Threshold Homomorphic Encryption in the Universally Composable Cryptographic Library

Peeter Laud and Long Ngo
Tartu University

August 25, 2008

Abstract

Protocol security analysis has become an active research topic in recent years. Researchers have been trying to build sufficient theories for building automated tools, which give security proofs for cryptographic protocols. There are two approaches for analysing protocols: formal and computational. The former, often called Dolev-Yao style, uses abstract terms to model cryptographic messages with an assumption about perfect security of the cryptographic primitives. The latter mathematically uses indistinguishability to prove that adversaries with computational resources bounds cannot gain anything significantly. The first method is easy to be automated while the second one can give sound proofs of security.

Therefore there is a demand to bridge the gap between two methods in order to have better security-proof tools. One idea is to prove that some Dolev-Yao style cryptographic primitives used in formal tools are computationally sound for arbitrary active attacks in arbitrary reactive environments, i.e universally composable. As a consequence, protocols that use such primitives can also be proved secure by formal tools.

In this paper, we prove that a homomorphic encryption used together with a non-interactive zero-knowledge proof in Dolev-Yao style are sound abstractions for the real implementation under certain conditions. It helps to automatically design and analyze a class of protocols that use homomorphic encryptions together with non-interactive zero-knowledge proofs, such as e-voting.

Contents

List of Figures

Abbreviations and Acronyms

electronic auction	e-auction
electronic voting	e-voting
IND-CCA2	Indistinguishable under Adaptively Chosen ciphertext attack
IND-TCPA	Indistinguishable under Threshold Chosen plaintext attack
NIZK	Non-interactive zero-knowledge
PCL	Protocol Compositional Logic
UC	Universally composable
VSS	verifiable secret sharing

Chapter 1

INTRODUCTION

1.1 Problem statement and our motivation

Network protocol analysis is a necessity because of the fact that many faults have been found in practical protocols, which had been believed to be secure until a successful attack happened. It means we need to model a protocol and prove how secure it is to prevent potential attacks.

There are two types of models for specification and verification of cryptographic protocols so far: *Computational* and *Formal*. Computational models use low-level notions, i.e bit-strings are used for modelling messages, giving a detailed view of cryptographic system and protocols, but complex and error-prone. Formal models abstract the behaviours of systems and protocols, modelling messages as abstract terms, and are therefore errorless. However, most of them, based on the model first proposed by Dolev and Yao [DY83], ignore the security problem of lower level and assume perfect security. Therefore, by a formal method we can not achieve computational sound proofs for the security of cryptosystems [Zun06, Mea03, AR00, CS02].

To the best of my knowledge, there are currently two ways to achieve computational sound proofs in automatic ways. The first approach is formulating syntactic calculi to model probabilism and polynomial-time considerations directly and thus we can build tools for computational sound proofs [Bla06, DDM$^+$05, IK06]. It is a promising direction because it may provide automated proofs of security for cryptosystems as we can do by hand. The other approach is combine advantages of the two analysis methods by justifying Dolev-Yao model, i.e. showing that we can have some sound Dolev-Yao sound abstractions [BPW03a]. This direction has some limits since we can not have sound abstractions for some cryptographic components [BPW06b] but it gives better simplification. Our paper is one that follows this direction and we discuss more about below.

Bridging the gap and related works: Bridging this gap has been attracting a lot of attention. Ideally one would have a formal method tool that analyzes security protocols at a lower level, where we have well-defined computational security properties [Zun06]. Among research projects to bridges the gap is a paper by Abadi and Rogaway [AR00], which shows the relation between computational and formal worlds by pointing out the computational soundness of formal symmetric encryptions. However, this work is limited to the case of passive adversaries. [DDMR07] proposed *Protocol Compositional Logic* (PCL) . This type of logic can be used to model protocols in Dolev-Yao style so we can apply automatic ways for analyzing. In addition, it is complemented with *Computational* PCL, which is sound with respect to the complexity-theoretic model of modern cryptography. Another work is by Backes et al. [BPW03a] in which they show the indistinguishability between an ideal cryptographic library based on abstract terms and a real one based on bit-strings. The advantage of this approach is that the library works in a reactive setting. It means that the library can provide cryptographic primitives for more complex protocols with more powerful adversaries while preserving their security properties.

This paper is an extension for the last approach listed above so we take a deeper look at it. The library is described using the model for asynchronous reactive systems proposed by Pfitzmann and Waidner[PW01]. The first version of the library contains signature and public-key encryption based on an IND-CCA2 encryption scheme. After that the authors extended to message authentication in [BPW03b] and symmetric encryption in [BP04]. Later, however Backes also pointed some limitations with this work, namely limits with hashes [BPW06a] and with XOR in [BP05]. To circumvent this problem, they proposed *Conditional Reactive Simulatability* in [BDHK08]. That work is useful because we can add some more cryptographic primitives and the new library is still UC under some conditions. Although conditions exist, we can still use the library if the conditions do not affect some practical uses. And that motivated our work.

Our motivation: We want to extend the library in [BPW03a] to have a threshold homomorphic encryption. Although the library works only under some certain conditions, it may still be useful for a number of applications such as e-voting and e-auction applications.

1.2 Contribution of this paper

In this paper, we define an ideal cryptographic library that contains a threshold homomorphic encryption. It follows another library, which has CCA2 public-key encryption and signature scheme, described in [BPW03a]. We define the ideal library in a way that it is suitable for formal protocol verification. Then we show the ideal library can be implemented by a real one, using real cryptographic components and that the real library actually UC-realizes the ideal one (Because of the same structure and notations, this work repeats part of the work in [BPW03a]. However, our library is actually an extension of the original one). Having this library, now the protocol design and verification become easier: we design an arbitrary protocol with the ideal library so a fully automated tool can verify it. And for implementing it, we replace the ideal with the real one. It should be noticed that even if we already have a number of useful UC components, such as a universally composable message board, in many cases we still need such low-level primitives like the public-key encryption in this library to design protocols.

We also describe how we can apply this library and propose some possible improvements in the future.

1.3 Structure os the paper

The rest of the paper is outlined as follows:

- Chapter 2 gives background knowledge for what will be used in the rest of the paper;
- Chapter 3 shows an overview of the ideal and real library for an easier understanding of the rest of the paper;
- Chapter 4 and 5 describe about the ideal and real system in more details;
- Chapter 6 shows the simulator for the real adversary
- Chapter 7 states the theorem related to the indistinguishablity between two systems;
- Chapter 8 discusses about applications that motivated us to do this research, what we need to do more in the future and finally concludes the paper.

Chapter 2

PRELIMINARIES

This chapter gives some background knowledge, which is useful for the rest of the thesis.

2.1 Indistinguishability and Negligibility

The notions of *Indistinguishability* and *Negligibility* are used often in this paper so it is important to give their definitions here.

Negligible functions. A function $g : \mathbb{N} \to [0, 1]$ is called *negligible* if for every positive polynomial p and all sufficiently large k's we have $g(k) < 1/p(k)$.

Probability ensembles. A *probability ensemble* indexed by $S \subseteq \{0, 1\}^*$ is family of distributions $\{X_w\}_{w \in S}$ so that each X_w is a distribution that ranges over the subset of $\{0, 1\}^{\mathsf{poly}(|w|)}$.

Identically distributed. We say that two probability ensembles, $\{X_w\}_{w \in S}$ and $\{Y_w\}_{w \in S}$, are *identically distributed* or *perfectly indistinguishable* if for every $w \in S$ and every α

$$\mathsf{Pr}[X_w = \alpha] = \mathsf{Pr}[Y_w = \alpha].$$

Computationally indistinguishable. We say that two probability ensembles, $\{X_w\}_{w \in S}$ and $\{Y_w\}_{w \in S}$, are*computationally indistinguishable* if for any probabilistic polynomial time algorithm A, there exists a negligible function $g : \mathbb{N} \rightarrow [0, 1]$ so that

$$|\mathsf{Pr}[A(w, X_w) = 1] - \mathsf{Pr}[A(w, Y_w) = 1]| < g(|w|).$$

2.2 Simulatable security

One may ask what we mean when we say that a protocol is secure. One of the most popular definitions is based on simulatability, i.e simulatable security. To the best of our knowledge, the first time simulatability was mentioned is in [GMR85]. After that several formalizations of this idea appeared.

Informally, we can say that a protocol is secure for some task if it emulates an "ideal setting", in which participants give their inputs to a *trusted party* who locally computes the desired outputs and sends the results back to the participants. More specifically, we say that a real protocol π is as secure as a trusted host TH if replacing TH with π in any context does not cause any more harm to any honest users. To formalize context and harm, we introduce the concepts of *environment* and *adversary*. The environment interacts with the protocol and the adversary as black boxes. The adversary can decide to take over the control of some parties. The environment always knows the states of the adversary and also the corrupted parties' ones. Now we can define the simulatable security of a protocol π as follows: "π securely realizes TH if for any adversary $\mathsf{A}_{\mathsf{real}}$ and any environment H, there exists a adversary $\mathsf{A}_{\mathsf{sim}}$ such that the output of H when running with π and $\mathsf{A}_{\mathsf{real}}$ is indistinguishable from that of H running with TH and $\mathsf{A}_{\mathsf{sim}}$".

The above definition is considered *standard simulatability* whereas in *universal simulatability* the adversary $\mathsf{A}_{\mathsf{sim}}$ does not depend on the environment H. Formally we define *universal simulatability* as: "π securely realizes TH if for any adversary $\mathsf{A}_{\mathsf{real}}$ there exists a adversary $\mathsf{A}_{\mathsf{sim}}$ such that for any environment H, the output of H when running with π and $\mathsf{A}_{\mathsf{real}}$ is indistinguishable from that of H running with TH and $\mathsf{A}_{\mathsf{sim}}$".

We can see that these definitions capture well the concept of security: One can learn nothing except information he is supposed to learn as specified in TH in any situation. However, the security definition itself does not guarantee the possibility of composing protocols in parallel. Therefore we need some composition theorem, which is discussed in Section 2.3.

2.3 Universal composability or Reactive simulatability

It turns out that simulatable security allows composition. There are two independent works [BPW04, Can01] that give some composition theorems for simulatable security. Here we consider the case of universal simulatability.

Let us define a $G-hybrid$ model as follows: G is an ideal functionality. A protocol π in this model is a real protocol which can make calls to G through its interface. π can run polynomial many copies of G simultaneously. Let π^ρ be the protocol that makes calls to the protocol ρ instead of G. We have the following theorem.

Theorem 2.3.1. (Composition theorem) If a protocol π in the $G-hybrid$ model securely realizes a functionality F and protocol ρ in the real world securely realizes G, then protocol π^ρ securely realizes F in real world. $\square$

Detailed proofs for this theorem can be found in [BPW04, Can01].

It is important to notice that there is a restriction in this composition theorem that the adversary $\mathsf{A}_{\mathsf{sim}}$ is not allowed to rewind the real adversary $\mathsf{A}_{\mathsf{real}}$ or the environment H.

2.4 Protocol analysis using Dolev-Yao sound abstraction

Now we have a look at how Universal composability or Reactive simulatability concepts can help to automatically analyze protocols.

Formal tools use a very strong assumption that cryptographic primitives are perfectly secure, therefore we can formally analyze protocols using primitive's abstraction without caring the security of their implementations. Now with Universal composability framework, we can prove that it is indistinguishable between some Dolev-Yao style abstractions and their real implementations, i.e the Dolev-Yao representation is faithful abstraction of the real one. In this case, we say that the Dolev-Yao representation is sound abstraction and by composition theorem, security of the system using the abstract representation implies the security of the system using the real one. See figure 2.1.

The idea is that we can use formal tools to automatically analyze the security of protocols in the Dolev-Yao model. As a consequence, with the sound ab-

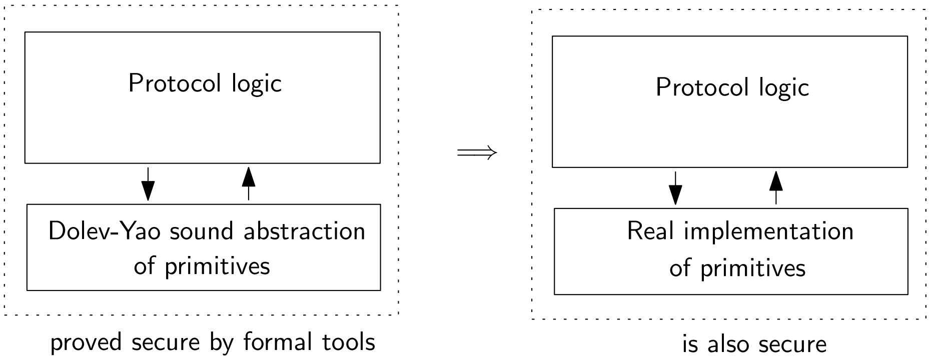

Figure 2.1: Automated protocol analysis using Dolev-Yao sound abstraction

straction, that security proof will also imply the security of the corresponding real systems.

2.5 (t, w)-threshold homomorphic encryption

We define a (t, w)-threshold homomorphic encryption scheme as a tuple of polynomial-time algorithms $\mathcal{E}\_\mathsf{thres}$ where

$$\mathcal{E}\_\mathsf{thres} = (\mathsf{FKEY}, \mathsf{E\_thres}, \mathsf{D\_thres}, \mathsf{C\_thres}, \mathsf{enc\_thres\_len}, \\ \mathsf{pke\_thres\_len}, \mathsf{ds\_thres\_len}).$$

The defined tuple $\mathcal{E}\_\mathsf{thres}$ works as follows

- For generating a public key and a list of secret share, we use the corresponding distributed key generation functionality.

  $$(pk, sk_1, ..., sk_w) \leftarrow \mathsf{FKEY}$$

  where the length of each pk is $\mathsf{pke\_thres\_len}(k) > 0$ and $sk_1, ..., sk_w$ are shares of the secret key, which will be given by FKEY to intended users securely.

  It should be noticed that we require this threshold homomorphic encryption scheme to have a secure distributed key generation protocol. Such protocols and encryption schemes can be found in [AF04, Wik04].

- For encrypting a message $m \in \{0, 1\}^+$ we use the deterministic algorithm E_thres

$$c \leftarrow \mathsf{E\_thres}_{pk}(m, r)$$

where the length of c is $\mathsf{enc\_thres\_len}(k, |m|) \geq |m| > 0$ for every $pk \in \{0,1\}^{\mathsf{pke\_thres\_len}(k)}$. r is the random coin to make the result of $\mathsf{E\_thres}()$ probabilistic. r will be part of witness for making a NIZK proof of validity.

- For generating a decryption share we use the deterministic algorithm $\mathsf{D\_thres}$

$$ds \leftarrow \mathsf{D\_thres}_{sk}(c)$$

where the length of ds is $\mathsf{ds\_thres\_len}(k, |c|) \geq |m| > 0$. $m =\downarrow$ if c is an invalid ciphertext otherwise $m \neq \downarrow$ for every correctly generated sk.

- For getting the plaintext by combining t decryption shares, we use the deterministic algorithm $\mathsf{C\_thres}$

$$m \leftarrow \mathsf{C\_thres}(ds_1, ..., ds_t)$$

where $ds_1, ..., ds_t$ are decryption shares.

- $\mathcal{E}\mathsf{\_thres}$ must have homomorphic property as follows.

  We denote M, C and R as the message, ciphertext and random coin space respectively. Assume that R is a groupoid with a groupoid operation $\circ$ and C and M are groups with group operation $\boxdot$ and $\boxplus$ respectively.

  Then for any messages m_1 and m_2, any random coins r_1 and r_2, and any valid public key pk we have

$$\mathsf{E\_thres}_{\mathsf{pk}}(m_1, r_1) \boxdot \mathsf{E\_thres}_{\mathsf{pk}}(m_2, r_2) = \mathsf{E\_thres}_{\mathsf{pk}}(m_1 \boxplus m_2, r_1 \circ r_2).$$

- $\mathcal{E}\mathsf{\_thres}$ must have simulatability of decryption share as follows.

  There is an algorithm that takes as input a ciphertext c, a message m and s secret shares and produces simulated decryption shares for all w secret-share holder so that

  – Any set of t simulated decryption shares can be combined to get m

– The set of simulated decryption shares is computationally indistinguishable from real decryption share even with knowledge of any subset of l elements of the corresponding s secret shares where $l \leq t - 1$[1].

Correctness of decryption. We need the following theorem to show that we can always get the correct plaintext even the adversary is given incorrect decryption shares.

Theorem 2.5.1. Correctness of decryption Given w decryption shares for a ciphertext c from w secret-share holders, there exists an algorithm to get the correct plaintext if the number of adversarial secret-share holder is less than $w/3$ and $t < w/3$. □

Proof. We claim that we can always find only one set of $2w/3$ decryption shares such that any t decryption shares from that set are combined to the same plaintext. Then we show that it is the correct plaintext m.

Obviously, there are at least $2w/3$ of honest secret-share holders so there exist such a set S and the corresponding plaintext is the correct one, say m. We have to show that there is no other set of $2w/3$ decryption shares such that any t decryption shares from that set are combined to another plaintext m_1.

Assume that we have another set S_1 like that. S_1 must have less than $w/3$ decryption share from honest users, otherwise S_1 contains at least one subset that contains decryption shares that are combined to m. Therefore the number of decryption share from malicious users in S_1 must be greater than $2w/3 - w/3 = w/3$. It is a contradiction.

Finally, because that set is unique, we can always build an algorithm to find it. Because t and w are constant, even a "brute force" algorithm also works in polynomial-time. □

[1] The idea here is if a simulator at least has the adversary's keys and the adversary has less than $t-1$ key shares, the simulator can always simulates the decryption shares. Basically, if the decryption-share combination algorithm is not "one way" and the encryption scheme is semantically secure, we will have this property. This property mentions here is a bit different from the property mentioned in [Gro04] as we do not require exactly $t-1$ secret share input.

2.6 Zero-knowledge and Non-interactive zero-knowledge proof systems

Informally, a proof system is a protocol for a prover P to convince a verifier V about something. The system has the *completeness* property if a prover P who has the corresponding witness can always convince V. The system has the *soundness* property if without the corresponding witness P cannot convince V.

Goldwasser, Mical and Rackoff [GMR85] have shown that it is possible to prove that some theorem is true without giving the hint. It means the verifier is convinced without getting any information of the witness. Such a system is called a *zero-knowledge proof system*. Furthermore, [GMW86] pointed that any NP language possesses zero-knowledge proof systems, under an assumption that secure encryption schemes exist.

However, the proof systems mentioned above are interactive, i.e. the prover and verifier have to interact to follow the protocol. [BFM88] then shown that we can achieve *non-interactive zero-knowledge* (nizk) proof system in the common random string model. Note that the construction here just gives us *computational* zero-knowledge property.

Recently, Groth, Ostrovsky and Sahai [GOS06] demonstrate a construction for perfect nizk proof for any NP language. This construction is used in our work.

2.7 Database notation

To describe the structure of the ideal and real libraries, we use the same notations in [BPW03a]. We will discuss them shortly here.

We use a sans serif font for $\mathsf{machines}$,$\mathsf{algorithms}$, $\mathsf{functions}$ and $\mathsf{constants}$, an italic font for *sets* and *variables* and a calligraphic font for $\mathcal{TYPES}$ and $\mathcal{INDEX}$ sets.

":=" means a deterministic assignment, "$\leftarrow$" means a probabilistic one and "$\xleftarrow{\mathcal{R}}$" means a uniform random choice from a set. When we write $x := y++$ we mean $y := y + 1$; $x := y$. We denote $g \in NEGL$ if a function $g : \mathbb{N} \to \mathbb{R}$ is negligible.

For data types, we use the following representations

- $\mathcal{TYPES}$ is the disjoint union of the following data types: $\mathcal{NAT}$ ("naturals"), $\mathcal{HNDS}$ ("handles"), $\mathcal{INDS}$ ("indices"), $\mathcal{BOOL}$ ("boolean"), $\mathcal{ERR}$ ("error"), $\mathcal{CHARSTR}$ ("charater strings"), $\mathcal{BITSTR}$ ("bit-strings"), $\mathcal{LIST}$ ("lists") and $\mathcal{NIZK}$ ("NIZK Proof").

- The types $\mathcal{NAT}$, $\mathcal{HNDS}$ ("handles") and $\mathcal{INDS}$ are isomorphic to the set of natural numbers $\mathbb{N} = \{1, 2, ...\}$. We write elements and operations for these sets in a similar way for the set of natural numbers. We also have $\mathcal{NAT}_0$ that is isomorphic to $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

- $\mathcal{BOOL} = \{\mathsf{true}, \mathsf{false}\}$ and $\mathcal{ERR} = \{\downarrow\}$. $\mathcal{CHARSTR}$ is isomorphic to Σ^+ where $\Sigma \subseteq \{a, b, .., z\} \cup \{0, 1\}$ and $\downarrow \notin \Sigma$. $\mathcal{BITSTR}$ is isomorphic to $\{0, 1\}^*$. The type $\mathcal{LIST} = \{\{0, 1\}^*\}^*$ and we write $(x^1, x^2, ..., x^n)$ for a list of $x^1, x^2, ..., x^n$. It should be noticed that "adding an element to a list" means that the element is put at the end.

- All algorithms and functions can get the error element $\downarrow$ of type $\mathcal{ERR}$ as an input and also can return it.

- We assume that elements of $\mathcal{TYPES}\backslash\mathcal{ERR}$ are uniquely encoded into the set $\{0, 1\}^*$. Also if a function is efficiently computable on abstract data types, it must be efficiently computable on encoding.

- $|w|$ denotes the length of the encoding w. We have a funtion $\mathsf{list\_len}$ such that $|(x_1, x_2, ..., x_i)| = \mathsf{list\_len}(|x_1|, |x_2|, ..., |x_i|) \geq |x_1| + |x_2| + ... + |x_i|$ for all $i \in \mathbb{N}_0$ and all $x_1, x_2, ..., x_i \in \mathcal{TYPES}\backslash\mathcal{ERR}$.

- A "database" D means a finite set of mappings from a finite subset of $\mathcal{CHARSTR}$ to $\mathcal{TYPES} \cup \{0, 1\}^*$. We call these mapping *tuples* and their arguments *attributes*. Therefore $t.a$ means the result of $t \in D$ on argument $a \in \Sigma^*$ and $t.a =\downarrow$ means $t.a$ is undefined.When we add t to the database, we write $D :\Leftarrow t$ that means $D := D \cup \{t\}$.

- There are some relational-database notations here:
  - Given a $P(A)$ that is a logical condition that has attributes of A as free variables, we define $\sigma_{P(A)}(D)$ as the set of all $t \in D$ such that $P(t)$ holds. We write $D[P(A)] := t$ if $\sigma_{P(A)}(D) = \{t\}$ and $D[P(A)] :=\downarrow$ if $|\sigma_{P(A)}(D)| \neq 1$.
  - A *key attribute* is an attribute such that $t.a$ is defined and unique for all $t \in D$. Then if a is the primary key attribute, we write $D[x]$ instead of $D[a = x]$.

Chapter 3

OVERVIEW OF THE UC CRYPTOGRAPHIC LIBRARY

This chapter's purpose is to give a broad view of real and ideal libraries and how they relate to each other before we analyze each of them in very details in next chapters. The idea of our library is mostly based on [BPW03a] but it offers different cryptographic operations. The details of both ideal and real versions will be discussed in proceeding chapters

In fact, what we can have in practice is just the real library. However, we need to define an ideal one to prove that, our real system is at least as secure as the ideal one. We will prove that the real system can UC realize the ideal one, meaning what ever can happen to the real adversary can also happen to the ideal adversary in an arbitrary environment.

3.1 The ideal system

The behaviours of the ideal system show abstractly how we want our real system work.

In principle, the ideal library offers cryptographic commands that are applied to abstract terms. This version of the library includes specific commands for an IND-TCPA threshold homomorphic encryption, while the library in [BPW03a] contains signature and IND-CCA2 public encryption. The commands have simple and deterministic semantics, which are based on the system's states. The ideal system stores its states in a database.

