A Formalisation of Cryptography with the Coq Proof Assistant

Ole Dalgaard Lauridsen, 20072961

Master's Thesis, Computer Science

September 2015

Advisor: Olivier Danvy

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In this dissertation, we model and prove properties about several cryptographic protocols using the Coq proof assistant. By doing so, we show that it is possible to use interactive proof assistants in the field of cryptography, thereby achieving certainty of the correctness of cryptographic proofs. Also, we demonstrate that this approach is not only possible in theory, but also feasible in practice: we undertake this challenge with only an elementary amount of experience with interactive proof assistants.

This dissertation documents the following contributions.

We first consider modelling and proving theorems about the one-time pad cryptosystem: we prove that encrypting and then decrypting any plaintext under the same key gives the plaintext, that the system is "one-time" in the sense that non-trivial information is revealed if the same key is used for encryption twice, and that the encryption function is onto by which we mean that any plaintext can be encrypted into any ciphertext of the same length by choosing the key appropriately.

We then consider modelling and proving properties about modes of operation and MAC schemes built on top of them. For the modes we consider, we show that encrypting and then decrypting a plaintext under the same key yields the plaintext and that changing the initialisation vector when encrypting results in all ciphertext blocks changing. For each mode, we show that it either propagates changes of plaintext blocks or not, i.e., we show that, for any mode, either all ciphertext blocks from that point are changed or only the corresponding ciphertext block is changed if a plaintext block is changed. Additionally, we prove that any MAC value output by one of the MAC functions we consider is changed if a block of the corresponding input message is changed. We prove these results about the modes of operation and MAC schemes by defining general specifications and proving general theorems about them. In this way, we are able to extend the results to several modes and MAC schemes with only little extra effort. We exploit this fact to a high degree: we model and prove the abovementioned properties about the already known CBC, CFB, and CTR modes and the CBC- and CFB-MAC schemes, and we do the same for three new modes and a MAC scheme defined by us: the IFB, f, and BENC modes and the IFB-MAC scheme.

Finally, we consider the RSA encryption and signature schemes and the ElGamal encryption scheme in $\mathbb{Z}_p^*$ where p is prime. Using the libraries of the SSReflect Coq extension, we model the schemes and prove several properties about them: for both of the encryption schemes, we show that ciphertexts produced by them are members of the designated ciphertext spaces, that first encrypting and then decrypting any plaintext yields the plaintext if corresponding public and private keys are used, and that the encryption functions are invertible regarding the plaintexts. In addition, we prove that any signature produced by the RSA signature scheme is in the designated signature space and that the verification function of the scheme only accepts the input message and signature if the signature is the signature of the message.

Resumé

I denne afhandling modellerer vi flere kryptografiske protokoller og beviser egenskaber af disse protokoller vha. bevisassistenten Coq. Derved viser vi, at det er muligt at bruge interaktive bevisassistenter i kryptologi, hvorved vi opnår sikkerhed for korrekthed af kryptografiske beviser. Desuden demonstrerer vi, at denne fremgangsmåde ikke blot er mulig i teorien, men også praktisk håndterbar: vi løser denne udfordring med en udelukkende elementær erfaring med interaktive bevisassistenter.

Denne afhandling dokumenterer de følgende bidrag.

Først ser vi på "one-time pad"-kryptosystemet: vi modellerer dette system og beviser sætninger om det. Vi beviser, at hvis man først krypterer og derefter dekrypterer en vilkårlig plaintext under den samme nøgle, så fås plaintexten. Vi beviser, at systemet er "one-time" i den forstand, at ikke-triviel information afsløres, hvis den samme nøgle bruges to gange til kryptering. Desuden beviser vi, at krypteringsfunktionen er surjektiv, hvormed vi mener, at en vilkårlig plaintext kan krypteres til en vilkårlig ciffertekst af samme længde ved at vælge krypteringsnøglen passende.

Derefter ser vi på modes of operation og MAC-systemer bygget på disse modes. For alle de modes, vi ser på, viser vi, at hvis man først krypterer og derefter dekrypterer en plaintext under den samme nøgle, så fås plaintexten. Vi viser, at hvis man ændrer initialiseringsvektoren, så ændres all ciffertekst-blokke. Derudover viser vi, at en mode enten propagerer eller ikke propagerer ændringer i plaintext-blokke, dvs., vi viser, at enten ændres alle ciffertekst-blokke efter en ændret plaintext-blok eller også ændres kun den tilsvarende ciffertekst-blok. Slutteligt viser vi, at MAC-værdier outputtet fra de MAC-funktioner, vi tager i betragtning, ændres, når blokke i inputbeskederne ændres. Vi viser disse resultater ved at definere generelle specifikationer og bevise sætninger om disse specifikationer. På denne måde er vi i stand til at udvide resultaterne til adskillige modes og MAC-systemer uden større besvær. Vi udnytter i høj grad dette faktum: vi modellerer og viser de ovennævnte egenskaber for de allerede kendte CBC, CFB og CTR modes og CBC- og CFB-MAC-systemer, og vi gør det samme for tre nye modes og et MAC-system defineret af os: de tre modes, IFB, f og BENC, og IFB-MAC-systemet.

Endeligt ser vi på RSA-kryptosystemet, RSA-signatursystemet og ElGamal-kryptosystemet i $\mathbb{Z}_p^*$, hvor p er et primtal. Ved hjælp af bibliotekerne i Coq-udvidelsen SSReflect modellerer vi disse systemer og viser flere af deres egenskaber: for begge kryptosystemer viser vi for det første, at ciffertekster produceret af disse systemer er medlemmer af de designerede ciffertekst-rum, for det andet at hvis man krypterer og derefter dekrypterer en vilkårlig plaintext fås plaintexten, hvis korresponderende offentlige og private nøgler bruges, og for det tredje at de tilhørende krypteringsfunktioner er invertible mht. plaintexts. Yderligere viser vi, at signaturer produceret af RSA-signatursystemet er i det designerede signatur-rum, og at verifikationsfunktionen for systemet kun accepterer dens input-besked og -signatur, hvis signaturen er signaturen for beskeden.

Acknowledgements

My acknowledgements primarily go to my advisor Olivier Danvy: had it not been for his enthusiasm and guidance, this dissertation would not have come to be. To start with, he motivated me to expand upon my term projects in his courses about functional programming and proving using the Coq proof assistant. Throughout the process of writing my dissertation and preparing my defence, his advice and motivation have been invaluable. Also, his willingness to take the time needed to listen to and guide me and to respond to questions almost round-the-clock—even while interacting with numerous other students—has been admirable.

I also want to acknowledge the people behind the Coq proof assistant and the SSReflect extension as well as the contributors to the SSReflect libraries: the Coq proof assistant has been central throughout my work, and the SSReflect extension and libraries have been instrumental in my work on the RSA and ElGamal schemes in the dissertation and the RSA blind signature scheme in my MSc thesis defence.

I want to thank my fellow students Claus Jespersen and Morten Larsson. During the last years, they have been great colleagues and workmates in several courses including the functional-programming courses leading to this dissertation. It is such great colleagues who make one's studies a delight.

In addition, I want to acknowledge the administrative staff and system administrators in the Department of Computer Science at Aarhus University for putting an office at my disposal for the last $1/2$ year of my MSc studies as well as for ensuring the availability of network and printing facilities ever since I enrolled as a first-year student.

Finally, I want to thank my family for their love and support.

Ole Dalgaard Lauridsen,
Aarhus, Monday 28th September, 2015.

Contents

Part I

Introduction

Chapter 1

Introduction

> The beginning of wisdom
> is the statement 'I do not know.'
>
> Thrall

In this chapter, we introduce the goals and contributions of this work and give an overview of the structure of this dissertation.

In Section 1.1, we list the goals and contributions of this work. In Section 1.2, we outline the structure of this dissertation. Finally, in Sections 1.3 and 1.4, we specify the prerequisites assumed of the reader and explain how to retrieve the code in this dissertation along with the structure of the code files.

1.1 Goals and Contributions

The goal of this work is to model several cryptographic protocols using the Coq proof assistant and use these models to prove theorems about the protocols. One of our main motivators for following this procedure is to reap the benefits of using interactive proof assistants: the proof assistant checks the proofs which helps identify errors in them. Hence, we can be certain (subject to the implementation of Coq and the hardware used to run it) that the proofs are correct, e.g., we know that any possible special case has been considered.

Often, at least in the field of cryptography, many details of proofs are left out—which is understandable: the proofs are much more succinct by doing so. In this dissertation, we show that it is possible to avoid this situation for the protocols and theorems we consider with only an elementary amount of experience with interactive proof assistants.

Below, we list the specific goals and contributions of this work.

The One-Time Pad Cryptosystem Our goal in Chapter 5 is to model and prove theorems about the one-time pad cryptosystem. We specify the encryption and decryption functions of the system after which we prove that the system is valid, i.e., encrypting and then decrypting any plaintext under the same key—the cryptosystem is a symmetric-key system[1]—yields the plaintext, that the system is "one-time" in the sense that the ciphertexts resulting from encrypting two plaintexts under the same key reveal non-trivial information about the plaintexts, and that the encryption function is onto for any plaintext which, in this case, means that, by choosing the key appropriately, any plaintext can be encrypted into any ciphertext of the same length.

Our work in this chapter is our first contribution.

[1] A symmetric-key cryptosystem is a cryptosystem where the same key is used for both encryption and decryption.

Modes of Operation The goal in Chapter 6 is to model and prove theorems about modes of operation and MAC schemes built on top of them. We specify general functions: general encryption, decryption, and MAC functions. These functions are generalisations of the (non-general) functions of the modes of operation and MAC schemes. Then, after proving theorems about these general functions, we are able to prove theorems about the non-general functions as corollaries of the former theorems. A major benefit of this procedure of first specifying general functions and proving general theorems is that we can easily extend our result to other and, potentially, new modes of operation. We exploit this property: we model and prove properties about three already known modes of operation while we do the same for three new modes defined by us: the IFB, f, and BENC modes.

First, we prove that all the modes we consider are valid, i.e., encrypting and then decrypting any plaintext under the same key—modes of operation are symmetric-key cryptosystems—gives the plaintext. Second, we show that the encryption function of each mode either propagates plaintext changes or not (not both), i.e., if a plaintext block is changed, then either all subsequent ciphertext blocks are changed or only the corresponding ciphertext block is changed. Third, we prove that all the modes propagate changes of the initialisation vector which is used to influence the resulting ciphertext, i.e., changing the IV changes all ciphertext blocks without changing the plaintext that is encrypted. Finally, we prove that the MAC functions we consider propagate changes in the messages input to them, i.e., if a block of the input message is changed, then the corresponding output MAC value is changed, too.

The work done in this chapter is our second contribution.

The RSA Encryption and Signature Schemes The goal of Chapter 7 is to model the RSA encryption and signature schemes and prove properties about them. We specify valid RSA key pairs—the schemes are public-key systems[2]—and the functions of the schemes. Having done so, we are able to prove several theorems about the schemes.

For the RSA encryption scheme, we prove that the ciphertexts produced by it are members of the designated ciphertext space, that the scheme is valid, i.e., encrypting and then decryption any plaintext under the public and private keys of a valid key pair results in the plaintext, and that the encryption function is invertible regarding the input plaintexts.

For the RSA signature scheme, we prove that the signatures produced by it are in the designated signature space and that the scheme is valid, i.e., on input a message and a signature, the verification function of the scheme accepts if and only if the input signature is the signature of the input message.

In this chapter, we use the SSReflect libraries described in Chapter 4. In this way, we can use the lemmas in these libraries in our proofs enabling us to focus on the cryptographic concepts and abstract away from proving the underlying mathematical properties, e.g., properties concerning primes.

Our work in this chapter is our third contribution.

The ElGamal Cryptosystem The goal of Chapter 8 is to model and prove properties about the ElGamal cryptosystem in $\mathbb{Z}_p^*$ where p is prime. As the RSA schemes, this system is a public-key system so we first specify valid key pairs of it after which we specify the encryption and decryption functions. With these specifications defined, we can prove theorems about the scheme.

We prove that ciphertexts computed by the encryption function are members of the designated ciphertext space, that the scheme is valid, i.e., first encrypting and then decrypting any plaintext under the public and private keys of a valid key pair yields the plaintext, and that the encryption function is invertible regarding the input plaintexts.

As for the RSA schemes, we use the SSReflect libraries when proving theorems about the ElGamal cryptosystem. Again, doing so enables us to focus on the cryptographic subject instead of the underlying

[2] A public-key system is a system where key pairs each consisting of a public and a private key are used. In the encryption scheme, the public key is used to encrypt plaintexts while the private key is used to decrypt ciphertexts. In the signature scheme, the private key is used to sign messages while the public key is used to verify signatures against messages.

mathematical properties.

The work in this chapter is our fourth and last contribution.

1.2 Structure of Dissertation

In this section, we outline the structure of this dissertation.

The dissertation is divided into four parts: the "Introduction", "Background", "Contributions", and "Conclusion" parts.

The first part, the "Introduction" part, consists of this chapter.

In the following part, the "Background" part, we cover the background. In Chapter 2, we look at the Coq proof assistant and describe parts of it that we use. In Chapter 3, we describe some of types that we use in the dissertation. Not all types are covered here: the types left out are explained when used. In Chapter 4, we describe the SSReflect extension and those of its libraries and lemmas we use.

The third part contains the contributions of this work. In Chapter 5, we model and prove theorems about the one-time pad cryptosystem. In Chapter 6, we consider modes of operation and MAC schemes built on top of them: we prove properties about several of such including three new modes and a new MAC scheme. In Chapter 7, we model and prove properties of the RSA encryption and signature schemes while, in Chapter 8, we do the same for the ElGamal cryptosystem in $\mathbb{Z}_p^*$ where p is prime. In these two latter chapters, we use the SSReflect lemmas in Chapter 4.

In the final part of this dissertation, the "Conclusion" part, we summarise and conclude on the contributions of this work.

1.3 Prerequisites

We assume throughout this dissertation that the reader is familiar with the Coq proof assistant and, to some degree, the SSReflect libraries. In addition, it is assumed that the reader has a basic knowledge about the cryptographic protocols we consider and the associated terminology.

1.4 Technical Details

All code in the dissertation is available for download at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_code.tar.gz`. The archive file at this link contains a directory named `Ole-Dalgaard-Lauridsen_code` which includes several files and directories:

- The `OTP` directory contains the "`one-time pad cryptosystem.v`" file with the specifications and proofs of Chapter 5 where we consider the one-time pad cryptosystem.
- The `MoO` directory contains several files. These files contain the code in Chapter 6 where we prove theorems about the modes of operation and corresponding MAC schemes.
- The `RSA` directory contains the `RSA.v` file with the code of Chapter 7 where the RSA encryption and signature schemes are handled.
- The `ElGamal` directory contains the "`ElGamal in Z_p*.v`" file with the code of Chapter 8 where the ElGamal cryptosystem in $\mathbb{Z}_p^*$ is considered.
- Finally, the `load_all.v` file loads all the other files and can be used to verify the entirety of the code.

To verify the code, the reader must install the Coq proof assistant and SSReflect extension—we assume that the reader uses a computer running the Linux operating system.

- Coq can be downloaded at `https://coq.inria.fr`. The code has been verified to run on Coq version 8.4pl5 at `https://coq.inria.fr/distrib/8.4pl5/files/coq-8.4pl5.tar.gz`. The `INSTALL` file in the `coq-8.4pl5` folder of this archive file describes how to install Coq.
- The SSReflect extension can be downloaded at `http://ssr.msr-inria.inria.fr`. The code has been verified with SSReflect version 1.4 for Coq 8.4 at `http://gforge.inria.fr/frs/download.php/file/31453/ssreflect-1.4-coq8.4.tar.gz`. The `INSTALL` file in the `ssreflect-1.4` folder of this archive file explains how to install the SSReflect extension.

This dissertation is available as a `.pdf` file at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_dissertation.pdf`. Should the dissertation or the code for some reason be unavailable at the URLs—and even if not—the author can be contacted at olelauridsen1337@gmail.com.

Part II

Background

Chapter 2

The Coq Proof Assistant

> Bureaucrat Conrad,
> you are technically correct—
> the best kind of correct.
>
> Number 1.0, head bureaucrat

In this chapter, we look at the Coq proof assistant [1] and describe parts of it that we use in this dissertation.

In Section 2.1, a short overview of Coq is given. After that, in Section 2.2, we have a short section about the tactics used when interactively proving theorems. Finally, we discuss the induction and coinduction capabilities of Coq in Sections 2.3 and 2.4.

2.1 Overview of Coq

The Coq proof assistant implements a specification language called Gallina. This language facilitates defining functions and predicates and stating mathematical theorems and specifications.

One can prove these theorems by applying different tactics to the immediate goals or subgoals appearing when proving them: these tactics are defined using a language called Ltac. The entirety of the proof of any theorem is a program that Coq then translates to a lambda term which is then checked up against the type corresponding to the theorem. The feasibility of this method is provided by the Curry-Howard isomorphism with its proofs-as-programs and propositions-as-types.

The proofs in Coq are checked by a small certification kernel that ensures that errors are unlikely when proving with Coq—provided that the compiler and hardware used work properly. This is a huge advantage which is often missing when proving mathematical statements in cryptography, but also in general. Here, many steps of the proofs are done implicitly for brevity, but this also carries with it ample opportunity for errors. The avoidance of this problem is one of the main reason for using Coq (or other proof assistants) and is a main motivator of this work.

2.2 Tactics

The Ltac language is part of Coq. This language facilitates the definition of tactics that we apply when proving theorems.

When we prove theorems, we always have a goal (or subgoal) that we work towards. At the same time, we (almost always) have premises given by the theorems. These premises include statements about the types of the variables given and hypotheses about these. We apply tactics to either the goals or the

hypotheses, thereby transforming them (or creating new hypotheses). In this way, we can make the goals and hypotheses converge until we at last can apply one of the hypotheses or another theorem directly.

The tactics often take different parameters as input. These can, e.g., be variables or hypotheses in our proof or other theorems or lemmas. An example of this is the `apply` tactic that takes a hypothesis or another theorem or lemma as input and applies it to the goal (or some hypothesis if specified).

2.3 Induction

We can define inductive types in Coq. An example of an already defined inductive type is the `nat` type representing Peano natural numbers.

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

A clear advantage of using inductive types is that we can use inductive reasoning when showing some proposition to hold for all members of a type: first, we show that the proposition holds for the base case, and, second, we show that the proposition holds for the inductive case assuming it holds for the values provided to the constructor. As an example, to show some proposition `P` to hold for all natural numbers, we show that `P O` and `(forall n : nat, P n → P (S n))`. This is captured by Coq in that when defining the above `nat` type, the lemma `nat_rect` is defined, too.

```
nat_rect
     : forall P : nat → Type,
       P O → (forall n : nat, P n → P (S n)) → forall n : nat, P n
```

For easy facilitation of the induction principle, a tactic `induction` is defined in Coq. When used with a variable of an inductive type, the tactic automatically splits up the proof into the two or more cases of the type: the base case and one or more inductive cases. For the inductive cases, the appropriate induction hypothesis is given enabling us to take the inductive step. Showing the subgoals for these cases then amounts to showing the proposition to hold for all members of the inductive type.

Using an inductive type, we can easily define recursive functions over the members of the type. An example is the `fact` function (defined in the `Coq.Arith.Factorial` standard library) calculating the factorial of a natural number.

```
Fixpoint fact (n : nat) : nat :=
  match n with
    | O ⇒ 1
    | S n ⇒ S n * fact n
  end.
```

The `fact` function is defined inductively over the input natural number `n`: if the `n` is equal to the base case `0`, the function outputs `1`, and if the `n` is in the inductive case, we multiply it with the output of the function applied recursively to the natural number whose successor is `n`.

2.4 Coinduction

Coinduction is similar to induction. For inductive types, we start from a base case and then inductively define other members of the type. In this way, all members for the type are finitely structured. For coinductive types, however, we allow infinite data structures.

```
CoInductive Stream (T : Type) : Type :=
| Cons : T → Stream T → Stream T.
```

We use the coinductive `Stream` type whose members can be thought of as infinite lists. Here, a stream consists of an element of some type `T` and another stream consed together: we do not start from any base case, but just build onto another (yet unconstructed) infinite stream.

Unlike for regular inductive types, we cannot make use of the `induction` tactic when we want to show some proposition for each of all the members of a coinductive type. Instead we use the tactic `cofix`. In this dissertation, however, most of the proving of theorems concerning coinductive types is done using the regular induction principle. We discuss this further in Section 3.4 where we examine streams and the pertaining concept of bisimulation.

We can define corecursive functions over the members of a coinductive type. As an example, we can define a function `map_Stream` that applies a function `f` to all elements in the stream.

```
CoFixpoint map_Stream T1 T2 (xss : Stream T1) (f : T1 → T2) : Stream T2 :=
  match xss with
    | Cons xs xss' ⇒
      Cons (f xs) (map T1 T2 xss' f)
  end.
```

The `f` function is applied to the head element of the input stream. The output of this application is then consed to the result of applying the `map_Stream` function corecursively to the tail of the input stream. In this way, `f` is applied elementwise.

Chapter 3

Types

In this chapter, we describe some of the types that we use throughout this dissertation. Other types are used to some degree; those are explained when used. We use the types in this chapter to model the data and functions that we prove theorems about. As an example, when we in reality encrypt bit strings in Chapter 5, we model the bit strings as lists of Boolean values which enables us to use any functions over Boolean values when we wish to show that the one-time pad cryptosystem works.

We start by describing the `nat` type representing natural numbers in Section 3.1. Then we look at the `bool` type in Section 3.2 where we list pertaining operators and lemmas used. Likewise, we describe the `list` and `Stream` types in Sections 3.3 and 3.4 where we also describe the various list and stream operators and several of the lemmas we use.

3.1 `nat`

In Coq, natural numbers are represented by the inductive type `nat`:

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

Here, `O` represents the base case of zero while `S` is the successor of a natural number. Thus, `(S O)` means 1, `(S (S O))` means 2, etc. This follows the Peano axioms where 0 is a natural number and any successor of a natural number is a natural number.

Coq allows us to use a shorthand notation of these Peano natural numbers using Arabic numbers (0, 1, ...).

A predecessor function for natural numbers is defined in the standard library.

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ n
    | S u ⇒ u
  end.
```

If the provided natural number actually is a successor of another, the output of the function is apparent. If, however, it is the base case of `O`, and thus has no predecessor, the output is defined to be `O` or the number itself.

A theorem from the library concerning the predecessor is used in the dissertation.

```
Theorem pred_Sn : forall n:nat, n = pred (S n).
```

Any natural number is equal to the predecessor of its successor.

3.2 bool

Boolean values are represented by the bool type:

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

As is normally the case, Boolean values can either be true represented by the true constructor or false represented by the false constructor.

We use this type to represent bits. In this way, we can do any bit operation by using the corresponding Boolean operation, and the lemmas proved in the library and in the dissertation can be used directly to prove theorems involving bits.

3.2.1 Operators

We use two of the operators defined for the bool type: first, the exclusive-or operator

```
Definition xorb (b1 b2 : bool) : bool :=
  match b1, b2 with
    | true, true ⇒ false
    | true, false ⇒ true
    | false, true ⇒ true
    | false, false ⇒ false
  end.
```

and, secondly, the negation operator.

```
Definition negb (b : bool) := if b then false else true.
```

The xorb function is used when we want to XOR the two bits represented by the bools given as inputs while the negb function is applied when we want to flip the bit represented by the input bool.

3.2.2 Lemmas

A number of lemmas concerning the above operators are defined and proved in the standard library. Here, we list the ones we use. The first lemma says that negation is involutive, i.e., it is its own inverse.

```
Lemma negb_involutive : forall b : bool, negb (negb b) = b.
```

A consequence of this is then the following lemma:

```
Lemma negb_sym : forall b b' : bool, b' = negb b → b = negb b'.
```

Likewise, we use some lemmas concerning the xorb operator from above. Often, when doing case analysis of one of the Boolean values in a proof involving this operator, we need to resolve the result of applying the operator. Here, we use four lemmas that describe the result when one of the operands is a constant bool:

```
Lemma xorb_false_l : forall b:bool, xorb false b = b.
Lemma xorb_false_r : forall b : bool, xorb b false = b.
```

false is neutral for xorb while true is the opposite:

```
Lemma xorb_true_l : forall b:bool, xorb true b = negb b.
Lemma xorb_true_r : forall b : bool, xorb b true = negb b.
```

Furthermore, we use the fact that the operator is commutative, i.e., the order of the operands is without significance:

```
Lemma xorb_comm : forall b b' : bool, xorb b b' = xorb b' b.
```

For use in Chapters 5 and 6 where XOR'ing is used extensively, we prove a few other lemmas about the `xorb` operator.

```
Lemma about_xorb_l : forall b b' : bool, xorb b (xorb b b') = b'.

Lemma about_xorb_r : forall b b' : bool, xorb (xorb b b') b' = b.
```

When XOR'ing twice with the same Boolean value on the left or the right, we end up with the original Boolean value. The first lemma is proven by case analysis of the `bool b` and using some of the library lemmas above. The other lemma is a direct consequence of the first lemma and that `xorb` is a commutative operator.

Additionally, we prove that the `xorb` operator is invertible regarding the first input.

```
Lemma xorb_invertible_wrt_first_input :
  forall x_1 x_2 y : bool,
    xorb x_1 y = xorb x_2 y →
    x_1 = x_2.
```

This lemma is proven by case analysis of the second input `y`.

3.3 `list`

Coq uses the inductive type `list` to represent lists. Here, a list is either an empty list (`nil`) or a list consisting of an element, which we call the head element, and another list, which we call the tail, consed together:

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

`A` represents the type of the elements in the list.

```
Infix "::" := cons (at level 60, right associativity) : list_scope.
```

We use the infix notation `::` that represents the constructor `cons`, e.g., `cons x nil` can be abbreviated `x :: nil` and represents the list with one solitary element, `x`.

3.3.1 Operators

Several operators are defined for lists. In this section, we look at the ones we use.

Associated with lists are a number of operators. One of these is the `length` operator that recursively computes the length of a list.

```
Definition length (A : Type) : list A → nat :=
  fix length l :=
  match l with
   | nil ⇒ 0
   | _ :: l' ⇒ S (length l')
  end.
```

The operator applied to the base case of an empty list (`nil`) gives `0` (the natural number 0) while the operator applied to the inductive case with a head element and a tail list gives the successor of the length of the tail. In this way, the structures of `nat` and `list` align with each other.

Another function is the `nth_error` operator that outputs the element of a list `l` at an index `n`.

```
Fixpoint nth_error (l : list A) (n : nat) {struct n} : Exc A :=
    match n, l with
      | O, x :: _ ⇒ value x
```

```
    | S n, _ :: l ⇒ nth_error l n
    | _, _ ⇒ error
  end.
```

This operator gives outputs of the type `Exc A` which equals the type `option A`.

```
Definition Exc := option.
Definition value := Some.
Definition error := ·None.
```

`option`, in return, is a sum type that enables functions and operators to only optionally output a value of some specified type.

```
Inductive option (A:Type) : Type :=
  | Some : A → option A
  | None : option A.
```

This is useful for the `nth_error` operator: if the index is out-of-bounds, it outputs `error` equal to `None`. If the index is in-bounds, the result is `value a` equal to `Some a` if the element at the index equals `a`.

A similar operator, `nth`, is defined in the library. Here, however, we must provide a default value to output when the index is out-of-bounds since this operator does not use the `option` type. This property is undesirable for our purposes since often a clear choice of a default value is non-existent.

We use the standard library operator `last`.

```
Fixpoint last (l : list A) (d : A) : A :=
  match l with
    | [] ⇒ d
    | [a] ⇒ a
    | a :: l ⇒ last l d
  end.
```

Here, `[]` and `[a]` are shorthand notations for `nil` and `cons a nil`. If the input list is empty, the provided default value `d` is output. If the input list only has one element, this element is output. If the input list is in the inductive case and has more than one element (so that it is not matched against the second pattern), the output of the operator applied to the tail of the list and the same default value is passed on.

When specifying the general MAC function in Section 6.8.2, we use a list operator, `last_block`, that on input a list of elements each of type `block_n` (defined in Section 6.5.1) outputs the last element if applicable: in the MAC schemes we consider, the MAC value of a message of blocks of bits (each represented by an element of type `block_n`) is defined to be the last block in the corresponding ciphertext.

```
Definition last_block (xss : list block_n) : block_n :=
  List.last xss zero_block_n.
```

We define this operator in terms of the `last` operator above. The `zero_block_n` constant is defined in Section 3.4.4 as an element (of type `block_n`) whose entries are `false`s. This element represents an all-zero block of length `n`.

3.3.2 Lemmas

We prove two lemmas concerning the `length` operator. The first lemma states that any list of length `0` is equal to the empty list.

```
Lemma about_length_0_nil :
  forall (T : Type) (xs : list T),
    length xs = 0 →
    xs = nil.
```

This lemma is proved by case analysis of the input list `xs`. When `xs` is `nil`, the conclusion is apparent: when it consists of an element and another list consed together, its length is the successor of another

natural number. Then the premise `length xs = 0` cannot possibly be true, and we can use the `discriminate` tactic.

The other lemma says that if two non-empty lists have the same length, then their respective tails have equal lengths.

```
Lemma about_length_tail :
  forall (T : Type) (x y : T) (xs' ys' : list T),
    length (x :: xs') = length (y :: ys') →
    length xs' = length ys'.
```

In the proof of this lemma, we use the `pred_Sn` lemma in Section 3.1. This lemma states that any natural number is the predecessor of its successor.

We also prove a few lemmas about the `nth_error` operator. Apart from three unfolding lemmas that handle the base case and the inductive case of the input index and the base case of the input list, we prove a lemma that enables us to translate a problem concerning equality of `values` to a similar problem concerning the equality of the actual values inside the `value` constructor.

```
Lemma values_with_equal_values_are_equal_and_vice_versa :
  forall T (a b : T),
    value a = value b ↔ a = b.
```

This lemma makes it possible to apply other lemmas about the values themselves directly. It is proven using the `inversion` tactic.

Also, we prove that if the `nth_error` operator is applied to an out-of-bounds index, then it outputs `error`.

```
Lemma nth_error_yields_error_if_index_out_of_bounds :
  forall T (xss : list T) (i : nat),
    ~ i < length xss →
    nth_error xss i = error.
```

This lemma is proven by induction in the input index and case analysis of the input list. It is used in the proofs of the `lists_of_equal_lengths_different_at_index_only_implies_index_inbounds` lemma in Section 6.7.2 and the general plaintext change non-propagation theorem using lists in Section 6.8.3.3.

3.4 `Stream`

In Chapter 6, we use streams of elements of the `block_n` type defined in Section 6.5.1 to represent streams of blocks (of length `n`) of bits when proving theorems about modes of operation using block ciphers.

Streams are defined using coinduction that enables us to work with infinite data structures. The definition of the `Stream` type in Coq is

```
CoInductive Stream : Type :=
  Cons : A → Stream → Stream.
```

In short, a stream consists of an element, which we call the head element, of some type `A` and another stream, which we call the tail, consed together.

3.4.1 Bisimilarity

Because streams are infinite, the regular notion of equality does not work. Hence, when proving theorems involving streams, we make use of a concept called bisimulation or bisimilarity. Bisimilarity is defined below as a relation with members being pairs of streams. In our case, the streams contains elements of the `block_n` type.

