}, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\text{UUHD}}$. $\mathcal{S}$ computes a commitment com_1 to a random value. If $\mathsf{flag} = 1$, $\mathcal{S}$ sets $m \leftarrow com_1$. Else, after following the case "Honest $\mathcal{R}_k$ starts ZK proof", $\mathcal{S}$ picks the stored tuple $(sid, P, ins_r, s, open')$ and sets $m \leftarrow \langle com_1, s, open'\rangle$. $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on input $(\mathsf{nym.send.ini}, sid, m, P)$. When $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ RECEIVES COMMITMENT (AND OPENING). On input $(\mathsf{nym.send.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.end}, sid, m, P)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ SENDS UPDATE. On input $(\mathsf{nym.reply.ini}, sid, m, P)$ from $\mathcal{F}_{\text{NYM}}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(m))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

HONEST $\mathcal{R}_k$ RECEIVES UPDATE. On input $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{NYM}}$ on that input. When the copy of $\mathcal{F}_{\text{NYM}}$ sends $(\mathsf{nym.send.end}, sid, m, P)$, $\mathcal{S}$ parses m as $(\mathbf{x}, s_2, sig)$, picks the stored tuple $(sid, P, ins_r, s, open')$ and follows the steps described in Π_{UUHD} to verify sig. Then $\mathcal{S}$ sends $(\mathsf{uuhd.update.ini}, sid, P, \mathbf{x})$ to $\mathcal{F}_{\text{UUHD}}$. When $\mathcal{F}_{\text{UUHD}}$ sends the message $(\mathsf{uuhd.update.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{uuhd.update.rep}, sid, qid)$ to the functionality $\mathcal{F}_{\text{UUHD}}$.

Theorem A.2.6 *When $\mathcal{U}$ is corrupt, Π_{UUHD} securely realizes $\mathcal{F}_{\text{UUHD}}$ in the $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\text{NYM}}$ and $\mathcal{F}_{\text{ZK}}^{R_r}$-hybrid model if the VC scheme is hiding and the commitment scheme is hiding.*

PROOF OF THEOREM A.2.6. We show by means of a series of hybrid games that $\mathcal{Z}$ cannot distinguish between the real-world protocol and our simulation with non-negligible probability.

Figure A.21: Proof of Theorem A.2.6

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that $\Pr[\textbf{Game } 0] = 0$.

GAME 1: **Game** 1 follows **Game** 0, except that **Game** 1 does not run a copy of $\mathcal{F}_{\text{ZK}}^{R_r}$. Instead, **Game** 1 sets the messages $(\mathsf{zk.prove.sim}, sid, qid, ins_r)$ and $(\mathsf{zk.prove.end}, sid, ins_r, P)$ directly. This change does not alter the view of $\mathcal{Z}$. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$

GAME 2: **Game** 2 follows **Game** 1, except that **Game** 2 replaces the VC vc' in ins_r by a commitment to a random vector.

The probability that **Game** 1 and **Game** 2 are distinguished by $\mathcal{Z}$ is bound by theorem A.2.7.

Theorem A.2.7 *Let M be the number of read operations sent by honest readers after their first read operation. Under the hiding property of the vector commitment scheme, $|Pr[\textbf{Game } 2] - Pr[\textbf{Game } 1]| \leq M \cdot Adv_{\mathcal{A}}^{\mathsf{hid-vc}}$.*

PROOF OF THEOREM A.2.7. We define a sequence of games. In **Game** $1.i$, for the last i read operations (after their first read operation) sent by honest readers, the vector commitment vc' is replaced by a vector commitment to a random vector. Therefore, **Game** 1.0 corresponds to **Game** 1, while **Game** $1.M$ corresponds to **Game** 2.

We construct an algorithm B that, given an $\mathcal{A}$ that distinguishes **Game** $1.i$ from **Game** $1.(i+1)$ with non-negligible probability, breaks the hiding property of the vector commitment scheme with non-negligible probability. B works as follows. When the challenger sends the parameters par, B stores par as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. B replaces the vector commitment vc' by a vector commitment to a random vector in the last $i+1$ read operations by the honest readers. For the read operation i, B sends to the challenger the vector $\mathbf{x}$ that should be committed in vc'. The challenger sends back a commitment vc'. As can be seen, if vc' is a commitment to a random vector, the situation corresponds to **Game** $1.(i+1)$, while otherwise the situation corresponds to **Game** $1.i$. Therefore, if $\mathcal{A}$ distinguishes **Game** $1.i$ from **Game** $1.(i+1)$ with non-negligible probability, B can use $\mathcal{A}$'s guess to break the hiding property of the VC scheme. This concludes the proof of Theorem A.2.7.

Figure A.23: Proof of Theorem A.2.7

GAME 3: **Game** 3 follows **Game** 2, except that **Game** 3 replaces the commitment com' in ins_r by a commitment to a random value. The probability that **Game** 2 and **Game** 3 are distinguished by $\mathcal{Z}$ is bound by theorem A.2.8.

Theorem A.2.8 *Let M be the number of read operations sent by honest readers. Under the hiding property of the commitment scheme, $|Pr[\textbf{Game } 3] - Pr[\textbf{Game } 2]| \leq M \cdot Adv_{\mathcal{A}}^{\mathsf{hid-com}}$.*

PROOF OF THEOREM A.2.8. We define a sequence of games. In **Game** $2.i$, for the last i read operations sent by honest readers, the commitment com' is replaced by a commitment to a random value. Therefore, **Game** 2.0 corresponds to **Game** 2, while **Game** $2.M$ corresponds to **Game** 3.

We construct an algorithm B that, given an $\mathcal{A}$ that distinguishes **Game** $2.i$ from **Game** $2.(i+1)$ with non-negligible probability, breaks the hiding property of the

commitment scheme with non-negligible probability. B works as follows. When the challenger sends the parameters par_c, B stores par_c as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. B replaces the commitment com' by a commitment to a random value in the last $i+1$ read operations by the honest readers. For the proof i, B sends to the challenger the value s that should be committed in com'. The challenger sends back a commitment com'. As can be seen, if com' is a commitment to a random value, the situation corresponds to **Game** $2.(i+1)$, while otherwise the situation corresponds to **Game** $2.i$. Therefore, if $\mathcal{A}$ distinguishes **Game** $2.i$ from **Game** $2.(i+1)$ with non-negligible probability, B can use the $\mathcal{A}$'s guess to break the hiding property of the commitment scheme. This concludes the proof of Theorem A.2.8.

The distribution of **Game** 3 is identical to our simulation. This concludes the proof of Theorem A.2.6.