The database creates its entries when a certain command is called and may output the new handle to users. Each entry in the database contains information about its type and pointers to its arguments. It also has handles, under which different users know the entry. In addition, users work with handles, which point to the terms they know, not directly with terms.

The system must be "trusted". In other words it is impossible for the users to cheat. It means that if a user encrypts a message m, the ideal library will return the handle for the ciphertext. Someone can decrypt only if he has the handles to both the ciphertext and the secret key.

In addition, the library offers send commands, which allow to send data to other users. There are three types of communication channels: secure, authentic and insecure, which are denoted $\{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$ respectively. After a successful send command, the receiver gets a handle to the sent term and a new handle for the receiver will also be added to the term's entry .

Backes et al. in [BPW03a] pointed the following differences between this library and the standard Dolev-Yao model:

- Encryption scheme must be probabilistic. It means that if we encrypt the same message twice we get different ciphertexts.
- Because of the commitment problem [BPW06a], correct parties are not allowed to send the secret key.
- Parties can always get the length of any message.
- Adversary can create invalid data, or "garbage".

In order to make the real system realize the ideal correctly, we have bounds for the length of an arbitrary message and also the number of input messages.

3.2 The real system

The real library works similarly to a cryptographic library but in the real life. The point is that we want our real library to be able replace the ideal one,i.e it must be similar to and at least as secure as the ideal system.

As a consequence, it offers the same commands to users as the ideal one does. However, here abstract terms are replaced by bitstrings, or real values. Also cryptographic objects are put in different databases in different machines. Thess machines communicate via different types of channels: secure, authentic and insecure.

In the real system, the adversary has only the ability to eavesdrop insecure channel and control the corrupted parties. He also can always modify a message in an insecure channel and forward to the intended recipient.

To make the real system "at least as secure as" the ideal one, we use some encryption scheme for communication between machines. However, we need some technical requirements according to robust protocol design [BPW03a]:

- All objects have a tag with a type field. Therefore, a ciphertext can never be misinterpreted as a public key.

Chapter 4

THE IDEAL LIBRARY

In this chapter we explain the details of the ideal system. Intuitively, the ideal system is the specification of a secure real cryptographic library. Although it is unreal, it serves the purpose of comparison with a real one. A real system is secure if it is indistinguishable from the ideal one.

We will describe the structure, including a threshold homomorphic encryption. Then we will define how we want it to work ideally, and also how users and an adversary can work with this library.

To describe the ideal system, we use notations in [PW01] and mainly follow the outline of the previous library proposed in [BPW03a].

4.1 Structure

First we define the overall structure of an ideal library. Given a number n of participants and a tuple L of parameters (Section 4.2 discusses more about L), the form of the ideal system is

$$Sys_{n,L}^{\mathsf{cry,id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})|\mathcal{H} \subseteq \{1, ..., n\}\}$$

where $\mathcal{H}$ denotes the set of honest participants.

Now we define how $\mathsf{TH}_{\mathcal{H}}$ communicates with other parties. There are two versions of this system, $Sys_{n,L}^{\mathsf{cry,id,stan}}$ and $Sys_{n,L}^{\mathsf{cry,id,loc}}$, which are stand-alone and localized respectively. The former's inputs and outputs for the users are scheduled by the adversary, the latter's inputs and outputs are not (in this version the library could be used locally as subroutines by protocols using it). In each version, $\mathsf{TH}_{\mathcal{H}}$ has the following intended ports for the users

$$userports_{\mathcal{H}}^{\mathsf{stan}} := \{\mathsf{in}_u?, \mathsf{out}_u!|u \in \mathcal{H}\};$$
$$userports_{\mathcal{H}}^{\mathsf{loc}} := \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!|u \in \mathcal{H}\}.$$

Users also have corresponding ports that connect to those ports respectively

$$S_{\mathcal{H}}^{\mathsf{stan}} := \{\mathsf{in}_u!, \mathsf{out}_u?|u \in \mathcal{H}\};$$
$$S_{\mathcal{H}}^{\mathsf{loc}} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}!|u \in \mathcal{H}\}.$$

For communicating with the adversary, $\mathsf{TH}_{\mathcal{H}}$ offers 2 ports $\mathsf{in_a}?$ and $\mathsf{out_a}!$. To allow the adversary to schedule messages even on secure channels, we define 3 sets of channels

$$ch\_honest := \{(u, v, x)|u, v \in \mathcal{H} \wedge x \in \{\mathsf{s,a}\}\}$$
$$ch\_from\_adv := \{(u, v, x)|v \in \mathcal{H} \wedge (u \notin \mathcal{H} \vee x \in \{\mathsf{s,a}\})\}$$
$$ch\_to\_adv := \{(u, v, x)|u \in \mathcal{H} \wedge (v \notin \mathcal{H} \vee x \in \{\mathsf{a,i}\})\}$$

where $\{\mathsf{s,a,i}\}$ are secure, authentic and insecure channels respectively.

Remark *4.1.1.* We require a condition that honest user always use *ch_honest* to send decryption shares. ◇

Figure 4.1 shows the stand-alone version of $\mathsf{TH}_{\mathcal{H}}$. Dotted arrows are clock control. Message between honest users will be put in the channel $\mathsf{net\_id_{u,v,x}}$. The adversary can schedule messages here even if they are sent securely. The localized version is shown in [BPW03a].

4.2 System parameters

We do not hide the length of messages as in the real system anyone can get the length of any message. Therefore $\mathsf{TH}_{\mathcal{H}}$ has the following functions with the domain $\mathbb{N}_0$ and the range $\mathbb{N}$: $\mathsf{data\_len}^*(l)$, $\mathsf{list\_len}^*(l_1, ..., l_j)$ for all $j \in \mathbb{N}_0$, $\mathsf{nonce\_len}^*(k)$, $\mathsf{nizk\_len}^*(k, l)$, $\mathsf{enc\_thres\_len}^*(k, l)$, $\mathsf{pke\_thres\_len}^*(k)$ and $\mathsf{ds\_thres\_len}^*(k, cl)$ (length of decryption share) where l and l_i's are lengths of messages, cl is length of ciphertext.

Also, we bound the length of messages so that the real system can realize the ideal one. Given k as the security parameter, $\mathsf{max\_len}(k)$, $\mathsf{max\_in}(k)$ are

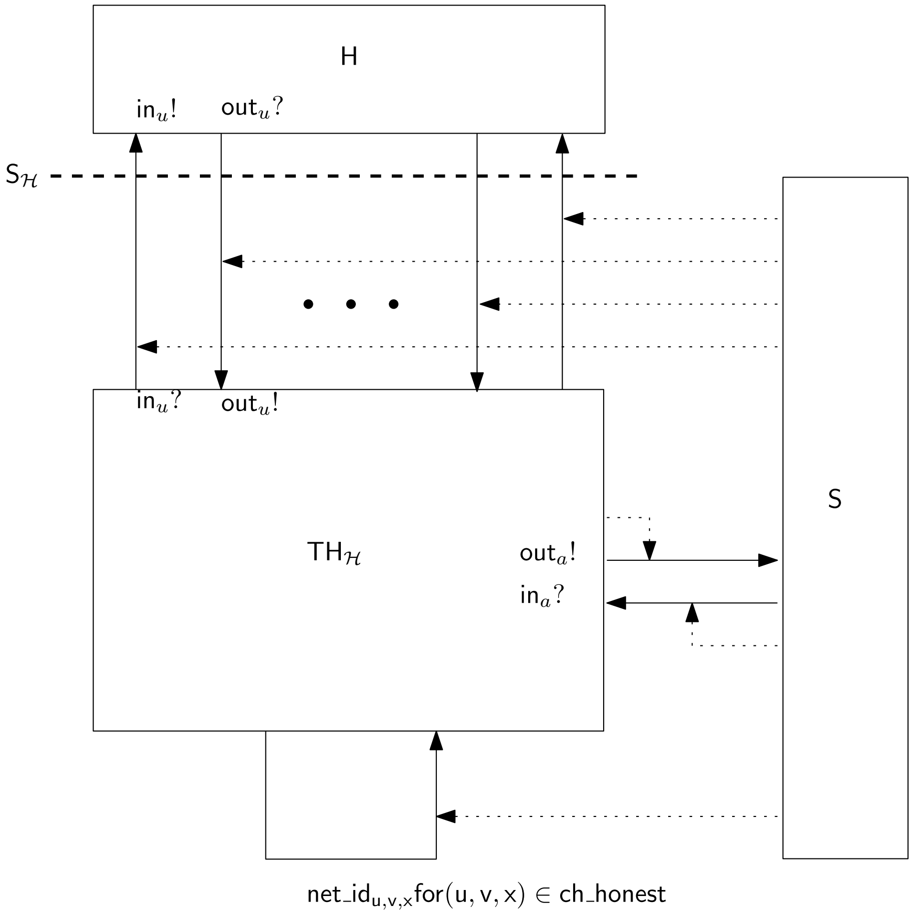

Figure 4.1: Localized version of $\mathsf{TH}_{\mathcal{H}}$

bounds for the length of messages and the maximum number of inputs at each port of correct machines, respectively. Basically, to avoid unrealistic cases, we must have: for all $k \in \mathbb{N}$ that $\mathsf{max\_len}(k) > 3$; $\mathsf{nonce\_len}^*(k)$ and $\mathsf{pke\_thres\_len}^*(k) < \mathsf{max\_len}(k)$.

We want users to encrypt only valid message, because it is essential in applications such as e-voting. For this purpose, we use a function $\mathsf{valid}^*(m) : \mathcal{BITSTR} \rightarrow \mathcal{BOOL}$ to check if m is valid or not. If m is valid then $\mathsf{valid}^*(m)$ returns true, otherwise it returns false.

The system parameter L is the tuple of all these functions. Every function here must be bounded by a polynomial and be efficiently computable.

4.3 States

The ideal system must store the current working information, or its state. The state of $\mathsf{TH}_{\mathcal{H}}$ is made of a database D and the variables *size* (to keep the current size of the database), $curhnd_u$ for each $u \in \mathcal{H} \cup \{\mathsf{a}\}$ (to index handles for user u) and $step_{p?}$ for each input port $p?$ (to count the number of of inputs has been put at port $p?$).

4.3.1 Database D

$\mathsf{TH}_{\mathcal{H}}$ uses a database D to keep abstract representations of data produced during a system run. Each entry in D consists the following attributes.

$$(ind, type, arg, hnd_{u_1}, ..., hnd_{u_n}, hnd_{\mathsf{a}}, len)$$

where $\mathcal{H} = \{u_1, ..., u_n\}$.

In more details, for each $x \in D$ we have:

- $x.ind \in \mathcal{INDS}$ is the index, consecutively numbers all entries. We use this attribute as the primary key so we write $D[i]$ for $D[ind = i]$. We use superscript "ind" to present an index. See Section 4.3.2 for more details.
- $x.type \in typeset = \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}, \mathsf{ske}, \mathsf{pke}, \mathsf{enc}, \mathsf{nizk}, \mathsf{decshr}, \mathsf{garbage}\}$ indicates the type of x.
- $x.arg = (a_1, ..., a_j)$ is a sequence of arguments. Each a_i can possibly be an index of another entry in D, therefore has the type $\mathcal{INDS}$.
- $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ where a stands for the adversary. If $x.hnd_u \neq \downarrow$ then $x.hnd_u$ is the *handle* for the corresponding user to the entry x. We user superscript "hnd" to present a handle and denote a handle to an entry $D[i]$ by i^{hnd}. See Section 4.3.2 for more details.
- $x.len \in \mathbb{N}$ denotes the length of entry x in the sense of Section 4.2.

4.3.2 Conventions about Indices and Handles

$\mathsf{TH}_{\mathcal{H}}$ uses a variable *size* to store the current number of element in the database D. Initially D is empty and $size = 0$. When a new entry x

is added, $size := size + 1$ and $x.ind := size$. After that, $x.ind$ is never changed.

For each $u \in \mathcal{H} \cup \{\mathsf{a}\}$, $\mathsf{TH}_{\mathcal{H}}$ uses a variable $curhnd_u \in \mathcal{HNDS}$ initialized with 0. When assigning a new handle for u, $\mathsf{TH}_{\mathcal{H}}$ sets $curhnd_u := curhnd_u + 1$ and sets the new handle $i^{\mathsf{hnd}} := curhnd_u$. By this way, $\mathsf{TH}_{\mathcal{H}}$ keeps handles for u uniquely and consecutively assigned. Also $\mathsf{TH}_{\mathcal{H}}$ uses the algorithm $\mathsf{ind2hnd}_u$ to determine a handle i^{hnd} for u to an entry $D[i]$. $\mathsf{ind2hnd}_u$ works as follows: On input $i \in \mathcal{INDS}$, return $D[i].hnd_u$ if it is not $\downarrow$. Otherwise set $curhnd_u := curhnd_u + 1$ and return $D[i].hnd_u := curhnd_u$. For a list, $\mathsf{TH}_{\mathcal{H}}$ uses another algorithm $\mathsf{ind2hnd}_u^*$ which applies $\mathsf{ind2hnd}_u$ on each element.

4.3.3 Input Counters

For each input port $\mathsf{p}?$, $\mathsf{TH}_{\mathcal{H}}$ uses a variable $steps_{\mathsf{p}?}$ to count the number of inputs at that port. These counter must not exceed a bound $bound_{\mathsf{p}?}$ to ensure polynomial runtime of $\mathsf{TH}_{\mathcal{H}}$. This bound is $\mathsf{max\_in}(k)$ for all port except $bound_{\mathsf{in_a}?}$ is $(2w + 10)n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ where w is the number of secret share per generation[1].

4.4 Inputs and their Evaluation

In this section we define how users can use the ideal system. Users and the adversary interact with the ideal system by using commands it offers. All possible operations can be done by calling commands, i.e encrypting, storing, sending, etc. It should be noticed that in the ideal system, all data are abstract terms and users keep the handles, which point to the corresponding abstract terms, not the terms themselves.

And because the ideal system is exactly what we desire a real system to be, the ideal one offers restricted extra commands for the adversary, namely some abilities the adversary can always do in a real environment, such as creating invalid data.

4.4.1 Overview of commands and possible inputs

There are some commands offered to users and they call them through input ports. Each input c to a port $\mathsf{in}_u?$ where $u \in \mathcal{H} \cup \{\mathsf{a}\}$ should be a list

[1]We explain later in the proof of Lemma 6.4.2.

$(cmd, x_1, ..., x_j)$. Then we write $y \leftarrow cmd(x_1, ..., x_j)$ meaning that y is the result that $\mathsf{TH}_{\mathcal{H}}$ returns at $\mathsf{out}_u!$.

The value *cmd* could be one of string here bellow:

$$basic\_cmds := \{\mathsf{get\_type}, \mathsf{get\_len}, \mathsf{store}, \mathsf{retrieve}, \mathsf{list}, \mathsf{list\_proj}, \mathsf{gen\_nonce}, \mathsf{gen\_enc\_thres\_keylist}, \mathsf{encrypt\_thres}, \mathsf{decrypt\_thres}, \mathsf{combine\_thres}, \mathsf{pk\_of\_enc\_thres}\};$$

$$send\_cmds := \{\mathsf{send\_s}, \mathsf{send\_a}, \mathsf{send\_i}\};$$

$$adv\_local\_cmds := \{\mathsf{adv\_garbage}, \mathsf{adv\_parse}, \mathsf{adv\_invalid\_ciph\_thres}, \mathsf{adv\_decrypt\_thres}, \mathsf{adv\_invalid\_share\_thres}\};$$

$$adv\_send\_cmds := \{\mathsf{adv\_send\_s}, \mathsf{adv\_send\_a}, \mathsf{adv\_send\_i}\}.$$

More specifically, if $u \neq \mathsf{a}$ then *cmd* must be in $basic\_cmds \cup send\_cmds$, otherwise it must be in $basic\_cmds \cup adv\_local\_cmds \cup adv\_send\_cmds$. Each of basic command represents one cryptographic operation. More details about commands are given in next sections.

Inputs at a port $\mathsf{net\_id}_{u,v,x}?$ is from $\mathsf{TH}_{\mathcal{H}}$ itself.

4.4.2 General working conventions

Some remarks:

- There are some commands to store real-world messages and users can retrieve them later via handles. Other commands work only with handles.
- Only lists are allowed to be sent by send commands. It does not reduce the generality but we can control that no secret information is sent with messages.
- $\mathsf{TH}_{\mathcal{H}}$ offers all types of channels, i.e. $\{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$, to participants[2].

General working conventions Upon each input at a port $\mathsf{p}?$, first $\mathsf{TH}_{\mathcal{H}}$ increments the counter $steps_{\mathsf{p}?}$. After that $\mathsf{TH}_{\mathcal{H}}$ uses the length function

[2]Except for the Remark 4.1.1

to check how many bits of input can be accepted at each port. This value depends on the current state. Once $steps_{\mathsf{p}?}$ has reached the bound $bound_{\mathsf{p}?}$, the length function for this port always returns 0, meaning that no input is accepted.

An input is called *well-formed* when it is in the correct domain. If it is not, then $\mathsf{TH}_{\mathcal{H}}$ can always verify and then aborts the transition. From now we consider only *well-formed* inputs.

For every output $\mathsf{out}_u!$ where $u \in \mathcal{H} \cup \{\mathsf{a}\}$, the localized version $\mathsf{TH}_{\mathcal{H}}^{\mathsf{loc}}$ schedules itself by making a clock output 1 at $\mathsf{out}_u{}^{\triangleleft}!$. The stand-alone version $\mathsf{TH}_{\mathcal{H}}^{\mathsf{loc}}$ does it only when $u = \mathsf{a}$.

4.4.3 Basic commands

In this section, we will list the basic commands that are offered to users. The basic commands cover all cryptographic operations that users can do with this library. We also define how $\mathsf{TH}_{\mathcal{H}}$ evaluates each command entered at port $\mathsf{in}_u?$ where $u \in \mathcal{H} \cup \{\mathsf{a}\}$. Most of these commands produce a result at the output port $\mathsf{out}_u!$ only, except the distributed key generation command will send secret-key shares to different output ports.[3] Each command may update the database D but does not touch any handle which does not belong to u.

Type and length queries

- *Type query*: $t \leftarrow \mathsf{get\_type}(x^{\mathsf{hnd}})$.
  Set $t := D[hnd_u = x^{\mathsf{hnd}}].type$.
- *Length query*: $l \leftarrow \mathsf{get\_len}(x^{\mathsf{hnd}})$.
  Set $l := D[hnd_u = x^{\mathsf{hnd}}].len$.

Storing and retrieving data $\mathsf{TH}_{\mathcal{H}}$ stores a string and give back to user u a handle that points to the entry. When a user wants to retrieve the string, that user must give a correct handle for it.

- *Storing*: $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$.

[3] This is a different point in this library compared to the original one in [BPW03a].

– If $i := D[type = \mathsf{data} \wedge arg = (m)].ind \neq \downarrow$ then return $m^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$.[4]

– Otherwise

* if $\mathsf{data\_len}^*(|\mathsf{m}|) > \mathsf{max\_len}(\mathsf{k})$ return $\downarrow$
* else set $m^{\mathsf{hnd}} := curhnd_u + +$; and

$$D :\Leftarrow (ind := size + +, type := \mathsf{data}, arg := (m), hnd_u := m^{\mathsf{hnd}}, len := \mathsf{data\_len}^*(|m|)).$$

- *Retrieval*: $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  $m := D[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].arg[1]$.

Lists Lists are combination of data sent to other players. Most of data can be included in a list, except secret shares because that kind of information is not allowed to be sent.

- *Generating a list*: $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, ..., x_j^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.

  – Set $x_i := D[hnd_u = x_i^{\mathsf{hnd}}].ind$ for $i = 1, ..., j$. If any $D[x_i].type \in \{\mathsf{ske}\}$, then set $l^{\mathsf{hnd}} :=\downarrow$.[5]

  – If $l := D[type = \mathsf{list} \wedge arg = (x_1, ..., x_j)].ind \neq \downarrow$, then return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.

  – Otherwise set $length := \mathsf{list\_len}^*(D[x_1].len, ..., D[x_j].len)$. If $length > \mathsf{max\_len}(k)$ then return $\downarrow$ else:

  * set $l^{\mathsf{hnd}} := curhnd_u + +$; and

$$D :\Leftarrow (ind := size + +, type := \mathsf{list}, arg := (x_1, ..., x_j), hnd_u := l^{\mathsf{hnd}}, len := length).$$

- *i-th element projection*: $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $0 \leq i \leq \mathsf{max\_len}(k)$.

  If $D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].arg = (x_1, ..., x_j)$ with $i \leq j$ then $x^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(x_i)$. Else $x^{\mathsf{hnd}} :=\downarrow$.

[4]The idea here is $\mathsf{TH}_{\mathcal{H}}$ uses 1 entry for all identical pieces of data.

[5]So we can guarantee that no secret-key share is sent to other parties.

Nonces $\mathsf{TH}_{\mathcal{H}}$ simply makes a new entry.

- *Generating a nonce*: $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Set $n^{\mathsf{hnd}} := curhnd_u + +$ and

  $$D :\Leftarrow (ind := size + +, type := \mathsf{nonce}, arg := (), hnd_u := n^{\mathsf{hnd}}, len := \mathsf{nonce\_len}^*(k)).$$

(t, w)-threshold homomorphic encryption with NIZK proof of plaintext validity with $(t-1) < w/3$. These are the main cryptographic operations offered in this library.

- *Key generation:*$pk^{\mathsf{hnd}} \leftarrow \mathsf{gen\_enc\_thres\_keylist}(u_1, ..., u_j)$

  for $(w-t) < j \leq w$, where $\{u_1, ..., u_j\}$ are honest users who will receive a secret-key share for each one. The rest of secret-key shares will belong to the adversary.

  If any $\{u_1, ..., u_j\} \backslash \mathcal{H} \neq \emptyset$ then set $pk^{hnd} :=\downarrow$. Else

  - Set $pk^{\mathsf{hnd}} := curhnd_u + +$, $pk^{\mathsf{hnd}}_{u_i} := curhnd_{u_i} + +$ for $i \in \{1, ..., j\}$, $pk^{\mathsf{hnd}}_{\mathsf{a}} := curhnd_{\mathsf{a}} + +$, and

    $$D :\Leftarrow (ind := size + +, type := \mathsf{pke}, arg := (u_1, ..., u_j), hnd_u := pk^{\mathsf{hnd}}, hnd_{\mathsf{a}} := pk^{\mathsf{hnd}}_{\mathsf{a}}, len := \mathsf{pke\_thres\_len}^*(k)).$$

    . Let $pk^{\mathsf{ind}} := size$.

  - For $i \in \{j+1, ..., w\}$
    - Set $sk^{\mathsf{hnd}}_i := curhnd_{\mathsf{a}} + +$; and

      $$D :\Leftarrow (ind := size + +, type := \mathsf{ske}, arg := (pk^{\mathsf{ind}}), hnd_{\mathsf{a}} := sk^{\mathsf{hnd}}, len := 0).$$

  - Output $(u, pk^{\mathsf{hnd}}_{\mathsf{a}}, sk^{\mathsf{hnd}}_{j+1}, ..., sk^{\mathsf{hnd}}_w)$ to the port $\mathsf{out_a}!$[6].

- *Encryption:*$(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{encrypt\_thres}(pk^{\mathsf{hnd}}, m^{\mathsf{hnd}})$ where p^{hnd} is the NIZK proof of plaintext validity.

  Set $pk := D[hnd_u = pk^{\mathsf{hnd}} \wedge type = \mathsf{pke}].ind$ and $m := D[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].arg[1]$ and $length := \mathsf{enc\_thres\_len}^*(k, D[m].len)$. If $length > \mathsf{max\_len}(k)$ or $pk =\downarrow$ or $m =\downarrow$ then return $\downarrow$. Otherwise check if $\mathsf{valid}^*(D[m].arg) = \mathsf{false}$ then return $\downarrow$. Else

[6]Send to the adversary public information along with his secret share.