```
CoInductive bisimilar_Stream_block_n :
  Stream block_n → Stream block_n → Prop :=
  | Bisimilar :
      forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n),
        xs_1 = xs_2 →
        bisimilar_Stream_block_n xss'_1 xss'_2 →
        bisimilar_Stream_block_n (Cons xs_1 xss'_1) (Cons xs_2 xss'_2).
```

As expressed in the definition, two streams are bisimilar if and only if the head element of the first stream is equal to that of the second stream and the tail of the first stream is bisimilar to the tail of the second.

Bisimilarity is our notion of equality of streams and is, e.g., used for showing equality of plaintext streams and plaintext streams that have been first encrypted and then decrypted.

In this dissertation, we exploit several properties of bisimilarity. The first two of these properties follow directly from the definition: if two streams are bisimilar, they have equal head elements and bisimilar tails. This is divided into two lemmas which are used extensively in the proofs involving streams.

```
Lemma bisimilarity_implies_equal_head_blocks :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n),
    bisimilar_Stream_block_n (Cons xs_1 xss'_1) (Cons xs_2 xss'_2) →
    xs_1 = xs_2.

Lemma bisimilarity_implies_bisimilar_tails :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n),
    bisimilar_Stream_block_n (Cons xs_1 xss'_1) (Cons xs_2 xss'_2) →
    bisimilar_Stream_block_n xss'_1 xss'_2.
```

These lemmas are proven by simply inverting the definition of bisimilarity of streams and applying the relevant resulting hypotheses.

The next three properties that the bisimilarity relation is both reflexive, symmetric, and transitive, i.e., any stream is bisimilar to itself, one stream being bisimilar to another implies the other is bisimilar to the first, and if two streams are bisimilar to one particular other stream, they themselves are bisimilar to each other. In short, the bisimilarity relation is an equivalence relation.

```
Lemma bisimilarity_is_a_reflexive_relation :
  forall (xss : Stream block_n),
    bisimilar_Stream_block_n xss xss.

Lemma bisimilarity_is_a_symmetric_relation :
  forall xss_1 xss_2 : Stream block_n,
    bisimilar_Stream_block_n xss_1 xss_2 →
    bisimilar_Stream_block_n xss_2 xss_1.

Lemma bisimilarity_is_a_transitive_relation :
  forall xss_1 xss_2 xss_3 : Stream block_n,
    bisimilar_Stream_block_n xss_1 xss_2 →
    bisimilar_Stream_block_n xss_2 xss_3 →
    bisimilar_Stream_block_n xss_1 xss_3.
```

These lemmas are shown using the `cofix` tactic. In the transitivity lemma, we use the lemmas above that state that bisimilarity implies equal heads and bisimilar tails.

3.4.2 `stream_ref`

We define the inductive `stream_ref` operator that takes a stream (of blocks of length `n`) and an index as inputs and outputs the element at that index in the stream.

```
Fixpoint stream_ref (xss : Stream block_n) (i : nat) : block_n :=
  match xss with
    | Cons xs xss' ⇒ match i with
                       | 0 ⇒ xs
                       | S i' ⇒ stream_ref xss' i'
                     end
  end.
```

If the index is 0, the head element is output. Else, the operator is applied recursively to the tail of the stream and the predecessor of the index. We prove two unfolding lemmas for this function: one for when the index is 0 and one for when it is a successor of another natural number.

Using the `stream_ref` operator, we can define a new notion of equality of streams:

```
Definition equal_values_at_equal_indices (xss yss : Stream block_n) :=
  forall i : nat,
    stream_ref xss i = stream_ref yss i.
```

This proposition is true for two streams if and only if they have pairwise equal elements at all indices.

We prove two translation lemmas to be able to translate between bisimilarity and the new notion of equality above.

```
Lemma bisimilarity_implies_equal_values_at_equal_indices :
  forall xss_1 xss_2 : Stream block_n,
    bisimilar_Stream_block_n xss_1 xss_2 →
    equal_values_at_equal_indices xss_1 xss_2.


Lemma equal_values_at_equal_indices_implies_bisimilarity :
  forall xss_1 xss_2 : Stream block_n,
    equal_values_at_equal_indices xss_1 xss_2 →
    bisimilar_Stream_block_n xss_1 xss_2.
```

The first lemma is proven by induction in the `i` index introduced by the predicate in the conclusion of the lemma while the second lemma is proven by using the `cofix` tactic. With these lemmas, we can transform any problem involving bisimilarity into a problem about pairwise equality of elements. This transformation enables us to use induction in the `i` index introduced by the `equal_values_at_equal_indices` predicate.

3.4.3 `cut_of_stream`

In Chapter 6, we state and prove theorems about modes of operation. There, we specify both stream and list versions of the associated encryption and decryption functions. To aid us in the proofs of the theorems concerning the list versions of the functions, we define alternative specifications of the list versions of the general encryption and decryption functions in Section 6.8.2: these specifications make use of the corresponding specifications using streams.

To help us in this regards, we a stream operator that on input a stream of elements of the `block_n` type and a natural number `m` outputs a list consisting of the first `m` elements in the stream (in order): a cut of the stream.

```
Fixpoint cut_of_stream (xss : Stream block_n) (m : nat) : list block_n :=
  match m with
    | 0 ⇒ List.nil
    | S m' ⇒
      match xss with
        | Cons xs xss' ⇒ xs :: cut_of_stream xss' m'
      end
  end.
```

If m is in the base case, the empty list is output. Else, the cons of the head element of the stream and the output of the operator applied recursively to the tail stream and the predecessor of m is output.

Apart from two unfolding lemmas handling the base and inductive cases of m, we prove a number of properties about the cut_of_stream operator above.

First, we prove that if two streams are bisimilar, then cuts of equal lengths of these streams are equal.

```
Lemma bisimilarity_implies_equal_cuts_of_stream :
  forall (xss_1 xss_2 : Stream block_n) (m : nat),
    bisimilar_Stream_block_n xss_1 xss_2 →
    cut_of_stream xss_1 m = cut_of_stream xss_2 m.
```

This lemma is proven by induction in m where we exploit the lemmas in Section 3.4.1 stating that any two bisimilar streams have equal heads and bisimilar tails. The lemma is used in the proofs of the lemmas (in Section 6.8.2) saying that any function either satisfy both the regular and alternative specifications of the general encryption function using lists or none of them. Also, it is used in the proof of the general validity theorem using lists in Section 6.8.3.1.

Secondly, we show that the length of the cut is equal to m.

```
Lemma about_length_of_cut_of_stream :
  forall (xss : Stream block_n) (m : nat),
    length (cut_of_stream xss m) = m.
```

This lemma, too, is proven by induction in m. As the lemma above, this lemma is used in the proof of the general validity theorem using lists.

Additionally, we prove that for two pairs each consisting of two bisimilar streams where the cuts of length m of the first stream of each pair are equal, the cuts of length m of the second stream of each pair are equal, too.

```
Lemma about_cut_of_stream_bisimilarity :
  forall (xss_1 yss_1 xss_2 yss_2 : Stream block_n) (m : nat),
    bisimilar_Stream_block_n xss_2 xss_1 →
    bisimilar_Stream_block_n yss_2 yss_1 →
    cut_of_stream xss_1 m = cut_of_stream yss_1 m →
    cut_of_stream xss_2 m = cut_of_stream yss_2 m.
```

The peculiar ordering of the inputs is for easier application of the lemma. The lemma is proven by induction in m. In the proof, we apply the lemmas in Section 3.4.1 saying the bisimilar streams have equal head elements and bisimilar tail streams.

The lemma above is used when proving the repl_ciphertext_blocks_after_cut_w_zero_blocks_n_does_not_change_corr_plaintext_cut_general lemma in Section 6.8.3.1 where we need to unfold the stream version of the general decryption function (specified in Section 6.8.1) whose output is input to the cut_of_stream operator: there, the lemma above and the specification of the general decryption function are used in conjunction.

We prove that the output of the nth_error operator applied to the cut of a stream equals the output of the stream_ref operator applied to the stream itself inside the value constructor if the element at the index to which the operators are applied is not cut away.

```
Lemma about_nth_error_cut_of_stream_stream_ref :
  forall (xss : Stream block_n) (i m : nat),
    i < m →
    nth_error (cut_of_stream xss m) i = value (stream_ref xss i).
```

This lemma is proven by induction in the i index and case analysis of the length m of the cut. It is used in the proof of the streams_different_at_index_implies_cuts_of_equal_lengths_different_at_index_if_index_in_cut lemma in Section 6.7.2.

In that proof, we also use the lemma below.

```
Lemma nth_error_yields_error_if_index_not_in_cut :
  forall (xss : Stream block_n) (i m : nat),
    ~ i < m →
    nth_error (cut_of_stream xss m) i = error.
```

This lemma states that if the `nth_error` operator is applied to a cut and an index whose corresponding element is cut away, then the operator outputs `error`. It is proven as a corollary of the `nth_error_yields_error_if_index_out_of_bounds` lemma in Section 3.3.2 and using the `about_length_of_cut_of_stream` lemma in Section 3.4.3: the first lemma states that the `nth_error` operator outputs `error` when applied to an out-of-bounds index while the second lemma states that the length of the cut output by the `cut_of_stream` operator is equal to the natural number input to it.

3.4.4 `extend_list_into_Stream`

In this section, we introduce a list operator complementary to the `cut_of_stream` operator in the section above. It is used to extend lists of elements of the `block_n` type into streams of such elements. Like the `cut_of_stream` operator, it is used when defining alternative specifications of the general encryption and decryption functions using lists in Section 6.8.2. In this section, the natural number `n` refers to the parameter used in Chapter 6 representing the length of blocks and used when defining the `block_n` type (see Section 6.5.1).

Before we define the list extension operator, we define a function below outputting all-zero blocks, i.e., blocks consisting of only zero-bits (`false`s). The list extension operator appends a stream of such blocks (of length `n`) to its input list.

```
Fixpoint zero_block_variable_length (m : nat) : block_variable_length m :=
  match m with
    | O ⇒ nil bool
    | S m' ⇒ cons_block false (zero_block_variable_length m')
  end.
```

Here, we use the `block_variable_length` function (defined in Section 6.5.1) that on input some natural number `m` outputs a type representing blocks of length `m`. The inductive `zero_block_variable_length` function outputs an all-zero block of length equal to the input `m`.

Using the function above, we define all-zero blocks of length `n`.

```
Definition zero_block_n : block_n :=
  zero_block_variable_length n.
```

As mentioned above, the list extension operator appends a stream of all-zero blocks of length `n` to the input list: we define such a stream.

```
CoFixpoint all_zero_blocks_n_stream : Stream block_n :=
  Cons zero_block_n all_zero_blocks_n_stream.
```

We can now define the list extension operator: `extend_list_into_Stream`.

```
Fixpoint extend_list_into_Stream (xss : list block_n) : Stream block_n :=
  match xss with
    | xs :: xss' ⇒ Cons xs (extend_list_into_Stream xss')
    | List.nil ⇒ all_zero_blocks_n_stream
  end.
```

If the input list is empty, the output is just the stream of all-zero blocks of length `n`. Else, the output is the cons of the head element of the list and the output of the recursive application of the operator on the tail of the list.

In addition to proving unfolding lemmas for the operator above, we prove that, for any list of blocks of length `n`, first extending the list using the above operator and then cutting it afterwards using the

`cut_of_stream` operator defined in the section above with the input `m` being the original length of the list, results in a list equal to the input list.

```
Lemma cutting_away_extension_yields_original_list :
  forall (xss : list block_n),
    cut_of_stream (extend_list_into_Stream xss) (length xss) = xss.
```

This lemma is proven by induction in the input list `xss`. The lemma is used in the proof of the general validity theorem using lists in Section 6.8.3.1.

Chapter 4

The SSReflect Extension

> But in our opinion
> truths of this kind
> should be drawn from notions
> rather than from notations.
>
> Carl Friedrich Gauss

In this chapter, we describe the SSReflect Coq extension [2] and the SSReflect libraries and lemmas we use.

In Section 4.1, we give a short overview of the extension, and, in Section 4.2, we describe the libraries and list the lemmas we use.

4.1 Overview of SSReflect

One of the major achievements accomplished through the use of the Coq proof assistant is the proving of the four colour theorem: no more than four colours are necessary to colour the regions of a simple planar map such that no two adjacent regions have the same colour. This work—done by Georges Gonthier and Benjamin Werner in 2004—led to the development of the SSReflect ("small-scale reflection") extension of Coq.

Although the SSReflect extension started from the work done in proving the four colour theorem, the extension is much more general: besides supporting reflection between Booleans and propositions, it is useful for proving theorems involving a multitude of mathematical concepts. In this dissertation, we use this fact in Chapters 7 and 8 where we model and prove theorems about the RSA and ElGamal public-key cryptosystems as well as the RSA signature scheme.

SSReflect extends the tactic language of Coq making it possible to write more compact proofs and lessen the tediousness of very long proofs, e.g., the proof of the four colour theorem where a lot of non-trivial cases must be considered. However, the main reason that we use the SSReflect extension is not the language changes, but the accompanying SSReflect libraries: in Section 4.2, we describe the relevant SSReflect libraries and pertaining operators and lemmas.

4.2 Libraries

In this section, we consider the operators, predicates, and lemmas we use from the SSReflect libraries. In doing so, we use a number of predicates for operators. These predicates are defined in the `ssrfun` SSReflect library and listed below. Here, `S` and `T` are types.

The first predicates, the `left_id` and `right_id` predicates, capture the concepts of left respectively right identities.

```
Implicit Type op : S → T → T.
Definition left_id e op := forall x, op e x = x.

Implicit Type op : S → T → S.
Definition right_id e op := forall x, op x e = x.
```

The next predicates, the `left_zero` and `right_zero` predicates, capture the concepts of left respectively right zero elements.

```
Implicit Type op : S → T → S.
Definition left_zero z op := forall x, op z x = z.

Implicit Type op : S → T → T.
Definition right_zero z op := forall x, op x z = z.
```

The `associative` predicate is satisfied by any associative operator.

```
Implicit Type op : S → S → S.
Definition associative op := forall x y z, op x (op y z) = op (op x y) z.
```

Likewise, the `commutative` predicate holds for any commutative operator.

```
Implicit Type op : S → S → T.
Definition commutative op := forall x y, op x y = op y x.
```

The `right_distributive` predicate is satisfied by two operators if the first operator is right distributive with regards to the second operator.

```
Implicit Type op : S → T → T.
Definition right_distributive op add :=
  forall x y z, op x (add y z) = add (op x y) (op x z).
```

Lastly, we consider the `cancel` predicate that holds for two unary operators if applying the first operator and then the second operator is extensionally equal to the identity operator.

```
Variables (rT aT : Type) (f : aT → rT).
Definition cancel g := forall x, g (f x) = x.
```

4.2.1 The `ssrnat` Library

When modelling and proving theorems about the RSA and ElGamal schemes in Chapters 7 and 8, we do not use the standard arithmetic operators (on natural numbers) defined in the standard Coq libraries—not directly, at least. Instead, we use the operators defined in the `ssrnat` SSReflect library.

Besides the definitions of the operators, the `ssrnat` library contains lemmas concerning the operators. In the following, we go through the operators and lemmas we use.

`addn` The `ssrnat` library defines a new addition operator for natural numbers: the `addn` operator seen below.

```
Definition addn := nosimpl addn_rec.
Notation "m + n" := (addn m n) : nat_scope.
```

The "+" notation is used for the `addn` operator. The operator is defined in terms of the `nosimpl` notation shown below and the `addn_rec` operator defined in the `ssrnat` library. The `addn_rec` operator, in turn, is set to the `plus` operator defined in the `Coq.Init.Peano` standard library.

```
Definition addn_rec := plus.
```

Hence, the `addn` and `plus` operators are equal as expressed by the `plusE` lemma in the `ssrnat` library:

```
Lemma plusE : plus = addn.
```

```
Notation nosimpl t := (let: tt := tt in t).
```

The only lemma we use primarily concerning the addn operator is the add1n lemma below—other lemmas involve the operator.

```
Lemma addn1 n : n + 1 = n.+1.
```

".+1" is notation for the succ successor operator described below.

subn Likewise, the ssrnat library defines a new subtraction operator for natural numbers: the subn operator shown below.

```
Definition subn := nosimpl subn_rec.
Notation "m - n" := (subn m n) : nat_scope.
```

The "−" notation is used for the operator. As above, we use the no_simpl notation. In addition, the operator uses the subn_rec operator, also defined in the ssrnat library:

```
Definition subn_rec := minus.
```

The subn_rec operator is set to the minus operator defined in the Coq.Init.Peano standard library, and, as a result, the subn and minus operators are equal as stated below.

```
Lemma minusE : minus = subn.
```

We exploit that the natural number 0 is a right identity for the any the subn operator. This property is stated in the subn0 lemma in the ssrnat library.

```
Lemma subn0 : right_id 0 subn.
```

Here, the right_id predicate shown above is used.

Additionally, we use a lemma to switch between the subn operator and predn predecessor operator described below.

```
Lemma subn1 n : n - 1 = n.-1.
```

Here, ".−1" is shorthand notation for the predn operator.

muln The ssrnat library defines a multiplication operator for natural numbers: the muln operator.

```
Definition muln := nosimpl muln_rec.
Notation "m * n" := (muln m n) : nat_scope.
```

The "∗" notation is used for the muln operator. Similarly to above, the operator is defined in terms of the nosimpl notation and the muln_rec operator also defined in the ssrnat library. This operator, in turn, is set to the mult operator defined in the Coq.Init.Peano standard library.

```
Definition muln_rec := mult.
```

Thus, the muln and mult operators are equal. This fact is stated in the multE lemma of the ssrnat library.

```
Lemma multE : mult = muln.
```

We exploit that the muln operator is associative, commutative, and distributive—here, right-distributive—with the subn subtraction operator above.

```
Lemma mulnA : associative muln.

Lemma mulnC : commutative muln.

Lemma mulnBr : right_distributive muln subn.
```

The natural number 0 is both the left and right zero element of multiplication and, hence, the muln operator.

```
Lemma mul0n : left_zero 0 muln.

Lemma muln0 : right_zero 0 muln.
```

The natural number 1 is the left and right identities of multiplication and the `muln` operator.

```
Lemma mul1n : left_id 1 muln.

Lemma muln1 : right_id 1 muln.
```

If a product of two factors is positive, then so is each factor and vice versa.

```
Lemma muln_gt0 m n : (0 < m * n) = (0 < m) && (0 < n).
```

If a two-factor product is less than or equal to another two-factor product with the same left factor, then the left factor is 0 or the first right factor is less than or equal to the second right factor and vice versa.

```
Lemma leq_mul2l m n1 n2 : (m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
```

Here, we use the "==" infix notation for the `eq_op` predicate in Section 4.2.6.

If the factors of one two-factor product are pairwise less than or equal to the factors of another two-factor product, then the first product is less than or equal to the second product, i.e., the "less than or equal" relation is closed under pairwise multiplication.

```
Lemma leq_mul m1 m2 n1 n2 : m1 <= n1 → m2 <= n2 → m1 * m2 <= n1 * n2.
```

The same property holds for the "less than" relation.

```
Lemma ltn_mul m1 m2 n1 n2 : m1 < n1 → m2 < n2 → m1 * m2 < n1 * n2.
```

If two (two-factor) products with a common positive factor `m` are equal, then the other factors are equal, too, and vice versa. Here, we consider the case where `m` is the first factor of each product.

```
Lemma eqn_pmul2l m n1 n2 : 0 < m → (m * n1 == m * n2) = (n1 == n2).
```

The result of multiplying a positive natural number `m` with a natural number `n` greater than 1 is greater than `m`. This property is satisfied regardlessly of the order of the factors.

```
Lemma ltn_Pmull m n : 1 < n → 0 < m → m < n * m.

Lemma ltn_Pmulr m n : 1 < n → 0 < m → m < m * n.
```

expn The `expn` exponentiation operator on natural numbers is defined in the `ssrnat` library and shown below.

```
Variable T : Type.
Implicit Types m n : nat.
Implicit Types x y : T.

Definition iteri n f x :=
  let fix loop m := if m is i.+1 then f i (loop i) else x in loop n.
Definition iterop n op x :=
  let f i y := if i is 0 then x else op x y in iteri n f.

Definition expn_rec m n := iterop n muln m 1.
Definition expn := nosimpl expn_rec.
Notation "m ^ n" := (expn m n) : nat_scope.
```

We use the regular "^" notation for this exponentiation operator.

First, we consider two lemmas involving the natural number 0: raising any natural number to the power of 0 gives 1, and raising 0 to any positive natural number gives 0.

```
Lemma expn0 m : m ^ 0 = 1.

Lemma exp0n n : 0 < n → 0 ^ n = 0.
```

Likewise, we consider two lemmas involving the natural number 1: raising any natural number to the power of 1 does not alter it, and raising 1 to any natural number gives 1.

```
Lemma expn1 m : m ^ 1 = m.

Lemma exp1n n : 1 ^ n = 1.
```

Incrementing the exponent by 1 is extensionally equal to multiplying by the base (here, on the right).

```
Lemma expnSr m n : m ^ n.+1 = m ^ n * m.
```

The operator is distributive.

```
Lemma expnD m n1 n2 : m ^ (n1 + n2) = m ^ n1 * m ^ n2.
```

Lifting to the power of a (two-factor) product is extensionally equal to lifting to the powers of the factors.

```
Lemma expnM m n1 n2 : m ^ (n1 * n2) = (m ^ n1) ^ n2.
```

The order of (two) exponentiations does not matter.

```
Lemma expnAC m n1 n2 : (m ^ n1) ^ n2 = (m ^ n2) ^ n1.
```

succn and predn The ssrnat library defines successor and predecessor notations for natural numbers in terms of the S constructor and pred operator in the Coq standard libraries.

```
Notation succn := Datatypes.S.
Notation predn := Peano.pred.
```

Additionally, the library defines further notation:

```
Notation "n .+1" := (succn n) (at level 2, left associativity,
  format "n .+1") : nat_scope.
Notation "n .+2" := n.+1.+1 (at level 2, left associativity,
  format "n .+2") : nat_scope.
Notation "n .+3" := n.+2.+1 (at level 2, left associativity,
  format "n .+3") : nat_scope.
Notation "n .+4" := n.+2.+2 (at level 2, left associativity,
  format "n .+4") : nat_scope.

Notation "n .-1" := (predn n) (at level 2, left associativity,
  format "n .-1") : nat_scope.
Notation "n .-2" := n.-1.-1 (at level 2, left associativity,
  format "n .-2") : nat_scope.
```

The succn and predn operators cancel each other if the succn operator is applied first.

```
Lemma succnK : cancel succn predn.
```

Likewise, the operators cancel each other if the second operator is applied first, but only if the original operand is positive.

```
Lemma prednK n : 0 < n → n.-1.+1 = n.
```

Ordering Operators The ssrnat library defines the leq operator that on input two natural numbers output a bool: if the first natural number is less than or equal to the second natural number, then true is output; else false is output.

```
Definition leq m n := m − n == 0.
```

In terms of this operator, notations for respectively "less than or equal", "less than", "greater than or equal", and "greater than" are defined.

```
Notation "m <= n" := (leq m n) : nat_scope.
Notation "m < n" := (m.+1 <= n) : nat_scope.
Notation "m >= n" := (n <= m) (only parsing) : nat_scope.
Notation "m > n" := (n < m) (only parsing) : nat_scope.
```

We use multiple lemmas concerning the ordering operators when proving theorems about the RSA and ElGamal schemes in Chapters 7 and 8. We consider these in the following.

0 is less than or equal to any natural number and less than the successor of any natural number.

```
Lemma leq0n n : 0 <= n.

Lemma ltn0Sn n : 0 < n.+1.
```

No natural number is less than 0.

```
Lemma ltn0 n : n < 0 = false.
```

Being positive is equivalent to being different from 0 for any natural number.

```
Lemma lt0n n : (0 < n) = (n != 0).
```

Here, we use the "!=" infix notation in Section 4.2.6.

"Less than or equal" is a reflexive relation.

```
Lemma leqnn n : n <= n.
```

Conversely, "less than" is an anti-reflexive relation.

```
Lemma ltnn n : n < n = false.
```

"Less than" is a transitive relation.

```
Lemma ltn_trans n m p : m < n → n < p → m < p.
```

We can substitute one of the "less than"s with a "less than or equal" in the above lemma.

```
Lemma leq_ltn_trans n m p : m <= n → n < p → m < p.
```

Being less than the successor of a natural number is equivalent to being less than or equal to the natural number.

```
Lemma ltnS m n : (m < n.+1) = (m <= n).
```

The "less than" relation is a subset of the "less than or equal" relation.

```
Lemma ltnW m n : m < n → m <= n.
```

If a natural number is less than another natural number, the second number is not less than or equal to the first number.

```
Lemma ltnNge m n : (m < n) = ~~ (n <= m).
```

The proposition m <= n is logically equivalent to the Boolean m <= n coerced into a proposition via the hidden is_true function—see Section 4.2.5 below for definitions of the reflect predicate and this function.

```
Lemma leP m n : reflect (m <= n)%coq_nat (m <= n).
```

4.2.2 The `div` Library

The SSReflect library `div` concerns divisibility for natural numbers. Amongst other operators, it defines the `modn` operator that performs the modulo operation for natural numbers. Additionally, the library defines the `dvdn` and `coprime` predicates for divisibility and coprimality. Finally, the library includes a lemma, the `chinese_remainder` lemma, stating a special case of the Chinese remainder theorem which we use in the proof of the validity theorem in Section 7.4.3.2 for the RSA cryptosystem.

The predicates are predicates in the sense that they output Boolean values, not propositions, when applied to natural numbers.

In the following paragraphs, we take a closer look at these parts of the library.

`modn` The `div` library the defines the binary `modn` operator that outputs the remainder of the first operand modulo the second operand.

```
Definition modn_rec d := fix loop m := if m - d is m'.+1 then loop m' else m.
Definition modn m d := if d > 0 then modn_rec d.-1 m else m.
```

Several notations pertaining to the operator are defined.

```
Notation "m %% d" := (modn m d) : nat_scope.
Notation "m = n %[mod d ]" := (m %% d = n %% d) : nat_scope.
Notation "m == n %[mod d ]" := (m %% d == n %% d) : nat_scope.
Notation "m <> n %[mod d ]" := (m %% d <> n %% d) : nat_scope.
Notation "m != n %[mod d ]" := (m %% d != n %% d) : nat_scope.
```

We use several lemmas proven in the `div` library about the `modn` operator. These lemmas are shown below.

The remainder of 0 modulo any natural number is 0.

```
Lemma mod0n d : 0 %% d = 0.
```

The library defines the remainder of any natural number m modulo 0 to be m.

```
Lemma modn0 m : m %% 0 = m.
```

The remainder of any natural number modulo 1 is 0.

```
Lemma modn1 m : m %% 1 = 0.
```

If the natural number d is positive, then any remainder modulo d is less than d and vice versa.

```
Lemma ltn_mod m d : (m %% d < d) = (0 < d).

Lemma ltn_pmod m d : 0 < d → m %% d < d.
```

The remainder of any natural number m modulo any greater, natural number is m.

```
Lemma modn_small m d : m < d → m %% d = m.
```

If a product is reduced modulo d, then reducing the factors individually does not matter.

```
Lemma modnMml m n d : m %% d * n = m * n %[mod d].

Lemma modnMmr m n d : m * (n %% d) = m * n %[mod d].

Lemma modnMm m n d : m %% d * (n %% d) = m * n %[mod d].
```

Likewise, reducing the base modulo d of an exponentiation whose result is also reduced modulo d does not matter.

```
Lemma modnXm m n a : (a %% n) ^ m = a ^ m %[mod n].
```

The operation of finding the remainder modulo the same natural number is idempotent.

```
Lemma modn_mod m d : m %% d = m %[mod d].
```

dvdn The `div` library defines the `dvdn` predicate that is satisfied by two natural numbers if the first number divides the second number as well as the "%|" notation.

```
Definition dvdn d m := m %% d == 0.
Notation "m %| d" := (dvdn m d) : nat_scope.
```

We use a few lemmas pertaining to the `dvdn` operator. These lemmas are shown below.

The `dvdnP` lemma below allows reflecting between a proposition stating that the natural number `m` is a multiple (with the multiplier being a natural number) of the natural number `d` and the Boolean expression saying that `d` divides `m`: the proposition is true if and only if the Boolean expression is true.

```
Lemma dvdnP d m : reflect (exists k, m = k * d) (d %| m).
```

See the definition of the `reflect` predicate in Section 4.2.5.

Two natural numbers being congruent modulo some natural number `d` is equivalent to `d` dividing the difference between the former two natural numbers if the difference is non-negative.

```
Lemma eqn_mod_dvd d m n : n <= m → (m == n %[mod d]) = (d %| m - n).
```

coprime The `coprime` predicate defined in the `div` library is satisfied by two natural numbers if they are coprime, i.e., if the greatest common divisor of the two numbers is 1.

```
Fixpoint gcdn_rec m n :=
  let n' := n %% m in if n' is 0 then m else
  if m - n'.-1 is m'.+1 then gcdn_rec (m' %% n') n' else n'.
Definition gcdn := nosimpl gcdn_rec.

Definition coprime m n := gcdn m n == 1.
```

The `coprime` predicate is defined in terms of the binary `gcdn` operator (also defined in the `div` library) that outputs the greatest common divisor of two natural numbers.

As for the `dvdn` predicate above, we use a number of lemmas concerning the `coprime` predicate. These lemmas are shown below.

Any natural number is coprime with 1.

```
Lemma coprimen1 n : coprime n 1.
```

Coprimality is a symmetric relation.

```
Lemma coprime_sym m n : coprime m n = coprime n m.
```

Reducing the first natural number modulo the second natural number preserves coprimality and non-coprimality. This lemma is true also if the inputs are switched—the `coprime_modr` lemma in the `div` library captures this property.

```
Lemma coprime_modl m n : coprime (m %% n) n = coprime m n.
```

If the natural number `k` is positive and there exists a multiple (with the multiplier being a natural number) of `k` whose remainder is 1 modulo some natural number `n`, then `k` and `n` are coprime.

```
Lemma modn_coprime k n : 0 < k → (exists u, (k * u) %% n = 1) → coprime k n.
```

Any natural number dividing a two-factor product is equivalent to the number dividing the second factor if the number is coprime with the first factor.

```
Lemma Gauss_dvdr m n p : coprime m n → (m %| n * p) = (m %| p).
```

This property remains a fact if the factors are swapped (see the `Gauss_dvdl` lemma in the `div` library).

If a natural number is coprime with a two-factor product of natural numbers, then it is coprime with each factor and vice versa. This property holds for any other of inputs to the predicate. Here, we consider the situation where the product is the first input.

```
Lemma coprime_mull p m n : coprime (m * n) p = coprime m p && coprime n p.
```

Exponentiating one of the natural numbers preserves coprimality. Here, we consider the case where the first number is raised.

```
Lemma coprime_expl k m n : coprime m n → coprime (m ^ k) n.
```

chinese_remainder The div library includes a lemma stating a special case of the Chinese remainder theorem. This theorem says that if two natural numbers are congruent modulo a product of pairwise coprime natural numbers, then the two former numbers are congruent modulo each of the factors of the product.

The chinese_remainder lemma states the special case of the theorem where the product has two factors.

```
Variables m1 m2 : nat.
Hypothesis co_m12 : coprime m1 m2.