A.3 SECURITY ANALYSIS OF OUR CONSTRUCTION FOR UD

Theorem A.3.1 Π_{UD} *securely realizes* $\mathcal{F}_{\text{UD}}$ *in the* $(\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{ZK}}^{R})$*-hybrid model if the NHVC scheme is binding.*

To prove that Π_{UD} securely realizes $\mathcal{F}_{\text{UD}}$, we must show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\text{UD}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\text{UD}}$ for all corrupt parties in the ideal world.

$\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{AUT}}$ and $\mathcal{F}_{\text{ZK}}^{R}$. In the descriptions of our simulators below, for brevity, we omit part of the communication between $\mathcal{S}$ and $\mathcal{A}$. Whenever a copy of those functionalities sends a message $(*.*.\mathsf{sim})$ to $\mathcal{S}$, $\mathcal{S}$ implicitly forwards that message to $\mathcal{A}$ and runs again a copy of that functionality on input the response provided by $\mathcal{A}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to $\mathcal{A}$ if the functionality sends the abortion message to a corrupt party.

In Section A.3.1, we analyse the security of Π_{UD} when $\mathcal{R}$ is corrupt. In Section A.3.2, we analyse the security of Π_{UD} when $\mathcal{U}$ is corrupt.

A.3.1 *Security Analysis when the Reader is Corrupt*

We describe the simulator $\mathcal{S}$ for the case in which $\mathcal{R}$ is corrupt.

Figure A.26: Security Analysis of Π_{UD} when the Reader is Corrupt

INITIALIZATION OF $\mathcal{S}$. $\mathcal{S}$ sets $cu \leftarrow 0$. $\mathcal{S}$ runs the $\mathsf{crs.get}$ interface of an instance of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ to get the parameters par.

HONEST $\mathcal{U}$ SENDS AN UPDATE. On input from functionality $\mathcal{F}_{\text{UD}}$ the message $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i \in [1,N]})$, $\mathcal{S}$ sends the message $(\mathsf{ud.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UD}}$. When $\mathcal{F}_{\text{UD}}$ sends

$(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i \in [1,N]})$, if $(sid, par, vc, \mathbf{x}, cu)$ is not stored, $\mathcal{S}$ does the following:

- $\mathcal{S}$ initializes a counter $cu \leftarrow 0$.
- $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for $i \in [1, N]$.
- $\mathcal{S}$ runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$ and stores $(sid, par, vc, \mathbf{x}, cu)$.

Else:

- $\mathcal{S}$ sets $cu' \leftarrow cu + 1$, sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $vc' \leftarrow vc$.
- For all $i \in [1, N]$ such that $vu_i \neq \perp$, the simulator $\mathcal{S}$ computes $\mathbf{x}'[i] \leftarrow vu_i$ and $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc', i, \mathbf{x}[i], vu_i)$.
- $\mathcal{S}$ replaces the stored tuple $(sid, par, vc, \mathbf{x}, cu)$ by $(sid, par, vc', \mathbf{x}', cu')$.

$\mathcal{S}$ uses the aut.send interface of $\mathcal{F}_{\mathrm{AUT}}$ to send $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$ to $\mathcal{A}$.

$\mathcal{A}$ REQUESTS AND RECEIVES par. When $\mathcal{A}$ invokes the crs.get interface, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ on that input to send par to $\mathcal{A}$.

$\mathcal{A}$ SENDS A PROOF. When $\mathcal{A}$ invokes the zk.prove interface on input the witness wit and the instance ins, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$ on that input. Then $\mathcal{S}$ parses ins as $(par', vc', parcom, c_i', cr_i', cr)$ and wit as $(w_i, i, o_i, vr_i, or_i)$. $\mathcal{S}$ sets the commitments $c_i \leftarrow (c_i', parcom, \mathsf{COM.Verify})$ and $cr_i \leftarrow (cr_i', parcom, \mathsf{COM.Verify})$. $\mathcal{S}$ sends the message $(\mathsf{ud.read.ini}, sid, i, vr_i, c_i, o_i, cr_i, or_i)$ to $\mathcal{F}_{\mathrm{UD}}$. When $\mathcal{F}_{\mathrm{UD}}$ sends the message $(\mathsf{ud.read.sim}, sid, qid, c_i, cr_i)$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, vc, \mathbf{x}, cu)$. If $cr \neq cu$, or if $par' \neq par$, or if $vc' \neq vc$, $\mathcal{S}$ sends $\mathcal{F}_{\mathrm{UD}}$ a message that makes $\mathcal{F}_{\mathrm{UD}}$ abort.
- Else, if $\mathbf{x}[i] \neq vr_i$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ sends $(\mathsf{ud.read.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{UD}}$.

Theorem A.3.2 *When $\mathcal{R}$ is corrupt, Π_{UD} securely realizes $\mathcal{F}_{\mathrm{UD}}$ in the $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^{R})$-hybrid model if the NHVC scheme is binding.*

PROOF OF THEOREM A.3.2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish the real-world protocol from the ideal-world protocol with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Figure A.28: Proof of Theorem A.3.2

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that $\Pr[\textbf{Game } 0] = 0$.

GAME 1: **Game** 1 follows **Game** 0, except that **Game** 1 runs an initialization phase to set a counter cu and the parameters par. **Game** 1 stores and updates a tuple $(sid, par, vc, \mathbf{x}, cu)$. These changes do not alter the view of the environment. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$.

GAME 2: **Game** 2 follows **Game** 1, except that, when the adversary sends a valid proof with witness wit and instance ins, **Game** 2 outputs failure if the values i and vr_i in the witness are such that $\mathbf{x}[i] \neq vr_i$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, vc, \mathbf{x}, cu)$.

The probability that **Game** 2 outputs failure is bound by the following claim.

Theorem A.3.3 *Under the binding property of the NHVC scheme, we have that* $|Pr[\textbf{Game } 2] - Pr[\textbf{Game } 1]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-nhvc}}$.

PROOF OF THEOREM A.3.3. We construct an algorithm B that, given an adversary that makes **Game** 2 fail with non-negligible probability, breaks the binding property of the NHVC scheme with non-negligible probability. B behaves as **Game** 2 with the following modifications:

- When the challenger sends the parameters par, B stores par as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$.
- When the adversary sends a valid proof with witness $wit = (w_i, i, o_i, vr_i, or_i)$ and instance $ins = (par, vc, parcom, c_i', cr_i', cr)$ such that the values i and vr_i in the witness fulfil $\mathbf{x}[i] \neq vr_i$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, vc, \mathbf{x}, cu)$, B runs $w_i' \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and sends $(vc, i, vr_i, \mathbf{x}[i], w_i, w_i')$ to the challenger.

This concludes the proof of Theorem A.3.3.