– Set $c^{\mathsf{hnd}} := curhnd_u++$, $p^{\mathsf{hnd}} := curhnd_u++$ and $c_{\mathsf{a}}^{\mathsf{hnd}} := curhnd_{\mathsf{a}}++$, $p_{\mathsf{a}}^{\mathsf{hnd}} := curhnd_{\mathsf{a}}++$; and

$$D :\Leftarrow (ind := size++, type := \mathsf{enc}, arg := (pk, m), hnd_u := c^{\mathsf{hnd}},\\ hnd_{\mathsf{a}} := c_{\mathsf{a}}^{\mathsf{hnd}}, len := length).$$

and

$$D :\Leftarrow (ind := size++, type := \mathsf{nizk}, arg := (size - 1), hnd_u := p^{\mathsf{hnd}},\\ hnd_{\mathsf{a}} := p_{\mathsf{a}}^{\mathsf{hnd}}, len := \mathsf{nizk\_len}^*(k, D[m].len));$$

– Output $(u, c_{\mathsf{a}}^{\mathsf{hnd}}, p_{\mathsf{a}}^{\mathsf{hnd}})$ to the port $\mathsf{out_a}!$, expecting $(\mathsf{nizkaccept}, u, p_{\mathsf{a}}^{\mathsf{hnd}})$ from the port $\mathsf{in_a}?$[7]. Finally return.

- *Generating a decryption share:* $ds^{\mathsf{hnd}} \leftarrow \mathsf{decrypt\_thres}(sk^{hnd}, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}, p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}})$ where $\{c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}\}$ is a set of ciphertexts and $\{p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}}\}$ is a set of corresponding proofs for plaintext validity. This command returns a decryption share of a possible ciphertext of the plaintext, which is the result of the algebraic operation on all original plaintexts[8].

  – Set $sk := D[hnd_u = sk^{\mathsf{hnd}} \wedge type = \mathsf{ske}].ind$ and $c_i := D[hnd_u = c_i^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$ and $p_i := D[hnd_u = p_i^{\mathsf{hnd}} \wedge type = \mathsf{nizk}].ind$ and $m_i := D[c_i].arg[2]$ for all $i \in \{1, ..., j\}$.
  If any $D[p_i].arg = \downarrow$[9] then output $(u, \mathsf{witness}, c^{\mathsf{hnd}}, p^{\mathsf{hnd}})$ to the port $\mathsf{out_a}!$, expecting $(u, \mathsf{witness}, m_i^{\mathsf{hnd}})$. If arg of c_i^{hnd} contains the index of m_i^{hnd} then set $D[p_i].arg := D[hnd_{\mathsf{a}} = m_i^{\mathsf{hnd}}].ind$, else return $\downarrow$[10].

  – Return $\downarrow$ if $sk = \downarrow$ or any $c_i = \downarrow$ or any $p_i = \downarrow$.

  – Return $\downarrow$ if $D[c_i].arg[1] \neq D[sk].arg[1]$ or $D[p_i].arg[1] \neq c_i$ for any i.

  – Return $\downarrow$ if any $m_i = \downarrow$.[11]

  – Else

    * Set $m\_arg := D[m_1].arg[1] \boxplus \cdots \boxplus D[m_j].arg[1]);$

[7] We have assumed that the adversary replies immediately in this case.

[8] Users can always make decryption shares for only one ciphertext. However, this command gives them more power to encrypt a combination of ciphertext as well. Note that users can not combine ciphertexts first then decrypt, because they can not give the corresponding proof of validity.

[9] Invalid nizk.

[10] The adversary may have inserted invalid ciphertexts to the database together with invalid proofs.

[11] As some ciphertexts may be results of the command adv_invalid_ciph_thres.

* Set $m\_ind := D[type = \mathsf{data} \wedge arg = (m\_arg)].ind$. If $m\_ind = \downarrow$ then
  * $D :\Leftarrow (ind := size++, type := \mathsf{data}, arg := (m\_arg), len := \mathsf{data\_len}^*(|m\_arg|))$;[12]
  * Set $m\_ind := size$;
* Set $ds^{\mathsf{hnd}} := curhnd_u ++$ and

$$D :\Leftarrow (ind := size++, type := \mathsf{decshr}, arg := (m\_ind, c_1, ..., c_j), hnd_{\mathsf{u}} := ds^{\mathsf{hnd}}, len := \mathsf{ds\_thres\_len}^*(k, \mathsf{enc\_thres\_len}^*(k, D[m].len))).$$

- *Combining decryption shares:* $m^{\mathsf{hnd}} \leftarrow \mathsf{combine\_thres}(ds_1^{\mathsf{hnd}}, ..., ds_w^{\mathsf{hnd}})$ where $\{ds_1^{\mathsf{hnd}}, ..., ds_w^{\mathsf{hnd}}\}$ is the set of all w decryption shares

  Let $ds_i := D[hnd_u = ds_i^{\mathsf{hnd}} \wedge type = \mathsf{decshr}].ind$ for all $i \in \{1, ..., t\}$.

  First, check if all of ds_i are decryption shares for the same combination of ciphertexts[13]. If not then return $\downarrow$.

  Otherwise, among all of $D[ds_i].arg[1]$, find the majority which share the same value, say m. If the majority has less than $2w/3$ elements then return $\downarrow$[14]. Otherwise return $m^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(m)$ [15].

- *Public key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_enc\_thres}(c^{\mathsf{hnd}})$.

  Let $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$. Return $\downarrow$ if $c = \downarrow$. Otherwise let $pk := D[c].arg[1]$ and return $pk^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(pk)$.

4.4.4 Adversarial commands

Now we see the power of the adversary in this ideal system. The adversary can use any command that an honest user can and some extra commands. The following commands are offered only to the adversary.

[12]Note the for this new data, no users has handle to it at this moment. It means that even some users have decryption shares, they still do not know the plaintext until they combine them.

[13]We even accept invalid decryption shares made by the adversary but they should point to the same combination of ciphertexts.

[14]In this case we do not have enough correct decryption share to continue.

[15]Note that honest users must use authentic channels to send their decryption share according to Remark 4.1.1

General adversarial commands

- *Creating invalid entry*: $y^{\mathsf{hnd}} \leftarrow \mathsf{adv\_garbage}(l)$ for $l \in \mathbb{N}$ and $l \leq \mathsf{max\_len}(k)$.[16]

  Set $y^{\mathsf{hnd}} := curhnd_{\mathsf{a}}++$ and

  $$D :\Leftarrow (ind := size++, type := \mathsf{garbage}, arg := (), hnd_{\mathsf{a}} := y^{\mathsf{hnd}}, \\ len := l).$$

- *Retrieving parameters*: $(type, arg) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$ where m^{hnd} is a handle that points to any kind of data.

  Let $m := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].ind$ and set $type := D[m].type$. In most cases, set $arg := \mathsf{ind2hnd^*}_{\mathsf{a}}(D[m].arg)$. There are two exceptions here.

  - When $type = \mathsf{enc}$ and $D[m].arg$ is of the form (pk, l) (a valid ciphertext) and the adversary has less than t secret shares in D[17], set $arg := (\mathsf{ind2hnd_a}(pk), D[l].len)$.
  - When $type = \mathsf{decshr}$ and $D[m].arg$ is of the form $(l, c_1, ..., c_j)$ (a valid decryption share) and the sum of the number of valid decryption shares from the same ciphertext and the number of secret shares that the adversary has is less than $t - 1$[18]. In this case, set $arg := (c_1, ..., c_j)$.

Advesarial commands for (t, n) threshold homomorphic encryption + NIZK proof of plaintext validity In the following we denote ϵ for unknown plaintext.

- *Invalid key generation:*$pk^{\mathsf{hnd}} \leftarrow \mathsf{adv\_gen\_enc}()$

  Set $pk^{\mathsf{hnd}} := curhnd_{\mathsf{a}}++$ and

  $$D :\Leftarrow (ind := size++, type := \mathsf{pke}, arg := (), hnd_{\mathsf{a}} := pk^{\mathsf{hnd}}, \\ len := \mathsf{pke\_thres\_len}^*(k)).$$

- *Creating invalid ciphertext of length l*
  $(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{adv\_invalid\_ciph\_thres}(pk^{\mathsf{hnd}}, l)$ for $1 \leq l \leq \mathsf{max\_len}(k)$ where p^{hnd} is also an invalid NIZK proof of plaintext validity.

[16] l is the data length.
[17] Key generation has been done by an honest user
[18] It means the adversary has had enough elements to get the plaintext.

Set $pk := D[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}} \wedge type = \mathsf{pke}].ind$. If $pk = \downarrow$ then return $\downarrow$. Otherwise set $c^{\mathsf{hnd}} := curhnd_a{+}{+}$ and $p^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{+}{+}$ and

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{enc}, arg := (pk), hnd_{\mathsf{a}} := c^{\mathsf{hnd}},\\ len := l).$$

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{nizk}, arg := (), hnd_{\mathsf{a}} := p^{\mathsf{hnd}},\\ len := \mathsf{nizk\_len}^*(k, l)).$$

- *Adversarial decryption:* $ds^{hnd} \leftarrow \mathsf{adv\_decrypt\_thres}(sk^{hnd}, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}})$ where $\{c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}\}$ is a set of ciphertexts. It should be noticed that the adversary does not need a list of the corresponding NIZK proofs of plaintext validity.
  - Set $sk := D[hnd_{\mathsf{a}} = sk^{\mathsf{hnd}} \wedge type = \mathsf{ske}].ind$ and $c_i := D[hnd_{\mathsf{a}} = c_i^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$ and $m_i := D[c_i].arg[2]$ for all $i \in \{1, ..., j\}$
  - Return $\downarrow$ if $sk = \downarrow$ or any $c_i = \downarrow$.
  - Return $\downarrow$ if $D[c_i].arg[1] \neq D[sk].arg[1]$ for any i.
  - Return $\downarrow$ if any $m_i = \downarrow$.[19]
  - Else
    - Let $m\_arg := \mathsf{multiply}(D[m_1].arg[1], ..., D[m_j].arg[1])$.
    - Let $m\_ind := D[type = \mathsf{data} \wedge arg = (m\_arg)].ind$. If $m\_ind = \downarrow$ then
      - $D :\Leftarrow (ind := size{+}{+}, type := \mathsf{data}, arg := (m\_arg), len := \mathsf{data\_len}^*(|m\_arg|))$;[20]
      - Set $m\_ind := size$.
    - Set $ds^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{+}{+}$ and

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{decshr}, arg := (m\_ind, c_1, ..., c_j),\\ hnd_{\mathsf{u}} := ds^{\mathsf{hnd}},\\ len := \mathsf{ds\_thres\_len}^*(k, \mathsf{enc\_thres\_len}^*(k, D[m].len))).$$

- *Creating invalid decryption-share for a plaintext of length l:* $ds^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_decshr}(l, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}})$ for all $i \in \{1, ..., j\}$.

[19] As some ciphertexts may be results of the command adv_invalid_ciph_thres.

[20] Note the for this new data, the adversary does not have handle to it at this moment. It means that even he has this decryption share, they still do not know the plaintext.

Let $c_i := D[hnd_u = c_i^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$ for all $i \in \{1, ..., j\}$. Return $\downarrow$ if any $c_i = \downarrow$.

Set $ds^{\mathsf{hnd}} := curhnd_{\mathsf{a}}++$ and

$$D :\Leftarrow (ind := size++, type := \mathsf{decshr}, arg := (\epsilon, c_1, ...c_j),$$
$$hnd_{\mathsf{a}} := ds^{\mathsf{hnd}}, len := \mathsf{ds\_thres\_len}^*(k, \mathsf{enc\_thres\_len}^*(k, l))).$$

- *Unblocking an honest user to receive his secret share*: $\mathsf{learn\_share}(pk_{\mathsf{a}}^{\mathsf{hnd}}$ $, u_i)$

  Let $pk^{\mathsf{ind}} := D[hnd_{\mathsf{a}} = pk_{\mathsf{a}}^{\mathsf{hnd}}].ind$. If u_i is not in $D[pk^{\mathsf{ind}}].arg$ then return.

  Otherwise let $pk_{u_i}^{\mathsf{hnd}} := curhnd_{u_i}$++ the $D[pk^{\mathsf{ind}}].hnd_{u_i} := pk_{u_i}^{\mathsf{hnd}}$.

  Then set $sk_{u_i}^{\mathsf{hnd}} := curhnd_{u_i}++$; and

  $$D :\Leftarrow (ind := size++, type := \mathsf{ske}, arg := (pk^{\mathsf{ind}}), hnd_{u_i} := sk_{u_i}^{\mathsf{hnd}},$$
  $$len := 0).$$

  Finally output $(u, pk_{u_i}^{\mathsf{hnd}}, sk^{\mathsf{hnd}})$ to the port $\mathsf{out}_{u_i}!$.

4.4.5 Send commands

Users send lists to others. All of messages from one honest user to another in secure or authentic channels are scheduled by the adversary. Messages to or from the adversary are output immediately.

- $\mathsf{send\_}x(v, l^{\mathsf{hnd}})$ for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$ and $v \in \{1, ..., n\}$

  Let $l := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l \neq \downarrow$ then

  - If $(u, v, x) \in ch\_honest$ then output l at $\mathsf{net\_id}_{u,v,x}!$.[21]
  - If $(u, v, x) \in ch\_to\_adv$ then output $(u, v, x, \mathsf{ind2hnd_a}(l))$ at $\mathsf{out_a}!$.[22]

- $\mathsf{adv\_send\_}x(u, v, l^{\mathsf{hnd}})$ for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$, $u \in \{1, ..., n\}$ and $v \in \mathcal{H}$. This command is for the adversary only, i.e. this command can be called only at the port $\mathsf{in_a}?$. The adversary uses this command to pretend u sent a message to v.

  Let $l := D[hnd_a = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l \neq \downarrow$ and $(u, v, x) \in ch\_from\_adv$ then output $(u, v, x, \mathsf{ind2hnd_v}(l))$ at $\mathsf{out}_v!$

[21]The message will be sent to $\mathsf{out_v}!$ later on. See Section 4.4.6.

[22]Note that both of these cases may happen for the same message, for example anauthentic channel.

4.4.6 Inputs from Secure channels

On input l at port $\mathsf{net\_id}_{u,v,x}?$ for $l \in \mathcal{INDS}$ where $l \leq size$:

$\mathsf{TH}_{\mathcal{H}}$ outputs $(u, x, \mathsf{ind2hnd_v}(l))$ at $\mathsf{out}_v!$

4.5 Properties of the Ideal library

We prove that the ideal library has some properties by giving the following lemmas. We need these lemma later in the security proof in Chapter 7. For every entry x in database D, let $\mathsf{owners}(x) := \{u \in \mathcal{H} \cup \{\mathsf{a}\} \,|\, x.hnd_u \neq \downarrow\}$.

Lemma 4.5.1. The ideal systems $Sys_{n,L}^{\mathsf{cry,id},t}$ for $t \in \{\mathsf{stan}, \mathsf{loc}\}$ have the following properties:

1. **Index and handle uniqueness:** The argument *ind* is a key attribute in D. Also $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 1$ for any $i^{\mathsf{hnd}} \leq curhnd_u$ and $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 0$ for any $i^{\mathsf{hnd}} > curhnd_u$.
2. **Well-defined terms:** If an entry $x = D[i]$ has an index argument $a := x.arg[j] \in \mathcal{INDS}$ for $j \in \mathbb{N}$, then $a < i$.
3. **Message correctness:** Let denote $net\_id_{u,v,x}[i]$ as the i-th element in the buffer of the channel. Then the following statement always holds: for all $(u, v, x) \in ch\_honest$, each message $l := net\_id_{u,v,x}[i]$ with $l \neq \downarrow$ has $D[l].type = \mathsf{list}$.
4. **Length bounds:** For all $x \in D$, $x.len \leq \mathsf{max\_len}(k)$ at all times.
5. **Key secrecy:**If $D[i].type = \mathsf{ske}$, then $D[i]$ is not a component of $D[j]$ for any $i \neq j$ and $|\mathsf{owner}(D[i])| = 1$ at all times[23].
6. There is only one modification to an existing entry in D is to update an entry of zero-knowledge proof when the adversary provide the witness. □

Proof. Part 1 is stated in Section 4.3.2 and all the usage of the database follow the conventions.

Part 2,3,4, 5 and 6 can be proved be inspection of all commands □

[23]In this case we write owner.

Lemma 4.5.2. The ideal systems $Sys_{n,L}^{\mathsf{cry,id},t}$ for $t \in \{\mathsf{stan}, \mathsf{loc}\}$ have the following properties:

1. The systems are polynomial-time.
2. In the ideal system, no input is rejected because a counter $steps_{\mathsf{net\_id}_{u,v,x}}?$ has reached its bound.
3. When $\mathsf{TH}_{\mathcal{H}}$ assigns a handle $x.hnd_u$, it outputs the handle to the port $\mathsf{out}_u!$ in the same transition. □

Proof. For part 1, from Section 4.4.2 we see that the input number for each port is bounded by $bound_{\mathsf{p}?}$, which is polynomial in k. Also for each accepted input, the maximum length is always bounded by lengths functions that are also polynomial in k. Furthermore, action in each command is also clearly polynomial. As a consequence, $\mathsf{TH}_{\mathcal{H}}$ is polynomial-time.

Part 2 holds because before messages get into a buffer $net\_id_{u,v,x}$ they must be sent by a command send_x at port $\mathsf{in}_u?$. However, the number of inputs for this port is also bounded by $\mathsf{max\_in}(k)$.

Part 3 can be seen by inspection of commands. □

Chapter 5

THE REAL LIBRARY

In this chapter, we describe a real library that we can build from some practical cryptographic components. We will show what cryptographic components we need, what the structure of the real system is and how to build it from the components. The main difference of a real system from an ideal one is that the real one is made from several machines working over the network with the existence of an adversary and any of these machines can be corrupted as the adversary wishes, while the ideal one is made in one piece and "trusted".

Similarly as describing the ideal system, we use notations in [PW01] and follow the outline of [BPW03a].

5.1 Cryptographic operations

In this section, we describe what cryptographic components we need to build a real library that is "equivalent" to the ideal in Section 4. We will give definitions of those components and also specify what security requirements they must satisfy. Range of all algorithms is $\{0,1\}^+ \cup \{\downarrow\}$. In addition, to show that building such a real system is possible, we will point out that all the components have been shown to exist.

5.1.1 Key distribution system

We need a way to securely distribute the secret-key shares of a homomorphic threshold cryptosystem. In the real system we assume that we have a UC-secure distributed key generation protocol, or in other word a functionality FKEY. For $u \in \mathcal{H} \cup \{\mathsf{a}\}$, FKEY has ports $\{\mathsf{in}_{\mathsf{key},u}?, \mathsf{out}_{\mathsf{key},u}!\}$, which are

Functionality FKEY

Parameterized with a security parameter k, a (t, w) homomorphic threshold key generation algorithm, running with parties $P_1, ..., P_m$ and adversary S

Key gen: On input $(\textbf{keygen}, d, sid, u_1, ..., u_j)$ from a party P_d, generate a set of keys $(pk, sk_1, ..., sk_w)$. After that send $(\textbf{publickey}, d, sid, pk, u_1, ..., u_w)$ to P_d, send $(\textbf{secretshare}, d, sid, sk_i, i, pk, u_1, ..., u_w)$ to party P_{u_i} for all $i \in \{1, ...j\}$, send $(\textbf{secretshare}, d, sid, sk_{j+1}, ..., sk_w, pk, u_1, ..., u_j)$ to S[a].

[a]The rest of secret shares belong to S

Figure 5.1: The distributed key generation functionality FKEY

bounded by $\mathsf{max\_in}(k)$. For simplicity we also assume that the adversary does not block and delay messages. We describe our FKEY in Figure 5.1. For realizing such a protocol we refer to [AF04, Wik04].

5.1.2 Semantic security for (t, w)-threshold homomorphic encryption scheme

Given a (t, w)-threshold homomorphic encryption scheme $\mathcal{E}\_\mathsf{thres}$ as described in Section 2.5, we require that $t - 1 < w/3$ and our threshold encryption scheme must be secure under chosen plaintext attack or "IND-TCPA" [FP00, FPS01]. Specifically, we require the threshold encryption must satisfy the following security definition

Definition 5.1.1. (IND-TCPA security) Given a (t, w)-threshold encryption scheme, the decryptor Dec is defined as follows: It has one input and one output port, variables $pk, sk_1, ..., sk_w, c'$ initialized with $\downarrow$ and it works by the following transition rules:

- In the beginning set $(pk, sk_1, ..., sk_w) \leftarrow \mathsf{FKEY}$.
- On input $(\mathsf{corrupt}, u_1, ..., u_{t-1})$ for the first time, where $u_1, ..., u_{t-1} \in \{1, ...w\}$, output $(pk, sk_{u_1}, ..., sk_{u_{t-1}})$.
- On input (share, m, c) and if (m, c) is a correct plaintext-ciphertext pair, then for $i = 1$ to w $\{$ output $ds_i \leftarrow \mathsf{D\_thres}_{sk\_i}(c)$ $\}$.

- On input (enc, m_0, m_1) and if $|m_0| = |m_1|$ and $c' = \downarrow$ then
  - pick a random coin r;
  - set $b \stackrel{\mathcal{R}}{\leftarrow} \{0,1\}$;
  - set $c' \leftarrow \mathsf{E\_thres}_{pk}(m_b)$;
  - finally store and output c'.

The threshold encryption scheme is called *indistinguishable under chosen plaintext attack* (IND-TCPA) if for every PPT machine A_{enc} that interacts with Dec and finally output a guess b^*, we have

$$|Pr[b^* = b] - 1/2| \leq g(k)$$

for a negligible function $g()$. $\diamond$

Informally this security definition means even the adversary has learnt $t-1$ secret shares, he gains nothing in a chosen plaintext attack.

5.1.3 Non-interactive zero-knowledge proof of knowledge

For implementation of the real system, we assume that we have a UC-secure NIZK protocol as introduced in Section 2.6. For $u \in \mathcal{H} \cup \{\mathsf{a}\}$, FNIZK has ports $\{\mathsf{in}_{\mathsf{nizk},u}?, \mathsf{out}_{\mathsf{nizk},u}!\}$, in which the number of new proofs is bounded by $\mathsf{max\_in}(k)$ for each port. Therefore, in our real system we will replace it by a NIZK functionality FNIZK. Thus our real library still realizes the ideal one according to UC composition theorem. For realizing a such NIZK functionality we refer to [GOS06]. We describe a NIZK functionality for a witnessing relation R and security parameter k in Figure 5.2.

With this functionality we have a function $\mathsf{nizk\_len}(k,l)$ to get the length of the proof where k is the security parameter and l is the length of witness. We also assume that we have a function $\mathsf{valid}(m)$ that returns true if the plaintext m is valid, otherwise returns false. Now we can define a witnessing relation R as follows.

Definition 5.1.2. Given an encryption scheme $\mathcal{E}\mathsf{\_thres}$ as defined in Section 2.5 and a function $\mathsf{valid}(m)$, a tuple (x, w) where $x = (c, pk)$ and $w = (m, r)$ is in relation R if:

Functionality FNIZK

Parameterized with a security parameter k, a witnessing relation R, running with parties P_1,...,P_n and the adversary S. FNIZK uses a database $nizks$ to store pairs of x and p.

Proof:

- On input $(\textbf{prove}, d, sid, x, w)$ from a party P_d, ignore if $(x, w) \notin R$. Send $(\textbf{prove}, d, sid, x)$ to S and wait for an answer $(\textbf{proof}, d, sid, \pi)$.
- On receiving the answer $(\textbf{proof}, d, sid, \pi)$ from S, store (x, π) and send $(\textbf{proof}, sid, \pi)$ to P_d.