Lemma chinese_remainder x y :
  (x == y %[mod m1 * m2]) = (x == y %[mod m1]) && (x == y %[mod m2]).
```

We use this lemma when proving that the RSA cryptosystem is valid in Section 7.4.3.2.

4.2.3 The prime Library

The SSReflect library prime defines the prime predicate that outputs the Boolean value true if applied to a prime natural number and else outputs the false value when applied to a non-prime natural number. The library defines several operators (most of them pertaining to primes): of these operators, we use the unary totient operator implements Euler's totient function (or Euler's phi function) that yields the number of totatives of the input natural number, i.e., the number of positive natural numbers that are both less than and coprime with the input natural number.[1]

We use both the prime predicate and the totient operator extensively in Chapters 7 and 8 where we model and prove theorems about the RSA and ElGamal schemes.

In the following, we consider the predicate and the operator in more detail.

prime The prime library defines the prime predicate (on natural numbers) which is satisfied for all primes only.

```
Definition edivn_rec d :=
  fix loop m q := if m - d is m'.+1 then loop m' q.+1 else (q, m).
Definition edivn m d := if d > 0 then edivn_rec d.-1 m 0 else (0, m).

Fixpoint elogn2 e q r {struct q} :=
  match q, r with
  | 0, _ | _, 0 ⇒ (e, q)
  | q'.+1, 1 ⇒ elogn2 e.+1 q' q'
  | q'.+1, r'.+2 ⇒ elogn2 e q' r'
  end.

Definition prime_decomp n :=
  let: (e2, m2) := elogn2 0 n.-1 n.-1 in
  if m2 < 2 then 2 ^? e2 :: 3 ^? m2 :: [::] else
  let: (a, bc) := edivn m2.-2 3 in
```

[1] By convention, Euler's totient function outputs 1 on input 0. However, the totient operator outputs 0 when the operand is 0—which is consistent with the definition of the function.

```
  let: (b, c) := edivn (2 - bc) 2 in
  2 ^? e2 :: [rec m2.*2.+1, 1, a, b, c, 0].

Definition prime p := if prime_decomp p is [:: (_ , 1)] then true else false.
```

These definitions are included in the `prime` library except for the definition of the `edivn` and `edivn_rec` operators from the `div` SSReflect library. The `prime_decomp` function outputs a list of the prime factors of the input natural number along with their multiplicities: the `prime` predicate evaluates to `true` if the prime factorisation of the input natural number consists of one prime factor with multiplicity 1—in this case, the natural number must be a prime.

We use a few lemmas in the `prime` library involving the `prime` predicate. We show these lemmas below.

Any prime number is greater than 1 and, in extension, 0.

```
Lemma prime_gt1 p : prime p → 1 < p.

Lemma prime_gt0 p : prime p → 0 < p.
```

For any prime p, p being coprime with some natural number m is equivalent to p not dividing m.

```
Lemma prime_coprime p m : prime p → coprime p m = ~~ (p %| m).
```

If a prime p divides a prime q, then they are equal and vice versa.

```
Lemma dvdn_prime2 p q : prime p → prime q → (p %| q) = (p == q).
```

totient The unary `totient` operator is defined in the `prime` library. As mentioned above, this operator implements Euler's totient function.

```
Fixpoint foldr s := if s is x :: s' then f x (foldr s') else z0.

Definition add_totient_factor f m := let: (p, e) := f in p.-1 * p ^ e.-1 * m.

Definition totient n := foldr add_totient_factor (n > 0) (prime_decomp n).
```

See the definition of the `prime_decomp` function in the paragraph above. The `foldr` function is defined in the `seq` SSReflect library.

In Chapters 7 and 8, we exploit some properties about the `totient` operator. These properties are stated in the lemmas below.

By convention, the totient of 0 is 1. However, the `totient` operator outputs 0 on input 0. The operator only outputs 0 on this input.

```
Lemma totient_gt0 n : (0 < totient n) = (0 < n).
```

The totient of any prime raised to the power of a positive natural number is equal to the predecessor of the prime multiplied with the prime raised to the power of the predecessor of the exponent.

```
Lemma totient_pfactor p e :
  prime p → e > 0 → totient (p ^ e) = p.-1 * p ^ e.-1.
```

If two natural numbers are coprime, then the totient of their product is the product of their totients.

```
Lemma totient_coprime m n :
  coprime m n → totient (m * n) = totient m * totient n.
```

4.2.4 The `cyclic` Library

The SSReflect library `cyclic` concerns properties of cyclic group. We only use only one theorem from this library: the `Euler_exp_totient` theorem stating Euler's theorem.

```
Theorem Euler_exp_totient a n : coprime a n → a ^ totient n = 1 %[mod n].
```

If two natural numbers a and n are coprime, then a lifted to the power of n is congruent to 1 modulo n.

The theorem above is used when proving that the RSA encryption scheme is valid in Section 7.4.3.2 and when proving the `about_coprime_inverses_modulo_Sp'` lemma in Section 8.4.2.1.

4.2.5 The `ssrbool` Library

The SSReflect library `ssrbool` is concerned with Boolean predicates and operators.

The Coq standard library `Coq.Init.Datatypes` defines the `is_true` function that interprets Booleans as propositions.

```
Definition is_true b := b = true.
```

`is_true` is activated as a coercion in the `ssrbool` library.

```
Coercion is_true : bool >→ Sortclass.
```

This coercion enables us to use Booleans in place of propositions: the Booleans are coerced to propositions.

The `ssrbool` library defines the `reflect` predicate that relate propositions and Booleans.

```
Inductive reflect (P : Prop) : bool → Set :=
  | ReflectT of P : reflect P true
  | ReflectF of ~ P : reflect P false.
```

Now, `reflect P b` means that `P` and `is_true b` are logically equivalent propositions [].

A multitude of lemmas involving the `reflect` predicate are proven in the library. The few we use of these lemmas are shown below.

If the `reflect` predicate is satisfied for the proposition `P` and Boolean `b`, then `P` implies `b`, `not P` implies `negb b`, and `b` implies `P`. Remember, that the Booleans are coerced to propositions via the `is_true` function where necessary.

```
Variables (P : Prop) (b : bool).
Hypothesis (Pb : reflect P b).

Lemma introT : P → b.
Lemma introN : ~ P → ~~ b.
Lemma elimT : b → P.
```

Using these lemmas, we can switch between propositions and coerced Booleans in our proofs.

We use two additional lemmas concerning the predicate.

```
Variables b1 b2 : bool.

Lemma idP : reflect b1 b1.
Lemma andP : reflect (b1 ∧ b2) (b1 && b2).
```

The `idP` lemma effectively states that `is_true b1` and `is_true b1` are logically equivalent propositions—remember, the Boolean `b1` is coerced to a proposition via the `is_true` function above when used as the first input to the `reflect` predicate. The `andP` lemma allows us to switch between the logical and Boolean conjunction.

Furthermore, the library includes several lemmas not pertaining to the `reflect` predicate. Below, the ones we use are shown. In the following, remember that Booleans might be coerced into propositions.

First, we have a trivial lemma stating that `not false` holds.

```
Lemma not_false_is_true : ~ false.
Definition notF := not_false_is_true.
```

Likewise, we have two negation lemmas stating that if a Boolean is `false`, then the negation of the Boolean holds and vice versa.

```
Lemma negbT b : b = false → ~~ b.
Lemma negbTE b : ~~ b → b = false.
```

We use three lemmas concerning `if-then-else` expressions.

```
Variables (A : Type) (b : bool) (vT vF : A).

Lemma ifT : b → (if b then vT else vF) = vT.
Lemma ifF : b = false → (if b then vT else vF) = vF.
Lemma if_same : (if b then vT else vT) = vT.
```

If `b` is `true`, then the `if-then-else` statement equals the `then` expression. If `b` is `false`, then the `if-then-else` statement equals the `else` expression. If the `then` and `else` expressions are equal, then the `if-then-else` statement equals these expressions.

If one Boolean being `true` implies another Boolean being `true` also, then the Boolean `orb` operator applied to the Booleans outputs a Boolean equal to the second Boolean. Here, we consider the case where the first Boolean is the first input to the operator.

```
Lemma orb_idl (a b : bool) : (a → b) → a || b = b.
```

4.2.6 The `eqtype` Library

The SSReflect library `eqtype` concerns types with a decidable equality via the `eqType` structure that encompasses such types.

The library uses the `eq_op` predicate. This predicate holds for two elements if and only if they are equal. The library defines notations involving the predicate.

```
Notation "x == y" := (eq_op x y)
  (at level 70, no associativity) : bool_scope.
Notation "x != y" := (~~ (x == y))
  (at level 70, no associativity) : bool_scope.
```

We use two lemmas in the library: the `eqP` and `eq_refl` lemmas.

The `eqP` lemma provides reflection between the standard Leibniz equality of Coq and the `eq_op` equality via the `reflect` predicate in Section 4.2.5 for elements of types encompassed by the `eqType` structure. Printing the lemma, we get the following:

```
eqP : forall (T : eqType) (x y : T), reflect (x = y) (x == y)
```

The `eq_refl` lemma states that the `eq_op` predicate is reflexive.

```
Lemma eq_refl (T : eqType) (x : T) : x == x.
```

Part III

Contributions

Chapter 5

The One-Time Pad

Let battle be joined!

Cenarius

In this chapter, we define and prove several lemmas and theorems about the one-time pad cryptosystem.

In Section 5.1, we introduce the one-time pad cryptosystem and list the goals and contributions of this chapter. In Section 5.3.1, we define the one-time pad cryptosystem and give an overview of the properties that we prove about it. In Section 5.4, we specify the functions of the one-time pad cryptosystem. These specifications set up the base for stating and proving theorems about the one-time pad functions. These theorems along with descriptions of their proofs are included in Section 5.5. Finally, we summarise and conclude on the results in Section 5.6.

5.1 Introduction

In this chapter, we consider the one-time pad cryptosystem. This cryptosystem encrypts and decrypts plaintexts or ciphertexts consisting of bits by bitwise XOR'ing them with the key itself consisting of bits. In [3], all plaintexts, ciphertexts, and keys are assumed to have the same number, n, of bits. We, however, allow different numbers such that, e.g., encrypting a plaintext under a key block with more bits is possible—doing so is equivalent to encrypting the plaintext under a key where the excess bits at the end of the key are ignored.

We seek to model and prove theorems about the cryptosystem in this chapter. Before doing so, however, we define the cryptosystem in mathematical terms so that any confusion about the details of the system is settled. After defining the system, we are able to reason about the properties of it: first, the system is a valid cryptosystem by which we mean that, for any plaintext and any key, first encrypting the plaintext and then decrypting the resulting ciphertext under the same key yields the original plaintext. Additionally, the keys can only be used once in the cryptosystem (hence its name): if two plaintexts are encrypted under the same key, then the key bits at which the plaintexts are equal are revealed—thus also revealing the indices at which they are unequal. Lastly, the one-time pad encryption function is onto for any plaintext in the sense that, for any plaintext and ciphertext of equal lengths, there exists a key such that the encryption of the plaintext under that key is the ciphertext.

Now that the cryptosystem is defined, we can specify the encryption and decryption functions of the system in Coq. These specifications facilitate stating the abovementioned properties about the system as theorems in Coq: we state and prove that the cryptosystem is valid, that encrypting twice under the same key reveals non-trivial information about the plaintexts encrypted, and that the encryption function is onto for any plaintext.

We only consider plaintexts, ciphertexts, and keys of finite lengths in this chapter. The use of streams and the `Stream` type in Section 3.4 is deferred to Chapter 6 where we consider modes of operation.

5.2 Technical Details

The code in this chapter is contained in the `one-time pad cryptosystem.v` file in the `Ole-Dalgaard-Lauridsen_code/OTP` directory in the archive file at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_code.tar.gz`.

5.3 Definitions and Properties

In this section, we define the one-time pad functions and state the properties we prove about them in this chapter: in Section 5.3.1, we define the functions while, in Section 5.3.2, we list the properties.

5.3.1 Definition

The one-time pad is a cryptosystem [3] which means that it is an encryption scheme that has both a family of encryption functions that map elements from a finite plaintext space ($\mathcal{P}$) to elements in a finite ciphertext space ($\mathcal{C}$) and a family of decryption functions that map the other way. Each of the families include a function for each key in a finite keyspace ($\mathcal{K}$). Now, it holds that first encrypting any plaintext and then decrypting it with the same key results in that same plaintext [3].

The one-time pad can be defined as follows [3]: Let n be a positive integer. Then $\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_2)^n$. For any key $K \in \mathcal{K}$, the encryption function e_K will on input $x \in \mathcal{P}$ output the vector sum modulo 2 of x and K. The corresponding decryption function d_K will be identical to e_K (recall, $\mathcal{P} = \mathcal{C}$). To clarify, for $x = (x_1, \ldots, x_n) \in \mathcal{P}$, $y = (y_1, \ldots, y_n) \in \mathcal{C}$, and $K = (K_1, \ldots, K_n) \in \mathcal{K}$, we have that

$$e_K(x) = (x_1 + K_1 \bmod 2, \ldots, x_n + K_n \bmod 2) \tag{5.1}$$

and

$$d_K(y) = (y_1 + K_1 \bmod 2, \ldots, y_n + K_n \bmod 2). \tag{5.2}$$

We think of $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{K}$ as sets of bit strings of length n.

5.3.2 Properties

It is seen that the one-time pad cryptosystem adheres to the rule that encrypting and then decrypting with the same key gives the original plaintext. This property follows from the fact that, if we think of bits as natural numbers being 0 or 1, adding any key bit K_i twice modulo 2 to some bit results in that bit again:

$$\forall 1 \leq i \leq n : (x_i + K_i \bmod 2) + K_i \bmod 2 = x_i. \tag{5.3}$$

The reason this cryptosystem has the prefix "one-time" is that it can only be used once for any particular key: if two plaintext bit strings are encrypted under the same key, then the indices at which they are equal (and hence also the indices at which they differ) are revealed by the corresponding ciphertext bit strings. This property follows from the fact that bits at equal indices are XOR'ed with the same key bits and that XOR is a permutation over $\{0, 1\}$ when the key bit is constant (regardlessly of the value of the key bit).

The one-time pad encryption function is onto for any plaintext in the sense that, for any plaintext and ciphertext of equal lengths, there exists a key such that the ciphertext is the result of encrypting the plaintext under the key: the key that equals the bitwise XOR of the plaintext and ciphertext is such a key since encrypting under this key amounts to bitwise XOR'ing the ciphertext twice with the plaintext

yielding the ciphertext. The reason we require the plaintext and ciphertext to be of equal lengths is that we allow for differing lengths. However, the length of the ciphertext resulting from encrypting a plaintext is always of the same length as the plaintext since we shorten or lengthen (by appending zero-bits) the key so that it has the same length as the plaintext and encrypt the plaintext under this modified key. Because of this property, we can only reach ciphertexts of the same length as the plaintext regardlessly of the chosen key.

5.4 Specifications

Before being able to prove any theorems about the one-time pad cryptosystem, we have to specify it. This specification is twofold—both the encryption and decryption functions are specified.

In Section 5.4.1, the encryption function is specified while the decryption function is specified in Section 5.4.2.

5.4.1 The Encryption Function

First, we specify the encryption function:

```
Definition specification_of_one_time_pad_encryption_function
           (one_time_pad_enc : list bool → list bool → list bool) :=
  (forall ks : list bool,
     one_time_pad_enc nil ks = nil)
  ∧
  (forall ms : list bool,
     one_time_pad_enc ms nil = ms)
  ∧
  (forall (m k : bool) (ms' ks' : list bool),
     one_time_pad_enc (m :: ms') (k :: ks') =
     xorb m k :: one_time_pad_enc ms' ks').
```

We use lists of Boolean values (`list bool`) to represent bit strings. Doing so facilitates the use of a number of functions and lemmas in the standard library of Coq. Here, we use the `xorb` operator that takes two `bool`s as input and outputs their exclusive-or. The `xorb` operator and some of the lemmas pertaining to it are described in Section 3.2.

The two first clauses handle special circumstances which are explained below. The third clause is the most important part of the specification. It says that the encryption of a plaintext `m :: ms'` using the key `k :: ks'` results in the XOR of `m` and `k` consed onto the result of the recursive call with the respective tails as input. In this way, the bits (represented by `bool`s) are XOR'ed one pair at a time.

The first and second clauses of the specification deal with situations where the plaintext and the key are of different lengths. Normally, we assume these to be of equal lengths, but we choose to include them. In the case of unequal lengths, we cannot just XOR them bitwise because at some point one of the lists runs out of bits. If the plaintext is shorter than the key, we ignore the rest of the key as if it were as short as the plaintext. If, on the other hand, the plaintext is longer than the key, we do not encrypt the rest of the plaintext and just pass it through which amounts to appending the key with zeroes (`false`s in our representation) so that the lengths of the key and the plaintext match and encrypting the plaintext regularly under the modified key. Of course, doing so reveals the last part of the plaintext to a potential adversary.

5.4.2 The Decryption Function

Second, we specify the decryption function. As mentioned earlier, this specification equals the specification above of the encryption function modulo renaming, so we could just make a wrapper function around the encryption function (or just use it directly). We choose to specify it in its entirety for clarity.

The specification is as follows:

```
Definition specification_of_one_time_pad_decryption_function
           (one_time_pad_dec : list bool → list bool → list bool) :=
  (forall ks : list bool,
     one_time_pad_dec nil ks = nil)
  ∧
  (forall cs : list bool,
     one_time_pad_dec cs nil = cs)
  ∧
  (forall (c k : bool) (cs' ks' : list bool),
     one_time_pad_dec (c :: cs') (k :: ks') =
     xorb c k :: one_time_pad_dec cs' ks').
```

Again, the three clauses handle three different situations: the first clause ensures that any surplus key is ignored. The second clause decrypts any surplus ciphertext as if the key were appended with zeroes (in Coq, `false`s). Lastly, the third clause says that the XOR of the head elements is consed onto the result of the output of the function applied to the tails.

5.5 Theorems and Proofs

Having specified the one-time pad functionalities, we are ready to state and describe the proofs of some of the properties of the one-time pad cryptosystem.

In Section 5.5.1, we show that the one-time pad actually is a cryptosystem—or rather that the last one of the requirements for being a valid cryptosystem is satisfied (see [3]): $\forall m \in \mathcal{P} \forall K \in \mathcal{K} : d_K(e_K(m)) = m$. In Section 5.5.2, we state and prove a theorem that implies that the one-time pad cryptosystem is insecure when keys are reused. Finally, in Section 5.5.3, we prove that the encryption function is onto for any plaintext when seen as a function from keys to ciphertexts (i.e., given any plaintext, any ciphertext of the same length can be the result of encrypting).

5.5.1 The One-Time Pad is a Valid Cryptosystem

The first theorem we prove about the one-time pad is that it satisfies one of the conditions of being a cryptosystem: $\forall m \in \mathcal{P} \forall K \in \mathcal{K} : d_K(e_K(m)) = m$ which means that for any given key, the result of encrypting and then decrypting any plaintext is the plaintext.

```
Theorem encrypting_and_then_decrypting_yields_original_plaintext :
forall one_time_pad_enc one_time_pad_dec :
         list bool → list bool → list bool,
  specification_of_one_time_pad_encryption_function one_time_pad_enc →
  specification_of_one_time_pad_decryption_function one_time_pad_dec →
  forall ms ks : list bool,
    one_time_pad_dec (one_time_pad_enc ms ks) ks = ms.
```

This theorem explicitly states that for any encryption function and decryption function satisfying the specifications of the encryption function respectively the decryption function, it holds that for any plaintext and key the original plaintext is output when first encrypting and then decrypting. The theorem is proved by induction in the plaintext `ms` and uses the lemma `about_xorb_r` (see Section 3.2.2).

5.5.2 The One-time Pad is "One-Time"

The name of the one-time pad cryptosystem is for a reason: the encryption function should not be allowed to be applied to two messages using the same key because of the fact that doing so reveals the

indices at which the plaintext bit strings are equal (and, hence, also the indices at which they are different) as described in Section 5.3.2. The theorem below states that the plaintexts are equal at some index `i` if and only if the corresponding ciphertexts are equal at that position.

```
Theorem encrypting_twice_with_same_key_reveals_positions_with_equal_bits :
  forall one_time_pad_enc : list bool → list bool → list bool,
    specification_of_one_time_pad_encryption_function
      one_time_pad_enc →
    forall (ms_1 ms_2 ks : list bool) (i : nat),
      nth_error (one_time_pad_enc ms_1 ks) i =
      nth_error (one_time_pad_enc ms_2 ks) i
      ↔
      nth_error ms_1 i = nth_error ms_2 i.
```

We make use of the standard library operator of `nth_error` described in Section 3.3: this operator takes a list and a natural number `i` as inputs and outputs the element at index `i`. If the index is out-of-bounds, it outputs an `error`. Else, it outputs the appropriate element inside the `value` constructor.

We use this operator to avoid having to submit default values to be output when out of bounds and to avoid having to include premises stating that the natural number `i` should be less than the length of either of the plaintexts: if the index is greater than or equal to both of the lengths of the plaintexts, then the outputs of the operator equal `error` when inputting any of the plaintexts or any of the ciphertexts (because of the length preservation of the one-time pad encryption function—see the `cs_same_length_as_ms` lemma in the code) in which case the theorem holds. If the index is only greater than or equal to one of the lengths of the plaintexts, we end up with a false expression on either side of the biimplication sign ↔ —again, because of the length preservation of the encryption function.

The proof of the theorem is done by induction in the `i` index and case analysis of the two plaintexts and the key which means that we have $2^4 = 16$ subgoals—one for each of the cases: both the `nat` and the `list` types have two cases. However, most of them are solvable with only a simple `rewrite` or two (using the hypotheses implied by the encryption function satisfying the specification of the one-time pad encryption function) followed by `reflexivity`.

In some of the cases, we have contradictory hypotheses, e.g., at one point (after using unfolding lemmas for the `nth_error` function), we have the subgoal `value (xorb m_1 k) = error ↔ value m_1 = error`. When we split up the biimplication in the subgoal, we get two subgoals where the premise before the → is contradictory. Hence, we can use `split; intro H; discriminate H.`—this splits the goal into two subgoals and then introduces for each subgoal the contradictory premise that we can use the `discriminate` tactic on.

5.5.3 The One-Time Pad Encryption Function is Onto for Any Plaintext

The last theorem we prove states that for any plaintext we can get any ciphertext of the same length by chosen an appropriate key.

```
Theorem one_time_pad_is_onto_for_any_particular_ms :
  forall one_time_pad_enc : list bool → list bool → list bool,
    specification_of_one_time_pad_encryption_function
      one_time_pad_enc →
    forall ms cs : list bool,
      length ms = length cs →
      exists ks : list bool,
        one_time_pad_enc ms ks = cs.
```

The proof of this theorem uses an auxiliary lemma that states that if the encryption function is applied to some plaintext `ms` and a key resulting from encrypting the plaintext under some key `ks`, then the resulting ciphertext is `ks` (the key). We require, however, that the plaintext and the key are of equal

lengths. The lemma follows directly from the fact that XOR'ing twice with the same `bool` does nothing (see `about_xorb_l` in Section 3.2.2) and is proved by induction in the plaintext and case analysis of the key.