The distribution of **Game** 2 is identical to our simulation. This concludes the proof of Theorem A.3.2.

A.3.2 *Security Analysis when the Updater is Corrupt*

We describe the simulator $\mathcal{S}$ for the case in which $\mathcal{U}$ is corrupt.

Figure A.30: Security Analysis of Π_{UD} when the Updater is Corrupt

INITIALIZATION OF $\mathcal{S}$. $\mathcal{S}$ sets $cr \leftarrow 0$. $\mathcal{S}$ runs the crs.get interface of an instance of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ to get the parameters par.

$\mathcal{A}$ REQUESTS AND RECEIVES par. $\mathcal{S}$ proceeds as in the case where $\mathcal{R}$ is corrupt.

$\mathcal{A}$ SENDS UPDATE. When $\mathcal{A}$ invokes the aut.send interface on input the message $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{AUT}}$ on that input. If $(sid, par, vc, \mathbf{x}, cr)$ is not stored.

- $\mathcal{S}$ initializes a counter $cr \leftarrow 0$.
- $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for $i = [1, N]$.
- $\mathcal{S}$ runs $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, and stores $(sid, par, vc, \mathbf{x}, cr)$.

Else:

- $\mathcal{S}$ sends an abortion message if $cu' \neq cr + 1$, or if, for all $i \notin [1, N]$, $vu_i \notin \mathbb{U}_v$.
- Otherwise $\mathcal{S}$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $vc' \leftarrow vc$. For all $i \in [1, N]$ such that $vu_i \neq \perp$, $\mathcal{S}$ computes $\mathbf{x}'[i] \leftarrow vu_i$ and $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc', i, \mathbf{x}[i], vu_i)$.
- $\mathcal{S}$ replaces the stored tuple $(sid, par, vc, \mathbf{x}, cr)$ by $(sid, par, vc', \mathbf{x}', cu')$.

$\mathcal{S}$ sends $(\mathsf{ud.update.ini}, sid, (i, vu_i)_{\forall i \in [1,N]})$ to $\mathcal{F}_{\text{UD}}$. When $\mathcal{F}_{\text{UD}}$ sends $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i \in [1,N]})$, $\mathcal{S}$ sends $(\mathsf{ud.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UD}}$.

HONEST $\mathcal{R}$ SENDS A PROOF. On input from $\mathcal{F}_{\text{UD}}$ the message $(\mathsf{ud.read.sim}, sid, qid, c_i, cr_i)$, $\mathcal{S}$ sends $(\mathsf{ud.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UD}}$ and receives the message $(\mathsf{ud.read.end}, sid, c_i, cr_i)$ from $\mathcal{F}_{\text{UD}}$. $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, vc, \mathbf{x}, cr)$.
- $\mathcal{S}$ parses c_i as $(c_i', parcom, \mathsf{COM.Verify})$.
- $\mathcal{S}$ parses cr_i as $(cr_i', parcom, \mathsf{COM.Verify})$.
- $\mathcal{S}$ sets $ins \leftarrow (par, vc, parcom, c_i', cr_i', cr)$.
- $\mathcal{S}$ sets the message corresponding to the zk.prove interface of $\mathcal{F}_{\text{ZK}}^{R}$ to send ins to $\mathcal{A}$. Note that $\mathcal{S}$ does not know the witness, so it does not run a copy of the functionality. Instead, $\mathcal{S}$ sets the message as if it was sent by a copy of $\mathcal{F}_{\text{ZK}}^{R}$.

Theorem A.3.4 *When $\mathcal{U}$ is corrupt, Π_{UD} securely realizes $\mathcal{F}_{\text{UD}}$ in the $(\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{ZK}}^{R})$-hybrid model.*

PROOF OF THEOREM A.3.4. The only difference between the real world protocol and $\mathcal{S}$ is that $\mathcal{S}$ does not run $\mathcal{F}_{\text{ZK}}^{R}$ because $\mathcal{S}$ does not know the witness of the proof. Because $\mathcal{F}_{\text{ZK}}^{R}$ does not leak the witness to the adversary, this change does not alter the view of the environment.

A.4 SECURITY ANALYSIS OF OUR CONSTRUCTION FOR UUD

Theorem A.4.1 *Π_{UUD} securely realizes $\mathcal{F}_{\mathrm{UUD}}$ in the $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{ZK}}^{R})$-hybrid model if the SVC scheme is binding.*

To prove that Π_{UUD} securely realizes $\mathcal{F}_{\mathrm{UUD}}$, we must show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{UUD}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{UUD}}$ for all corrupt parties in the ideal world.

$\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}$, $\mathcal{F}_{\mathrm{BB}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$. In the descriptions of our simulators below, for brevity, we omit part of the communication between $\mathcal{S}$ and $\mathcal{A}$. Whenever a copy of those functionalities sends a message $(*.*.\mathsf{sim})$ to $\mathcal{S}$, $\mathcal{S}$ implicitly forwards that message to $\mathcal{A}$ and runs again a copy of that functionality on input the response provided by $\mathcal{A}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to $\mathcal{A}$ if the functionality sends the abortion message to a corrupt party.

In Section A.4.1, we analyse the security of Π_{UUD} when (a subset of) readers $\mathcal{R}_k$ are corrupt. In Section A.4.2, we analyse the security of Π_{UUD} when $\mathcal{U}$ is corrupt. We do not analyse in detail the security of Π_{UUD} when $\mathcal{U}$ and (a subset of) readers $\mathcal{R}_k$ are corrupt. We note that, in Π_{UUD}, honest readers communicate with $\mathcal{U}$ but not with other readers. Therefore, for this case the simulator and the security proof are similar to the case where only $\mathcal{U}$ is corrupt.

A.4.1 *Security Analysis when the Readers are Corrupt*

We describe $\mathcal{S}$ for the case in which (a subset of) readers $\mathcal{R}_k$ are corrupt.

Figure A.33: Security Analysis of Π_{UUD} when the Readers are Corrupt

INITIALIZATION OF $\mathcal{S}$. $\mathcal{S}$ runs the $\mathsf{crs.get}$ interface of an instance of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}$ to get the parameters par.

HONEST $\mathcal{U}$ SENDS AN UPDATE. On input $(\mathsf{uud.update.sim}, sid, qid, (i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i \in [1,N]})$ from the functionality $\mathcal{F}_{\mathrm{UUD}}$, $\mathcal{S}$ runs the $\mathsf{bb.write}$ interface of a copy of $\mathcal{F}_{\mathrm{BB}}$ on input $(i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i \in [1,N]}$ to write the update into the bulletin board in $\mathcal{F}_{\mathrm{BB}}$. $\mathcal{S}$ follows the same steps described in Π_{UUD} in order to set and update a tuple $(sid, par, svc, \mathbf{x}, cu)$. Then $\mathcal{S}$ sends $(\mathsf{uud.update.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{UUD}}$.