Verification:

- On input $(\textbf{verify}, d, sid, x, \pi)$ from a party P_d, check if (x, π) is stored. If (x, π) has been stored then send $(\textbf{verification}, sid, 1)$ to P_d. If (x, π) has not been stored then send $(\textbf{verify}, d, sid, x, \pi)$ to S and wait for an answer $(\textbf{witness}, d, sid, w)$.
- On receiving $(\textbf{witness}, d, sid, w)$, check if $(x, w) \in R$. If $(x, w) \in R$ then store (x, π) and send $(\textbf{verification}, sid, 1)$ to P_d. Otherwise send $(\textbf{verification}, sid, 0)$ to P_d

Figure 5.2: The NIZK functionality FNIZK for a witnessing relation R.

- $\mathsf{valid}(m) = \mathsf{true}$ and
- $c = \mathsf{E\_thres}_{pk}(m, r)$. $\diamond$

***Remark** 5.1.1.* Note that we assume that for any message m, we have $\mathsf{valid}(1^{|m|}) = \mathsf{true}$ because we always encrypt $1^{|m|}$ instead of encrypting an unknown message m. However, this assumption does not cause any lost of generality.

For simplicity we also assume that the adversary does not block and delay messages. $\diamond$

5.2 Structures

Now we can describe the structure of the real system as follows. Given an encryption scheme $\mathcal{E}\_\mathsf{thres}$, a functionality FNIZK as a non-interactive zero-knowledge proof system for plaintext validity, a functionality FKEY as a distributed key generation system for $\mathcal{E}\_\mathsf{thres}$, a parameter n that denotes the number of participants, and L' is a tuple of parameters (We will discuss more about it in Section 5.3), we can define 2 versions of a real cryptographic library $Sys^{\mathsf{cry,real,stan}}_{n,\mathcal{E}\_\mathsf{thres},\mathsf{FNIZK},\mathsf{FKEY},L'}$ and $Sys^{\mathsf{cry,real,loc}}_{n,\mathcal{E}\_\mathsf{thres},\mathsf{FNIZK},\mathsf{FKEY}}$, which are stand-alone and localized versions respectively.

Now we give a full description of them, using also the model in [PW01]

- The intended structure contains n machines $\{\mathsf{M}_1, ..., \mathsf{M}_n\}$.
- The intended ports are the 2 sets

$$S^{*,\mathsf{stan}^c} := \{\mathsf{in}_u!, \mathsf{out}_u?|u \in \{1, ..., n\}\};$$
$$S^{*,\mathsf{loc}^c} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}!|u \in \{1, ..., n\}\}.$$

  It means that each M_u has corresponding ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$. In the localized version, M_u has also a local clock port $\mathsf{out}_u{}^{\triangleleft}!$.

- The communication between a machine M_u and another one M_v is via three connections $net_{u,v,\mathsf{s}}$, $net_{u,v,\mathsf{a}}$ and $net_{u,v,\mathsf{i}}$, which are the *secure*, *authentic* and *insecure* channels respectively. We call them *network connections* and call corresponding ports *network ports*. The adversary can schedule all of these connections, can get messages in $\{\mathsf{a}, \mathsf{i}\}$ connections and it should be noticed that the messages from an i connection is always come from the adversary. However, we require a condition that honest users always use $net_{u,v,\mathsf{s}}$ or $net_{u,v,\mathsf{a}}$ to send decryption shares.[1].
- The channel model $\mathcal{X}$ maps each network connection $net_{u,v,x}$ to x.
- Each machine M_u also has three ports $\mathsf{in}_{\mathsf{nizk},u}!$, $\mathsf{in}_{\mathsf{nizk},u}{}^{\triangleleft}!$ and $\mathsf{out}_{\mathsf{nizk},u}?$ to communicate with FNIZK and three ports $\mathsf{in}_{\mathsf{key},u}!$, $\mathsf{in}_{\mathsf{key},u}{}^{\triangleleft}!$ and $\mathsf{out}_{\mathsf{key},u}?$ to communicate with FKEY.
- The access structure $\mathcal{ACC}$ contains all subsets of $\mathcal{H} \in \{1, ..., n\}$.

Figure 5.3 depicts the structure of the real system.

[1]In multiparty computation we assume the adversary can not change the message sent between honest users.

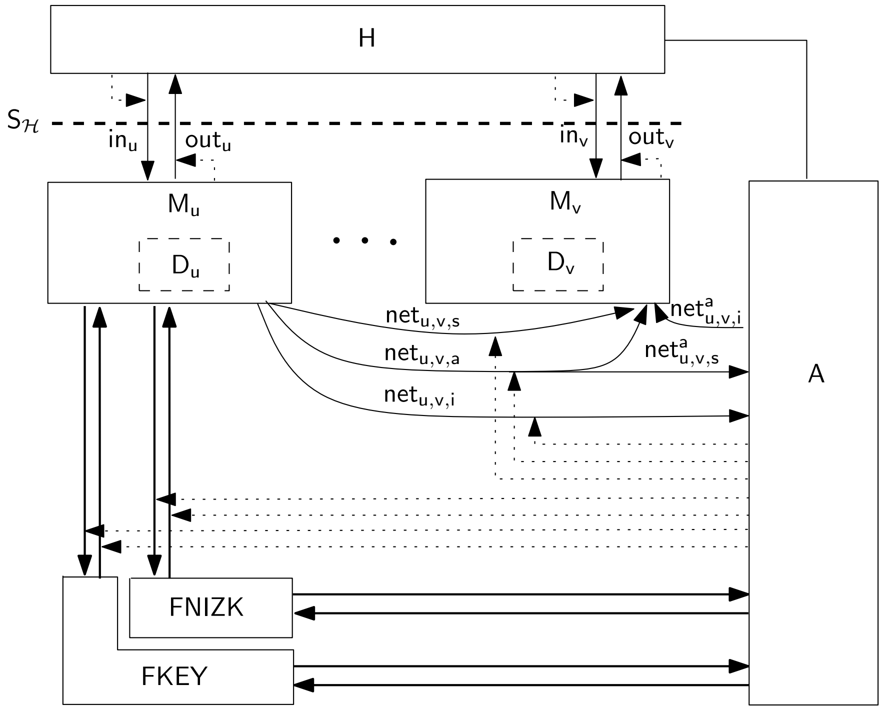

Figure 5.3: The real system that uses NIZK and key generation functionalities (Localized version)

5.3 System parameters

As we have defined tuple L that consists of functions to get lengths and bounds in the ideal system, in the real system we also want to bound the length of messages and also allow users to get length of any message. Correspondingly, the real system has a tuple L' that contains the functions $\mathsf{max\_len}(k)$, $\mathsf{max\_in}(k)$ as in the ideal system. It also has a function $\mathsf{nonce\_len}(k)$ such that $2^{-\mathsf{nonce\_len(k)}}$ is negligible.

Now we can map from the tuple L' in the real system to L in the ideal one. Given $\mathcal{E}$_thres, a NIZK functionality FNIZK and a distributed key generation system FKEY, we can define a function R2Ipar so that

$$L := \mathsf{R2Ipar}(\mathcal{E}\_\mathsf{thres}, \mathsf{FNIZK}, \mathsf{FKEY}, L').$$

Now L's functions mentioned in Section 4.2 can be defined as:[2]

[2]The idea here is that we use tags to distinguish different pieces of data.

- $\mathsf{data\_len}^*(l) := \mathsf{list\_len}(|data|, l)$;
- $\mathsf{list\_len}^*(l_1, ..., l_j) := \mathsf{list\_len}(|list|, l_1, ..., l_j)$;
- $\mathsf{nonce\_len}^*(k) := \mathsf{list\_len}(|nonce|, \mathsf{nonce\_len}(k))$;
- $\mathsf{pke\_len}^*(k) := \mathsf{list\_len}(|pke|, \mathsf{pke\_len}(k))$;
- $\mathsf{enc\_len}^*(k, l) := \mathsf{list\_len}(|enc|, \mathsf{pke\_len}(k), \mathsf{enc\_thres\_len}(k, l))$;
- $\mathsf{nizk\_len}^*(k, l) := \mathsf{list\_len}(|nizk|, \mathsf{nizk\_len}(k, l), \mathsf{pke\_len}^*(k)$ $, \mathsf{enc\_len}^*(k, l))$;
- $\mathsf{ds\_thres\_len}^*(k, cl) := \mathsf{list\_len}(|decshr|, \mathsf{ds\_thres\_len}(k, cl))$;

and $\mathsf{max\_len}(k)$, $\mathsf{max\_in}(k)$ are unchanged.

Lemma 5.3.1. Given a correct tuple L' of parameters for the real library, correct encryption scheme $\mathcal{E}\_\mathsf{thres}$, two correct functionalities FNIZK and FKEY, the algorithm $L := \mathsf{R2Ipar}(\mathcal{E}\_\mathsf{thres}, \mathsf{FNIZK}, \mathsf{FKEY}, L')$ yields a correct tuple L of parameters for the ideal library. □

Proof. It can be seen easily by verifying that the computed tuple L meet all conditions in Section 4.2. □

5.4 States of one machine

Basically, each machine M_u also has its own database D_u, a variable $curhnd_u$ pointing to the current handle and a variable $steps_{\mathsf{p}?}$ for each input port $\mathsf{p}?$.

5.4.1 Database D_u

Each machine M_u use its database D_u to store its running data. Each entry x in D_u has the following attributes

$$(hnd_u, word, type, add\_arg)$$

where

- $x.hnd_u \in \mathcal{HNDS}$ consecutively numbers all entries in D_u. We use it as the primary key, so we write $D_u[i^{\mathsf{hnd}}]$ for $D_u[hnd_u = i^{\mathsf{hnd}}]$.

- $x.word \in \{0,1\}^+$ is the real data of x.
- $x.type \in \mathsf{typeset} \cup \{\mathsf{null}\}$ is the type of x. A null value means the entry has not been parsed.
- $x.add\_arg$ is a list of additional data. This attribute depends on type of x.

5.4.2 Conventions about Handles

M_u uses a counter $curhnd_u$ of type $\mathcal{HNDS}$ to store the current number of elements in D_u. Initially D_u is empty and $curhnd_u = 0$. When a new entry x is added, M_u always set $x.hnd_u := curhnd_u{+}{+}$ and after that this attribute never changes.

When we write

$$(i^{\mathsf{hnd}}, D_u) :\leftarrow (i, type, add\_arg)$$

we mean "determine a handle i^{hnd} for that data in D_u" where $i \in \{0,1\}^+$, $type \in \mathsf{typeset} \cup \{\mathsf{null}\}$ and $add\_arg \in \mathcal{LIST}$. We use the following algorithm:

- If $i^{\mathsf{hnd}} := D_u[word = i \wedge type \neq \mathsf{ske}].hnd_u \neq \downarrow$ then
  - If $D_u[i^{\mathsf{hnd}}].type = \mathsf{null}$[3] then $D_u[i^{\mathsf{hnd}}].type = type$ and update add_arg of $D_u[i^{\mathsf{hnd}}]$.
  - return i^{hnd}.
- Otherwise
  - if $|i| \geq \mathsf{max\_len}(k)$ then return $i^{\mathsf{hnd}} :=\downarrow$.
  - Else set $i^{\mathsf{hnd}} := curhnd_u + +$ and $D_u :\Leftarrow (i^{\mathsf{hnd}}, i, type, add\_arg)$. Finally return i^{hnd}.

5.4.3 Input Counters

For each input port $\mathsf{p}?$, M_u uses a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ initialized with 0 to count the number of inputs at that port. The bound $bounds_{\mathsf{p}?}$ for the number of inputs equals $\mathsf{max\_in}(k)$.

[3] It has not been parsed

5.5 Inputs and their Evaluation

Now we look at the details of commands offered to users in this real library.

5.5.1 General working conventions

In this library we still use the same idea used for conventions in the ideal library.

Upon each input at a port $\mathsf{p}?$, first $\mathsf{TH}_{\mathcal{H}}$ increments the counter $steps_{\mathsf{p}?}$. After that $\mathsf{TH}_{\mathcal{H}}$ uses the length function to check how many bits of input can be accepted at each port. This value depends on the current state. Once $steps_{\mathsf{p}?}$ has reached the bound $bound_{\mathsf{p}?}$, the length function for this port always returns 0, meaning that no input is accepted.

An input is called *well-formed* when it is in the correct domain. If it is not, then M_u can always verify and then aborts the transition. From now we consider only *well-formed* inputs.

In the localized version, for every output $\mathsf{out}_u!$, M_u schedules itself by making a clock output 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

5.5.2 Constructors and One-level parsing

Constructors For each type, we define a main algorithm to construct data, $\mathsf{make\_}type()$. It is purely functional and uses only one global variable,the security parameter k.[4]

One-level parsing We define a functional parsing algorithm

$$(type, arg) \leftarrow \mathsf{parse}(m)$$

for $m \in \{0,1\}^+$. Here $type \in \mathsf{typeset}$ and arg is a list of elements of Σ^*.

The algorithm works as follows:

- On input m, check if it is of the form $(type, m_1, ..., m_j)$ with $type \in typeset \backslash \{\mathsf{ske}, \mathsf{garbage}\}$ and $j \geq 0$. If fail then return $(\mathsf{garbage}, ())$.

[4] This structure is useful for proving.

- Otherwise make a call to a type-specific parsing algorithm $arg \leftarrow \mathsf{parse\_}type(m)$. Such algorithm returns $\downarrow$ if m is not that type, otherwise it returns corresponding arguments (these algorithms are described in Section 5.5.3). If $\mathsf{parse\_}type(m)$ returns $arg =\downarrow$ then return $(\mathsf{garbage}, ())$ again.

By writing "parse m^{hnd}" we mean M_u calls $(type, arg) \leftarrow \mathsf{parse}(D_u[m^{\mathsf{hnd}}].word)$ then assigns $D_u[m^{\mathsf{hnd}}].type := type$ if this attribute is null and may use arg.

When we write "parse m^{hnd} if necessary" we mean the same thing but M_u does it only if $D_u[m^{\mathsf{hnd}}].type = \mathsf{null}$.

5.5.3 Basic commands and $\mathsf{parse\_}type$ algorithms

We describe how M_u evaluates a command when it is entered at port $\mathsf{in}_u?$.

Type and length queries

- *Type query*: $t \leftarrow \mathsf{get\_type}(x^{\mathsf{hnd}})$.

  Parse x^{hnd} if necessary. Then set $t := D_u[x^{\mathsf{hnd}}].type$.

- *Length query*: $l \leftarrow \mathsf{get\_len}(x^{\mathsf{hnd}})$.

  Parse x^{hnd} if necessary. If $D_u[x^{\mathsf{hnd}}].type \neq \mathsf{ske}$ then set $l := |D_u[x^{\mathsf{hnd}}].word|$, else set $l := 0$.

Storing and retrieving data

- *Constructor*: $d \leftarrow \mathsf{make\_data}(m)$, for $m \in \{0,1\}^*$.

  Set $d := (\mathsf{data}, m)$.

- *Storing*: $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$.

  Let $d \leftarrow \mathsf{make\_data}(m)$ then $(d^{\mathsf{hnd}}, D_u) :\leftarrow (d, \mathsf{data}, ())$.

- *Parsing*: $arg \leftarrow \mathsf{parse\_data}(m)$.

  If m is of the form (data, m') where $m' \in \{0,1\}^*$ then return (m'). Otherwise return $\downarrow$.

- *Retrieval*: $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  Parse x^{hnd} if necessary. Return $\mathsf{parse\_data}(D_u[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].word)[1]$.

Lists

- *Constructor*: $l \leftarrow \mathsf{make\_list}(x_1, ..., x_j)$ for $j \in \mathbb{N}_0$ and $x_i \in \{0,1\}^+$ for $i \in \{1, ..., j\}$.

  Set $l := (\mathsf{data}, x_1, ..., x_j)$.

- *Generating a list*: $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, ..., x_j^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.

  If $D_u[x_i^{\mathsf{hnd}}].type = \mathsf{ske}$ for any i, then return $\downarrow$. Otherwise set $l = \mathsf{make\_list}(D_u[x_1^{\mathsf{hnd}}].word, ..., D_u[x_j^{\mathsf{hnd}}].word)$ and $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

- *Parsing*: $arg \leftarrow \mathsf{parse\_list}(l)$.

  If l is of the form $(\mathsf{list}, x_1, ..., x_j)$ where $j \in \mathbb{N}_0$ and $x_i \in \{0,1\}^+$ for $i \in \{1, ..., j\}$, then return $(x_1, ..., x_j)$. Otherwise return $\downarrow$.

- *i-th element projection*: $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $0 \leq i \leq \mathsf{max\_len}(k)$.

  Parse l^{hnd}, getting arg. If $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$ then return $\downarrow$. Otherwise let $x := arg[i]$. If $x = \downarrow$ then return $\downarrow$, else $(x^{\mathsf{hnd}}, D_u) :\leftarrow (x, \mathsf{null}, ())$.

Nonces

- *Constructor*: $n \leftarrow \mathsf{make\_nonce}(m)$.

  Let $n' \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$ and then set $n := (\mathsf{nonce}, n')$.

- *Generating a nonce*: $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Let $n \leftarrow \mathsf{make\_nonce}(m)$, $n^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (n^{\mathsf{hnd}}, n, \mathsf{nonce}, ())$.

- *Parsing*: $arg \leftarrow \mathsf{parse\_nonce}(n)$

  If n is of the form (nonce, n') where $n' \in \{0,1\}^{\mathsf{nonce\_len}(k)}$ then return $()$, else return $\downarrow$.

(t, w)-threshold homomorphic encryption with NIZK proof of plaintext validity

- *Key generation:*$pk^{\mathsf{hnd}} \leftarrow \mathsf{gen\_enc\_thres\_keylist}(\mathsf{u}_1, ..., \mathsf{u}_\mathsf{j})$ for $(w - t) < j \leq w$.

  Activate FKEY with input $(u_1, ..., u_j)$, getting back a message (**publickey**, $d, sid, pk,$)[5]. Let $pk^* := (\mathsf{pke}, pk, d, sid, u_1, ..., u_j)$, set $pk^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, \mathsf{pke}, ())$.

[5]Here we need not a key-gen constructor. The secret-key shares are sent by FKEY to the intended users securely. See Section 5.5.5.

- *Public-key parsing*: $arg \leftarrow \mathsf{parse\_pke}(pk^*)$.

  If pk^* is of the form (pke, pk) where $pk \in \{0,1\}^{\mathsf{pke\_thres\_len}(k)}$ then return $()$, else return $\downarrow$.

- *Encryption constructor*: $c^*, p^* \leftarrow \mathsf{make\_enc}(pk^*, m)$ for $pk^*, m \in \{0,1\}^+$.

  Pick an $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$. Let $pk := pk^*[2]$, encrypt $c \leftarrow \mathsf{E\_thres}_{pk}(m, r)$ and set $c^* := (\mathsf{enc}, pk, c)$.

  Let $w := (m, r)$ and $x := (c, pk)$. Submit (x, w) to FNIZK, getting back p. If $p = \downarrow$ then return $\downarrow$. Otherwise let $p^* := (\mathsf{nizk}, p, c^*, pk^*)$.

- *Encryption:*$(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{encrypt\_thres}(pk^{\mathsf{hnd}}, m^{\mathsf{hnd}})$ where p^{hnd} is the NIZK proof of plaintext validity.

  Parse pk^{hnd} and m^{hnd} if necessary. If $D_u[pk^{\mathsf{hnd}}].type \neq \mathsf{pke}$ or $D_u[m^{\mathsf{hnd}}].type \neq \mathsf{data}$ then return $\downarrow$. Otherwise let $pk^* := D_u[pk^{\mathsf{hnd}}].word$, $m := D_u[m^{\mathsf{hnd}}].word$ and $c^*, p^* \leftarrow \mathsf{make\_enc}(pk^*, m)$.

  If $c^* = \downarrow$ or $c^* > \mathsf{max\_len}(k)$ then return $\downarrow$. Else

  - Set $c^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (c^{\mathsf{hnd}}, c^*, \mathsf{enc}, ())$.
  - Set $p^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (p^{\mathsf{hnd}}, p^*, \mathsf{nizk}, ())$.

- *Ciphertext parsing*: $arg \leftarrow \mathsf{parse\_enc}(c^*)$.

  If c^* is not of the form (enc, pk, c) where $pk \in \{0,1\}^{\mathsf{pke\_thres\_len}(k)}$ and $c \in \{0,1\}^+$ then return $\downarrow$, else set $arg := (pk^*) := (\mathsf{pke}, pk)$.

- *NIZK proof parsing*: $arg \leftarrow \mathsf{parse\_nizk}(p^*)$.

  If p^* is not of the form $(\mathsf{nizk}, p, c^*, pk^*)$ where $p \in \{0,1\}^{\mathsf{nizk\_len}(k)}$ then return $\downarrow$, else set $arg := (c^*, pk^*)$.

- *Functional decryption:* $ds^* \leftarrow \mathsf{make\_decrypt\_thres}(sk^*, c_1^*, ..., c_j^*, p_1^*, ..., p_j^*)$.

  Let $sk := sk^*[2]$, $c_i := c_i^*[3]$ and $p_i := p_i^*[2]$ for $i = 1, ..., j$. Give every pair $((c_i, c_i^*[2]), p_i)$ to FNIZK, if any proof is invalid then return $\downarrow$.

  Otherwise compute $c := c_1 \boxdot \cdots \boxdot c_j$ and let $ds := \mathsf{D\_thres}_{sk}(c)$. Set $c^* = (\mathsf{enc}, c_1^*[2], c)$. Return $ds^* := (\mathsf{dec\_shr}, ds, c^*, sk^*[4], c_1^*, ..., c_j^*)$.

- *Generating a decryption share:* $ds^{\mathsf{hnd}} \leftarrow \mathsf{decrypt\_thres}(sk^{hnd}, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}, p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}})$ where $\{c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}\}$ is a set of ciphertexts, $\{p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}}\}$ is a set of corresponding proofs for plaintext validity.

Parse c_i^{hnd} to have $arg_i := (pk_i^*)$. Return $\downarrow$ if all of pk_i^* are not the same. Otherwise say the same public key is pk^*. Return $\downarrow$ if $D_u[sk^{hnd}].type \neq$ ske or $D_u[sk^{hnd}].word[3] \neq pk^*$ or any $D_u[c_i^{hnd}].type \neq \mathsf{enc}$ or any $D_u[p_i^{hnd}].type \neq \mathsf{nizk}$.

Otherwise set $sk^* := D_u[sk^{hnd}].word$, $c_i^* := D_u[c_i^{hnd}].word$, $p_i^* := D_u[p_i^{hnd}].word$ then let $ds^* \leftarrow \mathsf{make\_decrypt\_thres}(sk^*, c_1^*, ..., c_j^*, p_1^*, ..., p_j^*)$. If $ds^* =\downarrow$ then return $\downarrow$. Otherwise $(ds^{\mathsf{hnd}}, D_u) :\leftarrow (ds^*, \mathsf{decshr}, ())$.

- *Decryption share parsing*: $arg \leftarrow \mathsf{parse\_decshr}(ds^*)$.

  If ds^* is not of the form $(\mathsf{decshr}, ds, c, i)$ where $ds \in \{0,1\}^{\mathsf{ds\_thres\_len}(k)}$ then return $\downarrow$, else set $arg := (c, i)$.

- *Functional combination:* $m \leftarrow \mathsf{make\_combi}(ds_1^*, ..., ds_t^*)$.

  Let $ds_i := ds_i^*[2]$. Set $m := \mathsf{C\_thres}(ds_1, ..., ds_t)$.

- *Combining decryption shares:*$m^{\mathsf{hnd}} \leftarrow \mathsf{combine\_thres}(ds_1^{\mathsf{hnd}}, ..., ds_w^{\mathsf{hnd}})$ where $\{ds_1^{\mathsf{hnd}}, ..., ds_w^{\mathsf{hnd}}\}$ is the set of all w decryption shares

  Let $ds_i^* := D_u[ds_i^{hnd}].word$ for all $i \in \{1, ..., w\}$. First check if all of $ds_i^*[3]$ are the same [6]. If not then return $\downarrow$.

  Otherwise, according to Theorem 2.5.1, from the set of $ds_i^*[2]$, M_u can find the majority of the correct result using $make\_combi(ds_1^*, ..., ds_t^*)$ (Because of the condition $(t-1) < w/3$). Say the correct plaintext is m. Let $(m^{\mathsf{hnd}}, D_u) :\leftarrow (m, \mathsf{data}, ())$[7].