```
Lemma about_one_time_pad_enc :
  forall one_time_pad_enc : list bool → list bool → list bool,
    specification_of_one_time_pad_encryption_function
      one_time_pad_enc →
    forall ms ks : list bool,
      length ms = length ks →
      one_time_pad_enc ms (one_time_pad_enc ms ks) = ks.
```

The theorem itself is proved by case analysis of the plaintext and the ciphertext, and, hence, we do not directly use induction. However, we use induction indirectly by applying the `about_xorb_l` lemma. Also, the `length` lemmas described in Section 3.3.2 are used.

5.6 Summary and Conclusion

Our goal in this chapter has been to model and prove theorems about the one-time pad cryptosystem: we have succeeded in doing so.

First, we have defined the system to get a clear grasp of the system and its properties after which we have specified the encryption and decryption functions of the system in Coq. Then, using these specifications, we have stated and proven the following properties: the system is valid, encrypting two plaintexts under the same key reveals the indices at which the plaintexts are equal (thereby also revealing the indices at which they are different), and, for any plaintext and ciphertext of equal lengths, a key exists such that the result of encrypting the plaintext under that key equals the ciphertext.

The work done in this chapter has been a fitting stepping stone: we are now better equipped to take on more difficult tasks. A lesson learned when specifying and proving properties about the one-time pad cryptosystem is that it would have been desirable to use vectors instead of lists. In this way, we could both have followed the definition of the system in [3] more precisely—there, the plaintext and ciphertext spaces and the keyspace were defined to be $(\mathbb{Z}_2)^n$ where n is a natural number—and avoided having to consider different lengths of input so that, e.g., in the `one_time_pad_is_onto_for_any_particular_ms` lemma in Section 5.5.3, we would not have to assume equal lengths of the plaintext and ciphertext.

With this new insight, we are ready to take on the next goal of this work.

Chapter 6

Modes of Operation

> Everything changes
> and nothing stands still.
>
> Heraclitus

In this chapter, we describe, model, and prove theorems about modes of operation and MAC schemes built on top of them.

First, in Section 6.1, we introduce block ciphers, modes of operation, and MAC (Message Authentication Code) schemes built on top of these modes and outline the goals of this chapter and the procedure followed to achieve them. In Section 6.2, we describe how to obtain the code in this chapter and how it is organised, and, in Section 6.3, we introduce some of the terminology used in this chapter. Then, in Section 6.4, we give an overview and precise definitions of the various modes of operation and MAC schemes and specify the theorems we prove. In Sections 6.5, 6.6, and 6.7, we describe the types, auxiliary functions, and predicates used when proving these theorems. In Section 6.8, we specify generalised versions of the functions of the modes and MAC schemes and prove theorems about them. These specifications and theorems are used in Section 6.10 where we specify and prove theorems about the modes and MAC schemes while following the procedures described in Section 6.9.

6.1 Introduction

In this section, we introduce block ciphers, modes of operation, and MAC schemes built on top of these modes and describe the goals of this chapter and the approach we follow to achieve them.

Block ciphers are deterministic encryption schemes that operate on blocks of bits. The corresponding encryption and decryption functions take as input a bit string of length n representing a plaintext respectively a ciphertext and a bit string of length k representing a key used for both encryption and decryption (the modes of operation are symmetric-key systems). Hence, blocks of n bits are encrypted into blocks of n bit and vice versa for decryption.

This functionality alone renders us able to encrypt any bit string of any length (even infinite strings): we can divide the string into blocks of length n (some padding might be necessary), encrypt them—maybe in parallel—and lastly concatenate the results. Then decryption will work analogously.

However, this naïve method brings about a major problem: equal plaintext blocks are encrypted into equal ciphertext blocks since the scheme is deterministic and we use the same key for all blocks. Hence, an adversary can—just from the ciphertext—tell some significant information about the plaintext which might be highly undesirable.

To remedy this issue, a concept called 'Modes of Operation' is used. This concept introduces feedback blocks which we use to influence the encryption and decryption of plaintext and ciphertext blocks. In

this way, we can obtain different ciphertext blocks when encrypting a plaintext block by using different feedback blocks, and vice versa for decryption. The modes use underlying block ciphers—in some cases, just the encryption function of the block cipher.[1]

The feedback blocks are computed when encrypting or decrypting: when encrypting or decrypting the i^{th} plaintext or ciphertext block, we compute the feedback block used to influence the encryption or decryption of the $(i+1)^{th}$ plaintext or ciphertext block. This latter feedback block can be dependent on the underlying block cipher, the key, the i^{th} plaintext or ciphertext block, and the feedback block computed when encrypting or decrypting the $(i-1)^{th}$ plaintext or ciphertext block. The feedback block for the encryption or decryption of the first plaintext or ciphertext block is given as input to the encryption or decryption function of the modes of operation. We call this initial feedback block the initialisation vector (or short, the IV).

In this chapter, we consider several modes of operation including three new modes. These modes differ in the ways in which they encrypt or decrypt plaintext or ciphertext blocks and compute feedback blocks. In Section 6.4, we define the modes and state the properties we show about them.

Our goal is to show the following properties of the modes.

- We wish to show that all the modes are valid, i.e., encrypting and then decryption a plaintext under the same key results in the plaintext again, under differing assumptions about the underlying block encryption function (or block cipher if also the block decryption function is used).

- We wish to show that each mode either does or does not propagate plaintext changes when encrypting (not both), i.e., if a plaintext block is changed, then either all ciphertext blocks after and including the corresponding ciphertext block are changed or only the corresponding ciphertext block is changed. For each encryption function of the modes we consider, we show that it propagates plaintext changes or not. Often, these properties are proven as consequences of assumptions about the underlying block encryption functions.

- Likewise, we wish to prove that all encryption functions of the modes of operation propagate IV changes, i.e., if the inputs to any of the functions remain unchanged except for the IV, then all resulting ciphertext blocks are changed. We prove this property as an implication of the underlying block encryption functions being invertible with regards to the in-the-context-of-these-functions plaintext blocks—and, in the case of the f mode defined in Section 6.4.4, the f function being invertible, too.

Before being able to prove these properties, however, we have to specify the encryption and decryption functions of the modes. Here, we consider two different versions: functions that encrypt or decrypt streams of plaintext or ciphertext blocks, i.e., the number of blocks is infinite, and functions that encrypt or decrypt lists of plaintext or ciphertext blocks, i.e., the number of blocks is finite. We consider both cases since modes of operation can handle both finite and infinite numbers of blocks.

We specify the encryption and decryption functions of each mode of operation in terms of general encryption and decryption functions specified in Sections 6.8.1 and 6.8.2. Here, as for the functions of the modes, we consider two versions of the general functions: ones handling streams of blocks and ones handling lists of blocks. The general functions are similar to the corresponding functions of the modes: they use IVs and feedback blocks to influence their outputs. However, how these blocks influence the outputs and how consecutive feedback blocks are computed are not determined: here, we let two functions input to the general functions settle these details. The first function is called the block function. This function settles how the feedback blocks influence the outputs. The second function is called the feedback function. This function settles how feedback blocks are computed.

[1] Throughout this chapter, we sometimes refer to the underlying block cipher as the underlying block encryption scheme while we refer to the encryption and decryption functions of the block cipher as the underlying block encryption and decryption functions.

Now, we can specify the encryption and decryption functions of the modes by first specifying the corresponding block and feedback functions for the modes (here, different functions might be used for encryption and decryption) and then specifying the encryption and decryption functions themselves in terms of the general functions applied to these block and feedback functions.

We wish to show the properties mentioned above of the modes. Here, the general encryption and decryption functions come to our aid. By showing that the properties hold for these general functions as consequences of the input block and feedback functions having certain properties, we can show the properties for each mode by first showing that these certain properties hold for the block and feedback functions corresponding to the mode and using this implication. Hence, all the theorems about each mode of operation are corollaries of theorems about the general functions. In Section 6.8.3, we prove the general theorems while, in Section 6.10, we prove the corresponding non-general corollaries for each mode.

We choose to use this procedure of first specifying general functions and then specifying the non-general functions in term of these general function to be able to be able to easily extend our results to other and, potentially, new modes of operation: now, when defining a mode, we merely have to specify the corresponding block and feedback functions, prove certain properties of these functions, and specify the encryption and decryption functions of the mode as being extensionally bisimilar or equal to the corresponding general functions applied to the block and feedback functions after which we can prove theorems stating properties of the mode as corollaries of the general theorems. We follow this procedure extensively in this chapter: all the modes of operation are handled doing so—both the already defined CBC, CFB, and CTR modes as well as the new IFB, f, and BENC modes. See Section 6.4 for definitions of these modes.

One can define MAC schemes[2] in terms of the encryption functions of the modes of operation. For each of these schemes, we define the MAC value, output by the MAC function, of a message of blocks to be the last block of the ciphertext resulting from encrypting the (finite) message using the encryption function of the particular mode of operation under the given key. Here, we use an all-zero-bits block as the IV.

However, for the MAC function to be interesting, the underlying encryption function of the particular mode should propagate plaintext changes. If so, the MAC value changes if one of the blocks of the message is changed. This property is desirable when using MAC schemes to protect integrity of messages: a change of a block can be discovered by recomputing the MAC value and comparing it to the old value.[3] Hence, we specify, in Section 6.10, corresponding MAC functions for the modes whose encryption functions propagate plaintext changes and prove that changes in the blocks of the input messages are propagated to the output MAC values.

We specify these MAC functions in terms of a general MAC function specified in Section 6.8.2. This general function is analogous to the regular MAC functions: it encrypts the message and output the last block. Here, the encryption is done using the abovementioned general encryption function handling lists of blocks. As the general encryption and decryption functions, the general MAC function takes as input a block function and a feedback function. These functions are passed on to the general encryption function. Hence, we can specify the MAC functions for the particular modes in terms of the general MAC function applied to corresponding block and feedback functions.

In Section 6.8.3.5, we prove a (general) theorem stating that the general MAC function propagates changes of the blocks of the input messages under certain assumptions about the input block and feedback functions. Then, in Section 6.10, we prove theorems stating that the same property holds for the non-general MAC functions as corollaries of the general theorem which is possible since we only consider MAC functions built on top of modes of operation for which the certain assumptions hold true for

[2] A MAC (message authentication code) scheme consists of a MAC function that, when applied to a message of blocks and a key, outputs a block called a MAC value. This MAC value can be used for integrity and authenticity.

[3] Note, however, that if more than one block is changed, this strategy of checking integrity of the message is not perfectly secure: in this case, the MAC value is changed more than once, and we can end up with the original MAC value again.

their block and feedback functions (the ones used for encryption).

We follow this procedure for all MAC functions we consider. In this way, we can easily extend our results to new MAC functions: we just specify the corresponding block and feedback functions and specify the MAC function in terms of the general MAC function applied to these block and feedback functions after which we can prove a theorem stating that the MAC function propagates changes in the blocks of the messages as a corollary of the corresponding general theorem. In this chapter, this procedure is followed for all the MAC functions—both the already defined CBC- and CFB-MAC functions and the newly defined IFB-MAC function. These functions are defined in Section 6.4.

6.2 Technical Details

The code in this chapter is contained in the `Ole-Dalgaard-Lauridsen_code/Mo0` directory of the archive file at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_code.tar.gz`. This directory contains several files:

- The `preamble.v`, `blocks.v`, `types.v`, `predicates.v`, and "`XOR-blocks-n function.v`" files contain the preliminary code.
- The `general.v` file considers the general encryption and MAC schemes.
- The "`CBC mode.v`", "`CFB mode.v`", "`IFB mode.v`", "`f mode.v`", "`CTR mode.v`", and "`BENC mode.v`" files consider each of the modes of operation (of corresponding names).

6.3 Terminology

In this section, we introduce some of the terminology we use in this chapter.

We say that a mode is valid if, for any plaintext, first encrypting the plaintext and then decrypting the resulting ciphertext with the same key and IV using the mode results in a plaintext equal (or bisimilar) to the original plaintext. To prove that the modes that use both the encryption and decryption functions of the underlying block encryption scheme are valid, we assume that the underlying block encryption scheme is valid, i.e., the abovementioned property holds for the underlying scheme, too.

When proving the validity theorems in Section 6.10, we assume that the block functions of the particular modes are inverse and that the corresponding feedback functions output equal values. In this context, we say that two block functions, i.e., the block function for encryption and the block function for decryption, are inverse if first applying the block function for encryption and then the block function for decryption to any plaintext block yields the plaintext block. Here, the same key and the same feedback block are given as inputs to the functions, the underlying block encryption function is given as input to the block function for encryption, and the corresponding, underlying block decryption function is given as input to the block function for decryption. However, if the mode of operation in question does not use the underlying block decryption function, the underlying block encryption function is input to both block functions.

We say that two feedback functions—again one for encryption and one for decryption—output equal values if the output of the function for encryption is equal to the output of the function for decryption where the function for encryption is applied to the plaintext block directly and the function for decryption is applied to the ciphertext block resulting from applying the corresponding block function for encryption to the plaintext block. All keys and IVs are the same, and the same underlying block encryption function is used for the block and feedback functions for encryption while the same block encryption function or the corresponding block decryption function is used for the feedback function for decryption.

When proving the propagation theorems in Section 6.10, we assume often assume that the block and feedback functions for encryption are invertible with regards to the input plaintext or feedback blocks. Regularly, we prove this property as a consequence of the underlying block encryption function being invertible with regards to the plaintext block. However, in these cases, with this latter block, we do not refer to the plaintext block input to the block and feedback functions, but the plaintext block as seen in the context of the block encryption function, i.e., the block encryption function is invertible with regards to the block it is set to encrypt.

When proving the plaintext change non-propagation theorems in Section 6.10, we assume that the feedback function for encryption ignores or disregards the input plaintext block. In other words, the output of the function is independent of the input plaintext block.

6.4 Definitions and Theorems

In this section, we define the modes of operation that we consider in this chapter and discuss the theorems we prove for each mode. Each of the following sections treat in order

- the Cipher Block Chaining (CBC) mode and the CBC-MAC function,
- the Cipher Feedback (CFB) mode and the CFB-MAC function,
- the Input Feedback (IFB) mode and the IFB-MAC function,
- and, finally, the f mode and its special cases: the Counter (CTR) mode and the `block_enc` (BENC) mode.

In the following, all plaintext, ciphertext, and message blocks have lengths equal to n.

Throughout this chapter, e_K and d_K denote encryption respectively decryption under the key K using an underlying block encryption respectively decryption function. Likewise, x_i and y_i denote the i^{th} plaintext respectively ciphertext block.

We say that a symmetric-key encryption scheme (with encryption and decryption functions using key K denoted by enc_K and dec_K) is valid if and only if $dec_K(enc_K(x)) = x \forall x \in \mathcal{P} \forall K \in \mathcal{K}$—i.e., any plaintext first encrypted and then decrypted under the same key remains the same.

6.4.1 The Cipher Block Chaining Mode

The first mode of operation we look at is the Cipher Block Chaining (CBC) mode. When encrypting using this mode, each plaintext block is encrypted by first bitwise XOR'ing it with the IV or feedback block and then encrypting the result with the underlying encryption function. When decrypting, each ciphertext block is decrypted by first decrypting it with the underlying decryption function and then XOR'ing the result with the IV or feedback block. In the CBC mode, encryption is sequential as the computation of the ciphertext block corresponding to any plaintext block after the first one is dependent on the ciphertext block corresponding to the prior plaintext block.

Figures 6.1 and 6.2 illustrate how the CBC mode works. The feedback from block $i-1$ to block i is the ciphertext block y_{i-1} so $y_i = e_K(x_i \oplus y_{i-1})$ for $i \geq 1$ where $y_0 := IV$. This fact ensures that if some plaintext block x_i is changed, then the rest of the ciphertext blocks are changed, too. Here, we use the fact that the underlying block encryption function is invertible with regards to its input plaintext block.[4] This propagation property of CBC encryption is proven in Section 6.10.1.4. Also, the CBC encryption function propagates IV changes, i.e., if the IV is changed and all other inputs remain the same, then

[4] Any proper encryption function is invertible with regards to its input plaintext: if any ciphertext were the encryption of two or more different plaintexts, the decryption function would not be able to reliably decrypt the ciphertext.

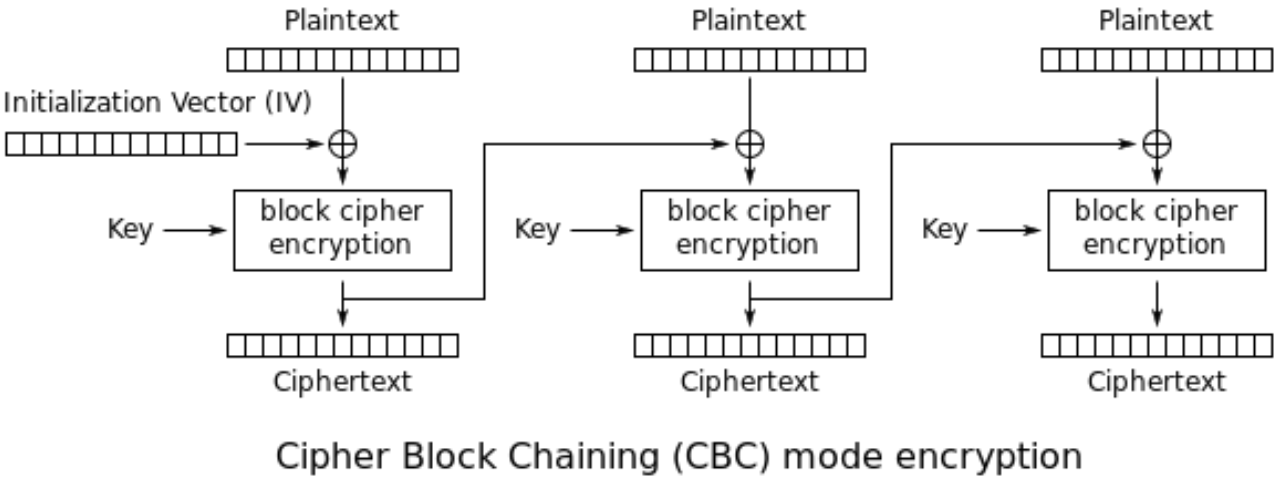

Figure 6.1: CBC encryption. Source: Wikimedia Commons.

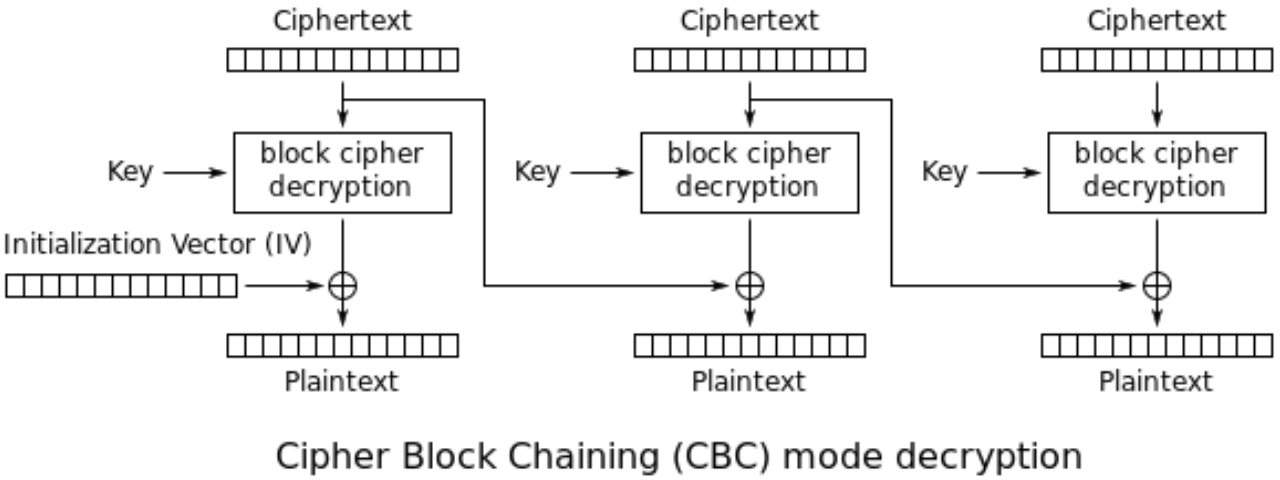

Figure 6.2: CBC decryption. Source: Wikimedia Commons.

all output ciphertext blocks are changed, if the underlying block encryption function is invertible. This property follows from the equation above and is proven in Section 6.10.1.5.

Decryption works analogously: here, the feedback block again equals the ciphertext block (now, however, this ciphertext block is already given) so the decryption of the i^{th} ciphertext block y_i is $d_K(y_i) \oplus y_{i-1} \forall i \geq 1$ where, again, $y_0 := \text{IV}$.

In Section 6.10.1.3, we prove that if the underlying block encryption scheme is valid, then so is the CBC mode of operation. The correctness of this theorem follows from the fact that $d_K(y_i) \oplus y_{i-1} = d_K(e_K(x_i \oplus y_{i-1})) \oplus y_{i-1} = (x_i \oplus y_{i-1}) \oplus y_{i-1} = x_i \forall i \geq 1$ where the second equality follows from the assumed validity of the underlying scheme.[5]

The CBC mode of operation can be used to construct a MAC scheme called CBC-MAC. In this scheme, the MAC value of a message consisting of blocks of bits is defined to be equal to the last block in the encryption of the message using the CBC encryption function under the given key. The scheme

[5]Since both x_i and y_{i-1} are blocks of length n, their bitwise XOR $x_i \oplus y_{i-1}$ is a block of length n, too. Hence, the XOR is in the plaintext space.

uses the all-zero block of length n as the IV. Figure 6.3 illustrates this scheme. There, m_n denotes the n^{th} block of the message input to the scheme.

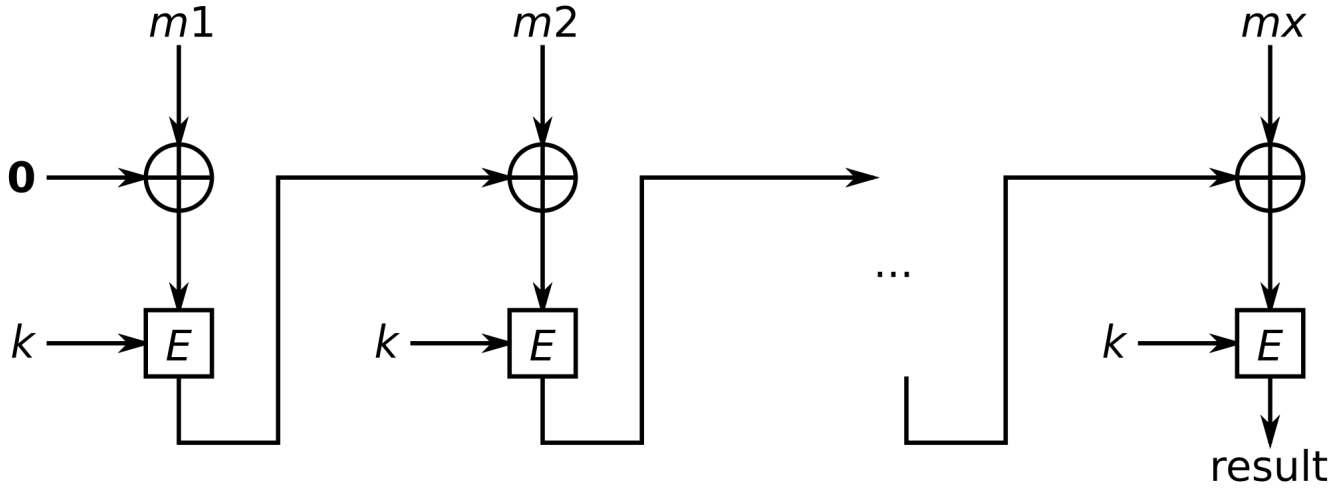

Figure 6.3: CBC-MAC. Source: Wikimedia Commons.

Because of the plaintext change propagation property mentioned above of the CBC encryption function, the CBC-MAC scheme has the desirable property that if a block of a message is changed, then the corresponding MAC value is changed, too: a change of any block in the message is propagated to all ciphertext blocks from that point forward resulting in the last ciphertext block and, in extension, the MAC value changing. We prove this property in Section 6.10.1.6.

6.4.2 The Cipher Feedback Mode

The Cipher Feedback (CFB) mode differs from the above CBC mode of operation in that it does not use any underlying decryption function. When encrypting using this mode, we for each plaintext block first encrypt the IV or feedback block using the underlying block encryption function under the given key and then bitwise XOR the result with the plaintext block. The corresponding ciphertext block is then defined to be this XOR. Decryption for this mode resembles encryption: the plaintext block is just replaced by the ciphertext block—we still use the underlying block *encryption* function to encrypt the IV or feedback block. As for the CBC mode, the feedback block equals the ciphertext block from the last block (both when encrypting and decrypting).

The encryption and decryption functionalities of the CFB mode are shown in Figures 6.4 and 6.5. If we define $y_0 := \text{IV}$, then $y_i = x_i \oplus e_K(y_{y-1}) \forall i \geq 1$. As for the CBC mode above, the encryption of each plaintext block cannot be parallelised because to compute any ciphertext block except for the first one we need the prior ciphertext block—any change in a plaintext block affects the ciphertext blocks from that point forward (if the underlying block encryption function is invertible regarding the in-the-context-of-that-function plaintext block). This propagation property is proven in Section 6.10.2.4. If the underlying block encryption function is invertible, the CFB encryption function, in addition, propagates IV changes, i.e., if the IV is changed and all other inputs remain the same, then the output ciphertext blocks are all changed. This property follows from the equation above and is proven in Section 6.10.2.5.

In Section 6.10.2.3, we prove that the CFB mode of operation is valid. Unlike the above CBC mode, we do not require any (valid) underlying block encryption scheme since we only use an encryption function (a corresponding decryption function does not even have to be specified). However, we still require the encryption function to be deterministic. Both encryption and decryption of x_i respectively y_i (for any $i \geq 1$) are done by XOR'ing it bitwise with $e_K(y_{i-1})$. Since bitwise XOR'ing a block twice with two equal blocks cancel out any effect, the validity of the CFB mode follows.

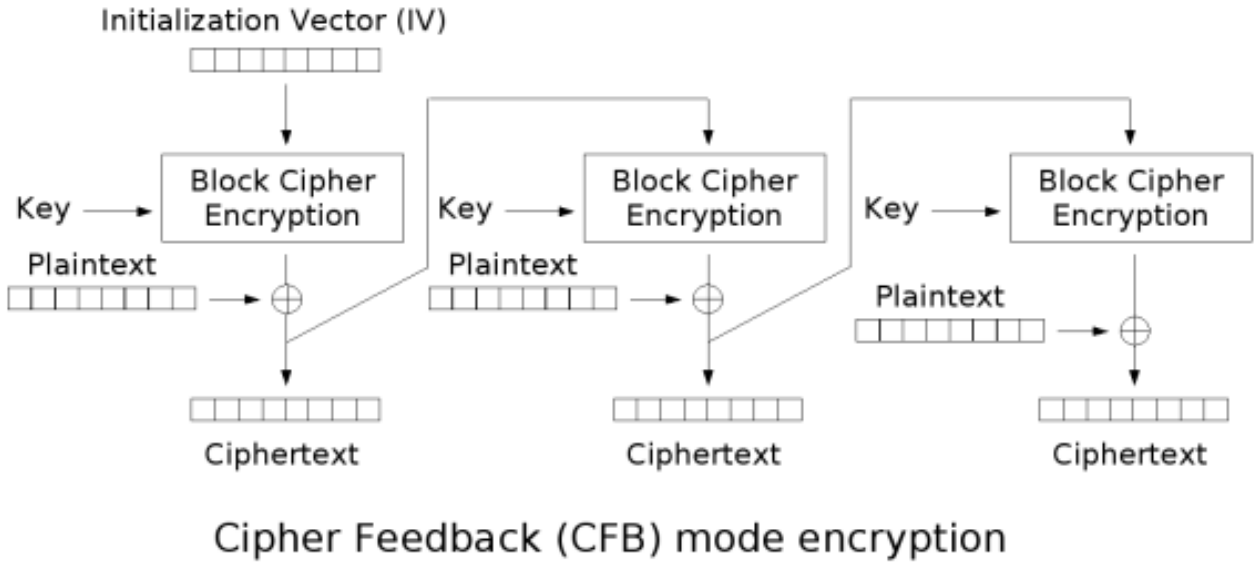

Figure 6.4: CFB encryption. Source: Wikimedia Commons.

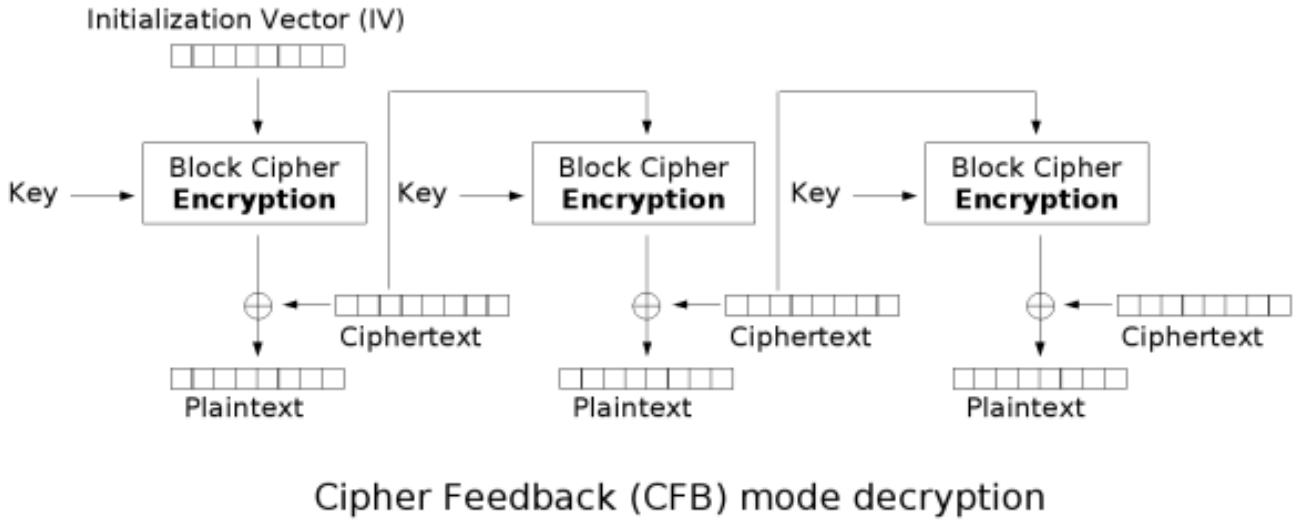

Figure 6.5: CFB decryption. Source: Wikimedia Commons.

Like the CBC mode in the above section, the CFB mode can be used to construct a MAC scheme. This scheme is called the CFB-MAC scheme. The MAC value of a message of blocks of bits is defined to be the last block of the ciphertext resulting from encrypting the message under the given key using the CFB encryption function. Here, the IV is set to the all-zero block of length n.

Since the CFB encryption function propagates changes of any plaintext block to all ciphertext blocks from that point forward, the CFB-MAC function has the property that a change of any block of a message results in the corresponding MAC value of that message changing also since the last block of the ciphertext is changed. This property of the CFB-MAC function is proven in Section 6.10.2.6.

6.4.3 The Input Feedback Mode

We define a new mode of operation which we call the Input Feedback (IFB) mode.

This mode, like the CBC mode defined in Section 6.4.1, uses both the encryption and decryption functions of the underlying block encryption scheme. When encrypting using this mode, we bitwise XOR each plaintext block with the IV or feedback block at that point and encrypt the resulting block with the underlying block encryption function under the given key forming the corresponding ciphertext block. When decrypting, each ciphertext block is first decrypted using the underlying block decryption function under the given key: the result of this decryption is then bitwise XOR'ed with the current IV or feedback block giving the corresponding plaintext block.

The feedback blocks are the inputs to the underlying block encryption function in the respective, prior rounds (hence the name of the mode), i.e., the feedback block is the bitwise XOR of the prior plaintext block and the prior IV or feedback block. When decrypting using the IFB mode, the feedback blocks are given as the outputs of the underlying block decryption function in the respective, prior round, i.e., the feedback block is the decryption of the prior ciphertext block.

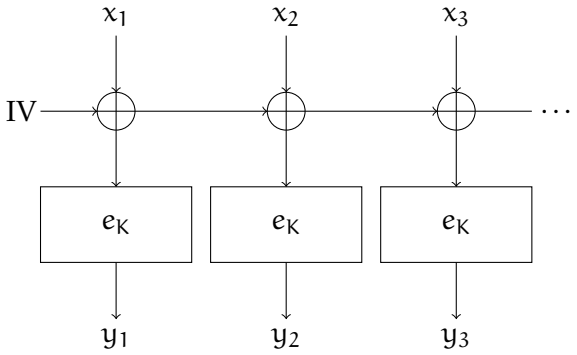

Figure 6.6: IFB encryption.

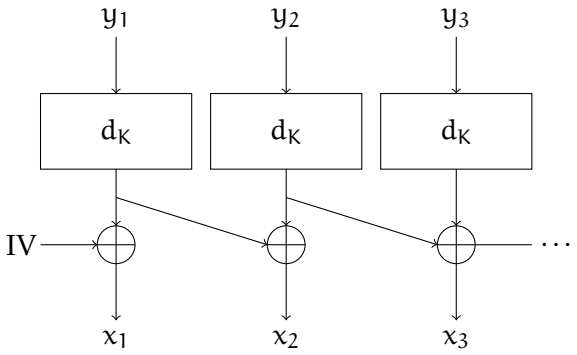

Figure 6.7: IFB decryption.

Figures 6.6 and 6.7 show the IFB encryption and decryption functionalities. If z_j denotes the j^{th} feedback block for any positive natural number j and we define $z_0 = IV$, then $z_i = x_i \oplus z_{i-1} \forall i \geq 1$, i.e., the feedback blocks are computed recursively by bitwise XOR'ing them with the corresponding plaintext blocks. The i^{th} ciphertext block y_i (starting with $i = 1$) is then $e_K(z_i)$. When we decrypt with the IFB mode, we compute the i^{th} plaintext block x_i (again, starting with $i = 1$) as $d_K(y_i) \oplus z_{i-1}$. These z_is are given in a timely fashion: for $i = 1$, $z_{i-1} = z_0 = IV$ is given as input when decrypting; for $i > 1$, z_{i-1} is

given as $d_K(y_{i-1}) = d_K(e_K(z_{i-1})) = z_{i-1}$ when decrypting the prior ciphertext block. The last equality follows from the validity of the underlying block encryption scheme.

In Section 6.10.3.3, we prove that if the underlying block encryption scheme is valid, then the IFB mode is valid, too. We can see this property from the fact that $d_K(y_i) \oplus z_{i-1} = d_K(e_K(z_i)) \oplus z_{i-1} = z_i \oplus z_{i-1} = (x_i \oplus z_{i-1}) \oplus z_{i-1} = x_i \forall i \geq 1$. Here, the second equality follows from the validity of the underlying scheme.

In Section 6.10.3.4, we show that if the underlying block encryption function is invertible with regards to the plaintext blocks input to it,[6] then any change of any plaintext block propagates when encrypting using the IFB mode. This property is seen from the fact that, for any $i \geq 1$, changing the i^{th} plaintext block x_i results in z_j changing for all $j \geq i$: for $j = i$, z_j is changed directly; for $j > i$, z_j is changed as a result of z_{j-1} changing. Now, since $y_i = e_K(z_i) \forall i \geq 1$, all ciphertext blocks y_j with $j \geq i$ are changed, too, because the underlying block encryption function is invertible with regards to the plaintext block (in the context of the function). In other words, the ciphertext block corresponding to the changed plaintext block and all subsequent ciphertext blocks are changed.

In Section 6.10.3.5, we state and prove that if two different IVs are used, then, for the IFB mode, the ciphertexts corresponding to some plaintext are blockwise different. This property follows from much the same reason as the propagation-when-encrypting property discussed above: changing the IV means changing z_0, and changing z_i results in z_{i+1} changing for all $i \geq 0$. As above, changing z_i implies y_i changing for all $i \geq 0$: all ciphertext blocks y_i are changed.

As for the CBC and CFB modes defined in Sections 6.4.1 and 6.4.2, we build a MAC function on top of the IFB mode: the IFB-MAC function illustrated in Figure 6.8. Like for the other MAC schemes, we define the MAC value of a message of blocks of lengths equal to n to be the last block in the ciphertext output by encrypting the message using the IFB mode under the given key. Again, the IV is set to the all-zero block of length n.

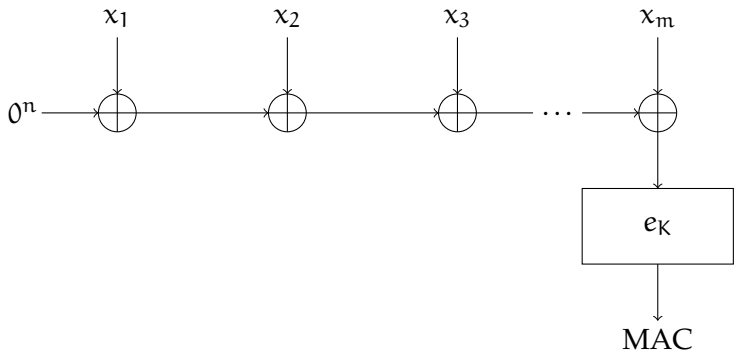

Figure 6.8: The IFB-MAC function.

In Section 6.10.3.6, we prove that if a block in a message is changed, then the corresponding MAC value computed by the IFB-MAC scheme is changed, too. This property follows from the propagation property mentioned above: if a block in the message is changed, then all blocks from that point forwards in the corresponding ciphertext are changed, too: in particular, the last ciphertext block, i.e., the MAC value, is changed also.

[6] If the underlying scheme is valid, then the underlying encryption function must be invertible with regards to the input plaintext blocks. Else, the corresponding decryption function cannot be sure to output the right plaintext block when decrypting a ciphertext block.

6.4.4 The f, Counter, and block_enc Modes

In this section, we define a new mode of operation: the f mode. This mode is a generalisation of the Counter (CTR) mode and the new block_enc (BENC) mode.

As for the CFB mode defined in Section 6.4.2, the f mode only uses the encryption function of the underlying block encryption scheme. When encrypting using this mode, we bitwise XOR each plaintext block with IV or feedback block encrypted using the underlying block encryption function under the given key forming the corresponding ciphertext block. Decrypting with the mode is similar: each ciphertext block is decrypted by bitwise XOR'ing it with the IV or feedback block encrypted using the underlying block encryption function under the given key.

Consecutive feedback blocks are computed by applying a unary function which we call the f function, i.e., if z_i is the i^{th} feedback block, then the $(i+1)^{th}$ feedback block is $z_{i+1} = f(z_i)$. This f function is not a fixed function. However, by fixing it to particular functions, we can define special cases of the f mode: we define the CTR and BENC modes in this section by doing so.

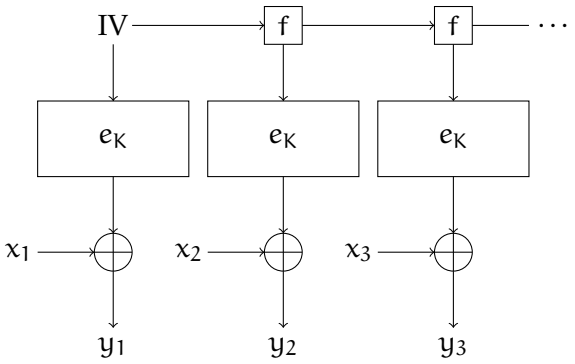

Figure 6.9: f mode encryption.

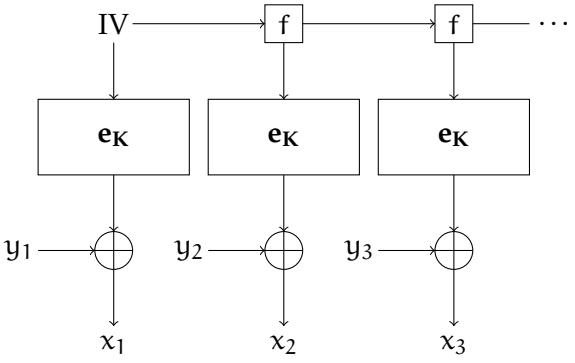

Figure 6.10: f mode decryption.

Figures 6.9 and 6.10 illustrate how the f mode encryption and decryption functions work. As mentioned above, if z_i is the i^{th} feedback block, then $z_1 = IV$ and $z_{i+1} = f(z_i) \forall i \geq 2$. Then, when we encrypt, the i^{th} ciphertext block y_i is computed as $x_i \oplus e_K(z_i)$ where x_i is the i^{th} plaintext block and K is the key. Analogously, when we decrypt, the i^{th} plaintext ${x_i}^{th}$ is computed as $y_i \oplus e_K(z_i)$ where, again, K is the key.

Since bitwise XOR'ing some block twice with the same block results in the first block, the f mode is valid for any underlying block encryption function and any f function: when encrypting, x_i is mapped

to $x_i \oplus e_K(z_i)$ which is mapped to $(x_i \oplus e_K(z_i)) \oplus e_K(z_i) = x_i$ when decrypting—the plaintext block x_i is output. In Section 6.10.4.4, we prove that the f mode and the CTR and BENC modes defined below are valid.

In Section 6.10.4.5, we prove that the encryption functions of the modes do not propagate plaintext changes for any underlying block encryption function and, for the f mode, any f function. This property follows immediately from the fact that the feedback blocks are independent of the plaintext blocks: any feedback block is computed by applying the f function to the prior feedback block (or the IV).

However, as for the other modes of operation we consider, the encryption functions of the f, CTR, and BENC modes propagate IV changes if the underlying block encryption function is invertible regarding the in-the-context-of-that-function plaintext block and, for the f mode, the f function is invertible. For the f mode, if the f function is invertible, then changing the IV results in every feedback block changing, too. Then, since every ciphertext block is computed as the bitwise XOR of the corresponding plaintext block and the block encryption of the IV or feedback block and the block encryption function is invertible, every ciphertext block is changed.

The CTR mode is a special case of the f mode where the f function is the unary counter-n function specified in Section 6.6.2. When the counter-n function is applied to a block representing the natural number b in binary notation, it outputs a block representing $b + 1 \bmod n$ (also in binary representation) where n is the length of the blocks.

In Section 6.10.4.4, we prove that, for any underlying block encryption function, the CTR mode is valid and the CTR encryption function does not propagate plaintext changes. In addition, we prove that if the block encryption function is invertible, the CTR encryption function propagates IV changes. These properties follow from the facts that the CTR mode is a special case of the f mode and that the counter-n function is invertible.

We define a new mode, the BENC mode, which also is a special case of the f mode. For this mode, the underlying block encryption function is used for the f function. However, it cannot be used as the f function directly since the block encryption function takes two inputs: a plaintext block and a key. In this regard, we use the function specified in Section 6.6.3: the make-`block_enc`-unary function. This function takes as inputs a block encryption function and a key and outputs a function that encrypts its inputs using the block encryption function under the key. This output function is the f function for the BENC mode. Hence, the f function is not fixed for this mode: it depends on the given key.

In Section 6.10.4.4, we prove that, for any underlying block encryption function, the BENC mode is valid and the encryption function of the mode does not propagate plaintext changes. Also, we show that if the block encryption function is invertible, then the BENC encryption function propagates IV changes. Similarly to the CTR mode described above, these properties about the BENC mode follow from the facts that the BENC mode is a special case of the f mode and that the function output by the make-`block_enc`-unary function is invertible if the input block encryption function is invertible.

6.5 Types

A number of types are used for modelling and proving theorems about the modes of operation and the associated MAC functions. First, in Section 6.5.1, we describe the basic types used to represent blocks. In Section 6.5.2, the types of the auxiliary functions and the encryption, decryption, and MAC functions for the modes of operation and the general schemes are defined and explained.

6.5.1 Basic Types

The modes of operation and MAC functions (and any underlying encryption schemes or functions) operate on blocks of bits. The number of blocks could in principle be infinite.

We represent each bit of the blocks with a `bool` which is Coq's version of a Boolean value: the Boolean value `false` represents the zero-bit while the Boolean value `true` represents the one-bit. In this way, we

can do all the regular operations on bits by doing the corresponding operations on `bools`. For instance, XOR'ing two bits b_1 and b_2 can be done by applying the function `xorb` on the two `bools` representing these bits: the result is the `bool` representing $b_1 \oplus b_2$. We could have chosen the `nat` type to represent blocks of bits. This choice would make sense since the blocks can be interpreted as natural numbers in binary notation. While this choice would also have made the specification of the counter function in Section 6.6.2 simpler, it would have made it more cumbersome when doing the computations defined by the modes of operations, e.g., XOR'ing two blocks bitwise could not just be done by adding the corresponding natural numbers.[7]

The plaintext, ciphertext, message, and feedback blocks, the IVs, and the MAC values are blocks of length n while the keys are blocks of length k: we define two parameters `n` and `k`.