$\mathcal{A}$ REQUESTS DATABASE. When $\mathcal{A}$ invokes the $\mathsf{bb.getbb}$ interface on input an index i, $\mathcal{S}$ sends $(\mathsf{uud.getdb.ini}, sid)$ to $\mathcal{F}_{\mathrm{UUD}}$. When $\mathcal{F}_{\mathrm{UUD}}$ sends $(\mathsf{uud.getdb.sim}, sid, qid)$, $\mathcal{S}$ sends the message $(\mathsf{uud.getdb.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{UUD}}$. When $\mathcal{F}_{\mathrm{UUD}}$ sends the message $(\mathsf{uud.getdb.end}, sid, \mathsf{DB})$, $\mathcal{S}$ runs the $\mathsf{bb.getbb}$ interface of $\mathcal{F}_{\mathrm{BB}}$ on input i to send the update $(i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i \in [1,N]}$ to

$\mathcal{A}$. We recall that the update was already stored in $\mathcal{F}_{\text{BB}}$ when it was sent by the honest $\mathcal{U}$.

$\mathcal{A}$ REQUESTS AND RECEIVES par. When $\mathcal{A}$ invokes the crs.get interface, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}$ on that input to send par to $\mathcal{A}$.

$\mathcal{A}$ SENDS A PROOF. When $\mathcal{A}$ invokes the zk.prove interface on input a pseudonym P, a witness wit and an instance ins, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{ZK}}^{R}$ on that input. Then $\mathcal{S}$ parses ins as $(par', svc', parcom, c'_i, \langle cr'_{i,j}\rangle_{\forall j\in[1,L]}, cr_k)$ and wit as $(w_I, I, i, o_i, \langle vr_{ii,j}, or_{ii,j}\rangle_{\forall j\in[1,L]})$. $\mathcal{S}$ sets $c_i \leftarrow (c'_i, parcom, \text{COM.Verify})$ and $\langle cr_{ii,j} \leftarrow (cr'_{i,j}, parcom, \text{COM.Verify})\rangle_{\forall j\in[1,L]}$. $\mathcal{S}$ sends $(\text{uud.read.ini}, sid, P, (i, c_i, o_i, \langle vr_{ii,j}, cr_{ii,j}, or_{ii,j}\rangle_{\forall j\in[1,L]}))$ to the functionality $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends the message $(\text{uud.read.sim}, sid, qid, (c_i, \langle cr_{ii,j}\rangle_{\forall j\in[1,L]}))$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, svc, \mathbf{x}, cu)$. If $cr_k \neq cu$, or if $par' \neq par$, or if $svc' \neq svc$, $\mathcal{S}$ sends $\mathcal{F}_{\text{UUD}}$ a message that makes $\mathcal{F}_{\text{UUD}}$ abort.
- Else, if for any $j \in [1, L]$, $\mathbf{x}[(i-1)L+j] \neq vr_{ii,j}$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ sends $(\text{uud.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$.

Theorem A.4.2 *When (a subset of) readers $\mathcal{R}_k$ are corrupt, Π_{UUD} securely realizes $\mathcal{F}_{\text{UUD}}$ in the $(\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{ZK}}^{R})$-hybrid model if the SVC scheme is binding.*

PROOF OF THEOREM A.4.2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish the real-world protocol from the ideal-world protocol with non-negligible probability. We denote by Pr [**Game** i] the probability that the environment distinguishes **Game** i from the real-world protocol.

Figure A.35: Proof of Theorem A.4.2

GAME 0: This game corresponds to the execution of the real-world protocol. Therefore, we have that Pr [**Game** 0] $= 0$.

GAME 1: **Game** 1 follows **Game** 0, except that **Game** 1 runs an initialization phase to set a counter cu and the parameters par. **Game** 1 stores and updates a tuple $(sid, par, svc, \mathbf{x}, cu)$. These changes do not alter the view of the environment. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$.

GAME 2: **Game** 2 follows **Game** 1, except that, when the adversary sends a valid proof with witness wit and instance ins, **Game** 2 outputs

failure if, for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq vr_{ii,j}$, where $\mathbf{x}$ is in the stored tuple $(sid, par, svc, \mathbf{x}, cu)$.

The probability that **Game** 2 outputs failure is bound by the following claim.

Theorem A.4.3 *Under the binding property of the SVC scheme, we have that* $|Pr[\textbf{Game}\ 2] - Pr[\textbf{Game}\ 1]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-svc}}$.

PROOF OF THEOREM A.4.3. We construct an algorithm B that, given an adversary that makes **Game** 2 fail with non-negligible probability, breaks the binding property of the SVC scheme with non-negligible probability. B behaves as **Game** 2 with the following modifications:

- When the challenger sends the parameters par, B stores par as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{SVC.Setup}}$.
- When the adversary sends a valid proof with witness $wit = (w_I, I, i, o_i, \langle vr_{ii,j}, or_{ii,j}\rangle_{\forall j\in[1,L]})$ and instance $ins = (par, svc, parcom, c'_i, \langle cr'_{i,j}\rangle_{\forall j\in[1,L]}, cr_k)$ such that, for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq vr_{ii,j}$, where $\mathbf{x}$ is in the stored tuple $(sid, par, svc, \mathbf{x}, cu)$, B runs $w_I \leftarrow \mathsf{SVC.Open}(par, I, \mathbf{x})$ and sends $(svc, I, I, \langle vr_{ii,j}\rangle_{\forall j\in[1,L]}, \langle \mathbf{x}[(i-1)L+j]\rangle_{\forall j\in[1,L]}, w_I, w'_I)$ to the challenger.

This concludes the proof of Theorem A.4.3.

The distribution of **Game** 2 is identical to our simulation. This concludes the proof of Theorem A.4.2.

A.4.2 *Security Analysis when the Updater is Corrupt*

We describe $\mathcal{S}$ for the case in which $\mathcal{U}$ is corrupt.

Figure A.37: Security Analysis of Π_{UUD} when the Updater is Corrupt

INITIALIZATION OF $\mathcal{S}$. $\mathcal{S}$ runs the crs.get interface of an instance of $\mathcal{F}_{\text{CRS}}^{\mathsf{SVC.Setup}}$ to get the parameters par.

$\mathcal{A}$ REQUESTS AND RECEIVES par. $\mathcal{S}$ proceeds as in the case where $\mathcal{R}_k$ is corrupt.