- *Public key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_enc\_thres}(c^{\mathsf{hnd}})$.

  Parse c^{hnd} to get arg. If $D_u[c^{\mathsf{hnd}}].type \neq \mathsf{enc}$ then return $\downarrow$. Else set $(pk^{\mathsf{hnd}}, D_u) :\leftarrow (arg[1], \mathsf{pke}, ())$.

5.5.4 Send commands

- $\mathsf{send\_}x(v, l^{\mathsf{hnd}})$ for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$ and $v \in \{1, ..., n\}$.

  Parse l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$ then output $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,x}!$.

---

[6] It means all of decryption share are supposed from the same ciphertext, even malicious decryption shares

[7] Note that honest user must use authentic or secure channels to send decryption share. See Section 5.2

5.5.5 Network Inputs

- On input l at a port $\mathsf{net}_{w,v,x}?$ for $l \in \{0,1\}^+$ and $|l| \leq \mathsf{max\_len}(k)$.

  Check if l is of the form $(\mathsf{list}, x_1, ...x_j)$ for some $j \in \mathbb{N}_0$ and $x_i \in \{0,1\}^+$. If yes then $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$ and output (w, x, l^{hnd}) at $\mathsf{out}_u!$.

5.5.6 Inputs from FKEY

- On input $(\mathbf{secretshare}, d, sid, sk, i, pk, u_1, ..., u_j)$ at a port $\mathsf{out}_{\mathsf{key},u}?$.

  Let $pk^* := (\mathsf{pke}, pk, d, sid, u_1, ..., u_j)$, set $pk^{\mathsf{hnd}} := curhnd_u + +$ and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, \mathsf{pke}, ())$.

  Let $sk^* := (\mathsf{ske}, sk, pk, i)$, set $sk^{\mathsf{hnd}} := curhnd_u + +$ and $D_u :\Leftarrow (sk^{\mathsf{hnd}}, sk^*, \mathsf{ske}, ())$.

  Output $(pk^{\mathsf{hnd}}, sk^{\mathsf{hnd}})$ to the port $\mathsf{out}_u!$.

5.6 Properties of the Real system

We show that the real library has some properties, which we will use later in security proof.

Lemma 5.6.1. The real systems $Sys^{\mathsf{cry,real,t}}_{n,\mathcal{E}\_\mathsf{thres},\mathsf{FNIZK},\mathsf{FKEY},L'}$ for $t \in \{\mathsf{stan}, \mathsf{loc}\}$ have the following properties:

1. The argument hnd_u is a key attribute in D_u.
2. Only two modifications may happen to existing entries x in D_u:
   - $x.type$ changes from null to another type;
   - $x.add\_arg$ changes from () to something else at the same time.[8]
3. All the time we have $|x.word| \leq \mathsf{max\_len}(k)$ for all $x \in D_u$ except possibly when $x.type = \mathsf{ske}$.
4. The systems are polynomial-time. □

[8] as a result of a parsing.

Proof. Part 1 holds as it is stated in Section 5.4.2.

Part 2 can be proved by inspection of commands.

Part 3 can be proved by inspection of the commands that produce new entries.

For part 4, first we see that all algorithms used have polynomial numbers of steps and also add polynomial numbers of entries to D_u. Every input is bounded by polynomial functions. Furthermore, each port also accepts a polynomial number of inputs. As a consequence, the whole system is polynomial-time. $\square$

Chapter 6

THE SIMULATOR

We have discussed in the Section 2.2 about using a simulator to prove a system is at least as secure as another one. In this chapter we will construct a simulator $\mathsf{Sim}_{\mathcal{H}}$ for each set $\mathcal{H}$ and in the chapter 7 we will prove that for every real adversary A, the combination $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$, in which $\mathsf{Sim}_{\mathcal{H}}$ uses A as a blackbox (blackbox simulation), makes the same effects in the ideal library as the real adversary A does in the real one. Then by simulatable security definition we can show that for any $t \in \{\mathsf{stan}, \mathsf{loc}\}$, the library system $Sys^{\mathsf{cry,real},t}_{n,\mathcal{E}\_\mathsf{thres},\mathsf{FNIZK},\mathsf{FKEY},L'}$ in chapter 5 is at least as secure as the library system $Sys^{\mathsf{cry,id},t}_{n,L}$ in chapter 4.

The basic idea about how $\mathsf{Sim}_{\mathcal{H}}$ works is just translating abstract messages form ideal to real system and vice versa as described in Figure 6.1.

6.1 Ports and Scheduling

$\mathsf{Sim}_{\mathcal{H}}$ communicates with $\mathsf{TH}_{\mathcal{H}}$ via ports $\mathsf{in_a}$ and $\mathsf{out_a}$, with A via all of network connections $net_{u,v,x}$.

For scheduling, A is still the master scheduler. It controls all of clock ports as it does with the real library, except the clock port $\mathsf{net}_{u,v,x}{}^{\triangleleft}!$ with $(u, v, x) \in ch\_honest$ are renamed to $\mathsf{net\_id}_{u,v,x}{}^{\triangleleft}!$ and A controls these new name-changed ports[1].

$\mathsf{TH}_{\mathcal{H}}$ still schedules its output port $\mathsf{out_a}!$.

$\mathsf{Sim}_{\mathcal{H}}$ outputs 1 to $\mathsf{in_a}{}^{\triangleleft}!$ immediately whenever it makes an output at $\mathsf{in_a}!$.

[1]Port renaming is allowed here

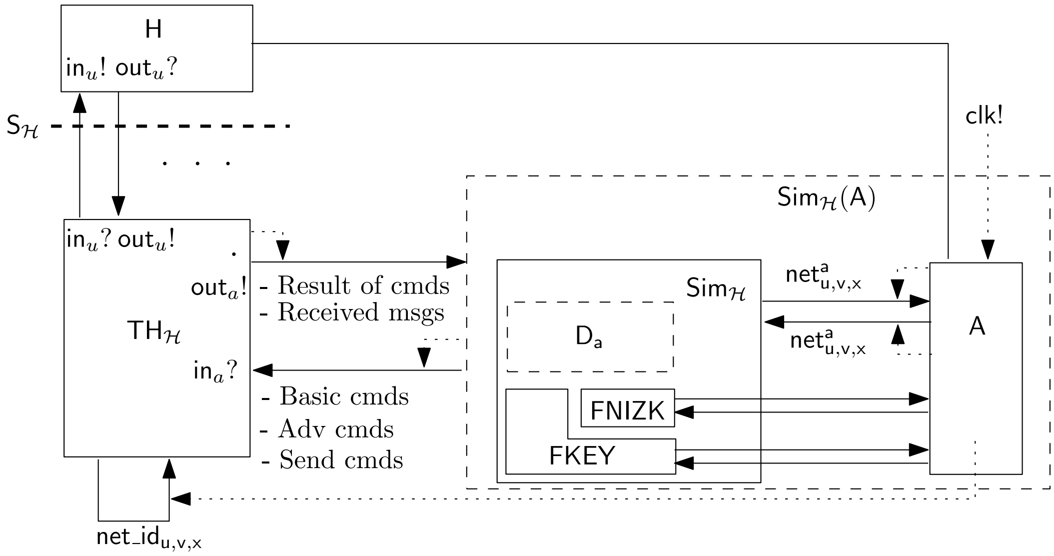

Figure 6.1: The simulator translates messages between $\mathsf{TH}_\mathcal{H}$ and the real adversary A

$\mathsf{Sim}_\mathcal{H}$ also also schedules the channels from FNIZK and FKEY (contained inside the simulator) to the adversary A.

We use the following abbreviation when we mean $\mathsf{Sim}_\mathcal{H}$ calls local commands:

"Call $y \leftarrow x$ at $\mathsf{in_a}$! expecting ...".

By writing this we mean $\mathsf{Sim}_\mathcal{H}$ outputs x at $\mathsf{in_a}$! and schedules it, waits for an input at $\mathsf{out_a}$? by setting other length functions to 0, and finally assigns the input to y. If y is not in the expected domain stated after "expecting", $\mathsf{Sim}_\mathcal{H}$ aborts all current recursive algorithms and its outermost transition. Note that $\mathsf{TH}_\mathcal{H}$ responses to local commands called by $\mathsf{Sim}_\mathcal{H}$ by giving the result at the port $\mathsf{out_a}$! and also scheduling it immediately, we get subroutine behaviour here.

6.2 States of the Simulator

The state of $\mathsf{Sim}_\mathcal{H}$ includes a database D_a, a variable $curhnd_\mathsf{a}$ and a counter $steps_{\mathsf{p?}}$ for each input port p?. For each $u \in \mathcal{H} \cup \{\mathsf{a}\}$, $\mathsf{Sim}_\mathcal{H}$ also uses variables $keysid_u$, and $nizksid_u$ to use as sid when calling to FKEY and FNIZK. We describe them in details below.

6.2.1 Database D_{a}

$\mathsf{Sim}_{\mathcal{H}}$ uses a database D_{a} to store already mapped handles. Each entry in D_{a} contains the following attributes

$$(hnd_{\mathsf{a}}, word, add\_arg)$$

For each entry $x \in D_{\mathsf{a}}$:

- $x.hnd_{\mathsf{a}} \in \mathcal{HNDS}$ is the primary key attribute.
- $x.word \in \{0,1\}^*$ is the real representation of x.
- $x.add\_arg$ is a list of additional arguments.

6.2.2 Input counters

For each input port $\mathsf{p}?$, $\mathsf{Sim}_{\mathcal{H}}$ uses a counter $steps_{\mathsf{p}?}$, initialized with 0, to count the number of inputs at $\mathsf{p}?$. The corresponding $bound_{\mathsf{p}?}$ are $\mathsf{max\_in}(k)$ in the case of network ports and $\mathsf{max\_in_a}(k) = (2w+10)n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ for the port $\mathsf{out_a}?$ where w is the number of secret share per generation[2].

6.3 Input Evaluation

6.3.1 General conventions

In general, on each input at a port $\mathsf{p}?$, first $\mathsf{Sim}_{\mathcal{H}}$ increments $steps_{\mathsf{p}?}$. When $steps_{\mathsf{p}?}$ has reached $bound_{\mathsf{p}?}$, the length function, which indicates the maximum of possible length of inputs at this port, always 0. Otherwise, the function depends on different domain of input in each command below.

6.3.2 Inputs from $\mathsf{TH}_{\mathcal{H}}$

Here we consider only the inputs that are the results of commands called by honest users.

[2] We explain later in the proof of Lemma 6.4.2.

- *Input from send commands:* On input m at port $\mathsf{out_a}?$ where $m = (u, v, x, l^{\mathsf{hnd}})$, $(u, v, x) \in ch\_to\_adv$ and $l^{\mathsf{hnd}} \le \mathsf{max\_hnd}(k) := (2w + 6)n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ (We explain why in the proof of Lemma 6.4.2).

  If $D_{\mathsf{a}}[l^{\mathsf{hnd}}] \neq \downarrow$, let $l := D_{\mathsf{a}}[l^{\mathsf{hnd}}].word$[3].

  Otherwise set $curhnd++$ and create a new real representation with the algorithm $l \leftarrow \mathsf{id2real}(l^{\mathsf{hnd}})$. This algorithm maps the handle of an abstract term to its corresponding real representation. We will describe the algorithm later in this Section.

  Output l at the port $\mathsf{net}_{u,v,x}!$.

- *New key generation:* On input m at port $\mathsf{out_a}?$ where m is of the form $(u, pk_{\mathsf{a}}^{\mathsf{hnd}}, sk_{j+1}^{\mathsf{hnd}}, ..., sk_{w}^{\mathsf{hnd}},)$.

  Increment $keysid_u$. Call $(type, arg) \leftarrow \mathsf{adv\_parse}(pk^{\mathsf{hnd}})$ at $\mathsf{in_a}!$, expecting $type = \mathsf{pke}$. arg must be of the form $(u_1, ..., u_j)$ (The list of honest users who keep secret shares).

  Activate FKEY to make a set of w secret shares $\{sk_1, ..., sk_w\}$ and a public key pk. For all $i \in \{1, ..., j\}$ let $sk_i^* = (\mathsf{ske}, sk_i, pk, i)$ and $pk^* = (\mathsf{ske}, pk, u, keysid_u, u_1, ..., u_j)$. Then $D_{\mathsf{a}} :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, (\mathsf{honest}, sk_1^*, ...$ $, sk_j^*))$.

  Output $(\mathbf{secretshare}, u, keysid_u, sk_{j+1}, ..., sk_w, pk, u_1, ..., u_j)$ at the port $\mathsf{out_{key,a}}!$. For all $i \in \{j+1, ..., w\}$ do $D_{\mathsf{a}} :\Leftarrow (sk_i^{\mathsf{hnd}}, sk_i, ())$. Wait until the adversary clocks connections from FKEY to $\mathsf{Sim}_{\mathcal{H}}$.

  For all $i \in \{1, ..., j\}$, when the adversary clocks the connection from FKEY to the port $\mathsf{out}_{\mathsf{key},u_i}?$ then call $\mathsf{learn\_share}(pk_{\mathsf{a}}^{\mathsf{hnd}}, u_i)$.

- *New zero-knowledge proof generation:* On input $(u, c^{\mathsf{hnd}}, p^{\mathsf{hnd}})$ at the port $\mathsf{out_a}?$.

  Increment $nizksid_u$. Make a call $c^* \leftarrow \mathsf{id2real}(c^{\mathsf{hnd}})$. Then set $x := (c^*[3], c^*[2])$ and ask FNIZK to make a proof p[4]. Let $pk^* \leftarrow \mathsf{parse\_enc}(c^*)$ then set $p^* := (\mathsf{nizk}, p, c^*, pk^*)$ and finally $D_{\mathsf{a}} :\Leftarrow (p^{\mathsf{hnd}}, p^*, ())$. Output $(\mathsf{nizkaccept}, u, p^{\mathsf{hnd}})$ to the port $\mathsf{in_a}!$.

- *Witness for a zero-knowledge proof generation:* On input $(u, \mathsf{witness}, c^{\mathsf{hnd}}$ $, p^{\mathsf{hnd}})$ at the port $\mathsf{out_a}?$ $\mathsf{out_a}?$.

  Increment $nizksid_u$. Make a call $c^* \leftarrow \mathsf{id2real}(c^{\mathsf{hnd}})$ and $p^* \leftarrow D_{\mathsf{a}}[p^{\mathsf{hnd}}].word$. Set $x := (c^*[3], c^*[2])$ and $p := (p)$ then ask FNIZK to verify. FNIZK will

[3] l^{hnd} is mapped already

[4] Without a witness here because the simulator control FNIZK. FNIZK will ask A for p. Note that we assume A replies without any delay

ask the adversary for witness in the form of (m, r). If FNIZK returns true then store m, yielding m^{hnd}. Then output $(u, \mathsf{witness}, m_i^{\mathsf{hnd}})$ to the port $\mathsf{in_a}!$. Otherwise output $(u, \mathsf{witness}, \downarrow)$ to the port $\mathsf{in_a}!$.

Now we describe the algorithm $\mathsf{id2real}$. Basically $\mathsf{id2real}$ recursively parses an abstract message, makes a corresponding real representation, and add all new messages parts into D_{a}. $\mathsf{id2real}$ may use local commands offered by $\mathsf{TH}_{\mathcal{H}}$. Note that $\mathsf{id2real}$ is called only for new handles. $\mathsf{id2real}$ is defined as follows.

1. Call $(type, (m_1^{\mathsf{hnd}}, ..., m_j^{\mathsf{hnd}})) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$ at $\mathsf{in_a}!$, expecting $type \in typeset \backslash \{\mathsf{ske}, \mathsf{garbage}\}$[5], $j \leq \mathsf{max\_len}(k)$ and $m_i^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ if $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and $|m_i^{\mathsf{hnd}}| \leq \mathsf{max\_len}(k)$ otherwise.
2. For $i \in \{1, ..., j\}$: If $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and $m_i^{\mathsf{hnd}} > curhnd_{\mathsf{a}}$ then set $curhnd_{\mathsf{a}}$++.[6]
3. For $i \in \{1, ..., j\}$: If $m_i^{\mathsf{hnd}} \notin \mathcal{HNDS}$, let $m_i := m_i^{\mathsf{hnd}}$. Otherwise if $D_{\mathsf{a}}[m_i^{\mathsf{hnd}}] \neq \downarrow$ then let $m_i := D_{\mathsf{a}}[m_i^{\mathsf{hnd}}].word$. Else make a recursive call $m_i \leftarrow \mathsf{id2real}(m_i^{\mathsf{hnd}})$. Finally let $arg^{\mathsf{real}} := (m_1, ..., m_j)$.
4. Construct the real version m and enter it into D_{a} depending on type[7]:
   - If $type \in \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}\}$, set $m \leftarrow \mathsf{make\_}type(arg^{\mathsf{real}})$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.
   - If $type = \mathsf{enc}$, then there are two cases here:
     - If $m_2^{\mathsf{hnd}} \in \mathcal{HNDS}$, i.e it is a cleartext here[8], then set $m \leftarrow \mathsf{make\_enc}(arg^{\mathsf{real}})$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.
     - Otherwise arg^{real} is of the form (pk^*, len). Let $pk := pk^*[2]$ and pick $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, and then encrypt $c \leftarrow \mathsf{E\_thres}_{pk}(1^{len}, r)$. Set $m \leftarrow (\mathsf{enc}, pk, c)$ and finally $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.
   - If $type = \mathsf{decshr}$, then there are two cases can happen.
     - If the adversary can see the plaintext, then arg^{real} is of the form $(m\_ind, c_1, ..., c_j)$. In this case, the simulator makes up

[5] We use superscript here even sometimes handles, e.g payload, user id, etc.

[6] In fact, $\mathsf{TH}_{\mathcal{H}}$ uses $l^{\mathsf{hnd}} = curhnd_{\mathsf{a}}$++ for new handles. Therefore this step is for $\mathsf{Sim}_{\mathcal{H}}$ to update its $curhnd_{\mathsf{hnd}}$ in order to restores "correct derivation".

[7] Note that we do not have type = pke here because the adversary knows every newly generated public key

[8] See Section 4.4.4

a new decryption which can be used to get the plaintext, say ds[9]. Finally $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, ds, ())$.
  - If the adversary can not see the plaintext, then just make a random one, say ds, and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, ds, ())$.

6.3.3 Inputs from A

- *Network input:* On input l at a port $\mathsf{net}_{z,u,x} \in ch\_from\_adv$, for $l \in \{0,1\}^+$ and $|l| \leq \mathsf{max\_len}(k)$.

  If l is not a list, then abort the transition. Else build the corresponding handle l^{hnd} by calling an algorithm $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ (This algorithm maps a real representation to the handle of its corresponding abstract term. Basically it works in the inverse way compared to $\mathsf{id2real}()$. We will describe the algorithm later in this Section.).

  Finally output the command $\mathsf{adv\_send\_i}(z, x, l^{\mathsf{hnd}})$ at the port $\mathsf{in_a}!$.

- *New key generation by adversary:* On input m at port $\mathsf{in}_{\mathsf{key,a}}?$ where m is of the form $(\textbf{keygen}, \mathsf{a}, sid, u_1, ..., u_j)$.

  Activate FKEY to make a set of w secret shares $\{sk_1, ..., sk_w\}$ and a public key pk. For all $i \in \{1, ..., w\}$ let $sk_i^* = (\mathsf{ske}, sk_i, pk, i)$ and $pk^* = (\mathsf{ske}, pk, u, keysid_u, u_1, ..., u_j)$. Also call $pk^{\mathsf{hnd}} \leftarrow \mathsf{gen\_enc\_thres\_keylist}(u_1, ..., u_j)$, getting back $(\mathsf{a}, pk_{\mathsf{a}}^{\mathsf{hnd}}, sk_{j+1}^{\mathsf{hnd}}, ..., sk_w^{\mathsf{hnd}},)$.

  Then $D_{\mathsf{a}} :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, (\mathsf{honest}, sk_1^*, ...sk_j^*))$ and $D_{\mathsf{a}} :\Leftarrow (sk_i^{\mathsf{hnd}}, sk_i^*, ())$ for all $i \in \{j+1, ..., w\}$. Output $(\textbf{secretshare}, \mathsf{a}, sid, sk_{j+1}, ..., sk_w, pk, u_1, ..., u_j)$ at the port $\mathsf{out}_{\mathsf{key,a}}?$.

  Wait until the adversary clocks connections from FKEY to $\mathsf{Sim}_{\mathcal{H}}$. For all $i \in \{1, ..., j\}$, when the adversary clocks the connection from FKEY to the port $\mathsf{out}_{\mathsf{key},u_i}?$ then call $\mathsf{learn\_share}(pk_{\mathsf{a}}^{\mathsf{hnd}}, u_i)$.

Now we describe the algorithm $\mathsf{real2id}$. Basically, $\mathsf{real2id}$ recursively parses a real message, and then build a corresponding term in $\mathsf{TH}_{\mathcal{H}}$ and enters all subterms into D_{a}. We define $\mathsf{real2id}$ as follows.

- $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$, for $m \in \{0,1\}^+$. If there is a m^{hnd} such that $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, then return it.

[9]See simulation for decryption shares in Section 2.5.

Else set $(type, arg) := \mathsf{parse}(m)$[10] and call a type-specific algorithm $add\_arg \leftarrow \mathsf{real2id\_}type(m, arg)$ (defined below).

Set $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$. If $type = \mathsf{enc}$ then $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}{++}, \epsilon, ())$[11].

If $type = \mathsf{nizk}$ then arg must be of the form (c^*, pk^*). Call $c^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(c^*)$. After that just update $D_{\mathsf{a}}[c^{\mathsf{hnd}} + 1].word = m$.

- **Garbage**: $add\_arg \leftarrow \mathsf{real2id\_garbage}(m, ())$.

  Call $m^{\mathsf{hnd}} \leftarrow \mathsf{adv\_garbage}(|m|)$ at $\mathsf{in_a}!$. Finally return $()$.

- **Data**: $add\_arg \leftarrow \mathsf{real2id\_data}(m, (m'))$.

  Call $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m')$ at $\mathsf{in_a}!$. Finally return $()$.

- **List**: $add\_arg \leftarrow \mathsf{real2id\_}list(m, (m_1, ..., m_j))$.

  For $i \in \{1, ..., j\}$ call $m_i^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m_i)$. After that call $m^{\mathsf{hnd}} \leftarrow \mathsf{list}(m_1^{\mathsf{hnd}}, ..., m_j^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Finally return $()$.

- **Nonce**: $add\_arg \leftarrow \mathsf{real2id\_nonce}(m, ())$.

  Call $m^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$ at $\mathsf{in_a}!$. Finally return $()$.

- (t, w) **threshold encryption**

  - $add\_arg \leftarrow \mathsf{real2id\_pke}(m, ())$.

    Call $pk^{\mathsf{hnd}} \leftarrow \mathsf{adv\_gen\_enc}()$ at $\mathsf{in_a}!$[12]. Finally return $adv\_arg := (\mathsf{adv})$.

  - $add\_arg \leftarrow \mathsf{real2id\_enc}(m, c^*, (pk^*))$.

    Make recursive call $pk^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(pk^*)$. Now there are two cases

    * If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{adv})$ then call $(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{adv\_invalid\_ciph\_thres}(pk^{\mathsf{hnd}}, |c^*|)$ at $\mathsf{in_a}!$ and return $()$.
    * If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk_1, ...sk_j)$ then use t secret share to decrypt it to get a plaintext m. If $\mathsf{valid}(m) = \mathsf{false}$ then

[10] See Section 5.5.2

[11] This handle is for the zero-knowledge proof. However, we use ϵ because $\mathsf{Sim}_{\mathcal{H}}$ does not ask the adversary for the real representation of the proof until the adversary sends it.