```
Parameter n : nat.

Parameter k : nat.
```

Using these parameters, we define two types: one type representing blocks of length n and another type representing blocks of length k. First, however, we define the `block_variable_length` function below.

```
Definition block_variable_length (m : nat) : Type := t bool m.
```

In this definition, we use the `t` function in the `VectorDef` standard library. This function is shown below.

```
Inductive t A : nat → Type :=
  | nil : t A 0
  | cons : forall (h : A) (n : nat), t A n → t A (S n).
```

The `t` function takes as input a type and a natural number and outputs a type representing vectors whose elements are of the input type and whose size is equal to the natural number. By passing on `bool` and `m` to the `t` function, we get a type representing vectors of Boolean values and of length m, i.e., blocks of length `m`. In this way, we can use the type output by the `block_variable_length` function to represent blocks of length m if the function is applied to `m`.

The function and parameters above allows us to define types representing blocks of lengths n and k.

```
Definition block_n : Type := block_variable_length n.

Definition block_k : Type := block_variable_length k.
```

While the block lengths are finite, we consider two cases for the number of blocks: either we have an infinite number of blocks of length n or just a finite number. In the first case, the `Stream` type fits: we represent the plaintexts and ciphertexts by the `Stream block_n` type. In the latter case, we use the `list` type. Here, we represent the plaintexts, ciphertexts, and messages by the `list block_n` type.

6.5.2 Function Types

When specifying and proving theorems about the modes of operation and the associated MAC functions, we use several functions. Several of the types of these functions are rather involved, and, hence, we define wrapper types for brevity and clarity: we go through these wrapper types in this section. When defining these types, we use the basic types defined in the section above.

Each mode of operation uses an underlying block encryption scheme: some of them use a decryption function while they all use an encryption function. Common to these functions is that they take as input two blocks (the plaintext, ciphertext, or message block and the key) of lengths n and k and output a block (the ciphertext or plaintext block) of length n. Therefore, the type of the these functions are as defined below. below: `block_enc_dec_Type`.

[7] A contradictory example is XOR'ing the blocks b_1 and b_2 where $b_1 = b_2 =$ `01`. Here, the result is `00`, but adding them would give `10` since `01` respectively `10` are the binary representations of 1 respectively 2.

```
Definition block_enc_dec_Type : Type :=
  block_n → block_k → block_n.
```

We specify the functions of each mode in terms of corresponding general functions, i.e., we specify the block and feedback functions for each mode such that we can specify the encryption and decryption functions of the mode in terms of general encryption or decryption functions applied to block and feedback functions. Likewise, we specify the MAC functions in terms of the block and feedback functions of the particular modes and a general MAC function. Both the block functions and the feedback functions take as input a plaintext, ciphertext or message block, a key, an IV or feedback block, and an underlying block encryption or decryption function. The block functions output a ciphertext or plaintext block while the feedback functions output a feedback block. Hence, the types of these functions are rather verbose. These identical types are defined below: the `block_fun_Type` type for the block functions, and the `feedback_fun_Type` type for the feedback functions.

```
Definition block_fun_Type : Type :=
  block_n → block_k → block_n → block_enc_dec_Type → block_n.

Definition feedback_fun_Type : Type :=
  block_n → block_k → block_n → block_enc_dec_Type → block_n.
```

The stream versions of the encryption and decryption functions of each mode of operation take as input a plaintext or ciphertext stream, a key, an IV, and an underlying block encryption or decryption function and output a ciphertext or plaintext stream. Therefore, for all the modes, the type of the encryption and decryption functions using streams is equal to the `MoO_enc_dec_Type_stream_version` type defined below.

```
Definition MoO_enc_dec_Type_stream_version : Type :=
  Stream block_n →
  block_k →
  block_n →
  block_enc_dec_Type →
  Stream block_n.
```

The list versions of the encryption and decryption functions of the modes are similar. They take the same inputs except that the plaintext or ciphertext streams are replaced by plaintext, ciphertext, or message lists. Additionally, the output is a plaintext or ciphertext list, not a plaintext or ciphertext stream. Thus, we arrive at the `MoO_enc_dec_Type_list_version` type defined below for these functions.

```
Definition MoO_enc_dec_Type_list_version : Type :=
  list block_n →
  block_k →
  block_n →
  block_enc_dec_Type →
  list block_n.
```

The general encryption and decryption functions that we use to define the functions of each specific mode of operation take the same inputs as above in addition to a block function and a feedback function that both define the inner operations of the mode. The output is still of the same type. Thus, we end up with the types defined below for the stream and list versions of these general functions.

```
Definition general_MoO_enc_dec_Type_stream_version : Type :=
  Stream block_n →
  block_k →
  block_n →
  block_enc_dec_Type →
  block_fun_Type →
  feedback_fun_Type →
  Stream block_n.
```

```
Definition general_MoO_enc_dec_Type_list_version : Type :=
  list block_n →
  block_k →
  block_n →
  block_enc_dec_Type →
  block_fun_Type →
  feedback_fun_Type →
  list block_n.
```

In Section 6.10, we specify MAC functions for each of the modes that propagate plaintext changes when encrypting. These functions take as input a finite message of blocks, a key, and an underlying block encryption function while they output a MAC value. Hence, we end up with the `MAC_fun_Type` type defined below for the non-general MAC functions.

```
Definition MAC_fun_Type : Type :=
  list block_n →
  block_k →
  block_enc_dec_Type →
  block_n.
```

In Section 6.8.2, we specify the general MAC function. This function takes the same inputs as the specific MAC functions and, in addition, a block function and a feedback function. The output remains of the same type. Thus, the general MAC function is of the `general_MAC_fun_Type` type defined below.

```
Definition general_MAC_fun_Type : Type :=
  list block_n →
  block_k →
  block_enc_dec_Type →
  block_fun_Type →
  feedback_fun_Type →
  block_n.
```

6.6 Auxiliary Functions

In this section, we specify, prove lemmas about, and implement three auxiliary functions that we use when specifying the block and feedback functions of the modes of operation.

The following three subsections handle, in order, the XOR-blocks-`n`, the counter-`n`, and the make-`block_enc`-unary functions.

6.6.1 The XOR-Blocks-`n` Function

In this section, we consider the XOR-blocks-`n` function. This function bitwise XOR's two blocks of length `n`. All the modes of operation defined in Section 6.4 use the function.

We specify the XOR-blocks-`n` function in terms of the XOR-blocks-variable-length function that can bitwise XOR two blocks of any equal lengths. This second function is specified below.

```
Definition specification_of_XOR_blocks_variable_length_function
           (XOR_blocks_var_length_fun :
              forall m : nat,
                block_variable_length m → block_variable_length m →
                block_variable_length m) :=
  (forall xs_1 xs_2 : block_variable_length 0,
     XOR_blocks_var_length_fun 0 xs_1 xs_2 = nil bool)
  ∧
  (forall (m' : nat) (x_1 x_2 : bool) (xs'_1 xs'_2 : block_variable_length m'),
```

```
    XOR_blocks_var_length_fun
      (S m') (cons_block x_1 xs'_1) (cons_block x_2 xs'_2) =
    cons_block (xorb x_1 x_2) (XOR_blocks_var_length_fun m' xs'_1 xs'_2)).
```

Recall from Section 6.5.1 that the `block_variable_length` function on input any natural number `m` outputs a type representing blocks of length `m`. The specification above states that the XOR-blocks-variable-length function outputs the empty vector if two empty vectors are input while it recursively XOR's the bits of the blocks if they are non-empty.

Above, the natural number `m` input to the XOR-blocks function could instead be implicit. In this way, we could apply the function without providing the length of the inputs blocks, and, particularly, we could apply it directly to blocks of length `n`. However, then the specification of the function would be more cumbersome.

Now, we specify the XOR-blocks-`n` function in terms of the above specification.

```
Definition specification_of_XOR_blocks_n_function
           (XOR_blocks_n_fun : block_n → block_n → block_n) :=
  forall XOR_blocks_var_length_fun :
  forall m : nat,
    block_variable_length m → block_variable_length m →
    block_variable_length m,
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun →
    forall xs_1 xs_2 : block_n,
      XOR_blocks_n_fun xs_1 xs_2 = XOR_blocks_var_length_fun n xs_1 xs_2.
```

The output of the XOR-blocks-`n` function is equal to the output of the XOR-blocks-variable-length function when applied to two blocks of length `n`.

We prove a number of properties about the XOR-blocks-`n` function. Each of these properties of XOR-blocks-`n` function are proven by first proving the property for the XOR-blocks-variable-length function and then proving it for the XOR-blocks-`n` function as a consequence.

We prove that the XOR-blocks-`n` function outputs the first input if applied twice to the same second input, i.e., for all blocks x_1 and x_2 of length `n`, $(x_1 \oplus x_2) \oplus x_2 = x_1$. This property is used when proving block functions to be inverse. First, we prove the property for the XOR-blocks-variable-length function (for any block length).

```
Lemma XOR_blocks_var_length_fun_applied_twice_to_same_second_input_block_yields_first_input_block :
  forall XOR_blocks_var_length_fun : forall m : nat,
    block_variable_length m → block_variable_length m → block_variable_length m,
    specification_of_XOR_blocks_variable_length_function XOR_blocks_var_length_fun →
    forall (l : nat) (xs_1 xs_2 : block_variable_length l),
      XOR_blocks_var_length_fun l (XOR_blocks_var_length_fun l xs_1 xs_2) xs_2 = xs_1.
```

The lemma above is proven by induction in the length `m` of the input blocks and by exploiting that the property holds for the `xorb` operator. As a corollary of the above lemma, we prove that the XOR-blocks-`n` function also has this property.

```
Lemma XOR_blocks_n_fun_applied_twice_to_same_second_input_yields_first_input :
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall xs_1 xs_2 : block_n,
      XOR_blocks_n_fun (XOR_blocks_n_fun xs_1 xs_2) xs_2 = xs_1.
```

The proof of this latter lemma consists of first unfolding the XOR-blocks-`n` function using the implementation below, `XOR_blocks_var_length_fun_v0`, of the XOR-blocks-variable-length function and then applying the first lemma.

Likewise, we prove that the XOR-blocks-`n` function is invertible with regards to both input block. First, we prove that the function is invertible with regards to the first input block as a consequence of the

XOR-blocks-variable-length function having this property which is proven below.

```
Lemma XOR_blocks_var_length_fun_invertible_wrt_first_input :
  forall XOR_blocks_var_length_fun : forall m : nat,
    block_variable_length m → block_variable_length m →
    block_variable_length m,
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun →
    forall (l : nat) (xs_1 xs_2 ys : block_variable_length l),
      XOR_blocks_var_length_fun l xs_1 ys =
      XOR_blocks_var_length_fun l xs_2 ys →
      xs_1 = xs_2.
```

The lemma above is proven by induction in the length `l` of the input blocks and exploiting that the `xorb` operator is invertible with regards to the second operand via the `xorb_invertible_wrt_first_input` lemma in Section 3.2.2. We prove the property for the XOR-blocks-n function.

```
Lemma XOR_blocks_n_fun_invertible_wrt_first_input :
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall xs_1 xs_2 ys : block_n,
      XOR_blocks_n_fun xs_1 ys = XOR_blocks_n_fun xs_2 ys →
      xs_1 = xs_2.
```

This lemma is proven by, again, unfolding the XOR-blocks-n function with the `XOR_blocks_var_length_fun_v0` function and applying the `XOR_blocks_var_length_fun_invertible_wrt_first_input` lemma above.

Secondly, we prove that the XOR-blocks-n function is invertible with regards to the second input block as a consequence of the XOR-blocks-variable-length function being so. The lemma below states this latter property.

```
Lemma XOR_blocks_var_length_fun_invertible_wrt_second_input :
  forall XOR_blocks_var_length_fun : forall m : nat,
    block_variable_length m → block_variable_length m →
    block_variable_length m,
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun →
    forall (l : nat) (xs ys_1 ys_2 : block_variable_length l),
      XOR_blocks_var_length_fun l xs ys_1 =
      XOR_blocks_var_length_fun l xs ys_2 →
      ys_1 = ys_2.
```

This lemma itself is proven by using the lemma above saying that the XOR-blocks-variable-length function is invertible with regards to the first input block and the fact that the function is commutative. This latter fact is stated in the lemma below.

```
Lemma XOR_blocks_var_length_fun_commutative :
  forall XOR_blocks_var_length_fun : forall m : nat,
    block_variable_length m → block_variable_length m →
    block_variable_length m,
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun →
    forall (l : nat) (xs_1 xs_2 : block_variable_length l),
      XOR_blocks_var_length_fun l xs_1 xs_2 =
      XOR_blocks_var_length_fun l xs_2 xs_1.
```

This lemma is proven by induction in the length `l` of the input blocks and using the `xorb_comm` lemma (in Section 3.2.2) saying that the `xorb` operator is commutative.

We can now prove that the XOR-blocks-n function is invertible, too, with regards to the second input.

```
Lemma XOR_blocks_n_fun_invertible_wrt_second_input :
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall xs ys_1 ys_2 : block_n,
      XOR_blocks_n_fun xs ys_1 = XOR_blocks_n_fun xs ys_2 →
      ys_1 = ys_2.
```

As before, we prove this lemma by unfolding the XOR-blocks-n function with the `XOR_blocks_var_length_fun_v0` function and applying the corresponding lemma above for the XOR-blocks-variable-length function.

We prove that there are only one XOR-blocks-variable-length function in the sense that if any two (or more, for that matter) functions satisfying the specification of the function are applied to the same inputs in the domain, then the outputs are equal. We use this property when proving that the `XOR_blocks_n_fun_v0` function defined below fits the specification of the XOR-blocks-n function.

```
Lemma there_is_only_one_XOR_blocks_variable_length_function :
  forall XOR_blocks_var_length_fun_1 XOR_blocks_var_length_fun_2 :
  forall m : nat,
    block_variable_length m → block_variable_length m →
    block_variable_length m,
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun_1 →
    specification_of_XOR_blocks_variable_length_function
      XOR_blocks_var_length_fun_2 →
    forall (l : nat) (xs_1 xs_2 : block_variable_length l),
      XOR_blocks_var_length_fun_1 l xs_1 xs_2 =
      XOR_blocks_var_length_fun_2 l xs_1 xs_2.
```

The lemma above is proven by induction in the length `l` of the input blocks.

Likewise, we prove that there is only one XOR-blocks-n function. This property about the function is used when proving that the implementations of block and feedback functions using the XOR-blocks-n function satisfy the corresponding specifications.

```
Lemma there_is_only_one_XOR_blocks_n_function :
  forall XOR_blocks_n_fun_1 XOR_blocks_n_fun_2 :
          block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun_1 →
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun_2 →
    forall xs_1 xs_2 : block_n,
      XOR_blocks_n_fun_1 xs_1 xs_2 = XOR_blocks_n_fun_2 xs_1 xs_2.
```

The lemma above is proven by just unfolding the XOR-blocks-n function.

We implement the XOR-blocks-n function; we implement block and feedback functions using the XOR-blocks-n function in terms of this implementation. First, however, we implement the XOR-blocks-variable-length function.

```
Fixpoint XOR_blocks_var_length_fun_v0
       (m : nat) (xs_1 xs_2 : block_variable_length m) :
  block_variable_length m :=
  let fix visit {l : nat} :
       block_variable_length l → block_variable_length l →
       block_variable_length l :=
     match l with
       | 0 ⇒ fun ys_1 ys_2 : block_variable_length 0 ⇒ nil bool
       | S l' ⇒ fun ys_1 ys_2 : block_variable_length (S l') ⇒
                   cons_block
                     (xorb (hd ys_1) (hd ys_2))
                     (visit (tl ys_1) (tl ys_2))
     end
```

```
  in
  visit xs_1 xs_2.
```

The function above uses an internal, recursive function `visit` that outputs a function that itself outputs the bitwise XOR of the blocks (of equal lengths) input to it. We structure the function in this style since Coq does not accept the more direct style where it has problems inferring that the (dependent) types of the blocks input to functions in the expressions on the right sides in the matching are correct—Coq cannot tell that the lengths of the blocks are correct. The function above is proved to fit the corresponding specification with the help of two unfolding lemmas for the function.

With the XOR-blocks-variable-length function implemented, the implementation below of the XOR-blocks-n function is straightforward.

```
Definition XOR_blocks_n_fun_v0 (xs ys : block_n) :
  block_n :=
  XOR_blocks_var_length_fun_v0 n xs ys.
```

Applying this function to two blocks of length `n` amounts to applying the implementation of the XOR-blocks-variable-length function and `n`. It is proved to satisfy the corresponding specification using an unfolding lemma for the function and the `there_is_only_one_XOR_blocks_variable_length_function` lemma above.

6.6.2 The Counter-n Function

The CTR mode defined in Section 6.4.4 uses a counter function that on input a block representing the natural number `b` outputs a block representing $b + 1 \bmod 2^m$. Here, `m` is the length of both the input and the output block.

We use the little-endian binary representation where the first bit (the head) of a block is the least significant. For instance, the block `0101` represents the natural number 5 in the binary representation when $m = 4$. Thus, 5 is represented by the block
`cons_block true (cons_block false (cons_block true (cons_block false (nil bool))))`—the block is reversed, and the bits are converted to `bool`s in a block.

Similarly to the section above where we specify the XOR-blocks-n function, we, in this section, specify first a counter function, the counter-variable-length function, that takes as input a natural number `m` and a block of length `m`. Then we specify another counter function, the counter-n function, that take as input a block of length `n`. This latter function is the one we use in the CTR mode.

Using the little-endian binary representation described above, we specify the counter-variable-length function below.

```
Definition specification_of_counter_variable_length_function
           (counter_var_length_fun :
              forall m : nat,
                block_variable_length m → block_variable_length m) :=
  (forall xs : block_variable_length 0,
    counter_var_length_fun 0 xs = nil bool)
  ∧
  (forall (l' : nat) (xs' : block_variable_length l'),
     counter_var_length_fun (S l') (cons_block false xs') =
     cons_block true xs')
  ∧
  (forall (l' : nat) (xs' : block_variable_length l'),
     counter_var_length_fun (S l') (cons_block true xs') =
     cons_block false (counter_var_length_fun l' xs')).
```

The second clause of this specification tells us what applying the counter-variable-length function does when the head element of the input block is `false`. In this case, the head element represents the zero-bit,

and incrementing the value represented by the block amounts to just changing the head element to `true` representing the one-bit since no carrying takes place.

The third clause handles the case where the head element is `true`. Now, the head element represents the bit 1 in which case incrementing the value represented by the block involves carrying. Therefore, we change the head to `false` and apply the function recursively to the tail: the carry increments the value represented by the tail block.

The first clause treat the case where $m = 0$. This case is only encountered in normal use when the carry goes all the way to the end of the block. As an example, when $m = 4$, it is only encountered when the input block represents 1111. In this case, all the 1s are flipped to 0s in the third clause, and the first clause then tells us to throw away the carry. In effect, we output a block representing `0000` which, in turn, is consistent with the function incrementing the represented number by 1 mod 2^m.

The natural number `m` input to the counter function could instead be implicit. Then, as explained similarly in the section above for the XOR-blocks functions, we would not have to input the length of the input block to the function, and, particularly, we could apply the function directly to a block of length `n`. Again, we do not follow this structure to ease the specifications of the XOR-blocks functions and, especially, the XOR-blocks-n function.

We specify the counter-`n` function in terms of the above specification.

```
Definition specification_of_counter_n_function
           (counter_n_fun : block_n → block_n) :=
  forall counter_var_length_fun :
  forall m, block_variable_length m → block_variable_length m,
    specification_of_counter_variable_length_function
      counter_var_length_fun →
    forall xs : block_n,
      counter_n_fun xs = counter_var_length_fun n xs.
```

The output of the counter-`n` function applied to a block of length `n` is specified to be equal to the output of the counter-variable-length function to `n` and the block.

We prove that the counter-`n` function is invertible which we exploit when showing that the CTR feedback function for encryption is invertible with regards to the IV or feedback block. We show the counter-`n` function is invertible as a consequence of the counter-variable-length function being invertible. The lemma below states this latter property.

```
Lemma counter_variable_length_function_invertible :
  forall counter_var_length_fun :
  forall m : nat, block_variable_length m → block_variable_length m,
    specification_of_counter_variable_length_function
      counter_var_length_fun →
    forall (l : nat) (xs_1 xs_2 : block_variable_length l),
      counter_var_length_fun l xs_1 = counter_var_length_fun l xs_2 →
      xs_1 = xs_2.
```

The lemma is proven by induction in the length `l` of the input blocks and case analysis of the head elements of the blocks since the behaviour of the function depends on the values of these head elements.

We prove that the counter-`n` function is invertible, too, as a corollary of the above lemma.

```
Lemma counter_n_function_invertible :
  forall counter_n_fun : block_n → block_n,
    specification_of_counter_n_function counter_n_fun →
    forall xs_1 xs_2 : block_n,
      counter_n_fun xs_1 = counter_n_fun xs_2 →
      xs_1 = xs_2.
```

In the proof of the `counter_n_function_invertible` lemma above, we unfold the counter-`n` using the `counter_var_length_fun_v0` implementation defined below of the counter-variable-length function after

which we have to show that this specific latter function is invertible: here, we apply the `counter_variable_length_function_invertible` lemma above.

When we prove that the implementation of the counter-`n` function below fits the corresponding specification, we exploit that any two counter-variable-length functions are extensionally equal stated in the lemma below.

```
Lemma there_is_only_one_counter_variable_length_function :
  forall counter_var_length_fun_1 counter_var_length_fun_2 :
  forall m : nat, block_variable_length m → block_variable_length m,
    specification_of_counter_variable_length_function
        counter_var_length_fun_1 →
    specification_of_counter_variable_length_function
      counter_var_length_fun_2 →
    forall (l : nat) (xs : block_variable_length l),
      counter_var_length_fun_1 l xs = counter_var_length_fun_2 l xs.
```

This lemma is proven by induction in the length `l` of the input block and case analysis of the head element of the block—again, because the behaviour of counter-variable-length functions depend on the value of this element.

Similarly, we show that any two counter-`n` functions are extensionally equal.

```
Lemma there_is_only_one_counter_n_function :
  forall counter_n_fun_1 counter_n_fun_2 : block_n → block_n,
    specification_of_counter_n_function counter_n_fun_1 →
    specification_of_counter_n_function counter_n_fun_2 →
    forall xs : block_n,
      counter_n_fun_1 xs = counter_n_fun_2 xs.
```

The proof of the lemma above consists essentially just of unfolding the counter-`n` function with the implementation below of the counter-variable-length function.

When implementing the CTR feedback functions, we need an implementation of the counter-`n` function. However, since this function is specified in terms of the counter-variable-length function, we implement that function first.

```
Fixpoint counter_var_length_fun_v0 (m : nat) (xs : block_variable_length m) :
  block_variable_length m :=
  let fix visit {l : nat} :
        block_variable_length l → block_variable_length l :=
      match l with
        | O ⇒ fun ys : block_variable_length 0 ⇒ nil bool
        | S l' ⇒ fun ys : block_variable_length (S l') ⇒
                    match (hd ys) with
                      | false ⇒ cons_block true (tl ys)
                      | true ⇒ cons_block false (visit (tl ys))
                    end
      end
  in
  visit xs.
```

The implementation above is proven to satisfy the corresponding specification with the aid of three unfolding lemmas: one for the base case of the length of the input block and two for the inductive case—one for each potential value of the input block.

Analogously to when we implement the XOR-blocks-variable-length function in the section above, we implement the counter-variable-length function using an internal, recursive function that outputs a function which is applied to the input block. We use this style since the in the more direct style, Coq has problems inferring that the input block given to the function recursively when the head element is `false` is of the right length.

With the counter-variable-length function handled, we implement the counter-n function.

```
Definition counter_n_fun_v0 (xs : block_n) :
  block_n :=
  counter_var_length_fun_v0 n xs.
```

Applying this function is equivalent to applying the implementation of the counter-variable-length function and n. It is proved to fit the corresponding specification using an unfolding lemma and applying the there_is_only_one_counter_variable_length_function lemma above.

6.6.3 The Make-block_enc-Unary Function

In the BENC mode defined in Section 6.4.4, we use the underlying block encryption function to compute the feedback blocks. The mode is a special case of the f mode described in the same section: we use the block encryption function as the f function. However, a problem arises here: the f function is unary while the encryption function is binary. Therefore, we specify a function, the make-block_enc-unary function, that on input a block encryption function and a key outputs a function that on input a block outputs the encryption of the block using the encryption function under the key: this output function is unary. Also, the domain and codomain of the function coincide with those of the f function: the function takes as input and outputs blocks of length n.

First, we define the make-block_enc-unary function.

```
Definition make_block_enc_unary
           (block_enc : block_enc_dec_Type) (ks : block_k) :
  block_n → block_n :=
  (fun IV :  block_n ⇒ block_enc IV ks).
```

The output function takes as input a block which is encrypted using the provided block encryption function under the given key.

We prove that the function output by make-block_enc-unary function is invertible if the block encryption function input to it is itself invertible with regards to the (in the context of the encryption function) plaintext block.

```
Lemma make_block_enc_unary_preserves_invertibility :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall ks :  block_k,
      f_invertible (make_block_enc_unary block_enc ks).
```

Here, we use the f_invertible predicate defined in Section 6.10.4.1 to state that the function output by the make-block_enc-unary function is invertible. The lemma is proven by using an unfolding lemma for the make-block_enc-unary function and exploiting that the provided encryption function is assumed to be invertible.

6.7 Predicates

In this section, we define a number of predicates and prove lemmas about them. These predicates and lemmas are part of our framework for proving theorems about the general encryption and MAC schemes in Section 6.8 and each mode of operation and non-general MAC function in Section 6.10.

The rest of this section is divided into four subsections. In the first section, we define predicates concerning streams and pairwise equality of or difference between their elements of the block_n type defined in Section 6.5.1. In the second section, we define corresponding predicates concerning lists. In these first two sections, we state and prove lemmas involving the predicates. Finally, in the third and fourth sections, we use the stream and list predicates to define other predicates: we define predicates concerning properties about the block and feedback functions and underlying block encryption schemes

or functions in the third section, and, in the fourth section, we define predicates capturing properties of the functions of the non-general encryption and MAC schemes.

6.7.1 Stream Predicates

In this section, we define predicates pertaining to streams and their elements and prove lemmas about them. The section is divided into subsections—one subsection for each stream predicate.

In the following, all elements of the streams considered are of the `block_n` type.

`streams_equal_at_index` We define the `streams_equal_at_index` predicate which holds for two streams and an index if and only if the streams are equal at that index.

```
Definition streams_equal_at_index (xss_1 xss_2 : Stream block_n) (i : nat) :=
  stream_ref xss_1 i = stream_ref xss_2 i.
```

In this definition, we use the `stream_ref` operator to access the elements of the streams.

We prove that if the tails of two streams are equal at some index, then the streams are equal at the index equal to the successor of the first index.