$\mathcal{A}$ SENDS UPDATE. When $\mathcal{A}$ invokes the bb.write interface on input $(i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i\in[1,N]}$, $\mathcal{S}$ sends $(\mathsf{uud.update.ini}, sid, (i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i\in[1,N]})$ to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends $(\mathsf{uud.update.sim}, sid, qid, (i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i\in[1,N]})$, $\mathcal{S}$ sends $(\mathsf{uud.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends $(\mathsf{uud.update.end}, sid)$, $\mathcal{S}$ follows the same steps described in Π_{UUD} in order to set and update a tuple $(sid, par, svc, \mathbf{x}, cu)$. Finally, $\mathcal{S}$ runs the bb.write interface of a copy of $\mathcal{F}_{\text{BB}}$ on input $(i, vr_{ii,1}, \ldots, vr_{ii,L})_{\forall i\in[1,N]}$.

HONEST $\mathcal{R}_k$ REQUESTS DATABASE. When $\mathcal{F}_{\text{UUD}}$ sends $(\mathsf{uud.getdb.sim}, sid, qid)$, the simulator $\mathcal{S}$ runs the $\mathsf{bb.getbb}$ interface of $\mathcal{F}_{\text{BB}}$ on input a random index i. Then $\mathcal{S}$ sends $(\mathsf{uud.getdb.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$.

HONEST $\mathcal{R}_k$ SENDS A PROOF. On input $(\mathsf{uud.read.sim}, sid, qid, (c_i, \langle cr_{ii,j}\rangle_{\forall j\in[1,L]}))$ from $\mathcal{F}_{\text{UUD}}$, $\mathcal{S}$ sends $(\mathsf{uud.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$ and receives the message $(\mathsf{uud.read.end}, sid, P, (c_i, \langle cr_{ii,j}\rangle_{\forall j\in[1,L]}))$ from the functionality $\mathcal{F}_{\text{UUD}}$. $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, svc, \mathbf{x}, cu)$.
- $\mathcal{S}$ parses c_i as $(c_i', parcom, \mathsf{COM.Verify})$ and $\langle cr_{ii,j}\rangle_{\forall j\in[1,L]}$ as $\langle (cr_{i\,i,j}', parcom, \mathsf{COM.Verify})\rangle_{\forall j\in[1,L]}$.
- $\mathcal{S}$ sets $ins \leftarrow (par, svc, parcom, c_i', \langle cr_{i\,i,j}'\rangle_{\forall j\in[1,L]}, cu)$.
- $\mathcal{S}$ sets the message corresponding to the $\mathsf{zk.prove}$ interface of $\mathcal{F}_{\text{ZK}}^{R}$ to send P and ins to $\mathcal{A}$. Note that $\mathcal{S}$ does not know the witness, so it does not run a copy of the functionality. Instead, $\mathcal{S}$ sets the message as if it was sent by a copy of $\mathcal{F}_{\text{ZK}}^{R}$.

Theorem A.4.4 *When $\mathcal{U}$ is corrupt, Π_{UUD} securely realizes $\mathcal{F}_{\text{UUD}}$ in the $(\mathcal{F}_{\text{CRS}}^{\mathsf{SVC.Setup}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{ZK}}^{R})$-hybrid model.*

PROOF OF THEOREM A.4.4. There are two differences between the real world protocol and $\mathcal{S}$. First, $\mathcal{S}$ uses a random index i to run the $\mathsf{bb.getbb}$ interface of $\mathcal{F}_{\text{BB}}$. This change does not alter the view of the environment because $\mathcal{F}_{\text{BB}}$ does not disclose i to the adversary. Second, $\mathcal{S}$ does not run $\mathcal{F}_{\text{ZK}}^{R}$ because $\mathcal{S}$ does not know the witness of the proof. Because $\mathcal{F}_{\text{ZK}}^{R}$ does not leak the witness to the adversary and the pseudonym and the instance are not modified, this change does not alter the view of the environment.

BIBLIOGRAPHY

[1] Masayuki Abe, Jan Camenisch, Maria Dubovitskaya, and Ryo Nishimaki. "Universally composable adaptive oblivious transfer (with access control) from standard assumptions." In: *DIM'13, Proceedings of the 2013 ACM Workshop on Digital Identity Management*. 2013, pp. 1–12.

[2] Masayuki Abe, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. "Optimal Structure-Preserving Signatures in Asymmetric Bilinear Groups." In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 2011, pp. 649–666.

[3] William Aiello, Yuval Ishai, and Omer Reingold. "Priced Oblivious Transfer: How to Sell Digital Goods." In: *Advances in Cryptology – EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Innsbruck, Austria: Springer, Heidelberg, Germany, 2001, pp. 119–135.

[4] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. "Charm: a framework for rapidly prototyping cryptosystems." In: *J. Cryptographic Engineering* 3.2 (2013), pp. 111–128.

[5] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. "A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Sigma-Protocols." In: *ESORICS 2010: 15th European Symposium on Research in Computer Security*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. Vol. 6345. Lecture Notes in Computer Science. Athens, Greece: Springer, Heidelberg, Germany, 2010, pp. 151–167.

[6] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols." In: *ACM CCS 2012: 19th Conference on Computer and Communications Security*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. Raleigh, NC, USA: ACM Press, 2012, pp. 488–500.

[7] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. *RELIC is an Efficient LIbrary for Cryptography*. https://github.com/relic-toolkit/relic.

[8] David Auerbach. "The Open-Source Question." In: *Slate* (2015). URL: https://slate.com/technology/2015/02/werner-koch-and-gpg-how-can-we-preserve-important-barely-funded-open-source-software.html.

[9] Lawrence Ausubel and Paul Milgrom. "The Lovely but Lonely Vickrey Auction." In: *Comb. Auct.* 17 (Jan. 2006).

[10] Wouter Biesmans, Josep Balasch, Alfredo Rial, Bart Preneel, and Ingrid Verbauwhede. "Private mobile pay-TV from priced oblivious transfer." In: *IEEE Transactions on Information Forensics and Security* 13.2 (2018), pp. 280–291.

[11] Alberto Blanco-Justicia and Josep Domingo-Ferrer. "Privacy-Preserving Loyalty Programs." In: *DPM 2014, SETOP 2014, QASA 2014*. 2014, pp. 133–146.

[12] Olivier Blazy, Céline Chevalier, and Paul Germouty. "Adaptive Oblivious Transfer and Generalization." In: *Advances in Cryptology – ASIACRYPT 2016, Part II*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, 2016, pp. 217–247.

[13] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. "Updatable Anonymous Credentials and Applications to Incentive Systems." In: *ACM CCS 2019*. 2019, pp. 1671–1685.