[12] In this case the public key is created by the adversary itself. If it generated keys by calling FKEY, then $\mathsf{Sim}_{\mathcal{H}}$ would get the public key and some secret share from FKEY

$(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{adv\_invalid\_ciph\_thres}(pk^{\mathsf{hnd}}, |c^*|)$ at $\mathsf{in_a}!$ and return ().
Otherwise make a recursive call $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$ then call $(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{encrypt\_thres}(pk^{\mathsf{hnd}}, m^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and return ().

– $add\_arg \leftarrow \mathsf{real2id\_nizk}(m, c^*)$.
Make recursive call $c^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(c^*)$. Return ()[13].

– $add\_arg \leftarrow \mathsf{real2id\_decshr}(ds, (c^*, i, c_1^*, ..., c_j^*))$.
Make recursive call $c^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(c^*)$ and $c_i^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(c_i^*)$ for all $i \in \{1, ..., j\}$
If $\mathsf{Sim}_{\mathcal{H}}$ does not have secret shares[14] for c^* or the c^* is not created from corresponding sk_i or c^* is not product of $c_1^*, ..., c_j^*$ then call $ds^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_decshr}(l, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and return ().
Otherwise call $ds^{hnd} \leftarrow \mathsf{adv\_decrypt\_thres}(sk^{hnd}, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and return ().

6.4 Properties of the Simulator

We claim that the simulator is polynomial-time

Lemma 6.4.1. For each polynomial-time adversary A, the joint machine $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ is also polynomial-time. □

Proof. Each input port of $\mathsf{Sim}_{\mathcal{H}}$ has a polynomially bounded counter and also polynomially bounded length functions. We also can inspect that $\mathsf{id2real}$ is polynomial-time. It means $\mathsf{Sim}_{\mathcal{H}}$ is polynomial time and the joint machine $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ is also polynomial-time. □

We also claim the following properties of $\mathsf{Sim}_{\mathcal{H}}$

Lemma 6.4.2. Each machine $\mathsf{Sim}_{\mathcal{H}}$ has the following properties

1. After each call $\mathsf{id2real}(m^{\mathsf{hnd}})$ we always have $D_{\mathsf{a}}[m^{\mathsf{hnd}}] \neq \downarrow$ unless $\mathsf{Sim}_{\mathcal{H}}$ aborts the execution.

2. The following holds for $\mathsf{id2real}$:

[13] The update of the entry related to this proof is mentioned in the beginning of this algorithm.
[14] $\mathsf{Sim}_{\mathcal{H}}$ always has at least $2w/3$ secret shares or has no secret share at all.

- Calls $\mathsf{id2real}(m^{\mathsf{hnd}})$ are made only for $m^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ and for each m^{hnd} there is at most one such call.
- At most $\mathsf{max\_hnd}(k)$ outputs can be made at $\mathsf{in_a}!$.
- No new entries in the database D of $\mathsf{TH}_{\mathcal{H}}$ is made.

3. For each call $\mathsf{real2id}(m)$ the following holds:
   - At most $|m|-1$ extra calls $\mathsf{real2id}(|m|)$ are made.
   - At most $|m|$ outputs can be made at $\mathsf{in_a}!$.
   - $curhnd_{\mathsf{a}}$ increases by at most $2|m|$, and at most $2|m|$ new entries can be made in D_{a}.
4. The following holds with D_{a}:
   - hnd_{a} of D_{a} is a key attribute.
   - Outside any execution of $\mathsf{id2real}$, entries in D_{a} are consecutively numbered.
5. The following holds for the interaction between $\mathsf{Sim}_{\mathcal{H}}$ and $\mathsf{TH}_{\mathcal{H}}$.
   - No handle output by $\mathsf{TH}_{\mathcal{H}}$ is rejected by $\mathsf{Sim}_{\mathcal{H}}$.
   - The counter $steps_{\mathsf{out_a}}?$ of $\mathsf{Sim}_{\mathcal{H}}$ and $steps_{\mathsf{in_a}}?$ of $\mathsf{TH}_{\mathcal{H}}$ never reach their bounds. □

Proof. • For part 1, if no abortion, any execution of $\mathsf{id2real}(m^{\mathsf{hnd}})$ ends with $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, ...)$.

- Part 2, 3, 4 are obvious by checking how the two algorithms work.
- Part 5 is proven as follows. In the given combination, the free ports are $\mathsf{in}_u?$ for $u \in \mathcal{H}$ and $\mathsf{net}_{w,u,x}?$ with $(w,u,x) \in ch\_from\_adv$ and the clock ports for the secure channels and $\mathsf{in_{key,a}}?$ and $\mathsf{in_{nizk,a}}?$ from simulated FKEY and FNIZK respectively. We consider each type of inputs. Recall that the library offers commands for (t,w) threshold encryption.
  - User inputs. There are n ports, which can make at most $(w+1)n\mathsf{max\_in}(k)$ entries in D[15].

    Also inputs from these ports can lead up to $n\mathsf{max\_in}()+\mathsf{max\_hnd}(k)$ output at $\mathsf{out_a}!$ and $\mathsf{max\_hnd}(k)$ outputs at $\mathsf{in_a}!$

[15]When all called commands are to generate keys

– Network inputs. There are at most $2n^2$ network ports which can give at most $2n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ outputs at $\mathsf{in_a}!$ and $2n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ outputs at $\mathsf{out_a}!$

  Also these inputs can lead up to at most $4n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ entries in D.

– Inputs from ideal secure channels. These inputs make neither entries in D nor outputs at $\mathsf{out_a}!$ and $\mathsf{in_a}!$.

– Inputs at $\mathsf{in_{nizk,a}}?$. These inputs make neither entries in D nor outputs at $\mathsf{out_a}!$ and $\mathsf{in_a}!$[16].

– Inputs at $\mathsf{in_{key,a}}?$. These inputs can lead up to $(w+1)\mathsf{max\_in}(k)$, $\mathsf{max\_in}(k)$ outputs at $\mathsf{in_a}!$.

Therefore, we can have at most $(w+1)n\mathsf{max\_in}(k) + 4n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k) + (w+1)\mathsf{max\_in}(k)$ entries in D. It means that the bound $\mathsf{max\_hnd}(k) = (2w+6)n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ is safe.

For the second part of part 5, we see that there are at most $n\mathsf{max\_in}() + \mathsf{max\_hnd}(k) + 2n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ outputs at $\mathsf{out_a}!$ and $\mathsf{max\_hnd}(k) + 2n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k) + \mathsf{max\_in}(k)$ outputs at $\mathsf{in_a}!$, both values are less than the value $(2w+10)n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ of $bound_{\mathsf{out_a}}?$ of $\mathsf{Sim}_{\mathcal{H}}$ and $bound_{\mathsf{in_a}}?$ of $\mathsf{TH}_{\mathcal{H}}$. □

[16]Because the call to FNIZK actually originated from inputs at other ports. FNIZK just provides proofs.

Chapter 7

SECURITY PROOF

7.1 Security of the cryptographic library

Theorem 7.1.1. (Security of the cryptographic library) Given $\mathcal{E}\_\mathsf{thres}$, FNIZK, FKEY, for all $n \in \mathbb{N}$, all correct parameters L' and all $\mathcal{H} \subseteq \{1, ..., n\}$, there exist a simulator $\mathsf{Sim}_{\mathcal{H}}$ that satisfies the following property: For all polynomial-time honest users H and adversary A, the view of H while interacting with correct machines $\mathsf{M}_{u,\mathcal{H}}$ for all $u \in \mathcal{H}$ and A is polynomially indistinguishable from the view of H while interacting with $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ with a parameter $L := \mathsf{R2Ipar}(\mathcal{E}\_\mathsf{thres}, \mathsf{FNIZK}, \mathsf{FKEY}^{\mathcal{E}\_\mathsf{thres}}, L')$. $\Box$

Intuitively, the theorem means that given these cryptographic components, we can always build a real cryptographic library that is "equivalent" to the ideal library, which is definitely secure or "trusted". In other words, using these primitives, we can always build a real cryptographic library that is similar and "at least as secure as" the ideal one.

Note that we require the following conditions:

- Decryption shares must be sent over authentic or secure channels as stated in Section 5.2.
- If honest users and the adversary share w secret keys for a public key, in which every group of t shares can decrypt a ciphertext, then the adversary must have less than t share and $t - 1 < w$, as stated in Section 5.1.2.

It turns out that even we require some conditions, we can still achieve the desired result, as Backes et al. [BDHK08] has demonstrated Conditional

Reactive Simulatability.

7.2 Outline of the proof

We follow the proof approach in [BPW03a]. For self containment, we repeat the outline here. Note that the details of the proof however are different due to some different cryptographic operations.

The proof will include the following steps, which are depicted in Figure 7.1:

- *Step 0 - Introducing encryption machines*: We introduce two encryption machines, $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$. The first one computes correct encryption of a message m, while the second one encrypts the fixed message $1^|m|$. Then we prove $\mathsf{Enc}_{\mathcal{H}}$ is at least as secure as $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.

- *Step 1 - Refactoring the real library*: We rewrite the real library so that machine M_u calls $\mathsf{Enc}_{\mathcal{H}}$ for all crytographic operations . We call new machines $\mathsf{M'}_u$.

- *Step 2 - Replacing with the ideal encryption machine*: We replace $\mathsf{Enc}_{\mathcal{H}}$ with $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and prove that the new composed system is as least as secure as the original one (composition theorem).

- *Step 3 - Combined system*: We introduce an intermediate system $\mathsf{C}_{\mathcal{H}}$, which has the combined state space of two systems: One is the combination $\mathsf{M}_{\mathcal{H}}$ of $\mathsf{M'}_u$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and the other is the combination $\mathsf{THSim}_{\mathcal{H}}$ of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$.

- *Step 4 - Bisimulation with error sets*: We prove that the joint view of H and A is indistinguishable between interacting with $\mathsf{C}_{\mathcal{H}}$ and two systems $\mathsf{M}_{\mathcal{H}}$ and $\mathsf{THSim}_{\mathcal{H}}$, except for certain runs called *error sets*. By transitivity and symmetric of indistinguishability we get the indistinguishability between $\mathsf{M}_{\mathcal{H}}$ and $\mathsf{THSim}_{\mathcal{H}}$ with error sets.

- *Step 5 - Reduction proof against the underlying cryptography*: We show that the aggregated probability of runs in error sets is negligible. It means $\mathsf{M}_{\mathcal{H}}$ is computationally at least as secure as $\mathsf{THSim}_{\mathcal{H}}$.

- *Step 6 - Proof conclusion*: We get our desired result: The real library is computationally at least as secure as the ideal one.

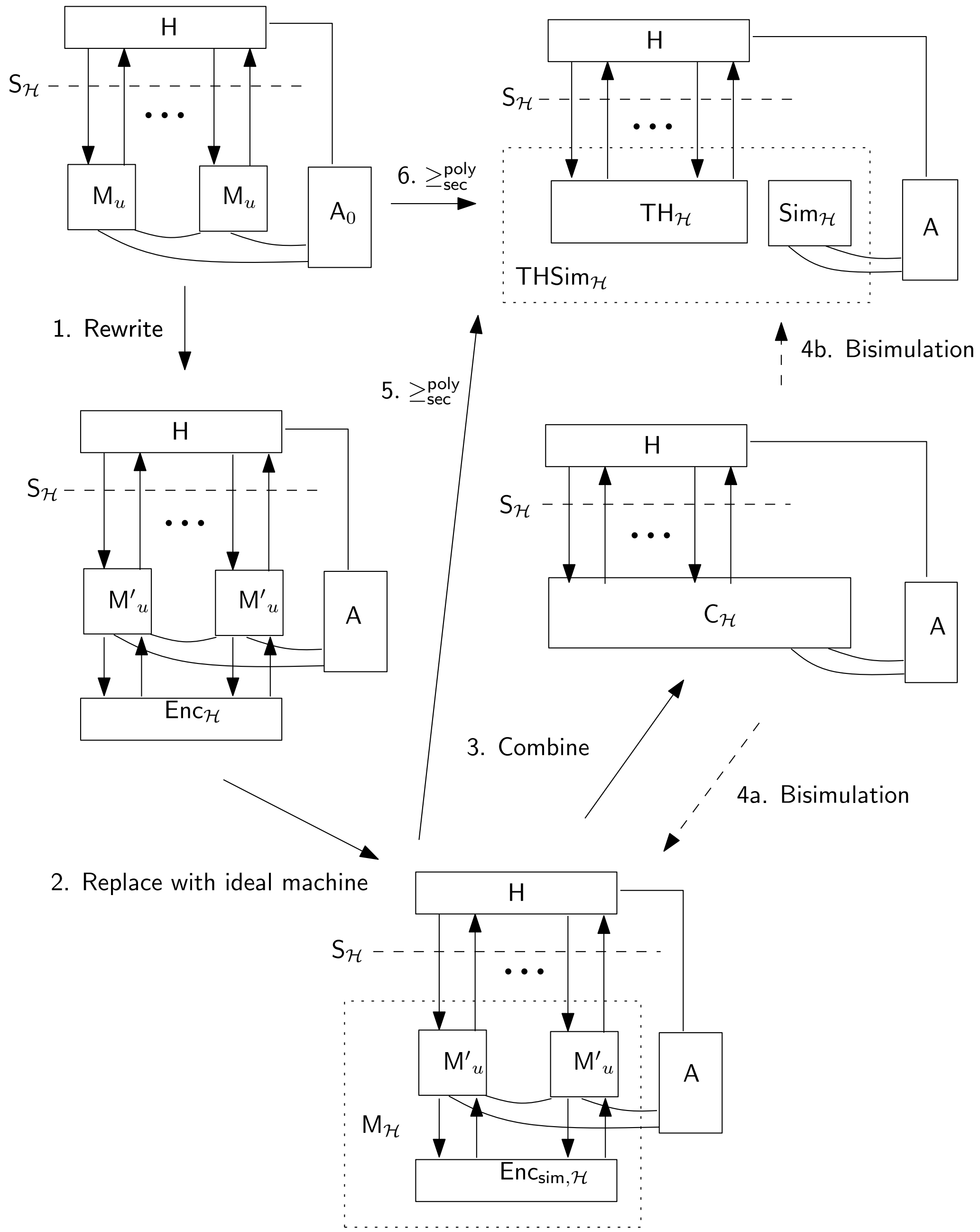

Figure 7.1: Steps of proof [BPW03a]

7.3 Encryption machines

We introduce here two encryption system $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$, which have been used in [PW01, BPW03a] but in different versions. These two encryption machines here also have database notations as that in [BPW03a], but have threshold homomorphic encryption commands instead of IND-CCA2 encrytion commands in [PW01, BPW03a].

The idea is that we replace each encryption of plaintext m with an encryption of a fixed one $1^{|m|}$, then the decryption task will be done by table look up. Basically, the machine are for honest users to use. However, the adversary in threshold setting can get some information, i.e some secret shares. For adversarial encrypting (decrypting) work, the adversary can do in any way he wants, e.g building his own encryption machine or do it by himself. Finally we will show that the view of honest users H is indistinguishable from using or not using the encryption machine.

Scheme 7.3.1. (Ideal and Real Encryption Machines) Let an encryption scheme $\mathcal{E}\_\mathsf{thres}$, two functionalities FKEY, FNIZK, parameter $n \in \mathbb{N}$ and two functions $s_{keys}, s_{encs} : \mathbb{N} \Rightarrow \mathbb{N}$ be given where

- $\mathcal{E}\_\mathsf{thres} = (\mathsf{FKEY}, \mathsf{E\_thres}, \mathsf{D\_thres}, \mathsf{C\_thres}, \mathsf{enc\_thres\_len},$ $\mathsf{pk\_thres\_len})$ as described in Section 2.5.
- $s_{keys}(k)$ denotes the maximum number of keys that can be generated for $\mathcal{E}\_\mathsf{thres}$. $s_{encs}(k)$ denotes the maximum number of encryptions per key in $\mathcal{E}\_\mathsf{thres}$. These two functions must be polynomially bounded in k.

We define two ecnryption systems

- $Sys^{\mathsf{enc,sim}}_{n,s_{keys},s_{encs}} := \{(\{\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}\}, S_{\mathsf{enc},\mathcal{H}}) | \mathcal{H} \subseteq \{1, ..., n\}\}$;
- $Sys^{\mathsf{enc,real}}_{n,s_{keys},s_{encs}} := \{(\{\mathsf{Enc}_{\mathcal{H}}\}, S_{\mathsf{enc},\mathcal{H}}) | \mathcal{H} \subseteq \{1, ..., n\}\}$.

And for every $\mathcal{H}$, the corresponding userports are

- $Ports_{\mathsf{Enc}_{\mathcal{H}}} := Ports_{\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}} := \{\mathsf{in}_{\mathsf{enc},u}?, \mathsf{out}_{\mathsf{enc},u}!, \mathsf{out}_{\mathsf{enc},u}{}^{\triangleleft}! | u \in \mathcal{H}\}$
- $S_{\mathsf{enc},\mathcal{H}} := \{\mathsf{in}_{\mathsf{enc},u}!, \mathsf{in}_{\mathsf{enc},u}{}^{\triangleleft}!, \mathsf{out}_{\mathsf{enc},u}? | u \in \mathcal{H}\}$

Each machine has a security parameter k, a key counter $curkey \in \mathbb{N}$ starting from 0 and initially empty databases $keys$, $keyscounter$ and $ciphers$. Each entry in $keys$ has attributes $(owner, skenc, pkenc)$, which are used to map key pairs. Each entry in $keyscounter$ has attributes $(owner, pkenc, ec)$, which are used to count the number of encryptions per public key. Each entry in $ciphers$ has attributes $(msg, pkenc, ciph)$, which are used to look up intended plaintexts. The offered commands of the two systems work by the following rules. Let $\mathsf{in}_{\mathsf{enc},u}?$ be the current input port. In the following rules, when we say that the machine outputs something, we mean the resulting output will be in $\mathsf{out}_{\mathsf{enc},u}!$, with $\mathsf{out}_{\mathsf{enc},u}{}^{\triangleleft}! := 1$. In other cases of different output ports, i.e output to the adversary, we will state it explicitly.

- $pk \leftarrow (\mathsf{generate}, u_1, ..., u_z)$: (In both $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and $\mathsf{Enc}_{\mathcal{H}}$) If $z \leq (w-t)$,i.e the adversary gets at least t secret shares, then output $\downarrow$ and stop. Otherwise if $curkey < s_{keys}(k)$ then
  - set $curkey := curkey + 1$;
  - activate FKEY and collect secret shares $\{pk, sk_1, ..., sk_w\}$ from output ports;
  - add $keyscounter :\Leftarrow (u, pkenc, 0)$;
  - for $i = 1$ to z do
    - $keys :\Leftarrow (u_i, (sk_i, i), pk)$;
  - for $i = z + 1$ to w do
    - output (sk_i, i) to the adversary via port $\mathsf{out}_a!$;
  - finally output pk.

  Otherwise output $\downarrow$.

- $c, p \leftarrow (\mathsf{encrypt}, pk, m)$, for $pk, m \in \{0,1\}^+$: Let $K := keyscounter[$ $pkenc = pk]$. If $K =\downarrow$ or $K.ec \geq s_{encs}(k)$, then output $\downarrow$, otherwise set $K.ec := K.ec + 1$ and
  - for $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$: Pick an $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$. Set $c \leftarrow \mathsf{E\_thres}_{pk}(1^{|m|}, r)$, give $((c, pk), (r, 1^{|m|}))$ to FNIZK and get proof p. If $p =\downarrow$ then return $\downarrow$. Otherwise add $ciphers :\Leftarrow (m, pk, c)$ and finally output (c, p).
  - for $\mathsf{Enc}_{\mathcal{H}}$: Pick an $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$. Set $c \leftarrow \mathsf{E\_thres}_{pk}(m, r)$. Give $((c, pk), (r, m))$ to FNIZK and get proof p. If $p =\downarrow$ then return $\downarrow$. Otherwise output (c, p).

- $ds \leftarrow (\mathsf{decrypt}, pk, c_1, ..., c_l, p_1, ..., p_l)$, for $pk, c_1, ..., c_l, p_1, ..., p_l \in \{0,1\}^+$: Use FNIZK to check the validity of every ciphertext in $\{c_1, ..., c_l\}$. If any of them is invalid, output $\downarrow$. Otherwise let $K := keys[pkenc = pk \wedge owner = u]$. If $K =\downarrow$ then output $\downarrow$, else let $sk := K.skenc$ and
  - for $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$: for $i = 1$ to l do
    * let $m_i := ciphers[pkenc = pk \wedge ciph = c_i].msg$;
    * let $m := m_1 \boxplus \cdots \boxplus m_l$
      Make a simulated output ds so that it can later be combined to $c_1 \boxdot \cdots \boxdot c_l$[1].
  - for $\mathsf{Enc}_{\mathcal{H}}$: Return $ds \leftarrow \mathsf{D\_thres}_{sk}(c_1 \boxdot \cdots \boxdot c_l)$. $\diamond$

- $m \leftarrow (\mathsf{combine}, ds_1, ..., ds_t)$, $pk, ds_1, ..., ds_t \in \{0,1\}^+$: (In both $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and $\mathsf{Enc}_{\mathcal{H}}$) output $m := \mathsf{C\_thres}(ds_1, ..., ds_t)$.

We prove that the following lemma holds for these two machines.

Lemma 7.3.1. The encryption machines have the following properties:

1. The two systems are computationally indistinguishable (even without a simulator), i.e $Sys^{\mathsf{enc,real}}_{n,s_{keys},s_{encs}} \geq^{\mathsf{f,poly}}_{\mathsf{sec}} Sys^{\mathsf{enc,sim}}_{n,s_{keys},s_{encs}}$ holds for the canonical mapping f and all parameters $n \in \mathbb{N}$ and $s_{keys}, s_{encs} \in \mathbb{N}[x]$.
2. Each transition is polynomial-time. $\square$

Proof. (This proof follows the idea in [PW01] where the author proved the indisinguishability of encryption machines with IND-CCA2 encryption scheme).

Part 1. Let n, s_{keys}, s_{encs} and $\mathcal{H}$ be fixed. The ideal adversary A_1 uses the real one A_2 as a black box without any change,i.e $\mathsf{A}_2 = \mathsf{A}_1 = \mathsf{A}$. It means that the two systems are indistinguishable even without a simulator. We use the hybrid argument technique by constructing intermediate systems that differ only in one encryption each.

For every $k \in \mathbb{N}$ let $\mathcal{I}_k := (\{1, ..., s_{keys}(k)\} \times \{1, ..., s_{encs}(k)\}) \cup \{\alpha\}$. Let $<_k$ be the lexicographic order on $\mathcal{I}_k \backslash \{\alpha\}$ and $\alpha \leq_k t$ for all $t \in \mathcal{I}_k$. Let $\mathsf{pred}_k(t)$ be the predecessor of $t \in \mathcal{I}_k$ in the order $<_k$ and $w(k) := (s_{keys}(k), s_{encs}(k))$.

For every $k \in \mathbb{N}$ and $t \in \mathcal{I}_k$ we define a hybrid machine $\mathsf{Enc}_{k,t,\mathcal{H}}$. It is the same as $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ except for encryption and decryption calls. Let $t' := (curkey, K.ec)$ for the value $curkey, K.ec$ at that moment, then

[1]Note that it is indistinguishable from the joint view of H and A. See Section 2.5.

- If $t' <_k t$, for an encryption call, it sets $c, r \leftarrow \mathsf{E\_thres}_{pk}(m)$, for the rest it works like $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$;
- If $t' =_k t$,
  - for an encryption call, it pick a random coin r and sets $c \leftarrow \mathsf{E\_thres}_{pk}(m, r)$;
  - for the rest it works like $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$;
- If $t' >_k t$, it works like $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.

Obviously we can see that

- Every $\mathsf{Enc}_{k,\alpha,\mathcal{H}}$ works like $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ with security parameter k.
- Every $\mathsf{Enc}_{k,w(k),\mathcal{H}}$ works like $\mathsf{Enc}_{\mathcal{H}}$ with security parameter k.