```
Lemma tail_streams_equal_at_index_implies_Conses_equal_at_subsequent_index :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n) (j' : nat),
    streams_equal_at_index xss'_1 xss'_2 j' →
    streams_equal_at_index (Cons xs_1 xss'_1) (Cons xs_2 xss'_2) (S j').
```

This lemma is proven by unfolding the predicate and the `stream_ref` operator. It is used in the proof of the `streams_different_at_index_only_implies_equal_at_all_other_indices` lemma below.

`streams_different_at_index` Likewise, we define the `streams_different_at_index` predicate that is satisfied by two streams and an index if and only if the streams are different at that index.

```
Definition streams_different_at_index
           (xss_1 xss_2 : Stream block_n) (i : nat) :=
  stream_ref xss_1 i <> stream_ref xss_2 i.
```

As above, we use the `stream_ref` operator to access the elements of the streams.

We show that if two streams are different at a positive index, then their tails are different at the prior index.

```
Lemma streams_different_at_index_implies_tails_different_at_prior_index :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n) (i' : nat),
    streams_different_at_index
      (Cons xs_1 xss'_1)
      (Cons xs_2 xss'_2)
      (S i') →
    streams_different_at_index xss'_1 xss'_2 i'.
```

As the lemma in the section above, this lemma is proven by unfolding the predicate and the `stream_ref` operator.

`streams_different_at_index_only` We define the `streams_different_at_index_only` predicate that holds for two streams and an index if and only if the streams are different at that index only.

```
Definition streams_different_at_index_only
           (xss_1 xss_2 : Stream block_n) (i : nat) :=
  forall j : nat,
    j = i ↔ streams_different_at_index xss_1 xss_2 j.
```

Here, we use the `streams_different_at_index` predicate in the section above.

We prove several lemmas about the `streams_different_at_index_only` predicate. The first of these lemmas state that the predicate is symmetric regards the two input streams.

```
Lemma streams_different_at_index_only_is_a_symmetric_relation :
  forall (xss_1 xss_2 : Stream block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only xss_2 xss_1 i.
```

This lemma is proven by unfolding first the `streams_different_at_index_only` predicate and then the `streams_different_at_index` predicate revealing the inner inequality of the latter predicate. Then, we exploit that the inequality relation is symmetric. The proven lemma is used in the proof of the general plaintext change propagation theorem using streams in Section 6.8.3.2.

We show that if two streams are different at some positive index only, then their tail streams are different at the prior index only.

```
Lemma streams_different_at_index_only_implies_tails_different_at_prior_index_only :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n) (i' : nat),
    streams_different_at_index_only
        (Cons xs_1 xss'_1)
        (Cons xs_2 xss'_2)
        (S i') →
    streams_different_at_index_only xss'_1 xss'_2 i'.
```

This lemma is proven by unfolding the predicates until the applications of the `stream_ref` operator are revealed. After that, we unfold the operator: now, the premise and the conclusion of the lemma are equal except that the left side of the biimplication of the premise is `S j = S i'` and the left side of the biimplication of the conclusion is `j = i'`. We overcome this difference by applying the `eq_S` and `eq_add_S` lemmas from the standard libraries. These lemmas collectively state that two natural numbers are equal if and only if their successors are equal. The proven lemma is used in the proof of the `streams_different_at_index_only_implies_equal_at_all_other_indices` lemma below. Additionally, it is used when proving the general plaintext change propagation and non-propagation theorems using streams in Sections 6.8.3.2 and 6.8.3.3.

If two streams are different at some particular index only, then they are equal at all other indices.

```
Lemma streams_different_at_index_only_implies_equal_at_all_other_indices :
  forall (xss_1 xss_2 : Stream block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    forall j : nat,
      j <> i → streams_equal_at_index xss_1 xss_2 j.
```

This lemma is proven by induction in the `i` index at which the streams are different. As the `streams_different_at_index_only_implies_tails_different_at_prior_index_only` lemma above, it is used when proving the general plaintext change propagation and non-propagation theorems using streams.

Lastly, we prove that if two streams are different at some particular index only, then any two streams each bisimilar to one of the first streams are different at that index only, too.

```
Lemma bisimilar_streams_interchangeable_for_streams_different_at_index_only_extended :
  forall (xss_1 yss_1 xss_2 yss_2 : Stream block_n) (i : nat),
    bisimilar_Stream_block_n xss_2 xss_1 →
    bisimilar_Stream_block_n yss_2 yss_1 →
    streams_different_at_index_only xss_1 yss_1 i →
    streams_different_at_index_only xss_2 yss_2 i.
```

The peculiar order of inputs to the bisimilarity predicate is for easier application of the lemma. The lemma is proven by unfolding predicates until the applications of the `stream_ref` operator are revealed and using the `bisimilarity_implies_equal_values_at_equal_indices` lemma in Section 3.4.2 to be able to

exploit that all elements of bisimilar streams are pairwise equal. The proven lemma is used in the proof of the f mode plaintext change non-propagation theorem using streams in Section 6.8.3.3.

streams_different_from_index_and_equal_before The streams_different_from_index_and_equal_before predicate is satisfied for two streams and an index if and only if the streams are different at that index and all subsequent indices and equal at all indices prior to that index.

```
Definition streams_different_from_index_and_equal_before
           (xss_1 xss_2 : Stream block_n) (i : nat) :=
  forall j : nat,
    j >= i ↔ streams_different_at_index xss_1 xss_2 j.
```

The predicate uses the streams_different_at_index predicate defined above.

First, we prove that the streams_different_from_index_and_equal_before predicate is symmetric with regards to the input streams.

```
Lemma streams_different_from_index_and_equal_before_is_a_symmetric_relation :
  forall (xss_1 xss_2 : Stream block_n) (i : nat),
    streams_different_from_index_and_equal_before xss_1 xss_2 i →
    streams_different_from_index_and_equal_before xss_2 xss_1 i.
```

The proof of this lemma is similar to the proof of the streams_different_at_index_only_is_a_symmetric_relation lemma in the section above: we unfold the predicates to reveal the inner applications of the stream_ref operator and use the fact that the inequality is symmetric. The lemma above is used in the proof of the general plaintext change propagation theorem using streams in Section 6.8.3.2.

We show that if two streams have equal heads and their tails streams are different from some index (in the context of the tails) and equal before, then the streams are different from the next index (in the context of the whole streams) and equal before.

```
Lemma heads_equal_tail_streams_different_from_index_and_equal_before_implies_Conses_different_at_
  subsequent_index_and_equal_before :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : Stream block_n) (i' : nat),
    xs_1 = xs_2 →
    streams_different_from_index_and_equal_before xss'_1 xss'_2 i' →
    streams_different_from_index_and_equal_before
      (Cons xs_1 xss'_1)
      (Cons xs_2 xss'_2)
      (S i').
```

This lemma is proven by case analysis of the natural number j introduced by the streams_different_from_index_and_equal_before predicate. As the streams_different_from_index_and_equal_before_is_a_symmetric_relation lemma, this lemma is used when proving the general plaintext change propagation theorem using streams.

Finally, we prove that substituting one of two streams for which the predicate holds with a bisimilar stream preserves the satisfaction of the predicate. Here, we consider the case where the second stream is substituted.

```
Lemma bisimilar_streams_interchangeable_for_streams_different_from_index_and_equal_before :
  forall (xss_1 xss_2 xss_3 : Stream block_n) (i : nat),
    bisimilar_Stream_block_n xss_2 xss_3 →
    streams_different_from_index_and_equal_before xss_1 xss_2 i →
    streams_different_from_index_and_equal_before xss_1 xss_3 i.
```

As for the bisimilar_streams_interchangeable_for_streams_different_at_index_only_extended lemma in the section above, we prove this lemma by unfolding the predicates until the applications of the stream_ref operator are revealed after which we use the bisimilarity_implies_equal_values_at_equal_indices lemma in Section 3.4.2 to exploit that any two bisimilar streams have equal elements at all indices.

streams_different_at_all_indices The streams_different_at_all_indices predicate holds for two streams if they are different at all indices.

```
Definition streams_different_at_all_indices
           (xss_1 xss_2 : Stream block_n) :=
  forall i : nat,
    streams_different_at_index xss_1 xss_2 i.
```

In the definition of the predicate, we use the streams_different_at_index predicate defined above.

We prove that if two streams are different at all indices, then they are different from index 0 (and equal before).

```
Lemma streams_different_at_all_indices_implies_different_from_index_0 :
  forall xss_1 xss_2 : Stream block_n,
    streams_different_at_all_indices xss_1 xss_2 →
    streams_different_from_index_and_equal_before xss_1 xss_2 0.
```

In the proof of this lemma, we unfold the predicates to arrive at a situation where we have to show that the two streams are different at some index if and only if that index is greater than or equal to 0 and where we are given that the streams are different at any index. Proving this implication is straightforward. The lemma is used in Section 6.8.3.2 where we prove the general plaintext change propagation theorem using streams.

6.7.2 List Predicates

In this section, we define predicates concerning lists and their elements and prove lemmas about them. The section is divided into subsections—one subsection for each predicate.

Throughout this section, all elements of the lists considered are of the block_n type.

lists_different_at_index We define the lists_different_at_index predicate that is satisfied by two lists and an index if and only if the lists are different at that index.

```
Definition lists_different_at_index (xss_1 xss_2 : list block_n) (i : nat) :=
  nth_error xss_1 i <> nth_error xss_2 i.
```

Here, we use the nth_error operator in Section 3.3.1. This operator outputs error if the index is out-of-bounds. Hence, the predicate would hold for two lists of different lengths and an index out-of-bounds for exactly one of the lists. This fact, however, is not an issue since we always consider lists of equal lengths when using the predicate.

We prove three lemmas involving the predicate. First, we show that if the heads of two lists are different, then the lists are different at index 0.

```
Lemma different_heads_implies_lists_different_at_index_0 :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : list block_n),
    xs_1 <> xs_2 →
    lists_different_at_index (xs_1 :: xss'_1) (xs_2 :: xss'_2) 0.
```

This lemma is proven by unfolding the predicate and the nth_error operator and exploiting that for any two values xs and ys, we have that xs <> ys ↔ value xs <> value ys—recall that the nth_error operator outputs an element inside the value constructor (if the index is in-bounds). The proven lemma is used in the proof of the streams_different_at_all_indices_implies_same_for_cuts_of_equal_lengths lemma below.

We show that if the tails of two lists are different at some index (in the context of the tails), then the lists are different at the successive index (in the context of the whole lists).

```
Lemma lists_different_at_index_implies_conses_different_at_subsequent_index :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : list block_n) (i' : nat),
```

```
    lists_different_at_index xss'_1 xss'_2 i' →
    lists_different_at_index (xs_1 :: xss'_1) (xs_2 :: xss'_2) (S i').
```

This lemma is proven by unfolding the predicate and the inner `nth_error` operator. It is applied in the proof of the `streams_different_at_all_indices_implies_same_for_cuts_of_equal_lengths` lemma below.

We show that two streams are different at some index less than the natural number `m` if and only if corresponding cuts of length `m` are different at that index.

```
Lemma streams_different_at_index_iff_cuts_different_at_index_if_index_inbounds :
  forall (xss_1 xss_2 : Stream block_n) (i m : nat),
    i < m →
    (streams_different_at_index xss_1 xss_2 i ↔
     lists_different_at_index
       (cut_of_stream xss_1 m)
       (cut_of_stream xss_2 m)
       i).
```

Here, we use the `streams_different_at_index` predicate and the `cut_of_stream` operator defined in Sections 6.7.1 and 3.4.3. The lemma is proven by case analysis of `m` and induction in the `i` index. It is applied in the proof of the `streams_different_from_index_and_equal_before_implies_cuts_of_equal_lengths_same` lemma below.

`lists_different_at_index_only` The `lists_different_at_index_only` predicate holds for two lists and an index if and only if the lists are different at that index only.

```
Definition lists_different_at_index_only
         (xss_1 xss_2 : list block_n) (i : nat) :=
  forall j : nat,
    j = i ↔ lists_different_at_index xss_1 xss_2 j.
```

Here, we use the `lists_different_at_index` predicate defined in the section above.

We show that if two lists of equal lengths are different at some particular index, then after extending the lists into streams via the `extend_list_into_Stream` operator defined in Section 3.4.4 they are still different at that index.

```
Lemma extending_lists_of_equal_lengths_preserves_difference_at_index :
  forall (xss_1 xss_2 : list block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only
      (extend_list_into_Stream xss_1)
      (extend_list_into_Stream xss_2)
      i.
```

Here, we use the `streams_different_at_index_only` predicate defined in Section 6.7.1. The lemma is proven by induction in the the natural number `j` introduced by the `streams_different_at_index_only` predicate in the conclusion of the lemma. It is applied in the proofs of the general plaintext change propagation and non-propagation theorems using lists in Sections 6.8.3.2 and 6.8.3.3.

Additionally, we prove that if two streams are different at some particular index only, then corresponding cuts of equal lengths of these streams are different at that index only, too, if the elements at that index are not cut away.

```
Lemma streams_different_at_index_implies_cuts_of_equal_lengths_different_at_index_if_index_in_cut :
  forall (xss_1 xss_2 : Stream block_n) (i m : nat),
    i < m →
    streams_different_at_index_only xss_1 xss_2 i →
    lists_different_at_index_only
      (cut_of_stream xss_1 m)
```

```
    (cut_of_stream xss_2 m)
    i.
```

In the proof of the lemma, we unfold the predicates until the inner applications of the `stream_ref` and `nth_error` operators are revealed after which we use the `about_nth_error_cut_of_stream_stream_ref` and `nth_error_yields_error_if_index_not_in_cut` lemmas in Section 3.4.3. The proven lemma is applied in the proof of the plaintext change non-propagation theorem using lists in Section 6.8.3.3.

We prove that if two lists of equal lengths are different at some particular index only, then that index is in-bounds.

```
Lemma lists_of_equal_lengths_different_at_index_only_implies_index_inbounds :
  forall (xss_1 xss_2 : list block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    i < length xss_1.
```

In the proof of this lemma, we consider two cases: either the index is in-bounds or it is not. In the first case, the conclusion of the lemma follows immediately. The second case is proven by contradiction: the lists cannot possibly be different at an out-of-bounds index—remember that the elements of the lists are accessed using the `nth_error` operator in Section 3.3.1: this operator outputs `error` when given any of the two lists and an out-of-bounds index. Here, we use the `nth_error_yields_error_if_index_out_of_bounds` lemma in Section 3.3.2.

The `lists_of_equal_lengths_different_at_index_only_implies_index_inbounds` lemma is applied in the proof of the general MAC propagation theorem in Section 6.8.3.5.

`lists_different_from_index_and_equal_before` We define the `lists_different_from_index_and_equal_before` predicate that holds for two lists and an index if and only if the lists are different from that index and equal before.

```
Definition lists_different_from_index_and_equal_before
           (xss_1 xss_2 : list block_n) (i : nat) :=
  forall j : nat,
    j < length xss_1 →
    (j >= i ↔ lists_different_at_index xss_1 xss_2 j).
```

In the definition of the predicate, we use the `lists_different_at_index` predicate defined above. The premise stating that the natural number `j` is less than the length of the first list is necessary since for any index out-of-bounds for both lists, the lists are equal—recall the definition of the `lists_different_at_index` predicate: this predicate uses the `nth_error` operator that outputs `error` when the input index is out-of-bounds. Hence, if this premise were not included, the predicate would not hold for any two lists.

We prove several lemmas concerning the predicate defined above. First, we show that if two lists are different from index 0 (and equal before), then their heads are different.

```
Lemma lists_different_from_index_0_implies_heads_different :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : list block_n),
    lists_different_from_index_and_equal_before
      (xs_1 :: xss'_1)
      (xs_2 :: xss'_2)
      0 →
    xs_1 <> xs_2.
```

By unfolding the `lists_different_from_index_and_equal_before` predicate and then the `lists_different_at_index` predicate, the applications of the `nth_error` operator are revealed: we unfold the operator. Now, we have that `value xs_1 <> value xs_2`, and we can conclude that `xs_1 <> xs_2`. The lemma is applied in the proof of the `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma in this section.

Secondly, we prove that if two lists are different from index 0 (and equal before), then their tails have the same property.

```
Lemma lists_different_from_index_0_implies_same_for_tails :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : list block_n),
    lists_different_from_index_and_equal_before
      (xs_1 :: xss'_1)
      (xs_2 :: xss'_2)
      0 →
    lists_different_from_index_and_equal_before xss'_1 xss'_2 0.
```

In the proof of this lemma, we unfold the predicate after which we have to show that

```
j >= 0 ↔ lists_different_at_index xss'_1 xss'_2 j
```

given the hypothesis below. Here, `j` is the index introduced by the `lists_different_from_index_and_equal_before` predicate in the conclusion of the lemma.

```
  H_whole_lists : forall j : nat,
                  j < length (xs_1 :: xss'_1) →
                  (j >= 0 ↔
                   lists_different_at_index (xs_1 :: xss'_1)
                     (xs_2 :: xss'_2) j)
```

Any natural number is greater than or equal to 0 which is shown by unfolding the `ge` operator with notation >= and applying the `le_0_n` lemma (in the standard library) stating that 0 is less than or equal to any natural number. Hence, to use the hypothesis to show the goal, we just have to show that `j` is less than the length of the whole first list (`xs_1 :: xss'_1`). However, this fact is given as a premise of the predicate in the conclusion.

The lemma above is applied in the proof of the `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma in this section.

We show that if two lists are different from some positive index (in the context of the whole lists) and equal before, then their tails are different from the prior index (in the context of the tails) and equal before.

```
Lemma lists_different_at_index_and_equal_before_implies_same_for_tails_at_prior_index :
  forall (xs_1 xs_2 : block_n) (xss'_1 xss'_2 : list block_n) (i' : nat),
    lists_different_from_index_and_equal_before
       (xs_1 :: xss'_1)
       (xs_2 :: xss'_2)
       (S i') →
    lists_different_from_index_and_equal_before xss'_1 xss'_2 i'.
```

We unfold the predicates in the proof of this lemma until the inner `nth_error` operator is revealed. After doing so, we essentially have to show that—for the `j` index introduced by the `lists_different_from_index_and_equal_before` predicate in the conclusion of the lemma—`j >= i' ↔ nth_error xss'_1 j <> nth_error xss'_2 j` given that `S j >= S i' ↔ nth_error xss'_1 j <> nth_error xss'_2 j` which amounts to showing that `j >= i'` if and only if `S j >= S i'`. In this regard, we unfold the `ge` operator (whose notation is >=), to be able to apply the `le_n_S` and `le_S_n` lemmas from the standard libraries. These latter lemmas collective say that `n <= m` if and only if `S n <= S m` for any natural numbers `n` and `m`. Additionally, we show that the `j` respectively `S j` are in-bounds indices for the `xss'_1` respectively `xs_1 :: xss'_1` lists. Here, we use the `lt_n_S` lemma (from the standard libraries) stating that if a natural number is less than another natural number, then the same relation holds for their respective successors.

The lemma above is applied in the proof of `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma in this section.

We show that if two streams are different from some index and equal before, then cuts of equal lengths of these streams are different from that index and equal before, too.

```
Lemma streams_different_from_index_and_equal_before_implies_cuts_of_equal_lengths_same :
  forall (xss_1 xss_2 : Stream block_n) (i m : nat),
    streams_different_from_index_and_equal_before xss_1 xss_2 i →
    lists_different_from_index_and_equal_before
      (cut_of_stream xss_1 m)
      (cut_of_stream xss_2 m)
      i.
```

In the proof of this lemma, we unfold the `streams_different_from_index_and_equal_before` and `lists_different_from_index_and_equal_before` predicates. Then, we can use the `streams_different_at_index_iff_cuts_different_at_index_if_index_inbounds` lemma above and the `about_length_of_cut_of_stream` lemma in Section 3.4.3. The lemma above is applied in the proof of the general plaintext change propagation theorem using lists in Section 6.8.3.2.

We show that if two lists of equal lengths are different from some in-bounds index and equal before, then the last blocks of each list are different from each other.

```
Lemma lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different :
  forall (xss_1 xss_2 : list block_n) (i : nat),
    lists_different_from_index_and_equal_before xss_1 xss_2 i →
    length xss_1 = length xss_2 →
    i < length xss_1 →
    last_block xss_1 <> last_block xss_2.
```

Here, we use the `last_block` operator defined in Section 3.3.1. When applied to a list of blocks of length (elements of type `block_n`), this operator outputs the last element of the list.

The lemma is proven by induction in the `xss_1` list and case analysis of the `xss_2`, the `i` index, and the tails of the lists. It is applied in the proof of the general MAC propagation theorem in Section 6.8.3.5.

`lists_different_at_all_indices` We define the last list predicate: the `lists_different_at_all_indices` predicate. This predicate is satisfied by two lists if and only if the lists are different at all indices in-bounds for both lists.

```
Definition lists_different_at_all_indices (xss_1 xss_2 : list block_n) :=
  forall i : nat,
    i < length xss_1 →
    i < length xss_2 →
    lists_different_at_index xss_1 xss_2 i.
```

In the definition above of the predicate, we use the `lists_different_at_index` predicate defined above.

We prove one lemma involving the `lists_different_at_all_indices` predicate. This lemma states that if two streams are different at all indices, then any corresponding cuts of equal lengths are different at all indices, too.

```
Lemma streams_different_at_all_indices_implies_same_for_cuts_of_equal_lengths :
  forall (xss_1 xss_2 : Stream block_n) (m : nat),
    streams_different_at_all_indices xss_1 xss_2 →
    lists_different_at_all_indices
        (cut_of_stream xss_1 m)
        (cut_of_stream xss_2 m).
```

In this lemma, we use the `streams_different_at_all_indices` predicate and the `cut_of_stream` operator defined in Sections 6.7.1 and 3.4.3.

The lemma is proven by induction in the index introduced by the `lists_different_at_all_indices` predicate in the conclusion of the lemma. It is applied in the proof of the general IV change propagation theorem using lists in Section 6.8.3.4.

6.7.3 Predicates of the Underlying Functions

In this section, we define predicates involving the underlying block encryption schemes or functions and the block and feedback functions of the modes of operation.

The first predicate, the `block_enc_scheme_valid` predicate, concerns the validity of an underlying encryption scheme.

```
Definition block_enc_scheme_valid
           (block_enc block_dec : block_enc_dec_Type) :=
  forall (xs :  block_n) (ks :  block_k),
    block_dec (block_enc xs ks) ks = xs.
```

We define validity of an underlying encryption scheme to mean that encrypting and then decrypting any plaintext using the functions of the scheme with the same key (for any key) yields the plaintext again. The predicate is used in Section 6.10 where we prove that the CBC and IFB modes of operation, which use both the encryption and decryption functions of underlying block encryption schemes, are valid.

The second predicate, the `block_functions_inverse` predicate, concerns the block functions.

```
Definition block_functions_inverse
           (block_fun_enc block_fun_dec : block_fun_Type)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xs :  block_n) (ks :  block_k) (IV :  block_n),
    block_fun_dec (block_fun_enc xs ks IV block_enc) ks IV block_enc_dec =
    xs.
```

As explained in Section 6.3, we say that the two block functions we use for encryption and decryption are inverse if it is the case that if we first apply the block function for encryption and then the block function for decryption under the same key on some block, `xs` (representing a plaintext block), that block is output. The other inputs are the same for the two function applications.

The third predicate, the `feedback_functions_output_equal_values` predicate, concerns the feedback functions.

```
Definition feedback_functions_output_equal_values
           (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
           (block_fun_enc : block_fun_Type)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xs :  block_n) (ks :  block_k) (IV :  block_n),
    feedback_fun_enc xs ks IV block_enc =
    feedback_fun_dec (block_fun_enc xs ks IV block_enc) ks IV block_enc_dec.
```

This predicate holds for two feedback functions for encryption and decryption if and only if they output equal values when applied to corresponding plaintext and ciphertext blocks and equal keys and IV or feedback blocks. If this predicate holds for the two feedback functions of a mode of operation, one can conclude by induction that all feedback blocks encountered when encrypting and decrypting corresponding plaintexts and ciphertexts using that mode are pairwise equal when the same key and IV are used. Note that the plaintext block, `xs`, is input directly to the feedback function for encryption while the corresponding ciphertext block, `(block_fun_enc xs ks IV block_enc)`, is input to the feedback function for decryption.

We assume the two predicates above to hold for the block and feedback functions in the general validity theorems in Section 6.8.3.1. Hence, in Section 6.10, we prove that these predicates are satisfied for the block and feedback functions of the specific modes of operations. In this way, we are able to prove the validity theorems for each mode as corollaries of the general validity theorems.

In the general plaintext change propagation and MAC propagation theorems in Sections 6.8.3.2 and 6.8.3.5, we assume that the block and feedback functions for encryption are invertible with regards to both the plaintext blocks and the IVs or feedback blocks. Additionally, in the general plaintext change

non-propagation theorems in Section 6.8.3.3, we assume that the block function for encryption is invertible with regards to the plaintext block—this requirement is not strictly necessary for plaintext change non-propagation in the sense that we only use this property of the block function to show that a change of a plaintext block results in a change of the corresponding ciphertext block, not to show that the subsequent ciphertext blocks remain unchanged. Furthermore, the block and feedback functions for encryption are assumed to be invertible with regards to the IV or feedback blocks in the general IV change propagation theorems in Section 6.8.3.4.

Because of these assumptions in the general theorems, we prove that the relevant properties hold for the specific block and feedback functions for encryption in Section 6.10: we prove the specific plaintext change propagation and non-propagation, MAC propagation, and IV change propagation theorems as corollaries of the corresponding general theorems. Hence, for the modes whose encryption functions propagate plaintext changes, we show that the corresponding block and feedback functions for encryption are invertible are invertible with regards to both the plaintext blocks and the IVs or feedback blocks, and, for the modes whose encryption functions do not propagate plaintext changes, we show that the corresponding block functions are invertible with regards to both the plaintext blocks and the IVs or feedback blocks and that the corresponding feedback functions are invertible with regards to the IVs or feedback blocks: for the former group of modes, we do not prove any plaintext change non-propagation theorems, and, for the latter group of modes, we do not prove any plaintext change and MAC propagation theorems.

We define the predicates concerning invertibility for the block functions for encryption.

```
Definition block_fun_invertible_wrt_plaintext
           (block_fun : block_fun_Type) (block_enc : block_enc_dec_Type) :=
  forall (xs_1 xs_2 : block_n) (ks :  block_k) (IV_feedback : block_n),
    block_fun xs_1 ks IV_feedback block_enc =
    block_fun xs_2 ks IV_feedback block_enc →
    xs_1 = xs_2.

Definition block_fun_invertible_wrt_IV_feedback
           (block_fun : block_fun_Type) (block_enc : block_enc_dec_Type) :=
  forall (xs :  block_n) (ks :  block_k)
         (IV_feedback_1 IV_feedback_2 : block_n),
    block_fun xs ks IV_feedback_1 block_enc =
    block_fun xs ks IV_feedback_2 block_enc →
    IV_feedback_1 = IV_feedback_2.
```

Similarly, we define the predicates concerning invertibility for the feedback functions for encryption.

```
Definition feedback_fun_invertible_wrt_plaintext
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xs_1 xs_2 : block_n) (ks :  block_k) (IV_feedback : block_n),
    feedback_fun xs_1 ks IV_feedback block_enc =
    feedback_fun xs_2 ks IV_feedback block_enc →
    xs_1 = xs_2.

Definition feedback_fun_invertible_wrt_IV_feedback
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xs :  block_n) (ks :  block_k)
         (IV_feedback_1 IV_feedback_2 : block_n),
    feedback_fun xs ks IV_feedback_1 block_enc =
    feedback_fun xs ks IV_feedback_2 block_enc →
    IV_feedback_1 = IV_feedback_2.
```

The predicates follow the definition of inverse functions: if the outputs of the function applications are equal, then the inputs—either the plaintext blocks or IVs or feedback blocks—in question must be equal, too, if all other inputs are pairwise equal.

When we prove that the specific block and feedback functions for encryption are invertible, we often assume that the underlying block encryption functions are invertible with regards to the in-the-context-of-these-functions plaintext block. In these cases, we use the `block_enc_invertible_wrt_plaintext_in_context` predicate defined below.

```
Definition block_enc_invertible_wrt_plaintext_in_context
           (block_enc : block_enc_dec_Type) :=
  forall (xs_1 xs_2 : block_n) (ks :  block_k),
    block_enc xs_1 ks = block_enc xs_2 ks →
    xs_1 = xs_2.
```

If two plaintext blocks are encrypted into equal ciphertext blocks using an invertible block encryption function under the same key, then the plaintext blocks are equal.

In the general plaintext change non-propagation theorems in Section 6.8.3.3, one of the premises states that the input feedback function (for encryption) ignores the plaintext block, i.e., the output of the function is independent on the value of the plaintext block. In this way, a change of a plaintext block does not alter the corresponding feedback block and, in extension, any subsequent feedback blocks resulting in only the corresponding ciphertext block changing (we assume in the theorem that the input block function is invertible with regards to the plaintext block). We define the predicate below.

```
Definition feedback_fun_ignores_plaintext
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xs_1 xs_2 : block_n) (ks :  block_k) (IV :  block_n),
    feedback_fun xs_1 ks IV block_enc = feedback_fun xs_2 ks IV block_enc.
```

All other inputs being equal, changing the input plaintext block does not alter the output of the function.

6.7.4 Non-General Function Predicates

In this section, we define predicates capturing the properties that we prove about the non-general encryption, decryption, and MAC functions.

The two first predicates concern the validity of the modes of operation using streams respectively lists. We define what it means for a mode using an underlying encryption function or scheme to be valid (if only the underlying encryption function is used, it is provided twice to the predicate). First, we consider the stream version of the predicate: `MoO_valid_stream_version`.

```
Definition MoO_valid_stream_version
           (MoO_enc MoO_dec : MoO_enc_dec_Type_stream_version)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss css :  Stream block_n) (ks : block_k) (IV :  block_n),
    bisimilar_Stream_block_n css (MoO_enc xss ks IV block_enc) →
    bisimilar_Stream_block_n (MoO_dec css ks IV block_enc_dec) xss.
```

If the above `MoO_valid_stream_version` predicate holds for a mode, then if some stream, `css`, is bisimilar to the ciphertext output by the encryption function, then the decryption of `css` is bisimilar to the original plaintext `xss`. Said differently, encrypting and then decrypting amounts to doing nothing: the original plaintext blocks are output. This property must hold for any key and IV.

The predicate above captures the definition of validity in Section 6.3. Notice that the underlying block encryption function (and underlying block decryption function if applicable) is input to the definition above, too. This inclusion is necessary as we cannot expect a mode of operation to be valid for any underlying block encryption function (and any underlying block decryption function if applicable).

Below, we define the `MoO_valid_list_version` predicate which is the list version of the predicate above.

```
Definition MoO_valid_list_version
           (MoO_enc_lv MoO_dec_lv : MoO_enc_dec_Type_list_version)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV : block_n),
    MoO_dec_lv (MoO_enc_lv xss ks IV block_enc) ks IV block_enc_dec = xss.
```

This predicate is used for the list versions of the modes of operation. It is similar to the predicate using streams above. Now, however, we do not have to use bisimulation as we use lists instead of streams. The conclusion of the predicate captures the concept of validity: first encrypting and then decrypting using the same key and IV yields the original plaintext list.

In Section 6.10, we show that the stream and list versions of the encryption functions of the CBC, CFB, and IFB modes of operation propagate plaintext changes. There, we use the following two predicates. First, we consider the stream version of the plaintext change propagation predicate: the `plaintext_change_propagation_stream_version` predicate defined below.