[14] Jan Bobolz, Fabian Eidens, Stephan Krenn, Daniel Slamanig, and Christoph Striecks. "Privacy-Preserving Incentive Systems with Highly Efficient Point-Collection." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 382.

[15] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. "Efficient Protocols for Set Membership and Range Proofs." In: *ASIACRYPT 2008*. 2008, pp. 234–252.

[16] Jan Camenisch, Maria Dubovitskaya, Robert R. Enderlein, and Gregory Neven. "Oblivious Transfer with Hidden Access Control from Attribute-Based Encryption." In: *SCN 12: 8th International Conference on Security in Communication Networks*. Ed. by Ivan Visconti and Roberto De Prisco. Vol. 7485. Lecture Notes in Computer Science. Amalfi, Italy: Springer, Heidelberg, Germany, 2012, pp. 559–579.

[17] Jan Camenisch, Maria Dubovitskaya, and Gregory Neven. "Oblivious transfer with access control." In: *ACM Conference on Computer and Communications Security, CCS 2009*. 2009, pp. 131–140.

[18] Jan Camenisch, Maria Dubovitskaya, and Gregory Neven. "Unlinkable Priced Oblivious Transfer with Rechargeable Wallets." In: *FC 2010: 14th International Conference on Financial Cryptography and Data Security*. Ed. by Radu Sion. Vol. 6052. Lecture Notes in Computer Science. Tenerife, Canary Islands, Spain: Springer, Heidelberg, Germany, 2010, pp. 66–81.

[19] Jan Camenisch, Maria Dubovitskaya, Gregory Neven, and Gregory M. Zaverucha. "Oblivious Transfer with Hidden Access Control Policies." In: *PKC 2011: 14th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi. Vol. 6571. Lecture Notes in Computer Science. Taormina, Italy: Springer, Heidelberg, Germany, 2011, pp. 192–209.

[20] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. "UC Commitments for Modular Protocol Design and Applications to Revocation and Attribute Tokens." In: *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*. 2016, pp. 208–239.

[21] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. "Concise UC Zero-Knowledge Proofs for Oblivious Updatable Databases." In: *CSF 2021: IEEE 34st Computer Security Foundations Symposium*. Ed. by Ralf Küsters and Dave Naumann. Virtual Conference: IEEE Computer Society Press, 2021, pp. 1–16.

[22] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. "Compact E-Cash." In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. Lecture Notes in Computer Science. Aarhus, Denmark: Springer, Heidelberg, Germany, 2005, pp. 302–321.

[23] Jan Camenisch, Aggelos Kiayias, and Moti Yung. "On the Portability of Generalized Schnorr Proofs." In: *Advances in Cryptology – EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Cologne, Germany: Springer, Heidelberg, Germany, 2009, pp. 425–442.

[24] Jan Camenisch, Stephan Krenn, and Victor Shoup. "A Framework for Practical Universally Composable Zero-Knowledge Protocols." In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Seoul, South Korea: Springer, Heidelberg, Germany, 2011, pp. 449–467.

[25] Jan Camenisch, Anja Lehmann, Gregory Neven, and Alfredo Rial. "Privacy-Preserving Auditing for Attribute-Based Credentials." In: *ESORICS 2014*. 2014, pp. 109–127.

[26] Jan Camenisch and Anna Lysyanskaya. "An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation." In: *Advances in Cryptology – EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Innsbruck, Austria: Springer, Heidelberg, Germany, 2001, pp. 93–118.

[27] Jan Camenisch, Gregory Neven, and abhi shelat. "Simulatable Adaptive Oblivious Transfer." In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by Moni Naor. Vol. 4515. Lecture Notes in Computer Science. Barcelona, Spain: Springer, Heidelberg, Germany, 2007, pp. 573–590.

[28] Jan Camenisch and Markus Stadler. "Efficient Group Signature Schemes for Large Groups (Extended Abstract)." In: *Advances in Cryptology – CRYPTO'97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 1997, pp. 410–424.

[29] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols." In: *FOCS 2001*. 2001, pp. 136–145.

[30] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. "Universally composable two-party and multi-party secure computation." In: *34th Annual ACM Symposium on Theory of Computing*. Montréal, Québec, Canada: ACM Press, 2002, pp. 494–503.

[31] Ran Canetti and Tal Rabin. "Universal Composition with Joint State." In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2003, pp. 265–281.

[32] Dario Catalano and Dario Fiore. "Vector Commitments and Their Applications." In: *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*. 2013, pp. 55–72.

[33] Scott E. Coull, Matthew Green, and Susan Hohenberger. "Controlling Access to an Oblivious Database Using Stateful Anonymous Credentials." In: *Public Key Cryptography - PKC 2009*. 2009, pp. 501–520.

[34] Ivan Damgård and Eiichiro Fujisaki. "A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order." In: *Advances in Cryptology – ASIACRYPT 2002*. Ed. by Yuliang Zheng. Vol. 2501. Lecture Notes in Computer Science. Queenstown, New Zealand: Springer, Heidelberg, Germany, 2002, pp. 125–142.

[35] Aditya Damodaran, Maria Dubovitskaya, and Alfredo Rial. "UC Priced Oblivious Transfer with Purchase Statistics and Dynamic Pricing." In: *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*. Vol. 11898. Lecture Notes in Computer Science. Springer, 2019, pp. 273–296. URL: http://hdl.handle.net/10993/39424.

[36] Aditya Damodaran and Alfredo Rial. *SZK Deliverable D1.1: Definitions for Stateful Zero Knowledge*. Ed. by Alfredo Rial. 2020.

[37] Aditya Damodaran and Alfredo Rial. *SZK Deliverable D2.1: Constructions for Stateful Zero Knowledge*. Ed. by Alfredo Rial. 2020.

[38] Aditya Damodaran and Alfredo Rial. "UC Updatable Databases and Applications." In: *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*. Vol. 12174. Lecture Notes in Computer Science. Springer, 2020, pp. 66–87. URL: http://hdl.handle.net/10993/42984.

[39] Aditya Damodaran and Alfredo Rial. "Unlinkable Updatable Databases and Oblivious Transfer with Access Control." In: *Information Security and Privacy - 25th Australasian Conference, ACISP 2020, Perth, WA, Australia, November 30 - December 2, 2020, Proceedings*. Vol. 12248. Lecture Notes in Computer Science. Springer, 2020, pp. 584–604. URL: http://hdl.handle.net/10993/43250.

[40] Aditya Damodaran and Alfredo Rial. *Implementations for Unlinkable Updatable Hiding Databases and Privacy-Preserving Loyalty Programs*. 2021. URL: http://hdl.handle.net/10993/49102.