We will show that if the theorem is wrong then we have a contradiction. Assume that the theorem is wrong. Let $conf_{\mathsf{real}} := (\{\mathsf{Enc}_{\mathcal{H}}\}, S_{\mathsf{enc},\mathcal{H}}, \mathsf{H}, \mathsf{A})$ and $conf_{\mathsf{sim}}$ similarly with $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and let $coll_{k,t}$ denote the collection of $\{\mathsf{Enc}_{k,t,\mathcal{H}}, \mathsf{H}, \mathsf{A}\}$. Because we assume that the theorem is wrong so $view_{conf_{\mathsf{real}}}(\mathsf{H}) \not\approx_{\mathsf{poly}} view_{conf_{\mathsf{sim}}}(\mathsf{H})$ and it implies

$$(view_{coll_{k,w(k)}}(\mathsf{H}))_{k\in\mathbb{N}} \not\approx_{\mathsf{poly}} (view_{coll_{k,\alpha}}(\mathsf{H}))_{k\in\mathbb{N}}$$

.

We abbreviate $view_{k,t} := view_{coll_{k,t}}(\mathsf{H})$. The indistinguishability means that there exists a probabilistic polynomial-time distinguisher $\triangle$ and $p \in \mathbb{N}[x]$ such that for all k in an infinite set $\mathcal{K} \subset \mathbb{N}$,

$$|P(\triangle(view_{k,w(k)}) = 1) - P(\triangle(view_{k,\alpha}) = 1)| > \frac{1}{p(k)}$$

Now we construct an adversary $\mathsf{A}_{\mathsf{enc}}$ to play IND-TCPA game against the encryption scheme according to IND-TCPA security definition in Section 5.1.2. $\mathsf{A}_{\mathsf{enc}}$ uses the machines H, A, FKEY, its modified FNIZK and its own encryption functionality and communicates with the decryption oracle Dec to simulate a view of H and A using a hybrid machine (See figure 7.2). After that $\mathsf{A}_{\mathsf{enc}}$ submit the simulated view to Δ and use the answer from Δ to win the IND-TCPA game. We describe the construction in details as follows.

With security parameter k, $\mathsf{A}_{\mathsf{enc}}$ randomly choses $t \in_{\mathcal{R}} \mathcal{I}_k \backslash \{\alpha\}$, say $t = (kc, s')$, interacts with Dec to get a public-key pke and then try to play the

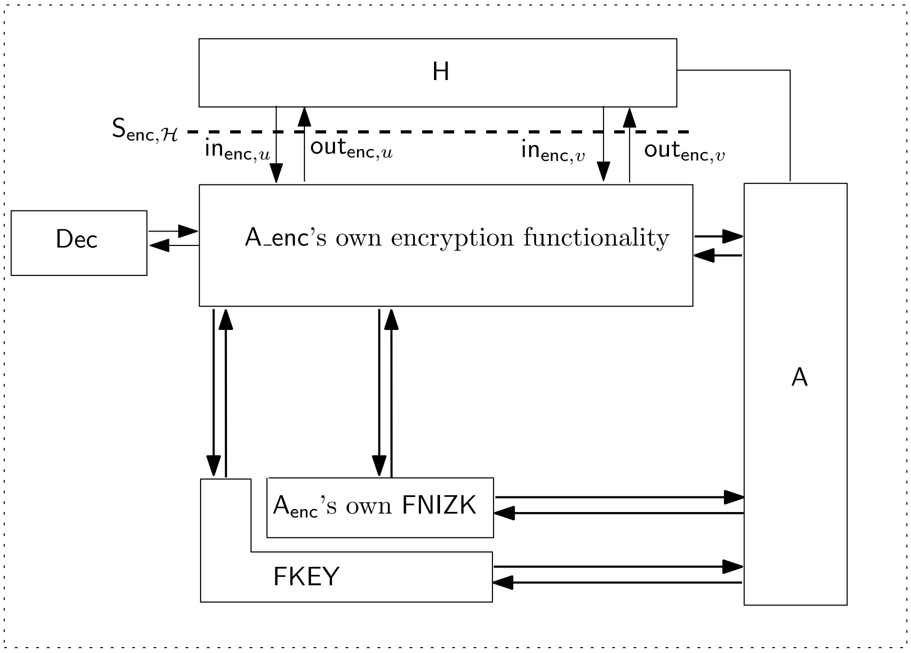

Figure 7.2: $\mathsf{A}_{\mathsf{enc}}$ produces a simulated view of H and A using a hybrid machine

IND-TCPA security game. $\mathsf{A}_{\mathsf{enc}}$ now can simulate $coll_{k,t}$ as exactly as $\mathsf{Enc}_{k,t,\mathcal{H}}$ does, except the following rules:

- if H makes the kc-th input $(\mathsf{generate}, u_1, ...u_n)$ from a certain port, say $\mathsf{in}_{\mathsf{enc},u}$? then $\mathsf{A}_{\mathsf{enc}}$ adds $(u_i, 0, pke)$ to $keys$ for every i from 1 to n and add $(u, pke, 0)$ to $keyscounter$, instead of calling FKEY to get new keys.

  After that, if H wants to get a decryption share which is not in the database, instead of using $\mathsf{D\_thres}()$[2], $\mathsf{A}_{\mathsf{enc}}$ gives the corresponding NIZK proofs to FNIZK. Of course FNIZK does not have the corresponding witnesses, so it gets from the adversary. If FNIZK does not accept any proof, $\mathsf{A}_{\mathsf{enc}}$ returns $\downarrow$. Otherwise $\mathsf{A}_{\mathsf{enc}}$ can capture the witnesses and then extracts the plaintexts[3]. Then $\mathsf{A}_{\mathsf{enc}}$ combines the plaintexts and ciphertext to the final plaintext and ciphertext, say m and c, and submits $(\mathsf{share}, c, m, u)$ to Dec get the correct decryption share. Note that it is allowed in IND-TCPA security game, as long as $\mathsf{A}_{\mathsf{enc}}$ knows a correct pair of plaintext and ciphertext.

- If H makes an input $(\mathsf{encrypt}, pke, m)$ which reaches the index $t =$

[2]In this case $\mathsf{A}_{\mathsf{enc}}$ does not have enough secret shares to make all possible decryption shares

[3]$\mathsf{A}_{\mathsf{enc}}$ can do that because everything happens inside an environment simulated by $\mathsf{A}_{\mathsf{enc}}$.

(kc, s'), then $\mathsf{A_{enc}}$ sends $(m_0, m_1) := (m, 1^{|k|})$ to Dec. Dec flips a bit $b \in \{0,1\}$ and returns the ciphertext $c \leftarrow \mathsf{E}_{pke}(m_b)$. $\mathsf{A_{enc}}$ adds (m, pke, c) to $ciphers$, gets a proof p from its own FNIZK [4] and then returns (c, p).

Later, if H wants to get a decryption share from the this ciphertext c, $\mathsf{A_{enc}}$ then makes a simulated output ds so that it can later be combined to m This action makes the simulated view still the same as actual view from H using $\mathsf{Enc}_{k,t,\mathcal{H}}$.

Everything else should be the same as $\mathsf{Enc}_{k,t,\mathcal{H}}$ does.

At the end, $\mathsf{A_{enc}}$ runs $\triangle$ on the resulting simulated view on H and gets an output bit b^*_k. $\mathsf{A_{enc}}$ then outputs b^*_k. Recall that in an IND-TCPA security game, if there exists an adversary that has non-negligible advantage, then the encryption scheme is not IND-TCPA secure.

Let $view^{(b)}_k$ be the random variable of the view of H in $\mathsf{A_{enc}}$ for parameter k and bit b. We denote $pr_{k,t} := P(\triangle(view_{k,t} = 1))$ and $pr^{(b)}_k := P(\triangle(view^{(b)}_k = 1))$. Note that for $b = 0$ the simulated run is the same as a run of $coll_{k,t}$ whereas for $b = 1$ the simulated run is the same as a run of $coll_{k,\mathsf{pred}_k(t)}$. We can compute

$$pr^{(0)}_k = \frac{1}{w(k)} \sum_{t \in \mathcal{I}_k \setminus \{\alpha\}} pr_{k,t};$$
$$pr^{(1)}_k = \frac{1}{w(k)} \sum_{t \in \mathcal{I}_k \setminus \{\alpha\}} pr_{k,\mathsf{pred}_k(t)}.$$

It implies

$$|pr^{(0)}_k - pr^{(1)}_k| = \frac{1}{w(k)} |pr_{k,w(k)} - pr_{k,\alpha}| > \frac{1}{w(k)p(k)}.$$

Therefore

$$\begin{aligned} P(b^*_k = b) &= P(b = 0 \wedge \triangle(view^{(b)}_k) = 0) + P(b = 1 \wedge \triangle(view^{(b)}_k) = 1) \\ &= \frac{1}{2}(P(\triangle(view^{(0)}_k = 0)) + P(\triangle(view^{(1)}_k = 1))) \\ &= \frac{1}{2} + \frac{1}{2}(pr^{(1)}_k - pr^{(0)}_k). \end{aligned}$$

[4] Although the proof is provided from a different FNIZK, the simulated view is still the same from real view, because $\mathsf{A_{enc}}$ guarantees that the plaintext is valid (everything is controlled by $\mathsf{A_{enc}}$)

Finally we get

$$|Pr[b_k^* = b] - 1/2| > \frac{1}{w(k)p(k)} \text{ , for all } k \in \mathcal{K}.$$

And because the advantage $\frac{1}{w(k)p(k)}$ is not a negligible function, we get the desired contradiction.

Part 2. This part can be proved easily by inspection the work of the two encryption systems. □

7.4 Refactoring the real library

We rewrite the real machines M_u so that they use the encryption machine $\mathsf{Enc}_{\mathcal{H}}$ instead of doing all encryption work by themselves (See Figure 7.3). We call the new machines M'_u. The parameters for the encryption systems are now set to $s_{keys} := s_{encs} := \mathsf{max\_in}(k) + 1$[5].

Each M'_u uses the port $\mathsf{in}_{\mathsf{enc},u}!, \mathsf{out}_{\mathsf{enc},u}?, \mathsf{in}_{\mathsf{enc},u}{}^{\triangleleft}!$ to communicate with the encryption machine. It also use a counter $steps_{\mathsf{out}_{\mathsf{enc},u}?}$ that is bounded by $bound_{\mathsf{out}_{\mathsf{enc},u}?} := \mathsf{max\_in}(k) + 1$.

Now we define the modification of some transitions. Note that the state is unmodified, except that the entries for secret shares are different because now the encryption machine keeps the shares.

- $pk^{\mathsf{hnd}} \leftarrow \mathsf{gen\_enc\_thres\_keylist}(\mathsf{u_1}, ..., \mathsf{u_j})$ for $(w - t) < j \leq w$.

  Call the encryption machine with $pk \leftarrow (\mathsf{generate}, u_1, ..., u_z)$. Let $pk^* := (\mathsf{pke}, pk)$, set $pk^{\mathsf{hnd}} := curhnd_u + +$ and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, \mathsf{pke}, ())$.

  If a M'_u is supposed to get a secret share now just have a public key[6]. Therefore in addition to adding an entry for the public key, it also adds another entry for its unknown secret share. Let $sk^* := (\mathsf{ske}, \epsilon, pk, i)$[7], set $sk^{\mathsf{hnd}} := curhnd_u + +$ and $D_u :\Leftarrow (sk^{\mathsf{hnd}}, sk^*, \mathsf{ske}, ())$.

- $c^*, p^* \leftarrow \mathsf{make\_enc}(pk^*, m)$ for $pk^*, m \in \{0,1\}^+$.

  It returns $\downarrow$ if $\mathsf{valid}(m) = \mathsf{false}$[8].

  Otherwise let $pk := pk^*[2]$, call the encryption machine with $c, p \leftarrow (\mathsf{encrypt}, pk, m)$.

---

[5]So the encryption machine never reaches its bounds
[6]Its secret share is kept by the encryption machine
[7]ϵ stands for an unknown secret share.
[8]We do not accept invalid messages

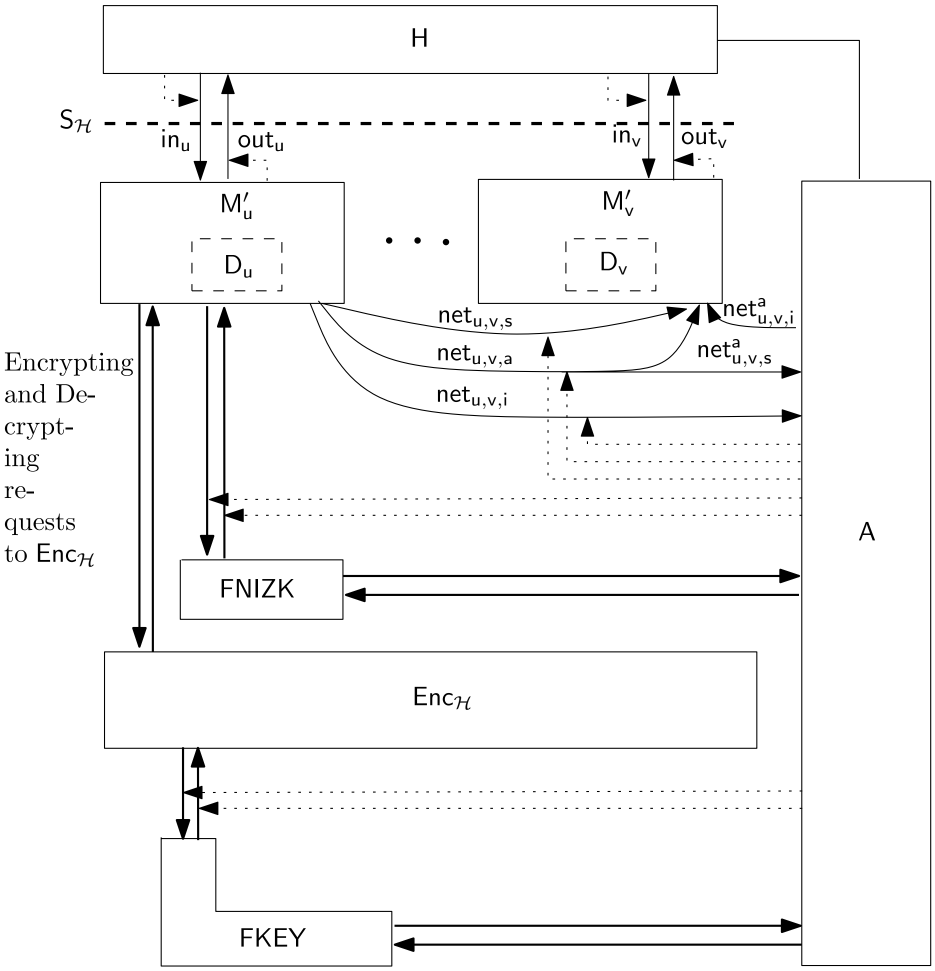

Figure 7.3: Refactoring the real library

If $c \neq \downarrow$ then set $c^* := (\mathsf{enc}, pk, c)$.

Else encrypt directly[9]: Pick an $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$. Let $pk := pk^*[2]$, encrypt $c \leftarrow \mathsf{E\_thres}_{pk}(m, r)$ and set $c^* := (\mathsf{enc}, pk, c)$. Let $w := (m, r)$ and $x := (c, pk)$. Submit (x, w) to FNIZK, getting back p. If $p = \downarrow$ then return $\downarrow$.

Let $p^* := (\mathsf{nizk}, p, c, pk)$.

- $ds^{\mathsf{hnd}} \leftarrow \mathsf{decrypt\_thres}(sk^{hnd}, c_1^{\mathsf{hnd}}, ..., c_j^{\mathsf{hnd}}, p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}})$ where $\{c_1^{\mathsf{hnd}}, ...$ $, c_j^{\mathsf{hnd}}\}$ is a set of ciphertexts, $\{p_1^{\mathsf{hnd}}, ..., p_j^{\mathsf{hnd}}\}$ is a set of corresponding proofs for plaintext validity.

[9]It was an adversary's public key

Parse c_i^{hnd} to have $arg_i := (pk_i^*)$. Return $\downarrow$ if all of pk_i^* are not the same. Otherwise say the same public key is pk^*. Return $\downarrow$ if $D_u[sk^{hnd}].type \neq$ ske or $D_u[sk^{hnd}].word[3] \neq pk^*$ or any $D_u[c_i^{hnd}].type \neq \mathsf{enc}$ or any $D_u[p_i^{hnd}].type \neq \mathsf{nizk}$.

Otherwise set $pk := pk^*[2]$, $c_i := D_u[c_i^{hnd}].word[3]$, $p_i := D_u[p_i^{hnd}].word[2]$ then call the encryption machine with $ds \leftarrow (\mathsf{decrypt}, pk, c_1, ..., c_l, p_1, ..., p_l)$. If $ds =\downarrow$ then return $\downarrow$. Otherwise set $ds^* := (\mathsf{dec\_shr}, ds, c, D_u[sk^{hnd}].word[4])$ and $(ds^{\mathsf{hnd}}, D_u) :\leftarrow (ds^*, \mathsf{decshr}, ())$.

$m \leftarrow \mathsf{make\_combi}(ds_1^*, ..., ds_t^*)$.

Let $ds_i := ds_i^*[2]$. Call the encryption machine with $m \leftarrow (\mathsf{combine}, ds_1, ...$ $, ds_t)$.

We claim that the modified system has the following properties

Lemma 7.4.1. The modified machines $\mathsf{M'}_u$ have the following properties:

1. $\mathsf{M'}_u$ are polynomial-time.
2. When they are used together with $\mathsf{Enc}_{\mathcal{H}}$ (even with $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$), no counter $steps_{\mathsf{out}_{\mathsf{enc},u}?}$, $curkey$ or $K.ec$ reaches its bound.
3. If they are used with $\mathsf{Enc}_{\mathcal{H}}$, their behaviour is perfectly indistinguishable from the real library for A and H. □

Proof.

1. Part 1 holds because for each machine $\mathsf{M'}_u$ the new input port $\mathsf{out}_{\mathsf{enc},u}!$ accepts only a polynomial number of inputs (its is bounded), the length function for this port is also bounded and every rewritten transition takes only a polynomial number of steps.
2. Part 2 can be proven by comparing these bounds with the bounds for other ports.
3. For part 3, we can see that the only change is that secret shares are stored in $keys$ instead of D_u. However, in every case that M_u uses a secret share, $\mathsf{M'}_u$ also use the same one. Therefore, actually $\mathsf{M'}_u$ always output the same thing as M_u do. □

7.5 Replacing with the ideal encryption machine

Because $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ are indistinguishable (Lemma 7.3.1) and the rewritten real library is polynomial-time (Lemma 7.4.1), according to the composition theorem in [PW01, Can01] we can replace $\mathsf{Enc}_{\mathcal{H}}$ with $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and the new system's behaviour is computationally indistinguishable from the original real library for A and H.

7.6 Combined system

It is difficult to compare $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$ directly. Therefore we define an intermediate machine call $\mathsf{C}_{\mathcal{H}}$ that has a combined state space of $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$.

Then we will prove that $\mathsf{C}_{\mathcal{H}}$ has a *bisimulation* relation to each of $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$. By transitivity of indistinguishability we have the desired proof.

7.6.1 Timing

Here we consider macro-transitions, in which for an input all sub-machines run until the control is returned back to A or H.

7.6.2 Definition of $\mathsf{THSim}_{\mathcal{H}}$

State of $\mathsf{C}_{\mathcal{H}}$

A part of state of $\mathsf{C}_{\mathcal{H}}$ is a database D^* that is structured like D of $\mathsf{TH}_{\mathcal{H}}$. However, each entry may have the following additional attributes:

- $x.word \in \{0,1\}^*$ contains real data as in $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{Sim}_{\mathcal{H}}$ under the same handle. If $x.type = ske$ and it is a key for adversary, then $x.word = \epsilon$. In other cases it is always non-empty.
- $x.parsed_u \in \{\mathsf{true}, \mathsf{false}, \downarrow\}^*$ for $u \in \{\mathcal{H}\}$. This value is $\downarrow$ if $x.hnd_u = \downarrow$, true if the entry would be parsed in D_u, or false if the entry would be still of type null[10].

[10]We use it when making derivations.

- $x.owner$ for ciphertext where honest users keeps at least $2w/3$ secret shares. This value is adv if the ciphertext was received from the adversary, otherwise honest. In other cases it is $\downarrow$.
- $x.ec$ for public key. This value corresponds to the encryption counter in $keyscounter$ of encryption machines.

Transition of $\mathsf{C}_{\mathcal{H}}$

The D-part of D^*, the variables $size$, $curhnd_u$ and the ideal secure channels are treated exactly as in $\mathsf{TH}_{\mathcal{H}}$. Entries created by basic commands from H get the words created M'_u and real secret shares created by $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$. Words received from A are parsed and entered as by $\mathsf{Sim}_{\mathcal{H}}$. Outputs to H are made as in $\mathsf{TH}_{\mathcal{H}}$, outputs to A are made as in $\mathsf{M}_{\mathcal{H}}$.

7.6.3 Derivations

Now we define the derivation of each original system from the combined system. The idea is to compare $\mathsf{C}_{\mathcal{H}}$ to $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$, we then show that the derived states and outputs are the same as in the original systems, except for some certain cases we call $errorsets$.

We use the following notations:

- $\omega(i)$ denotes word lookup for index i, i.e. $\omega(i) = D^*[i].word$ if $i \in \mathcal{HNDS}$, otherwise $\omega(i) = i$.
- We use superscript * to denote derived states, except the derived state of $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{TH}_{\mathcal{H}}$.

Given a state of $\mathsf{C}_{\mathcal{H}}$, we have

- $\mathsf{TH}_{\mathcal{H}}$:
  - D : This is exactly the restriction of D^* except $word$ and $parsed_u$.
  - $curhnd_u, size, steps_{\mathsf{p?}}$: These variables equal those in $\mathsf{C}_{\mathcal{H}}$.
- $\mathsf{M}^*_{\mathcal{H}}$:
  - D^*_u : We start from an empty database. For every $x^{\mathsf{hnd}} \leq curhnd_u$, let $x := D^*[hnd_u = x^*\mathsf{hnd}].ind$, $type := D^*[x].type$, and $m := D^*[x].word$.

* If $D^*[x].parsed_u = \mathsf{false}$, then $D^*_u :\Leftarrow (x^*\mathsf{hnd}, m, \mathsf{null}, ())$.
  * Else if $type \neq \mathsf{ske}$, then $D^*_u :\Leftarrow (x^*\mathsf{hnd}, m, type, ())$.
  * Else if $type = \mathsf{ske}$, then $D^*_u :\Leftarrow (x^*\mathsf{hnd}, \epsilon, \mathsf{ske}, ())$[11].
- $curhnd^*_u$: Equal to $curhnd_u$ of $\mathsf{C}_{\mathcal{H}}$.
- $steps^*_{\mathsf{p}?}$: Equal $steps_{\mathsf{p}?}$ of $\mathsf{C}_{\mathcal{H}}$, except for $\mathsf{p}? = \mathsf{net}_{u,v,x}?$ with $(u, v, x) \in ch\_honest$, where they are equal to $steps_{\mathsf{net\_id}_{u,v,x}?}$.
- $net^*_{u,v,x}$: (For every $(u, v, x) \in ch\_honest$.) Let $net^*_{u,v,x} := \omega^*(net\_id_{u,v,x})$.

• $\mathsf{Sim}^*_{\mathcal{H}}$:

- D^*_{a} : We start from an empty database. For every $x^{\mathsf{hnd}} \leq curhnd_{\mathsf{a}}$, let $x := D^*[hnd_{\mathsf{a}} = x^{\mathsf{hnd}}].ind$, $type := D^*[x].type$, and $m := D^*[x].word$.
  * If $type \neq \mathsf{pke}$, then $D^*_{\mathsf{a}} :\Leftarrow (x^{\mathsf{hnd}}, m, ())$.
  * Else if $D^*[x].arg = ()$ then $D^*_u :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{adv}))$. Otherwise for all $sk^{\mathsf{ind}}_{u_i}$ with $D^*[ind = sk^{\mathsf{ind}}_{u_i} \wedge type = \mathsf{ske}].arg = (x)$, set $sk^*_{u_i} = D^*[sk^{\mathsf{ind}}_{u_i}].word$. Finally $D^*_{\mathsf{a}} :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{honest}, sk^*_{u_1}, ..., sk^*_{u_j}))$.
- $curhnd^*_{\mathsf{a}}$: Equal to $curhnd_{\mathsf{a}}$ of $\mathsf{C}_{\mathcal{H}}$.
- $steps_{\mathsf{p}?}$: Identical to $steps_{\mathsf{p}?}$ of $\mathsf{C}_{\mathcal{H}}$.