```
Definition plaintext_change_propagation_stream_version
           (MoO_enc : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_from_index_and_equal_before
      (MoO_enc xss_1 ks IV block_enc)
      (MoO_enc xss_2 ks IV block_enc)
      i.
```

The predicate above holds for an encryption function of a mode of operation using streams and an underlying block encryption function if and only if any change of a single plaintext block results in all ciphertext blocks from that point forward changing also (and the ciphertext blocks before not changing). The predicate uses the `streams_different_at_index_only` predicate respectively the `streams_different_from_index_and_equal_before` predicate defined Section 6.7.1 which, as their names imply, hold for two streams and an index if the streams are different at that index only respectively are different from that index and equal before.

Likewise, we define the list version of the predicate above: the `plaintext_change_propagation_list_version` predicate.

```
Definition plaintext_change_propagation_list_version
           (MoO_enc_lv : MoO_enc_dec_Type_list_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_from_index_and_equal_before
      (MoO_enc_lv xss_1 ks IV block_enc)
      (MoO_enc_lv xss_2 ks IV block_enc)
      i.
```

This predicate is analogous to the `plaintext_change_propagation_stream_version` predicate above. Now, however, we consider plaintext lists, not plaintext streams. This change spurs a few other changes: we use the `lists_different_at_index_only` and `lists_different_from_index_and_equal_before` predicates defined Section 6.7.2 which correspond to the `streams_different_at_index_only` and `streams_different_from_index_and_equal_before` predicates used above, and we require that the two plaintext lists are of equal lengths: changing a plaintext block does not change the number of plaintext blocks.

In Section 6.10, we show that the stream and list versions of the encryption functions of the f, CTR, and BENC modes of operation do not propagate plaintext changes. There, we use the plaintext

change non-propagation predicates below. We consider the stream version of the predicate first: the `plaintext_change_non_propagation_stream_version` predicate below.

```
Definition plaintext_change_non_propagation_stream_version
           (MoO_enc : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only
      (MoO_enc xss_1 ks IV block_enc)
      (MoO_enc xss_2 ks IV block_enc)
      i.
```

This predicate holds for a encryption function of a mode of operation using streams in conjunction with an underlying block encryption function if and only if a change of plaintext block results in the corresponding ciphertext block changing and all other ciphertext blocks remaining unchanged. As above, we use the `streams_different_at_index_only` predicate to state that two streams are different at some particular index only.

Similarly, we define the list version of the plaintext change non-propagation predicate.

```
Definition plaintext_change_non_propagation_list_version
           (MoO_enc_lv : MoO_enc_dec_Type_list_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_at_index_only
      (MoO_enc_lv xss_1 ks IV block_enc)
      (MoO_enc_lv xss_2 ks IV block_enc)
      i.
```

This predicate is similar to the `plaintext_change_non_propagation_stream_version` predicate above. However, we replace the plaintext streams with plaintext lists which results in the use of other predicates: now, we use the `lists_different_at_index_only` predicate defined in Section 6.7.2 to specify that two lists are different at some particular index only. Also, we require that the two plaintext list have equal lengths: again, change a plaintext block does not change the number of plaintext blocks.

All encryption functions of the modes of operation propagate IV changes, i.e., for any of the encryption functions, if the input IV is changed and all other inputs remain the same, then all blocks of the output ciphertext stream or list are changed. In Section 6.10, we prove this property for all the encryption functions.

We define the `IV_change_propagation_stream_version` and `IV_change_propagation_list_version` predicates below. These predicates capture the aforementioned property for encryption functions using streams respectively lists.

```
Definition IV_change_propagation_stream_version
           (MoO_enc_sv : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : Stream block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    streams_different_at_all_indices
      (MoO_enc_sv xss ks IV_1 block_enc)
      (MoO_enc_sv xss ks IV_2 block_enc).

Definition IV_change_propagation_list_version
           (MoO_enc_lv : MoO_enc_dec_Type_list_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV_1 IV_2 : block_n),
```

```
    IV_1 <> IV_2 →
    lists_different_at_all_indices
      (MoO_enc_lv xss ks IV_1 block_enc)
      (MoO_enc_lv xss ks IV_2 block_enc).
```

The predicates use the `streams_different_at_all_indices` and `lists_different_at_all_indices` predicates defined in Sections 6.7.1 and 6.7.2: all blocks of the ciphertext streams or lists are changed if the IV is changed.

In Section 6.10, we prove that the CBC-, CFB-, and IFB-MAC functions propagate changes in the input messages. Below, we define the `MAC_propagation` predicate capturing this notion.

```
Definition MAC_propagation
           (MAC_fun : MAC_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    MAC_fun xss_1 ks block_enc <> MAC_fun xss_2 ks block_enc.
```

This predicate resembles the `plaintext_change_non_propagation_list_version` predicate above. Now, however, we do not have any IV, and the conclusion is different: we require that the MAC value changes as a consequence of a single block in the message changing. Again, we use the `lists_different_at_index_only` predicate stating that two lists are different at some particular index only. We require that the plaintext lists considered have the same lengths: changing a plaintext block does not change the number of plaintext blocks.

6.8 The General Encryption and MAC Schemes

Before we state and prove theorems about the modes of operation and the associated MAC schemes, we look at the general encryption, decryption, and MAC functions. These general functions take as input in addition to the inputs for the corresponding, non-general functions also a block function and a feedback function both specifying the inner operations of the particular mode of operation used.

In Section 6.8.1, we specify the general functions using streams. Likewise, in Section 6.8.2, we specify the general functions using lists. Then, in Section 6.8.3, we prove theorems involving these specifications.

6.8.1 Specifications Using Streams

We specify first the stream version of the general encryption function.

```
Definition specification_of_general_MoO_encryption_function_stream_version
           (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version) :=
  forall (xs : block_n) (ks : block_k) (IV : block_n) (xss' : Stream block_n)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    bisimilar_Stream_block_n
      (general_MoO_enc_sv
         (Cons xs xss') ks IV block_enc block_fun feedback_fun)
      (Cons
         (block_fun xs ks IV block_enc)
         (general_MoO_enc_sv
            xss' ks (feedback_fun xs ks IV block_enc) block_enc
            block_fun feedback_fun)).
```

Recall the definition of the `general_MoO_enc_dec_Type_stream_version` type in Section 6.5.2. The definition above specifies that the output of the general function applied to a plaintext stream is bisimilar to

the stream whose head element consists of the output of the block function applied to the first plaintext block and the original IV and whose tail consists of the output of the corecursive application of the general encryption function to the tail of the plaintext stream with the IV replaced with the output of the feedback function applied to the first plaintext block.

The stream version of the general decryption function is specified similarly below.

```
Definition specification_of_general_MoO_decryption_function_stream_version
           (general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version) :=
  forall (cs : block_n) (ks : block_k) (IV : block_n) (css' : Stream block_n)
         (block_dec : block_enc_dec_Type)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    bisimilar_Stream_block_n
      (general_MoO_dec_sv
         (Cons cs css') ks IV block_dec block_fun feedback_fun)
      (Cons
         (block_fun cs ks IV block_dec)
         (general_MoO_dec_sv
            css' ks (feedback_fun cs ks IV block_dec) block_dec
            block_fun feedback_fun)).
```

This specification resembles the specification of the stream version of the general encryption function above except that the head element is decrypted with the block function instead of encrypted. Further, the block and feedback functions now potentially use an underlying block *decryption* function instead of an underlying block encryption function as is the case for the CBC mode defined in Section 6.4.1.

Below, we define two corecursive functions: the `general_MoO_enc_sv_v0` function that fits the specification of the stream version of the general encryption function and the `general_MoO_dec_sv_v0` function that fits the specification of the stream version of the general decryption function.

```
CoFixpoint general_MoO_enc_sv_v0
           (xss : Stream block_n)
           (ks : block_k)
           (IV : block_n)
           (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type) :
  Stream block_n :=
  match xss with
    | Cons xs xss' ⇒
      Cons
        (block_fun xs ks IV block_enc)
        (general_MoO_enc_sv_v0
           xss' ks (feedback_fun xs ks IV block_enc)
           block_enc block_fun feedback_fun)
  end.

CoFixpoint general_MoO_dec_sv_v0
           (css : Stream block_n)
           (ks : block_k)
           (IV : block_n)
           (block_dec : block_enc_dec_Type)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type) :
  Stream block_n :=
  match css with
    | Cons cs css' ⇒
      Cons
        (block_fun cs ks IV block_dec)
```

```
      (general_MoO_dec_sv_v0
         css' ks (feedback_fun cs ks IV block_dec) block_dec
         block_fun feedback_fun)
end.
```

That these functions satisfy the specifications is proven using corresponding unfolding lemmas and the `equal_values_at_equal_indices_implies_bisimilarity` lemma (defined in Section 3.4.2) that says that two bisimilar streams have pairwise equal elements at all positions.

6.8.2 Specifications Using Lists

When encrypting and decryption with the modes of operation, the plaintexts or ciphertexts can in theory be infinite—they do not have to terminate. However, we want to show that the modes of operation work when the plaintexts or ciphertexts have finite lengths. Therefore, we specify list versions of the functionalities in the section above. Additionally, these specifications are used when stating and proving theorems about the MAC schemes that can be constructed on top of the propagating modes of operation.

First, we specify the list version of the general encryption function.

```
Definition specification_of_general_MoO_encryption_function_list_version
          (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version) :=
  forall (ks : block_k)
         (IV : block_n)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    general_MoO_enc_lv List.nil ks IV block_enc block_fun feedback_fun =
    List.nil
    ∧
    forall (xs : block_n) (xss' : list block_n),
      general_MoO_enc_lv (xs :: xss') ks IV block_enc
      block_fun feedback_fun =
      (block_fun xs ks IV block_enc)
        ::
        (general_MoO_enc_lv
           xss' ks (feedback_fun xs ks IV block_enc) block_enc
           block_fun feedback_fun).
```

Recall the definition of the `general_MoO_enc_dec_Type_list_version` type in Section 6.5.2. The plaintext is represented by the type of `list block_n`: it is a list of blocks of length `n`. The specification above is similar to the corresponding specification using streams in the above section.

Now, contrary to the specification using streams, this specification considers the case where the input plaintext list is empty. In that case, it is specified that the ciphertext list is also the empty list. When the plaintext list is in the inductive case, the block function is applied to the head of the plaintext list to form the head of the ciphertext list while the tail of the ciphertext list is the output of the general encryption function applied recursively to the tail of the plaintext list with the IV replaced by the output of the feedback function applied to the head of the plaintext list.

Below, the list version of the general decryption function is specified analogously.

```
Definition specification_of_general_MoO_decryption_function_list_version
          (general_MoO_dec_lv : general_MoO_enc_dec_Type_list_version) :=
  forall (ks : block_k)
         (IV : block_n)
         (block_dec : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    general_MoO_dec_lv List.nil ks IV block_dec block_fun feedback_fun =
```

```
    List.nil
    ∧
    forall (cs : block_n) (css' : list block_n),
      general_MoO_dec_lv
        (cs :: css') ks IV block_dec
        block_fun feedback_fun =
      (block_fun cs ks IV block_dec)
        ::
        (general_MoO_dec_lv
           css' ks (feedback_fun cs ks IV block_dec) block_dec
           block_fun feedback_fun).
```

This specification resembles the specification of the list version of the general encryption function above: conceptually, the only differences are that the plaintext list is replaced by a ciphertext list and that the functions are replaced by their decryption counterparts.

In Sections 6.8.3.1 to 6.8.3.4, we prove validity, plaintext change propagation and non-propagation, and IV change propagation theorems for the list version of the general encryption scheme. Since we prove the corresponding theorems for the stream version of the scheme, we want to specify the list versions of the general encryption and decryption functions in terms of the corresponding functions using streams. In this way, we can prove theorems for the list version of the scheme as corollaries of the corresponding theorems using streams.

First, we define the alternative specification of the list version of the general encryption function.

```
Definition specification_of_general_MoO_encryption_function_list_version_alt
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version) :=
  forall general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version,
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
    forall (xss : list block_n)
           (ks : block_k)
           (IV : block_n)
           (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type),
      general_MoO_enc_lv xss ks IV block_enc block_fun feedback_fun =
      cut_of_stream
        (general_MoO_enc_sv
           (extend_list_into_Stream xss) ks IV block_enc
           block_fun feedback_fun)
        (length xss).
```

The plaintext list is now encrypted by first extending it into a stream, then encrypting it using the general encryption function for streams, and, finally, cutting the resulting ciphertext stream into a list.

In more detail, the general encryption function using lists is specified to be extensionally equal to following this procedure: first, the `extend_list_into_Stream` operator defined in Section 3.4.4 is applied to the plaintext list. This operator appends a stream of all-zero blocks of length `n` (elements of type `block_n` whose entries are all equal to the Boolean `false`) onto the list thereby turning it into a stream. The output stream is then input to a function satisfying the specification of the general encryption function using streams. This application yields a ciphertext stream which is then cut into a list using the `cut_of_stream` operator defined in Section 3.4.3: the ciphertext stream is cut off so only the beginning of it remains and such that the resulting list has the same length as the original plaintext list.

Likewise, we define the alternative specification of the list version of the general decryption function.

```
Definition specification_of_general_MoO_decryption_function_list_version_alt
           (general_MoO_dec_lv : general_MoO_enc_dec_Type_list_version) :=
  forall general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version,
```

```
  specification_of_general_MoO_decryption_function_stream_version
    general_MoO_dec_sv →
  forall (css : list block_n)
         (ks : block_k)
         (IV : block_n)
         (block_dec : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    general_MoO_dec_lv css ks IV block_dec block_fun feedback_fun =
    cut_of_stream
      (general_MoO_dec_sv
         (extend_list_into_Stream css) ks IV block_dec
         block_fun feedback_fun)
      (length css).
```

This specification is very similar to the alternative specification of the list version of the general encryption function. Now, however, we use a general *decryption* function using streams, and the plaintext list is replaced by the ciphertext list.

In Sections 6.8.3.1 to 6.8.3.4, where we prove theorems involving the list version of the general encryption scheme, we use functions fitting the original, non-alternative specifications of the general encryption and decryption functions. We show that these specifications are equivalent to the two alternative specifications: in this way, we shorten the gap between the theorems for the general encryption scheme using streams and the theorems for the corresponding scheme using lists: the theorems using lists are proven as corollaries of the corresponding theorems using streams.

We show that the specifications of the general encryption function using lists are equivalent in the sense that any function satisfying one of the specifications also satisfies the other.

```
Lemma specifications_of_general_MoO_encryption_function_list_version_are_equivalent :
  forall general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version,
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv ↔
    specification_of_general_MoO_encryption_function_list_version_alt
      general_MoO_enc_lv.
```

The proof of the above lemma is done by splitting the biimplication. The implication going in the right direction is proven by induction in the plaintext list introduced by the alternative specification while the main part of the proof of the opposite implication is rewriting via the alternative specification.

Likewise, we prove a similar lemma for the specifications of the general decryption function using lists.

```
Lemma specifications_of_general_MoO_decryption_function_list_version_are_equivalent :
  forall general_MoO_dec_lv : general_MoO_enc_dec_Type_list_version,
    specification_of_general_MoO_decryption_function_list_version
      general_MoO_dec_lv ↔
    specification_of_general_MoO_decryption_function_list_version_alt
      general_MoO_dec_lv.
```

This lemma is proven by merely applying the lemma above: the specifications in the lemmas are alike.

We define two functions that are proven to satisfy the original, non-alternative specifications of the general encrypting and decryption functions using lists—and, by the lemmas above, the corresponding, alternative specifications, too.

```
Fixpoint general_MoO_enc_lv_v0
         (xss : list block_n)
         (ks : block_k)
         (IV : block_n)
         (block_enc : block_enc_dec_Type)
```

```
          (block_fun : block_fun_Type)
          (feedback_fun : feedback_fun_Type) :
  list block_n :=
  match xss with
    | List.nil ⇒ List.nil
    | xs :: xss' ⇒
      (block_fun xs ks IV block_enc)
        ::
        (general_MoO_enc_lv_v0
           xss' ks (feedback_fun xs ks IV block_enc) block_enc
           block_fun feedback_fun)
  end.

Fixpoint general_MoO_dec_lv_v0
          (css : list block_n)
          (ks : block_k)
          (IV : block_n)
          (block_dec : block_enc_dec_Type)
          (block_fun : block_fun_Type)
          (feedback_fun : feedback_fun_Type) :
  list block_n :=
  match css with
    | List.nil ⇒ List.nil
    | cs :: css' ⇒
      (block_fun cs ks IV block_dec)
        ::
        (general_MoO_dec_lv_v0
           css' ks (feedback_fun cs ks IV block_dec)
           block_dec block_fun feedback_fun)
  end.
```

These definitions follow the structures of the original, non-alternative specifications. Hence, the proofs that they satisfy these specifications consist primarily of unfolding the functions.

We prove that the `general_MAC_v0` function fits the specification of the general MAC function (see further below). In that proof, we use a lemma stating that any two functions satisfying the original, non-alternative specification of the general encryption function using lists are extensionally equal.

```
Lemma there_is_only_one_general_MoO_encryption_function_list_version :
  forall general_MoO_enc_lv_1 general_MoO_enc_lv_2 :
          general_MoO_enc_dec_Type_list_version,
    specification_of_general_MoO_encryption_function_list_version
       general_MoO_enc_lv_1 →
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv_2 →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
          (block_enc : block_enc_dec_Type)
          (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
      general_MoO_enc_lv_1 xss ks IV block_enc block_fun feedback_fun =
      general_MoO_enc_lv_2 xss ks IV block_enc block_fun feedback_fun.
```

This lemma is proven by induction in the plaintext list `xss`.

We now specify the general MAC function.

```
Definition specification_of_general_MAC_function
          (general_MAC : general_MAC_fun_Type) :=
  forall general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version,
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
```

```
  forall (xss : list block_n) (ks : block_k)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    general_MAC xss ks block_enc block_fun feedback_fun =
    last_block
      (general_MoO_enc_lv
         xss ks zero_block_n block_enc block_fun feedback_fun).
```

The specification tells us that the output of the general MAC function is equal to the last block of the ciphertext corresponding to the input message consisting of a finite number of blocks. The encryption is done using the list version of the general encryption function under the given key since the messages input to the MAC functions are finite. The IV is set to the all-zero block of length `n` via the `zero_block_n` constant defined in Section 3.4.4. The last block is accessed by using the `last_block` operator defined in Section 3.3.1.

When proving the CBC-, CFB-, and IFB-MAC propagation theorems in Section 6.10, we need the existence of a function fitting the above specification. Thus, we define such a function below.

```
Definition general_MAC_fun_v0
         (xss : list block_n)
         (ks : block_k)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type) :
  block_n :=
  last_block
    (general_MoO_enc_lv_v0
       xss ks zero_block_n block_enc block_fun feedback_fun)..
```

In this definition, we use the `general_MoO_enc_lv_v0` function defined above. That function is proven to fit the specification of the general encryption function using lists. Thus, we can prove that the `general_MAC_fun_v0` function fits the specification of the general MAC function by rewriting with the `there_is_only_one_general_MoO_encryption_function_list_version` lemma above.

6.8.3 The General Theorems and Proofs

In this section, we state and prove the general theorems concerning the general specifications.

In Section 6.8.3.1, we prove the general validity theorems. In Section 6.8.3.2, we prove the general plaintext change propagation theorems while, in Section 6.8.3.3, we prove the general plaintext change non-propagation theorems. In Section 6.8.3.4, we prove the general IV change propagation theorems. Finally, in Section 6.8.3.5, we prove the general MAC propagation theorem.

6.8.3.1 The General Validity Theorems

All the modes we consider are valid, i.e., first encrypting and then decrypting any plaintext under the same key using any of the modes results in the plaintext.

We have two general validity theorems: one for the general encryption scheme using streams in Section 6.8.1 and one for the general encryption scheme using lists in Section 6.8.2.

The General Validity Theorem Using Streams We prove the validity theorem for the stream version of the general encryption scheme specified in Section 6.8.1.

```
Theorem general_validity_theorem_stream_version :
  forall (general_MoO_enc_sv general_MoO_dec_sv :
            general_MoO_enc_dec_Type_stream_version)
         (block_fun_enc block_fun_dec : block_fun_Type)
```

```
      (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
      (block_enc block_dec : block_enc_dec_Type),
  specification_of_general_MoO_encryption_function_stream_version
    general_MoO_enc_sv →
  specification_of_general_MoO_decryption_function_stream_version
    general_MoO_dec_sv →
  block_functions_inverse
    block_fun_enc block_fun_dec
    block_enc block_dec →
  feedback_functions_output_equal_values
    feedback_fun_enc feedback_fun_dec block_fun_enc
    block_enc block_dec →
  general_MoO_valid_stream_version
    general_MoO_enc_sv general_MoO_dec_sv
    block_fun_enc block_fun_dec
    feedback_fun_enc feedback_fun_dec
    block_enc block_dec.
```

This theorem states that if the block functions are inverse and the feedback functions output equal values, then the general encryption scheme using these functions is valid.

The theorem above uses the `general_MoO_valid_stream_version` predicate defined below in its conclusion. This predicate captures what it means for a general encryption scheme using streams to be valid.

```
Definition general_MoO_valid_stream_version
          (general_MoO_enc_sv general_MoO_dec_sv :
             general_MoO_enc_dec_Type_stream_version)
          (block_fun_enc block_fun_dec : block_fun_Type)
          (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
          (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss css : Stream block_n) (ks : block_k) (IV : block_n),
    bisimilar_Stream_block_n
      css
      (general_MoO_enc_sv
         xss ks IV block_enc block_fun_enc feedback_fun_enc) →
    bisimilar_Stream_block_n
      (general_MoO_dec_sv
         css ks IV block_enc_dec block_fun_dec feedback_fun_dec)
      xss.
```

This definition follows the structure of the corresponding, non-general predicate (the `MoO_valid_stream_version` predicate in Section 6.7.4), but now we also provide the block and feedback functions specifying the inner operations of the particular modes. The motivations of the details of the predicate above and the `MoO_valid_stream_version` predicate coincide: confer Section 6.7.4.

The proof of the theorem is done by proving the bisimilarity in the conclusion of the `general_MoO_valid_stream_version` predicate. This bisimilarity is translated to a problem of proving two streams to be equal at all indices involving the `stream_ref` operator using the `equal_values_at_equal_indices_implies_bisimilarity` lemma in Section 3.4.2 and unfolding the resulting `equal_values_at_equal_indices` predicate also defined in that section. This translation introduces an index `i`: we prove the theorem by induction in this index.

For the base case of the index, we have to prove that the heads of the `xss` plaintext stream and the stream resulting from decrypting the `css` ciphertext stream are equal. By unfolding the decryption function, we arrive at a situation where we have to prove that the output of the block function for decryption applied to the head of the ciphertext stream is equal to the head of the plaintext stream. Here, we use the last premise of the `general_MoO_valid_stream_version` predicate to assert that the head

of the ciphertext stream is equal to the output of the block function for *encryption* applied to the head of the plaintext stream. Hence, after rewriting with this assertion, we can apply the premise (of the theorem we are proving) that states that the block functions are inverse.

For the inductive case of the index `i` where `i = S i'` for some natural number `i'`, we have to show that the plaintext stream and the stream resulting from decrypting the ciphertext stream are equal at index `i = S i'` given the induction hypothesis below.

```
IHi' : forall (xss css : Stream block_n) (IV0 : block_n),
       bisimilar_Stream_block_n css
         (general_MoO_enc_sv xss ks IV0 block_enc block_fun_enc
            feedback_fun_enc) →
       stream_ref
         (general_MoO_dec_sv css ks IV0 block_dec block_fun_dec
            feedback_fun_dec) i' = stream_ref xss i'
```

Here, the streams and the IV are parameterised. Hence, we can show that the inductive case holds: showing that streams are equal a index `S i'` is equivalent to showing that the corresponding tail streams are equal at index `i'`.

The tail of the output of the decryption function applied to the ciphertext stream is equal to the output of the function applied to the tail of the ciphertext stream, but with the IV being the output of the feedback function for decryption applied to the head of the ciphertext stream. This property follows from the specification of the function. Hence, after unfolding the function via the specification, we have to show that the tails are equal. We do so by applying the induction hypothesis above.

After applying the hypothesis, we have to show that the premise of it holds: we have to show the goal below—after applying the `equal_values_at_equal_indices_implies_bisimilarity` lemma in Section 3.4.2 and introducing the index `j`.

```
stream_ref css' j =
stream_ref
  (general_MoO_enc_sv xss' ks (feedback_fun_dec cs ks IV block_dec)
     block_enc block_fun_enc feedback_fun_enc) j
```

At this point in the proof, however, we have the hypothesis shown below from the premise of the `general_MoO_valid_stream_version` predicate in the conclusion of the theorem we are proving—after applying the `bisimilarity_implies_equal_values_at_equal_indices` lemma in Section 3.4.2 to it and specialising it with `S j`.

```
H_css_S_j : stream_ref (Cons cs css') (S j) =
            stream_ref
              (general_MoO_enc_sv (Cons xs xss') ks IV block_enc
                 block_fun_enc feedback_fun_enc)
              (S j)
```

Now, we unfold the general encryption function in the hypothesis via its specification and the unfolding lemma for the `stream_ref` operator handling the inductive case of the input index: the tail of the output of the general encryption function applied to the plaintext stream is equal to the output of the encryption function applied to the tail of the plaintext stream, but with the IV being the output of the feedback function for encryption applied to the head of the plaintext stream. After the unfolding, the hypothesis is as below.

```
H_css_S_j : stream_ref css' j =
            stream_ref
              (general_MoO_enc_sv xss' ks
                 (feedback_fun_enc xs ks IV block_enc) block_enc
                 block_fun_enc feedback_fun_enc) j
```

This hypothesis resembles the goal above that we have to show. However, the IVs differ: in the goal, the IV is `feedback_fun_dec cs ks IV block_dec` while, in the hypothesis, the IV is `feedback_fun_enc xs ks IV block_enc`. Here, we use the premise (in the theorem we are proving) that states that the feedback functions output equal values via the `block_functions_inverse` predicate defined in Section 6.7.3.

We first have to show that the head of the ciphertext stream, `cs`, is the output of the block function for encryption applied to the head of the plaintext stream, `xs`, and `IV`: this fact follows from the specification of the general encryption function and the premise in the `general_MoO_valid_stream_version` predicate stating that the output of the general encryption function applied to the plaintext stream is the ciphertext stream.

After rewriting the `H_css_S_j` above using this relation between `xs` and `cs`, we can apply it to solve the goal: the proof is done.

The General Validity Theorem Using Lists Having proven validity of the general encryption scheme using streams, we can now prove validity of the corresponding scheme using lists.

```
Corollary general_validity_theorem_list_version :
  forall (general_MoO_enc_lv general_MoO_dec_lv :
            general_MoO_enc_dec_Type_list_version)
         (block_fun_enc block_fun_dec : block_fun_Type)
         (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
         (block_enc block_dec : block_enc_dec_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    specification_of_general_MoO_decryption_function_list_version
      general_MoO_dec_lv →
    block_functions_inverse block_fun_enc block_fun_dec
      block_enc block_dec →
    feedback_functions_output_equal_values
      feedback_fun_enc feedback_fun_dec block_fun_enc
      block_enc block_dec →
    general_MoO_valid_list_version
      general_MoO_enc_lv general_MoO_dec_lv
      block_fun_enc block_fun_dec
      feedback_fun_enc feedback_fun_dec
      block_enc block_dec.
```

The premises of this theorem are identical to the premises of the general validity theorem using streams except that the general encryption and decryption functions satisfy list versions of the specifications (see Section 6.8.2). The conclusion of the theorem states that the general scheme (using the block and feedback functions input to the theorem) using lists is valid using the `general_MoO_valid_list_version` predicate defined below.

```
Definition general_MoO_valid_list_version
           (general_MoO_enc_lv general_MoO_dec_lv :
              general_MoO_enc_dec_Type_list_version)
           (block_fun_enc block_fun_dec : block_fun_Type)
           (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV : block_n),
    general_MoO_dec_lv
      (general_MoO_enc_lv
         xss ks IV block_enc block_fun_enc feedback_fun_enc)
      ks IV block_enc_dec block_fun_dec feedback_fun_dec =
    xss.
```

The predicate captures the concept of validity: encrypting and the decrypting any plaintext block with the same key and IV yields the block.

The theorem is a corollary of the validity theorem for the general scheme using streams in the section above. In Section 6.8.2, we have lemmas stating that the regular and alternative specifications of the encryption respectively decryption function using lists are equivalent. Exploiting these equivalences, we transform the goal in the proof into the goal below in terms of the general encryption functions using streams.

```
cut_of_stream
  (general_MoO_dec_sv_v0
     (extend_list_into_Stream
        (cut_of_stream
           (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV
              block_enc block_fun_enc feedback_fun_enc)
           (length xss))) ks IV block_dec block_fun_dec feedback_fun_dec)
  (length
     (cut_of_stream
        (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV
           block_enc block_fun_enc feedback_fun_enc)
        (length xss))) = xss
```

We want to be able to apply the general validity theorem using streams. To be able to do so, we must transform the goal into one matching the conclusion of the `general_MoO_valid_stream_version` predicate (defined in the section above) in the conclusion of that theorem.

First, we use the `about_length_of_cut_of_stream` lemma in Section 3.4.3. This lemma states that the length of the list output by the `cut_of_stream` operator is equal to the natural number input to the operator.

Secondly, we use the `cutting_away_extension_yields_original_list` lemma in Section 3.4.4. This lemma says that first extending a list via the `extend_list_into_Stream` operator and then cutting the resulting stream with the `cut_of_stream` operator yields the original list if the natural number input to the last operator is equal to the length of the list.

At this point in the proof, we have the goal below.

```
cut_of_stream
  (general_MoO_dec_sv_v0
     (extend_list_into_Stream
        (cut_of_stream
           (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV
              block_enc block_fun_enc feedback_fun_enc)
           (length xss))) ks IV block_dec block_fun_dec feedback_fun_dec)
  (length xss) =
  cut_of_stream (extend_list_into_Stream xss) (length xss)
```

We wish to remove `cut_of_stream` and `extend_list_into_Stream` operators applied to the output of the general encryption function and before the general decryption function is applied. Hence, we use the lemma below.

```
Lemma repl_ciphertext_blocks_after_cut_w_zero_blocks_n_does_not_change_corr_plaintext_cut_general :
  forall general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version,
    specification_of_general_MoO_decryption_function_stream_version
      general_MoO_dec_sv →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type)
           (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type)
           (m : nat),
      cut_of_stream
        (general_MoO_dec_sv
```

```
      (extend_list_into_Stream (cut_of_stream css m))
      ks IV block_dec block_fun feedback_fun)
    m =
  cut_of_stream
    (general_MoO_dec_sv css ks IV block_dec block_fun feedback_fun)
    m.
```

This lemma states that for any cut of any length `m` of the output of the general decryption function, it does not matter if the ciphertext stream input to the decryption function is first cut to the length of `m` and then extending into a stream—effectively replacing all the blocks after the cut with all-zero blocks of length `n`. The lemma is proven by induction in the length `m`.

After using the lemma above and applying the `bisimilarity_implies_equal_cuts_of_stream` lemma (in Section 3.4.3) saying that bisimilar streams have equal cuts for any cut length, we arrive at the goal below.