[41] Aditya Damodaran and Alfredo Rial. "Unlinkable Updatable Hiding Databases and Privacy-Preserving Loyalty Programs." In: *Proc. Priv. Enhancing Technol.* 2021.3 (2021), pp. 95–121. URL: http://hdl.handle.net/10993/49090.

[42] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004.

[43] Josep Domingo-Ferrer, Alberto Blanco-Justicia, and Carla Ràfols. "Dynamic group size accreditation and group discounts preserving anonymity." In: *Int. J. Inf. Sec.* 17.3 (2018), pp. 243–260. URL: https://doi.org/10.1007/s10207-017-0368-y.

[44] Jannik Dreier, Hugo Jonker, and Pascal Lafourcade. "Defining verifiability in e-auction protocols." In: May 2013, pp. 547–552.

[45] Charles Duhigg. "How Companies Learn Your Secrets." In: *The New York Times* (2012). URL: https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html.

[46] Matthias Enzmann and Markus Schneider. "A Privacy-Friendly Loyalty System for Electronic Marketplaces." In: *2004 IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 04)*. 2004, pp. 385–393.

[47] Martin Farrer. "Airline data hack: hundreds of thousands of Star Alliance passengers' details stolen." In: *The Guardian* (2021). URL: https://www.theguardian.com/world/2021/mar/05/airline-data-hack-hundreds-of-thousands-of-star-alliance-passengers-details-stolen.

[48] Cédric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. "ZQL: A Compiler for Privacy-Preserving Data Processing." In: *USENIX Security 2013: 22nd USENIX Security Symposium*. Ed. by Samuel T. King. Washington, DC, USA: USENIX Association, 2013, pp. 163–178.

[49] M. Franklin and M. Reiter. "The Design and Implementation of a Secure Auction Service." In: *IEEE Trans. Software Eng.* 22 (1996), pp. 302–312.

[50] Matthew Fredrikson and Benjamin Livshits. "ZØ: An Optimizing Distributing Zero-Knowledge Compiler." In: *USENIX Security 2014: 23rd USENIX Security Symposium*. Ed. by Kevin Fu and Jaeyeon Jung. San Diego, CA, USA: USENIX Association, 2014, pp. 909–924.

[51] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs." In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Athens, Greece: Springer, Heidelberg, Germany, 2013, pp. 626–645.

[52] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks." In: *SIAM J. Comput.* 17.2 (1988), pp. 281–308.

[53] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "New Techniques for Noninteractive Zero-Knowledge." In: *J. ACM* 59.3 (2012), 11:1–11:35.

[54] Hashicorp. *Vagrant*. https://www.vagrantup.com/.

[55] Ryan Henry, Femi G. Olumofin, and Ian Goldberg. "Practical PIR for electronic commerce." In: *ACM CCS 2011: 18th Conference on Computer and Communications Security*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. Chicago, Illinois, USA: ACM Press, 2011, pp. 677–690.

[56] Oliver Hinz, Eva Gerstmeier, Omid Tafreschi, Matthias Enzmann, and Markus Schneider. "Customer loyalty programs and privacy concerns." In: *BLED 2007 Proceedings* (2007), p. 32.

[57] Malika Izabachène, Benoît Libert, and Damien Vergnaud. "Block-Wise P-Signatures and Non-interactive Anonymous Credentials with Efficient Attributes." In: *13th IMA International Conference on Cryptography and Coding*. Ed. by Liqun Chen. Vol. 7089. Lecture Notes in Computer Science. Oxford, UK: Springer, Heidelberg, Germany, 2011, pp. 431–450.

[58] Tun-Min Catherine Jai and Nancy J King. "Privacy versus reward: Do loyalty programs increase consumers' willingness to share personal information with third-party advertisers and data brokers?" In: *Journal of Retailing and Consumer Services* 28 (2016), pp. 296–303.

[59] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications." In: *Advances in Cryptology – ASIACRYPT 2010*. Ed. by Masayuki Abe. Vol. 6477. Lecture Notes in Computer Science. Singapore: Springer, Heidelberg, Germany, 2010, pp. 177–194.

[60] H. Kikuchi, S. Hotta, K. Abe, and S. Nakanishi. "Distributed auction servers resolving winner and winning bid without revealing privacy of bids." In: *Proceedings Seventh International Conference on Parallel and Distributed Systems: Workshops*. 2000, pp. 307–312.

[61] Joe Kilian. "A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract)." In: *24th Annual ACM Symposium on Theory of Computing*. Victoria, BC, Canada: ACM Press, 1992, pp. 723–732.

[62] Werner Koch. *Libgcrypt code repository*. 2000. URL: https://git.gnupg.org/.

[63] Markulf Kohlweiss and Alfredo Rial. "Optimally private access control." In: *WPES 2013*. Ed. by Ahmad-Reza Sadeghi and Sara Foresti. ACM, 2013, pp. 37–48.

[64] Russell W. F. Lai and Giulio Malavolta. "Subvector Commitments with Application to Succinct Arguments." In: *Advances in Cryptology – CRYPTO 2019, Part I*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11692. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2019, pp. 530–560.

[65] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. "Why Does Cryptographic Software Fail? A Case Study and Open Problems." In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. APSys '14. Beijing, China: Association for Computing Machinery, 2014.

[66] Benoît Libert, San Ling, Fabrice Mouhartem, Khoa Nguyen, and Huaxiong Wang. "Adaptive Oblivious Transfer with Access Control from Lattice Assumptions." In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Hong Kong, China: Springer, Heidelberg, Germany, 2017, pp. 533–563.

[67] Benoît Libert, Thomas Peters, and Moti Yung. "Group Signatures with Almost-for-Free Revocation." In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2012, pp. 571–589.

[68] Benoît Libert, Somindu C. Ramanna, and Moti Yung. "Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions." In: *ICALP 2016: 43rd International Colloquium on Automata, Languages and Programming*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol. 55. LIPIcs. Rome, Italy: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 30:1–30:14.

[69] Benoît Libert and Moti Yung. "Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs." In: *TCC 2010: 7th Theory of Cryptography Conference*. Ed. by Daniele Micciancio. Vol. 5978. Lecture Notes in Computer Science. Zurich, Switzerland: Springer, Heidelberg, Germany, 2010, pp. 499–517.

[70] Helger Lipmaa, N. Asokan, and Valtteri Niemi. "Secure Vickrey Auctions without Threshold Trust." In: *Financial Cryptography*. Ed. by Matt Blaze. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 87–101.

[71] Wouter Lueks, Bogdan Kulynych, Jules Fasquelle, Simon Le Bail-Collet, and Carmela Troncoso. "zksk." In: *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society - WPES'19*. ACM Press, 2019. URL: https://doi.org/10.1145%2F3338498.3358653.