• $\mathsf{THSim}_{\mathcal{H}}$:

- $out_{\mathsf{a}}, in_{\mathsf{a}}$: Equal to those in $\mathsf{C}_{\mathcal{H}}$. They are empty after each macro-transition.
- $net\_id_{u,v,x}$: (For every $(u, v, x) \in ch\_honest$.) Equal to those in $\mathsf{C}_{\mathcal{H}}$.

• $\mathsf{Enc}^*_{\mathsf{sim},\mathcal{H}}$:

- $keys^*$: For every $sk^{\mathsf{ind}} \leq size$ with $D^*[sk^{\mathsf{ind}}].type = \mathsf{ske}$, $(pk^{\mathsf{ind}}) = D^*[sk^{\mathsf{ind}}].arg$, $u = \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$, let $keys^* :\Leftarrow (u, D^*[sk^{\mathsf{ind}}].word[2], D^*[pk^{\mathsf{ind}}].word[2])$.
- For every $pk^{\mathsf{ind}} \leq size$ with $D^*[pk^{\mathsf{ind}}].type = \mathsf{pke}$, let $keyscounter^*$ $:\Leftarrow (D^*[pk^{\mathsf{ind}}].word[3], D^*[pk^{\mathsf{ind}}].word[2], D^*[pk^{\mathsf{ind}}].ec)$
- For every $c^{\mathsf{ind}} \leq size$ with $D^*[c^{\mathsf{ind}}].type = \mathsf{enc}$ and $D^*[c^{\mathsf{ind}}].owner = \mathsf{honest}$, let $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}) := D^*[c^{\mathsf{ind}}].arg$ and $(l, pk^*, c^*) := \omega(l^{\mathsf{ind}}, pk^{\mathsf{ind}}, c^{\mathsf{ind}})$. Then $ciphers^* :\Leftarrow (l, pk^*[2], c^*[3])$.

[11] The secret share is kept by $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.

- $curkey^*$: $curkey^* :=$ the number of different $keys^*.pkenc$.
- Buffers to and from $\mathsf{Enc}^*_{\mathsf{sim},\mathcal{H}}$ are empty after every macro-transition.

- FNIZK^* :

  - $nizks^*$: Starting from an empty database. For every $p^{\mathsf{ind}} \leq size$ with $D^*[p^{\mathsf{ind}}].type = \mathsf{nizk}$, $(c^{\mathsf{ind}}) = D^*[p^{\mathsf{ind}}].arg \neq ()$, let $x := D^*[c^{\mathsf{ind}}].word[3]$, $p := D^*[p^{\mathsf{ind}}].word[2]$. Then $nizks^* :\Leftarrow (x, p)$.

We do not have derivation for FKEY because it is stateless.

7.6.4 Invariants in $\mathsf{C}_{\mathcal{H}}$

In addition properties of D-part mentioned in Lemma 4.5.1, $\mathsf{C}_{\mathcal{H}}$ has some more invariants as follows.

- *Fully defined.* For every $x \in D^*$, $x.ind, x.type, x.arg, x.length, x.word$ are never $\downarrow$.

- *Word uniqueness* For each word $m \in \{0,1\}^*$, we always have $|D^*[word = m \wedge type \neq \mathsf{ske}]| \leq 1$.

- *Correct length.* For all $i \leq size$, we have $D^*[i].len = |D^*[i].word|$, except when $D^*[i].type = \mathsf{ske}$.

- *No unparsed secret share.* If $u \in \mathsf{owner}(D^*[i])$ and $D^*[i].type = \mathsf{ske}$ then $D^*[i].parsed_u = \mathsf{true}$.

- *Correct arguments.* For all $i \leq size$, there is no conflict between a real message $m := D^*[i].word$ and its corresponding abstract type and argument, $type^{\mathsf{id}} := D^*[i].type$ and $arg^{\mathsf{ind}} := D^*[i].arg$.Let $arg^{\mathsf{real}} := \omega^*(arg^{\mathsf{ind}})$, then we require:

  - If $m = \epsilon$ then $type^{\mathsf{id}} = \mathsf{ske}$.
  - If $type^{\mathsf{id}} \neq \mathsf{ske}$, let $(type, arg^{\mathsf{parse}}) := \mathsf{parse}(m)$, then we require that $type = type^{\mathsf{id}}$ and also:
    - If $type \neq \mathsf{enc}$ then $arg^{\mathsf{parse}} = arg^{\mathsf{real}}$.
    - Else then $arg^{\mathsf{parse}}[1] = arg^{\mathsf{real}}[1]$ and let $o := D^*[i].owner$ then
      - If $D^*[arg^{\mathsf{ind}}[1]].arg = ()$ then $o = \downarrow$ and vice versa[12].

[12] All secret shares belong to the adversary.

- $\cdot$ If $o = \mathsf{honest}$ then $arg^{\mathsf{ind}}[2] \neq \downarrow$.
- $\cdot$ If $o = \mathsf{adv}$ then the decrypted and looked-up plaintext must equal.

- *Strongly correct arguments* if $\mathsf{a} \notin \mathsf{owners}(D^*[i])$ or $D^*[i].owner = \mathsf{honest}$. Intuitively it means that the probability of the real representation in each entry must coincide the probability of those in the real library.Let $type^{\mathsf{id}} := D^*[i].type$ and $arg^{\mathsf{ind}} := D^*[i].arg$.Let $arg^{\mathsf{real}} := \omega^*(arg^{\mathsf{ind}})$ and $m := D^*[i].word$ has the following distribution:

  - If $type \in \{\mathsf{type}, \mathsf{list}, \mathsf{nonce}\}$ then $m = \mathsf{make\_}type(arg^{\mathsf{real}})$.
  - If $type \in \{\mathsf{pke}, \mathsf{ske}\}$, then the corresponding key list $(pk^*, sk_1^*, ..., sk_w^*)$ are created from FKEY from corresponding inputs, which can be looked-up.
  - If $type = \{\mathsf{enc}, \mathsf{nizk}\}$ then the corresponding pair of ciphertext and zero-knowledge proof (c^*, p^*) must be compatible. c^* is of the form enc, pk, c and has $carg^{\mathsf{real}}$ of the form (pk^*, l) , while p^* is of the form $(\mathsf{nizk}, p, c^*, pk^*)$ with $pk = pk^*[2]$ and p is created by FNIZK from corresponding inputs.

    The distribution of c depends on o:

    * If $o = \mathsf{honest}$ then $c = \mathsf{E\_thres}_{pk}(1^{|(l,r)|})$ where $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$.
    * If $o = \downarrow$ (a ciphertext encrypted using the adversary's public key), then $c = \mathsf{E\_thres}_{pk}(l, r)$ where $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$.

- *Word secrecy.* It means the adversary never gets information from nonce-like words without adversary handles. We define a set Pub_Var data that A may know or get to know later:

  - All $D^*[i].word$ where $D^*[i].hnd_{\mathsf{hnd}} \neq \downarrow$;
  - The state of A and H;
  - The $\mathsf{TH}_{\mathcal{H}}$-part of $\mathsf{C}_{\mathcal{H}}$ and the ideal secure channels;
  - Secret shares where the corresponding public keys are public[13].

  We require that at all time, no other information gets into Pub_Var in the sense of information flow in programming languages.

Now we can define our bisimulation property as follows

[13]The idea is information about secret shares flows into public keys and decryptions. However, the adversary does not have handles to them [BPW03a].

Definition 7.6.1. Bisimulation property "An input retains all invariants" means:

- Resulting macro-transitions of $\mathsf{C}_{\mathcal{H}}$ retain the invariants that were true before the input.
- If some input is given to $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{THSim}_{\mathcal{H}}$ in a state derived from $\mathsf{C}_{\mathcal{H}}$, then the probability distribution of the next state coincides with that of the sate derived from the next state of $\mathsf{C}_{\mathcal{H}}$. We denote it "correct derivation". $\diamond$

Note that these conditions hold initially.

Before starting to compare the systems, we claim that $\mathsf{C}_{\mathcal{H}}$ has the following properties.

Lemma 7.6.1. $\mathsf{C}_{\mathcal{H}}$ has the following properties:

1. There are only the following modifications to existing entries x in D^*: assignments to previously undefined $x.hnd_u$ and then $x.parse_u := \mathsf{false}$, changes of $x.parse_u$ from false to true, and increase of the encryption counter in entries of type ske.
2. The function *owner* of a secret share never changes.
3. If a state change in $\mathsf{C}_{\mathcal{H}}$ contains only an assignment to a previously undefined $x.hnd_u$ and then $x.parse_u := false$ then only the following invariants have to be proven:
   - "Correct derivation" of D_u^* and $curhnd_u^*$.
   - "Word secrecy" if $u = \mathsf{a}$. $\square$

Proof. Part 1, 2 can be proven easily be checking the definition of $\mathsf{C}_{\mathcal{H}}$ and commands.

Part 3 is also obvious by checking that no other derivations are affected and other invariants still hold. $\square$

7.7 Comparison of the two pairs of systems

We will compare the systems based on all possible inputs. Because of the complex proving process, we will just briefly describe the comparison. For original proving approach, we refer to [BPW03a].

7.7.1 Comparison of basic commands

We consider a command c at a port $\mathsf{in}_u?$ and the corresponding output at the port $\mathsf{out}_u!$ with $u \in \mathcal{H}$. We consider only *well-formed* inputs (inputs in the correct domains).

General comparison

Lemma 7.7.1. For each command c at a port $\mathsf{in}_u?$ where c is not a command related to zero-knowledge proofs or gen_enc_thres_keylist, we have to show only the following properties:

- The outputs at $\mathsf{out}_u!$ in $\mathsf{C}_\mathcal{H}$ and $\mathsf{M}_\mathcal{H}$ are the same.
- "Correct derivation" of D^*_u and $curnhd^*_u$.
- The invariants in D^* are retained, except "word secrecy" is already obvious. □

Proof. All of the properties above can be easily proven if we do not consider commands related to key generation and zero-knowledge proofs because those commands are not local. They output something to the adversary. □

Type and Length queries, data, lists and nonces

For all of commands here, we can easily prove that they retain invariants and produce the same output in all three systems because they are just local commands. They produce outputs without interacting with other machines.

Note that sometimes a new nonce collides with an old noncefor the case of generating nonces. In that case we run in an error set called $Nonce\_Coll$. However, we will show later in Section 7.7.5 that this probability is negligible.

(t, w) threshold homomorphic encryption

Now we analyse more complicated commands.

- *Key generation.* $\mathsf{TH}_\mathcal{H}$, and thus $\mathsf{C}_\mathcal{H}$ outputs a new public key handle while M'_u calls $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$, which then calls FKEY to gets a new pubic key. M'_u gets that new public key, inserts it into D^*_u and output a new handle.

$\mathsf{TH}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$ outputs some new secret shares handles to some parties while FKEY also send new secret shares to the corresponding machines and those machines update their databases then output the new handles.

$\mathsf{TH}_{\mathcal{H}}$ outputs the secret share handle for adversary along with public key information then $\mathsf{Sim}_{\mathcal{H}}$ calls his own FKEY and this simulated FKEY output the secret shares for adversary together with public key information. On the other hand, after $\mathsf{M'}_u$ calls the real FKEY, it also output the secret shares for adversary together with public key information.

$\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ retains "correct derivation" because both it and $\mathsf{C}_{\mathcal{H}}$ updates the new public and honest user secret shares in to the database of each.

Here we may have an error where the new public key is equal to an old one, then "word uniqueness" does not retains. We call this error set Key_Coll.

- *Encryption.* It is clear that outputs, which are new handles, to honest users are the same.

  $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ will ask for a zero-knowledge proof from the adversary, while $\mathsf{TH}_{\mathcal{H}}$ also output the new proof handle together with the ciphertext, then $\mathsf{Sim}_{\mathcal{H}}$ can ask the same thing from the adversary[14].

  Here we may run into an error set called $Nonce\_Coll$ when the new nonce generated for the encryption is not new, then the new encryption may be equal to an old one, then "word uniqueness" is destroyed.

- *Decryption.* Honest users get the same decryption share handle if succeed. If there is any zero-knowledge proof received from the adversary before, both $\mathsf{Sim}_{\mathcal{H}}$ and FNIZK will ask the adversary for the witness.

- *Combination.* Honest users get the same decryption share handle and no state changes.

- *Public key retrieval.* Honest users get the same decryption share handle and no state changes.

[14]Note that we assume the adversary replies without delay.

7.7.2 Comparison of send commands by honest users

Secure channel

It is easy to check that all invariants are retained and right outputs are produced here.

To adversary

$(u, v, x \in ch\_to\_adv)$ Basically $\mathsf{Sim}_{\mathcal{H}}$ maintains his database D_{a} to store mappings between abstract and real representations. It is obvious to prove that $\mathsf{C}_{\mathcal{H}}$ out put the same message like M'_u at the end. We can see that $\mathsf{Sim}_{\mathcal{H}}$ provides output similar to M'_u by inspection of $\mathsf{Sim}_{\mathcal{H}}$. Furthermore, $\mathsf{Sim}_{\mathcal{H}}$ use the algorithm id2real to make real version of new received abstract terms. By inductive proof technique, we can show that id2real retains all invariants and also produce right outputs.

7.7.3 Comparison of input from the adversary

Network input

On input from a port $\mathsf{net}_{w,u,x}?$, $\mathsf{Sim}_{\mathcal{H}}$ and thus $\mathsf{C}_{\mathcal{H}}$ use the algorithm real2id to get the handle to abstract version. Also by inductive proof technique, we can show that by using real2id, $\mathsf{Sim}_{\mathcal{H}}$ and thus $\mathsf{C}_{\mathcal{H}}$ provide the same outputs and retain all invariants.

Note that in this case we may run in two error sets when the adversary has guessed correctly an existing nonce or an existing public key. We call these sets $Nonce\_Guess$ and Key_Guess .

Input to FKEY and FNIZK

- FKEY:The adversary input something to FKEY only when he wants to generate keys (The condition of $t < w/3$ must still holds). $\mathsf{M}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$ will get some new secret shares from that input and then output the handles to honest users. $\mathsf{Sim}_{\mathcal{H}}$ maintains also FKEY as a black box and then also create new abstract keys in $\mathsf{TH}_{\mathcal{H}}$. Therefore honest users will receive the same new handles and invariants hold.

  FNIZK :The adversary just reacts to some FNIZK's request, which are caused by some input from other input port, which have been shown for retaining all invariants and leading to the same outputs.

7.7.4 Comparison in scheduling of a secure channel

For $(w, u, x) \in ch\_honest$

Because of the invariant "correct derivation", before the input we have the buffer $net_{w,u,x}$ equals the real version of abstract buffer $net\_id_{w,u,x}$. So basically if the adversary give the same clock input, we have the same output and all invariants are still retained.

7.7.5 Error sets

There are following error sets that occur when we are comparing the three systems. We will show that their probability are negligible.

Nonce collisions

Nonce collisions of the error set$Nonce_Coll$ may happen as shown in Section 7.7.1 and Section 7.7.1. Because a nonce is picked randomly, so the probability of a collision is bounded by $2^{-\mathsf{nonce\_len}(k)}$, which is clearly negligible.

Key collisions

Key collisions of the error setKey_Coll may happen as shown in Section 7.7.1. It happens when a new public key is the same as an old one. Because of the semantic security of the underlying encryption scheme, we claim that the probability of the error setKey_Coll is negligible. Otherwise, an adversary can always attack the encryption scheme with non-negligible probability of success.

Nonce guessing

Nonce guessing of the error set$Nonce_Guess$ may happen as shown in Section 7.7.3. Similarly, the probability for this error to happen is bounded by $2^{-\mathsf{nonce\_len}(k)}$, which is clearly negligible.

Key guessing

Key guessing of the error setKey_Guess may happen as shown in Section 7.7.3. Similarly, the probability for this error to happen must also be negligible.

7.8 Proof conclusion

To conclude, by transitivity of indistinguishability, we claim that the real library is at least as secure as the ideal library. It also means the ideal library is a sound abstraction of the real library.

With the notations defined in [PW01] we can write the conclusion as follows: For all n and L' and $L := \mathsf{R2Ipar}(\mathcal{E}\_\mathsf{thres}, \mathsf{FNIZK}, \mathsf{FKEY}^{\mathcal{E}\_\mathsf{thres}}, L')$, we have

$$Sys^{\mathsf{cry,real}}_{n,\mathcal{E}\_\mathsf{thres},\mathsf{FNIZK},\mathsf{FKEY},L'} \geq^{\mathsf{f},\mathsf{poly}_{\mathsf{sec}}} Sys^{\mathsf{cry,id}}_{n,L}$$

for the canonical mapping f with blackbox simulatability.

Chapter 8

CONCLUSION

8.1 Applications

Although our library has to satisfy the conditions stated in Section 7, this extended UC library is still useful for a class of applications that need homomorphic encryptions. Generally speaking, in such applications some parties give their secret information to other parties for a computation but do not want to reveal it. Basically the computation will be done on encrypted data, but with homomorphic encryptions, we can guarantee that when decrypting the computed result, we will have the desired value without the original input. Furthermore, threshold version of homomorphic encryption reduces the probability that authorities cheat, because they have to get enough parties to do so. Some examples are electronic voting (e-voting), electronic auctions (e-auction), lottery and so on.

In e-voting, voters submit encryptions of their original votes. The authorities combine the ciphertext to have the ciphertext of the final result. By decrypting this combined ciphertext, we get the result of the election without decrypting every encryption of individual vote.

In e-auction, homomorphic encryptions help to hide the prices from bidders. Fortunately, we can still know who proposes the highest price using some mathematical techniques on the homomorphic property.

In lottery, what we need is a random number, which indicates the winner, chosen from a set of players. However, we do not want each player to see others' choices, because then they can cheat. Therefore each player encrypts his random number and submit to authorities. Authorities will combine all encrypted choices to get a final number without revealing every player's choice. Of course this final number must be random, if there is at least one player honest.

With the library, the design work for such applications could be easier. For example if one wants to design a protocol that needs homomorphic encryption, he can just use the ideal library. Because the ideal one is "trusted", it is easier to analyze the security of the new protocol. For implementation, he will replace the ideal one with the real one without losing security due to the composition theorem.

Even though there are some conditions when using our library, it is still applicable because the conditions are all reasonable and achievable. We even can improve it for a broader area of applications. In the next section we mention what we may improve in the future.

8.2 Future work

In this work, we add a simple version of threshold encryption. A more complex but more useful one is verifiable secret sharing (VSS) threshold encryption scheme. With a VSS version, one can always check if a decryption share is correct or not, therefore we do not need decryption shares from all authorities for detecting corrupted decryption shares. Instead, we use verification keys for that job. One can continue extend this library by analyzing the case of (VSS) threshold encryption scheme and we believe it will be more useful.

As we can see that the proof for the security of the library is quite complex and long with detailed notations. We hope that in the future we can find a better way to prove security for such type of problems.

8.3 Conclusion

In this paper we have given an extended version of a UC library proposed by Backes et al. [BPW03a]. This version contains a homomorphic encryption scheme used together with nizk proof and key distribution functionalities.

We have described the ideal library, which is considered the specification for our real library, and also the real one. The real library works on abstract terms while the real one works on bit-strings data. However, both offer to users commands that get handles as input and also output handles (Except the commands to store and retrieve data). Therefore, users get the same interface from these two libraries.

After that, we showed a simulator that acts as the ideal library, using the real adversary as a black box. Then we proved that the joint view of honest users and the adversary is indistinguishable between using the ideal library and the real one under some conditions. Therefore we concluded that the ideal library is a sound abstraction of our real one, i.e the real library is at least as secure as the ideal one.

Finally we gave some examples of applications on this library and proposed some possible future work.

Bibliography

[AF04] M. Abe and S. Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. *Advances in Cryptology–Crypto*, 3152:317–334, 2004.

[AR00] M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *Theoretical Computer Science: Exploring New Frontiers of Theoretical Information: Intermational Conference Ifip Tcs 2000 Sendai, Japan, August 17-19, 2000 Proceedings*, 2000.

[BDHK08] M. Backes, M. Dürmuth, D. Hofheinz, and R. Küsters. Conditional reactive simulatability. *International Journal of Information Security*, 7(2):155–169, 2008.

[BFM88] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112, 1988.

[Bla06] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.

[BP04] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 204–218, 2004.

[BP05] M. Backes and B. Pfitzmann. Limits of the Cryptographic Realization of Dolev-Yao-Style XOR. *Computer Security-Esorics 2005: 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, 2005.

[BPW03a] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[BPW03b] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric Authentication within a Simulatable Cryptographic Library. *Computer Security-ESORICS 2003: 8th European Symposium on Research in Computer Security, Gjøvik, Norway, October 13-15, 2003: Proceedings*, 2003.

[BPW04] M. Backes, B. Pfitzmann, and M. Waidner. A General Composition Theorem for Secure Reactive Systems. *Theory of Cryptography: First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004: Proceedings*, 2004.

[BPW06a] M. Backes, B. Pfitzmann, and M. Waidner. Limits of the reactive simulatability/UC of Dolev-Yao models with hashes. *Proc. of the 11th European Symposium on Research in Computer Security. Springer-Verlag*, 2006.

[BPW06b] M. Backes, B. Pfitzmann, and M. Waidner. Soundness Limits of Dolev-Yao Models. *Proceedings of the Workshop on Formal and Computational Cryptography (FCC 2006)*, 2006.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. *42nd IEEE Symposium on Foundations of Computer Science, 2001. Proceedings.*, pages 136–145, 2001.

[CS02] H. Comon and V. Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not. *Journal of Telecommunications and Information Technology*, 4:5–15, 2002.

[DDM+05] A. Datta, A. Derek, J.C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, 3580:16–29, 2005.

[DDMR07] A. Datta, A. Derek, J.C. Mitchell, and A. Roy. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[FP00] P.A. Fouque and D. Pointcheval. Threshold Cryptosystems Secure against Chosen-Ciphertext Attacks. *Proc. of Asiacrypt*, pages 573–84, 2000.

[FPS01] P.A. Fouque, G. Poupard, and J. Stern. Sharing Decryption in the Context of Voting or Lotteries. *Financial Cryptography: 4th International Conference, FC 2000, Anguilla, British West Indies, February 20-24, 2000: Proceedings*, 2001.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *Proceedings of the seventeenth annual ACM Symposium on Theory of Computing*, pages 291–304, 1985.

[GMW86] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. *27th Annual Symposium on Foundations of Computer Science*, pages 174–187, 1986.

[GOS06] J. Groth, R. Ostrovsky, and A. Sahai. Perfect non-interactive zero knowledge for NP. *Proceedings of EUROCRYPT-06, LNCS series*, 4004:339–358, 2006.

[Gro04] J. Groth. Evaluating security of voting schemes in the universal composability framework. *Proceedings of ACNS04, LNCS series*, 3089:46–60, 2004.

[IK06] R. Impagliazzo and B.M. Kapron. Logics for reasoning about cryptographic constructions. *Journal of Computer and System Sciences*, 72(2):286–320, 2006.

[Mea03] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, 2003.

[PW01] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. *IEEE Symposium on Security and Privacy*, pages 184–200, 2001.

[Wik04] Douglas Wikström. Universally composable dkg with linear number of exponentiations. Cryptology ePrint Archive, Report 2004/124, 2004.

[Zun06] R. Zunino. *Models for Cryptographic Protocol Analysis*. PhD thesis, Ph. D. thesis, Universita di Pisa, Italy, 2006.