[72] "Mark Zuckerberg in his own words: The CNN interview." In: *CNN Money* (2018). URL: https://money.cnn.com/2018/03/21/technology/mark-zuckerberg-cnn-interview-transcript/index.html.

[73] Philip Marquardt, David Dagon, and Patrick Traynor. "Impeding Individual User Profiling in Shopper Loyalty Programs." In: *FC 2011*. 2011, pp. 93–101.

[74] Sarah Meiklejohn, C. Christopher Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. "ZKPDL: A Language-Based System for Efficient Zero-Knowledge Proofs and Electronic Cash." In: *USENIX Security 2010: 19th USENIX Security Symposium*. Washington, DC, USA: USENIX Association, 2010, pp. 193–206.

[75] Milica Milutinovic, Italo Dacosta, Andreas Put, and Bart De Decker. "An Advanced, Privacy-Friendly Loyalty System." In: *Privacy and Identity Management for Emerging Services and Technologies - 8th IFIP WG 9.2, 9.5, 9.6/11.7, 11.4, 11.6 International Summer School*. 2013, pp. 128–138.

[76] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. "Sublinear Zero-Knowledge Arguments for RAM Programs." In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Heidelberg, Germany, 2017, pp. 501–531.

[77] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. ""Jumping Through Hoops": Why do Java Developers Struggle with Cryptography APIs?" In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 935–946.

[78] Moni Naor, Benny Pinkas, and Reuban Sumner. "Privacy Preserving Auctions and Mechanism Design." In: *Proceedings of the 1st ACM Conference on Electronic Commerce*. EC '99. Denver, Colorado, USA: Association for Computing Machinery, 1999, pp. 129–139. URL: https://doi.org/10.1145/336992.337028.

[79] Phong Q. Nguyen. "Can we trust cryptographic software? cryptographic flaws." In: *in GNU Privacy Guard v1.2.3. In EUROCRYPT 2004, LNCS*. Springer, 2004, pp. 555–570.

[80] Torben P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing." In: *CRYPTO '91*. 1991, pp. 129–140.

[81] Alfredo Rial. "Blind attribute-based encryption and oblivious transfer with fine-grained access control." In: *Des. Codes Cryptogr.* 81.2 (2016), pp. 179–223.

[82] Alfredo Rial, Josep Balasch, and Bart Preneel. "A privacy-preserving buyer–seller watermarking protocol based on priced oblivious transfer." In: *IEEE Transactions on Information Forensics and Security* 6.1 (2011), pp. 202–212.

[83] Alfredo Rial and George Danezis. "Privacy-preserving smart metering." In: *WPES 2011*, pp. 49–60.

[84] Alfredo Rial, George Danezis, and Markulf Kohlweiss. "Privacy-preserving smart metering revisited." In: *Int. J. Inf. Sec.* 17.1 (2018), pp. 1–31.

[85] Alfredo Rial, Markulf Kohlweiss, and Bart Preneel. "Universally Composable Adaptive Priced Oblivious Transfer." In: *Pairing-Based Cryptography - Pairing 2009*. 2009, pp. 231–247.

[86] Alfredo Rial and Bart Preneel. "Optimistic Fair Priced Oblivious Transfer." In: *AFRICACRYPT 10: 3rd International Conference on Cryptology in Africa*. Ed. by Daniel J. Bernstein and Tanja Lange. Vol. 6055. Lecture Notes in Computer Science. Stellenbosch, South Africa: Springer, Heidelberg, Germany, 2010, pp. 131–147.

[87] Matthew Rosenberg, Nicholas Confessore, and Carole Cadwalladr. "How Trump Consultants Exploited the Facebook Data of Millions." In: *The New York Times* (2018). URL: https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html.

[88] Kouichi Sakurai and Shingo Miyazaki. "An Anonymous Electronic Bidding Protocol Based on a New Convertible Group Signature Scheme." In: *Information Security and Privacy*. Ed. by E. P. Dawson, A. Clark, and Colin Boyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 385–399.

[89] Bruce Schneier. *Secrets & Lies: Digital Security in a Networked World*. 1st. USA: John Wiley & Sons, Inc., 2000.

[90] Erez Shinan. *Lark Parser*. https://github.com/lark-parser/lark.

[91] Jarrod Trevathan and Wayne Read. "Undesirable and Fraudulent Behaviour in Online Auctions." In: Jan. 2006, pp. 450–458.

[92] Jarrod Trevathan and Wayne Read. "Detecting Collusive Shill Bidding." In: May 2007, pp. 799 –808.

[93] William Vickrey. "Counterspeculation, Auctions, and Competitive Sealed Tenders." In: *The Journal of Finance* 16.1 (1961), pp. 8–37. URL: http://www.jstor.org/stable/2977633.

[94] Kurt Wagner. "Here's how Facebook allowed Cambridge Analytica to get data for 50 million users." In: *Vox* (2018). URL: https://www.vox.com/2018/3/17/17134072/facebook-cambridge-analytica-trump-explained-user-data.

[95] Arrianto Mukti Wibowo, Kwok-Yan Lam, and Gary S. H. Tan. "Loyalty Program Scheme for Anonymous Payment System." In: *Electronic Commerce and Web Technologies, EC-Web 2000*. 2000, pp. 253–265.

[96] Anna-Katharina Wickert, Michael Reif, Michael Eichberg, Anam Dodhy, and Mira Mezini. "A Dataset of Parametric Cryptographic Misuses." In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 96–100.

[97] Douglas Wikström. "A Universally Composable Mix-Net." In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Heidelberg, Germany, 2004, pp. 317–335.

[98] Ling-Ling Xu and Fang-Guo Zhang. "Oblivious Transfer with Threshold Access Control." In: *J. Inf. Sci. Eng.* 28.3 (2012), pp. 555–570.

[99] Lingling Xu and Fangguo Zhang. "Oblivious Transfer with Complex Attribute-Based Access Control." In: *Information Security and Cryptology - ICISC 2010*. 2010, pp. 370–395.

[100] Ye Zhang, Man Ho Au, Duncan S. Wong, Qiong Huang, Nikos Mamoulis, David W. Cheung, and Siu-Ming Yiu. "Oblivious Transfer with Access Control : Realizing Disjunction without Duplication." In: *PAIRING 2010: 4th International Conference on Pairing-based Cryptography*. Ed. by Marc Joye, Atsuko Miyaji, and Akira Otsuka. Vol. 6487. Lecture Notes in Computer Science. Yamanaka Hot Spring, Japan: Springer, Heidelberg, Germany, 2010, pp. 96–115.

COLOPHON

This document was typeset using a modified version of the typographical look-and-feel `classicthesis` template developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".