```
bisimilar_Stream_block_n
  (general_MoO_dec_sv_v0
    (general_MoO_enc_sv_v0
      (extend_list_into_Stream xss) ks IV block_enc
      block_fun_enc feedback_fun_enc)
    ks IV block_dec block_fun_dec feedback_fun_dec)
  (extend_list_into_Stream xss)
```

Now, the conclusion of the `general_MoO_valid_stream_version` predicate matches the goal, and we can apply the general validity theorem using streams.

Applying the general theorem using streams, we have to show that its premises hold. The first two premises state that the general encryption and decryption functions input to the theorem must satisfy the corresponding specifications. These premises hold since we use the `general_MoO_enc_sv_v0` and `general_MoO_dec_sv_v0` implementations defined in Section 6.8.1: these implementations have been proven to fit the specifications.

The two next premises state that the input block functions must be inverse and that the input feedback functions must output equal values. Since we input the block and feedback functions introduced and assumed to have these properties by the corollary we are proving, these premises hold, too.

The last premise of the applied theorem is the premise of the `general_MoO_valid_stream_version` predicate in the conclusion of the theorem. Applying the theorem to the goal shown above, showing this premise amount to showing the goal below.

```
bisimilar_Stream_block_n
  (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV block_enc
    block_fun_enc feedback_fun_enc)
  (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV block_enc
    block_fun_enc feedback_fun_enc)
```

Here, we use the `bisimilarity_is_a_reflexive_relation` lemma in Section 3.4.1: bisimilarity is a reflexive relation.

Now, all premises of the applied theorem have been shown, and the proof is done.

6.8.3.2 The General Plaintext Change Propagation Theorems

For some modes of operation, changes of the plaintext blocks are propagated when encrypting, i.e., when a single plaintext block is changed, the corresponding ciphertext block and all subsequent ones are changed, too. While this property is interesting by itself, it is also a prime motivator for using these particular modes to construct MAC functions: here, a change of a block in some message should be reflected by the MAC value being changed also. This property is desirable when using MAC schemes for checking integrity of messages.

We have two general plaintext change propagation theorems: one for the general encryption function using streams in Section 6.8.1 and one for the general encryption function using lists in Section 6.8.2.

The General Plaintext Change Propagation Theorem Using Streams We state the general plaintext change propagation theorem using streams.

```
Theorem general_plaintext_change_propagation_theorem_stream_version :
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    block_fun_invertible_wrt_IV_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    general_plaintext_change_propagation_stream_version
      general_MoO_enc_sv block_fun feedback_fun block_enc.
```

The general encryption function using streams propagates plaintext changes if the block and feedback functions for encryption—in conjunction with the underlying block encryption function—are invertible with regards to both the plaintext blocks and the IVs or feedback blocks.

The conclusion of the theorem uses the `general_plaintext_change_propagation_stream_version` predicate defined below.

```
Definition general_plaintext_change_propagation_stream_version
           (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_from_index_and_equal_before
      (general_MoO_enc xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

This predicate is true for a general encryption function—in conjunction with block and feedback functions and an underlying block encryption function—if changing the $\mathtt{i}^{\text{th}}$ plaintext block results in all ciphertext blocks from that point forward being changed, too.

The proof of the theorem is done by induction in the `i` index introduced by the predicate above: the base case is proven by applying the `plaintext_change_propagation_first_block_changed` helping lemma below considering the case where the first plaintext block is changed while the inductive case is proven by unfolding the general encryption function using streams and applying the induction hypothesis.

The `plaintext_change_propagation_first_block_changed` lemma states that a change of the first plaintext block is propagated by the general encryption function using streams under the same assumptions as in the theorem above.

```
Lemma plaintext_change_propagation_first_block_changed :
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
```

```
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    block_fun_invertible_wrt_IV_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    general_plaintext_change_propagation_first_block_changed
      general_MoO_enc_sv block_fun feedback_fun block_enc.
```

In the conclusion of the lemma above, we use the predicate defined below.

```
Definition general_plaintext_change_propagation_first_block_changed
           (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV :  block_n),
    streams_different_at_index_only xss_1 xss_2 0 →
    streams_different_at_all_indices
      (general_MoO_enc xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc xss_2 ks IV block_enc block_fun feedback_fun).
```

This predicate is similar to the `general_plaintext_change_propagation_stream_version` predicate above. Now, however, it is always the first plaintext block that is changed, and the ciphertext blocks are all changed, i.e., from the first block onward.

The lemma is proven by case analysis of the index introduced by the `streams_different_at_all_indices` predicate in the conclusion of the `general_plaintext_change_propagation_first_block_changed` predicate. In the base case, we show that the first pair of ciphertext blocks are different: this fact follows directly from the block function being invertible with regards to the plaintext block and the first pair of plaintext blocks being different.

The inductive case is more involved: it is proved by induction in `i'` where `i = S i'`. Hence, the base case of the induction part of the proof consists of showing that the second pair of ciphertext blocks are unequal. Here, we use the fact that the feedback function is invertible with regards to the plaintext block and that the block function is invertible with regards to the IV or feedback block: the fact that the feedback function is invertible enables us to show that the feedback blocks for encryption of the second plaintext block are different since the first blocks of each plaintext are different. Then, the fact that the block function is invertible implies that the second blocks of the ciphertexts are different. This implication follows from the fact that the second blocks of the plaintexts are equal (only the first plaintext blocks are different).

The inductive step (in the induction part of the proof) is taken by applying the `about_propagation` helping lemma below. This helping lemma states that if for some natural number `j` the $\texttt{j}^{\text{th}}$ and $(\texttt{S j})^{\text{th}}$ plaintext blocks are pairwise equal and the $\texttt{j}^{\text{th}}$ ciphertext blocks are different, then the $(\texttt{S j})^{\text{th}}$ ciphertext blocks are different, too. To take the inductive step, we have to show that ciphertext blocks number `i = S i'` are different from each other as a consequence of the induction hypothesis saying that the $(\texttt{i'})^{\text{th}}$ ciphertext blocks are unequal. We show that by applying the `about_propagation` lemma with `j` equal to `i'`. Hence, we just have to show that the $(\texttt{i'})^{\text{th}}$ and the $(\texttt{i = S i'})^{\text{th}}$ plaintext blocks are pairwise equal and that the $(\texttt{i'})^{\text{th}}$ ciphertext blocks are unequal. The latter is given by the induction hypothesis while the plaintext blocks being equal follows from the fact that the plaintext streams are only different at index 0.

The `about_propagation` lemma is shown below.

```
Lemma about_propagation :
  forall general_enc : general_MoO_enc_dec_Type_stream_version,
    specification_of_general_MoO_encryption_function_stream_version
      general_enc →
    forall (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
      block_fun_invertible_wrt_IV_feedback block_fun block_enc →
```

```
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    forall (xss_1 xss_2 : Stream block_n)
           (ks : block_k) (IV_1 IV_2 : block_n)
           (i : nat),
      streams_equal_at_index xss_1 xss_2 i →
      streams_equal_at_index xss_1 xss_2 (S i) →
      streams_different_at_index
        (general_enc xss_1 ks IV_1 block_enc block_fun feedback_fun)
        (general_enc xss_2 ks IV_2 block_enc block_fun feedback_fun)
        i →
      streams_different_at_index
        (general_enc xss_1 ks IV_1 block_enc block_fun feedback_fun)
        (general_enc xss_2 ks IV_2 block_enc block_fun feedback_fun)
        (S i).
```

As explained above, the three latter premises of this lemma hold. The premises concerning the general encryption function using streams and the block and feedback functions are also premises of the `plaintext_change_propagation_first_block_changed` lemma we are proving: all premises hold.

We prove the lemma above by induction in the `i` index introduced by the lemma. In the base case, we have to show that the ciphertext streams are different at index 1, i.e., the second blocks are different. Here, we use the fact that the block function is invertible with regards to the IV or feedback block: we, therefore, have to show that the feedback blocks from the first block to the second are different for each plaintext stream. However, because the feedback function is invertible with regards to the IV or feedback block, we, in extension, have to show that the IVs (`IV_1` and `IV_2` in the lemma) are distinct. This fact can be shown by contradiction: if the IVs were equal, then, since the first plaintext blocks are equal also in the premises (remember, we are in the base case of `i`), the first ciphertext blocks would be equal, too: such an equality contradicts the premise stating that they are different (remember, `i` = 0 in the base case). Here, we implicitly use the fact that the block function is deterministic as is always the case.

In the inductive case, we have to show that the $(\texttt{S}\ (\texttt{S}\ \texttt{i}'))^{\text{th}}$ ciphertext blocks are distinct where `i` = `S i'`. Here, we unfold the `general_enc` function and the `stream_ref` operator using the corresponding unfolding lemmas for the inductive cases to transform the goal into one of showing that the $(\texttt{S}\ \texttt{i}')^{\text{th}}$ blocks of the ciphertext tail streams are different. This transformation enables us to apply the induction hypothesis below.

```
IHi' : forall (xss_1 xss_2 : Stream block_n) (IV_1 IV_2 : block_n),
       streams_equal_at_index xss_1 xss_2 i' →
       streams_equal_at_index xss_1 xss_2 (S i') →
       streams_different_at_index
         (general_MoO_enc_sv
            xss_1 ks IV_1 block_enc block_fun feedback_fun)
         (general_MoO_enc_sv
            xss_2 ks IV_2 block_enc block_fun feedback_fun)
         i' →
       stream_ref
         (general_MoO_enc_sv
            xss_1 ks IV_1 block_enc block_fun feedback_fun)
         (S i') <>
       stream_ref
         (general_MoO_enc_sv
            xss_2 ks IV_2 block_enc block_fun feedback_fun)
         (S i')
```

The conclusion of the induction hypothesis consists of the unfolded `streams_different_at_index` predicate defined in Section 6.7.1.

We apply the hypothesis to the tails of the plaintext streams, i.e., the tails of `xss_1` and `xss_2`, in the lemma. We, now, have to show that the premises of the induction hypothesis hold.

The two first premises of the induction hypothesis say that the streams input to the hypothesis are different at indices `i'` and `S i'`. These premises follow from the premises of the lemma being proved since if two streams are different at an index equal to some successor of a natural number, then their respective tails are different at the index equal to that natural number. The premises of the lemma state that the plaintext streams, `xss_1` and `xss_2`, are different at indices `i = S i'` and `S i = S (S i')` which, by the discussion above, implies that the plaintext tails input to the induction hypothesis are different at indices `i'` and `S i'`.

The last premise of the induction hypothesis states that the $(\texttt{i'})^{\text{th}}$ ciphertext blocks corresponding to the plaintext tails input to the hypothesis are different. These ciphertext streams are the tails of the original ciphertext blocks since the IVs input to the hypothesis are the feedback blocks from the first to the second blocks of the original plaintext streams. Hence, we just have to show that the $(\texttt{S i'})^{\text{th}}$ blocks of the original ciphertext streams are different: however, this fact is a premise of the lemma.

All premises of the induction hypothesis hold, and the proof is done.

The General Plaintext Change Propagation Theorem Using Lists We prove the general plaintext change propagation theorem using lists as a corollary of the corresponding theorem using streams in the section above.

```
Corollary general_plaintext_change_propagation_theorem_list_version :
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    block_fun_invertible_wrt_IV_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    general_plaintext_change_propagation_list_version
      general_MoO_enc_lv block_fun feedback_fun block_enc.
```

Here, the premises are the same as for the corresponding theorem for the scheme using streams except that the general encryption function fits the specification of the general encryption function using lists instead of the one using streams.

The corollary uses the `general_plaintext_change_propagation_list_version` predicate defined below in its conclusion.

```
Definition general_plaintext_change_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_from_index_and_equal_before
      (general_MoO_enc_lv xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc_lv xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

This predicate is similar to the `general_plaintext_change_propagation_stream_version` predicate in the section above. It uses the `lists_different_at_index_only` and `lists_different_from_index_and_equal_before`

predicates defined in Section 6.7.2 corresponding to the streams_different_at_index_only and streams_different_from_index_and_equal_before predicates used in the general_plaintext_change_propagation_stream_version predicate: now, we consider *lists* different at index i only and *lists* different from index i onwards and equal before instead of streams.

The proof of the corollary consists primarily of applying the plaintext change propagation theorem using streams in the section above. However, since that theorem concerns the encryption of plaintext *streams*, we have to translate the goal involving ciphertext lists in the proof to another goal involving ciphertext streams. Here, we use the specifications_of_general_MoO_encryption_function_list_version_are_equivalent lemma in Section 6.8.2 stating that any function satisfying the specification_of_general_MoO_encryption_function_list_version specification also satisfies the alternative specification_of_general_MoO_encryption_function_list_version_alt specification and vice versa. This alternative specification specifies the general encryption function using lists in terms of the general encryption function using streams (and the cut_of_stream and extend_list_into_Stream operators defined in Sections 3.4.3 and 3.4.4).

Unfolding the general_MoO_enc_lv function via the latter specification using the general_MoO_enc_sv_v0 function (defined in Section 6.8.1) fitting the specification of the general encryption function using streams, we get the goal below.

```
lists_different_from_index_and_equal_before
  (cut_of_stream
     (general_MoO_enc_sv_v0 (extend_list_into_Stream xss_1) ks IV
        block_enc block_fun feedback_fun) (length xss_1))
  (cut_of_stream
     (general_MoO_enc_sv_v0 (extend_list_into_Stream xss_2) ks IV
        block_enc block_fun feedback_fun) (length xss_2)) i
```

Now, we have a goal involving ciphertext streams—the extend_list_into_Stream operator extends the input list into a stream. We use the streams_different_from_index_and_equal_before_implies_cuts_of_equal_lengths_same lemma in Section 6.7.2. This lemma states that if two streams are different from index i and equal before, then so are cuts of equal lengths of these streams. Using the lemma and the premise stating that the plaintext lists xss_1 and xss_2 have equal lengths, we transform the goal into the goal below.

```
streams_different_from_index_and_equal_before
  (general_MoO_enc_sv_v0
     (extend_list_into_Stream xss_1) ks IV block_enc
     block_fun feedback_fun)
  (general_MoO_enc_sv_v0
     (extend_list_into_Stream xss_2) ks IV block_enc
     block_fun feedback_fun)
  i
```

Now, we can apply the stream version of the general theorem from the section above with the input plaintext streams being extend_list_into_Stream xss_1 and extend_list_into_Stream xss_2, the stream encryption function being general_MoO_enc_sv_v0, and the i index being the index of the same name introduced by the general_plaintext_change_propagation_list_version in the conclusion of the corollary we are proving. The rest of the proof, hence, consists of showing that the premises of that theorem hold.

First, the general_MoO_enc_sv_v0 function satisfies the specification of the general encryption function using streams. Secondly, the premises involving the block and feedback functions are given as premises in the corollary that we are proving.

Additionally, the general_plaintext_change_propagation_stream_version predicate in the conclusion of the applied theorem has a premise that we must show: the input plaintext streams, extend_list_into_Stream xss_1 and extend_list_into_Stream xss_2, are assumed to be different at index i only. Here, we use the extending_lists_of_equal_lengths_preserves_difference_at_index lemma from Section 6.7.2: if

two lists of equal lengths are different at index i only, then the streams obtained by extending the lists using the extend_list_into_Stream operator are different at index i only, too. That xss_1 and xss_2 have equal lengths and that they are different at index i only are given as premises of the general_plaintext_change_propagation_list_version predicate in the conclusion of the corollary. Thus, all the premises of the applied theorem hold, and the proof is done.

6.8.3.3 The General Plaintext Change Non-Propagation Theorems

For some modes of operation, changes of the plaintext blocks are not propagated when encrypting, i.e., *only* the corresponding ciphertext block is changed, and, in particular, the other ciphertext blocks remain unchanged. This property holds for the f, CTR, and BENC modes defined in Section 6.4.4: their feedback functions ignore the plaintext blocks, and, hence, the subsequent ciphertext blocks are not influenced by a plaintext block changing.

In this section, we show that a mode of operation does not propagate plaintext changes when encrypting if the feedback function for encryption ignores the plaintext block, i.e., it is independent of the plaintext block, and the block function for encryption is invertible with regards to the plaintext block. The latter property is used to show that the ciphertext block corresponding to the changed plaintext block is still changed. In the section below, we show the plaintext change non-propagation property for the general encryption function using streams while we, in Section 6.8.3.3, show the property for the general encryption function using lists.

The General Plaintext Change Non-Propagation Theorem Using Streams In this section, we show that the general encryption function using streams does not propagate plaintext changes under the assumptions about the block and feedback functions for encryption mentioned above. First, we state the theorem.

```
Theorem general_plaintext_change_non_propagation_theorem_stream_version :
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    feedback_fun_ignores_plaintext feedback_fun block_enc →
    general_plaintext_change_non_propagation_stream_version
      general_MoO_enc_sv block_fun feedback_fun block_enc.
```

The general_plaintext_change_non_propagation_stream_version predicate defined below is used in the conclusion of the theorem.

```
Definition general_plaintext_change_non_propagation_stream_version
           (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only
      (general_MoO_enc xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

This predicate states that if two plaintext streams are different at index i only, i.e., only the i^{th} plaintext blocks are different, then the corresponding ciphertext streams are only different at index i also.

In the following, we only cursory describe the proof of the theorem. The proof is more technical, in part because of the definition of the `streams_different_at_index_only` predicate (in Section 6.7.1) involving a biimplication.

We prove the theorem above by induction in the `i` index introduced by the `general_plaintext_change_non_propagation_stream_version` predicate. First, we show the base case: if only the first pair of plaintext blocks are unequal, then only the first pair of the corresponding ciphertext blocks are unequal.

By unfolding the general encryption function, we can transform the problem of showing that the first ciphertext blocks are unequal into one of showing that the block function outputs different ciphertext blocks when applied to different plaintext blocks (when the other inputs are equal), i.e., the block function is invertible with regards to the plaintext block: this property is one of the premises of the theorem so we apply that premise.

In the base case, we also have to show that the blocks of the tails of the ciphertext streams are pairwise equal. Here, we use the fact given as a premise that the feedback function ignores the plaintext block. Since the same IV is used when encrypting the two plaintext streams, the feedback blocks, by induction, are equal at equal indices. Hence, the rest of the ciphertext blocks are pairwise equal since the rest of the plaintext blocks are so: all inputs to the block function are equal at each index greater than or equal to 1 for the two plaintext streams.

In the inductive case, we have to show that only the $(\texttt{S i'})^{\text{th}}$, where `S i' = i`, ciphertext blocks are different. However, we are given an induction hypothesis that states that if two plaintext streams are different at index `i'`, then the corresponding ciphertext streams are different at that index, too. We unfold the general encryption function in the goal to reveal the corecursive applications of the function to the tails of the plaintext streams. Then we transform the goal of showing the ciphertext streams being different at index `S i'` to a goal of showing that the corresponding tail streams are different at index `i'` by unfolding the `stream_ref` operator which is used to access the elements of the streams.

The induction hypothesis requires that equal IVs are used for both encryptions of the tails of the plaintext streams. Thus, we have to show that the feedback blocks given from the first to the second plaintext blocks, i.e., to the first blocks of the tails of the original plaintexts, are equal for each plaintext stream. Here, we use the fact that the feedback function ignores the plaintext blocks. Hence, since the original IVs are equal, the feedback blocks given to the second plaintext blocks are equal even though the first plaintext blocks are unequal. We apply the induction hypothesis, and the proof is done.

The General Plaintext Change Non-Propagation Theorem Using Lists We prove the general plaintext change non-propagation theorem using lists as a corollary of the corresponding theorem using streams in the section above.

```
Corollary general_plaintext_change_non_propagation_theorem_list_version :
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
        general_MoO_enc_lv →
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    feedback_fun_ignores_plaintext feedback_fun block_enc →
    general_plaintext_change_non_propagation_list_version
      general_MoO_enc_lv block_fun feedback_fun block_enc.
```

The premises of this corollary are the same as for the plaintext change non-propagation theorem using streams except that now we use a general encryption function using lists instead of streams. The corollary uses the predicate defined below in its conclusion.

```
Definition general_plaintext_change_non_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
```

```
        (block_fun : block_fun_Type)
        (feedback_fun : feedback_fun_Type)
        (block_enc : block_enc_dec_Type) :=
forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
  length xss_1 = length xss_2 →
  lists_different_at_index_only xss_1 xss_2 i →
  lists_different_at_index_only
    (general_MoO_enc_lv xss_1 ks IV block_enc block_fun feedback_fun)
    (general_MoO_enc_lv xss_2 ks IV block_enc block_fun feedback_fun)
    i.
```

This predicate is very similar to the `general_plaintext_change_non_propagation_stream_version` predicate defined in the section above. The difference is that we now consider lists instead of streams which spurs the use of predicates involving lists instead of streams. Also, the plaintext lists are assumed to have equal lengths—a property not applicable to streams. However, the idea behind the predicate remains the same: if two plaintexts are blockwise different only at some index `i`, then the corresponding ciphertexts are also only blockwise different at index `i`.

The proof of the corollary is similar to the proof of the list version of the general plaintext change propagation theorem (in Section 6.8.3.2) which itself is a corollary of the stream version of the general plaintext change propagation theorem in Section 6.8.3.2: by exploiting the `specifications_of_general_MoO_encryption_function_list_version_are_equivalent` lemma in Section 6.8.2, we can unfold the general encryption function in the corollary via the alternative specification of the general encryption function using lists (see Section 6.8.2) to get a goal in terms of a function satisfying the specification of the general encryption function using streams and where the plaintext lists have been extended in to streams. This transformation of the goal enables us to apply the general plaintext change non-propagation theorem using streams.

After the abovementioned transformation and after using the premise stating that the two plaintext lists have equal lengths, we have the goal below—we use the `general_MoO_enc_sv_v0` function (defined in Section 6.8.1) fitting the specification of the general encryption function using streams.

```
lists_different_at_index_only
  (cut_of_stream
     (general_MoO_enc_sv_v0 (extend_list_into_Stream xss_1) ks IV
        block_enc block_fun feedback_fun) (length xss_2))
  (cut_of_stream
     (general_MoO_enc_sv_v0 (extend_list_into_Stream xss_2) ks IV
        block_enc block_fun feedback_fun) (length xss_2)) i
```

Now, we apply the `streams_different_at_index_implies_cuts_of_equal_lengths_different_at_index_if_index_in_cut` helping lemma in Section 6.7.2. This lemma states that if the two streams are different at some particular index only, then the respective cuts of equal lengths of these streams are different at that index only, too. However, it is assumed that the blocks at the particular index are not cut away. By applying, we transform the goal into the one below involving the `streams_different_at_index_only` predicate defined in Section 6.7.1.

```
streams_different_at_index_only
  (general_MoO_enc_sv_v0
     (extend_list_into_Stream xss_1) ks IV block_enc
     block_fun feedback_fun)
  (general_MoO_enc_sv_v0
     (extend_list_into_Stream xss_2) ks IV block_enc
     block_fun feedback_fun)
  i
```

However, to be able to apply the helping lemma, the `i` index (introduced by the predicate in the conclusion of the corollary we are proving) must be less than `length xss_2`. Hence, we consider the dichotomous

cases of this requirement holding and of this requirement not holding. If the requirement does not hold, then the plaintext lists must (according to the premises of the corollary) be different at an index out of bounds for each of the lists. In that case, we have a contradiction, and we prove the corollary using the `False_ind` lemma from the standard library. This lemma says that the `False` proposition implies any proposition.

If the complementary case, i.e., the case where `i < length xss_2`, holds, then we have to show the latter goal above. Here, we apply the general plaintext change non-propagation theorem using streams (in the section above) with the input plaintext streams being `extend_list_into_Stream xss_1` and `extend_list_into_Stream xss_2`, the input general encryption function using streams being `general_MoO_enc_sv_v0`, and the input `i` index being the index of the same name in the goal above.

The rest of the proof is showing that the premises of the applied theorem hold. The first premise holds since `general_MoO_enc_sv_v0` fits the specification of the general encryption function using streams. The next two premises involving the block and feedback functions for encryption are also premises of the corollary we are proving and, therefore, hold, too.

The `general_plaintext_change_non_propagation_stream_version` predicate in the conclusion of the applied theorem contains an additional premise: the input plaintext streams are different at index `i` only. Here, we use the premise of the `general_plaintext_change_non_propagation_list_version` predicate in the conclusion of the corollary stating that the plaintext lists `xss_1` and `xss_2` are different at index `i` only. Extending these lists into stream by using the `extend_list_into_Stream` operator (defined in Section 3.4.4) preserves this property. This fact follows from the `extending_lists_of_equal_lengths_preserves_difference_at_index` lemma in Section 6.7.2.

All premises of the applied theorem hold: the proof is done.

6.8.3.4 The General IV Change Propagation Theorems

In this section, we state and prove the general IV change propagation theorems. For the general encryption scheme, it holds that if the input block and feedback functions for encryption both are invertible with regards to the IVs or feedback blocks, then, for any plaintext and key, changing the IV changes all blocks in the corresponding ciphertext.

In Section 6.8.3.4 below, we prove the above property for the general encryption scheme using streams while we prove the property for the corresponding scheme using lists in Section 6.8.3.4.

The General IV Change Propagation Theorem Using Streams In this section, we prove the stream version of the general IV propagation theorem.

```
Theorem general_IV_change_propagation_theorem_stream_version :
  forall (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc →
    block_fun_invertible_wrt_IV_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    general_IV_change_propagation_stream_version
      general_MoO_enc block_fun feedback_fun block_enc.
```

The theorem above states that the stream version of the general encryption function (specified in Section 6.8.1) propagates changes of the IV if applied to block and feedback functions (for encryption) that are invertible with regards to the IVs or feedback blocks input to them. The theorem uses the `general_IV_change_propagation_stream_version` predicate defined below in its conclusion.

```
Definition general_IV_change_propagation_stream_version
```

```
        (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
        (block_fun : block_fun_Type)
        (feedback_fun : feedback_fun_Type)
        (block_enc : block_enc_dec_Type) :=
forall (xss : Stream block_n) (ks : block_k) (IV_1 IV_2 : block_n),
  IV_1 <> IV_2 →
  streams_different_at_all_indices
    (general_MoO_enc_sv xss ks IV_1 block_enc block_fun feedback_fun)
    (general_MoO_enc_sv xss ks IV_2 block_enc block_fun feedback_fun).
```

This predicate captures exactly the notion that if the IV is changed, then all ciphertext blocks are changed, too.

The theorem is proven by induction in the `i` index introduced by the `streams_different_at_all_indices` predicate (defined in Section 6.7.1) in the conclusion in the predicate above: we prove by induction that the two ciphertext streams corresponding to the same plaintext stream—but with different IVs—are unequal at each index.

In the base case where `i = 0`, we have to show that the first blocks of the ciphertext streams are different. Here, we use the premise stating that the block function is invertible with regards to the IV or feedback block: since the IVs are different for the two encryptions of the plaintext stream, the block function is given different IVs when computing the first block of each ciphertext stream, and, therefore, since the block function is invertible with regards to the IV, the resulting first blocks of the ciphertext streams are different.

In the inductive case where `i = S i'` for some natural number `i'`, we have to show that the two ciphertext streams are different at index `i = S i'` given the induction hypothesis below stating that any two ciphertext streams (computed with two different IVs) corresponding to any plaintext stream are different at index `i'`.

```
IHi' : forall (xss : Stream block_n) (IV_1 IV_2 : block_n),
       IV_1 <> IV_2 →
       stream_ref
         (general_MoO_enc
            xss ks IV_1 block_enc block_fun feedback_fun) i' <>
       stream_ref
         (general_MoO_enc
            xss ks IV_2 block_enc block_fun feedback_fun) i'
```

In the hypothesis, the `streams_different_at_index` predicate in its conclusion has been unfolded. That predicate stems from the `streams_different_at_all_indices` predicate in the conclusion of the `general_IV_change_propagation_stream_version` predicate in the conclusion of the theorem we are proving.

By unfolding the general encryption function and the `stream_ref` operator defined in Section 3.4.2, we turn the goal of showing that the two ciphertext streams corresponding to the plaintext stream `xss = Cons xs xss'` are different at index `S i'` into a goal of showing that two ciphertext streams corresponding to the plaintext tail `xss'` are different at index `i'`. This transformation allows us to apply the induction hypothesis above to `xss'` and the feedback blocks given from the first to the second blocks when encrypting the original `xss` plaintext stream with the original IVs (`feedback_fun xs ks IV_1 block_enc` and `feedback_fun xs ks IV_2 block_enc`): the rest of the proof consists of showing that the premise of the hypothesis holds.

The premise of the induction hypothesis states that the feedback blocks input to the hypothesis are unequal. Now, we use the premise of the corollary saying that the feedback function (for encryption) is invertible with regards to the IVs or feedback blocks: since the original IVs are different, then so are the feedback blocks given as input to the hypothesis since these are output by the feedback function on input the original IVs.

The General IV Change Propagation Theorem Using Lists In this section, we prove the list version of the general IV propagation theorem.

```
Corollary general_IV_change_propagation_theorem_list_version :
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    block_fun_invertible_wrt_IV_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc →
    general_IV_change_propagation_list_version
      general_MoO_enc_lv block_fun feedback_fun block_enc.
```

The theorem above states that if the block and feedback functions (for encryption) are invertible with regards to the IV or feedback block, then the general encryption function using lists propagates changes of the IV when applied to these block and feedback functions. The theorem uses the `general_IV_change_propagation_list_version` predicate defined below in its conclusion.

```
Definition general_IV_change_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    lists_different_at_all_indices
      (general_MoO_enc_lv xss ks IV_1 block_enc block_fun feedback_fun)
      (general_MoO_enc_lv xss ks IV_2 block_enc block_fun feedback_fun).
```

This predicate resembles the `general_IV_change_propagation_list_version` predicate in the section above. Now, however, we consider plaintext lists instead of plaintext streams. The notion is the same: changing the IV changes all ciphertext blocks.

The theorem is a corollary of the stream version of the general IV change propagation theorem in the section above However, the goal that we have to show in the proof involves the general encryption function using lists. Therefore, we use the `specifications_of_general_MoO_encryption_function_list_version_are_equivalent` lemma in Section 6.8.2 to transform the goal into one involving the general encryption function using streams. After unfolding via the alternative specification (also in Section 6.8.2), we have to show the goal below—we use the `general_MoO_enc_sv_v0` implementation defined in Section 6.8.1 of the general encryption function using streams.

```
  lists_different_at_all_indices
    (cut_of_stream
       (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV_1
          block_enc block_fun feedback_fun) (length xss))
    (cut_of_stream
       (general_MoO_enc_sv_v0 (extend_list_into_Stream xss) ks IV_2
          block_enc block_fun feedback_fun) (length xss))
```

Here, the `lists_different_at_all_indices` predicate defined in Section 6.7.2 comes from the conclusion of the `general_IV_change_propagation_list_version` predicate in the conclusion of the corollary we are proving. We can now apply the `streams_different_at_all_indices_implies_same_for_cuts_of_equal_lengths` lemma in Section 6.7.2. This lemma says that cuts of equal lengths of two streams that are different at all indices are different at all (in-bounds) indices.

By applying the lemma, we transform the goal into the one below.

```
  streams_different_at_all_indices
    (general_MoO_enc_sv_v0
       (extend_list_into_Stream xss) ks IV_1 block_enc
       block_fun feedback_fun)
    (general_MoO_enc_sv_v0
       (extend_list_into_Stream xss) ks IV_2 block_enc
       block_fun feedback_fun)
```

We can now apply the stream version of the general IV change propagation theorem of which the theorem we are currently proving is a corollary: the rest of the proof consists of showing that the premises of the applied theorem hold. Here, the plaintext stream input to the theorem is `extend_list_into_Stream xss`.

The first premise of the applied theorem says that the used general encryption function using lists must fit the corresponding specification. We use the `general_MoO_enc_sv_v0` function defined in Section 6.8.1: this function is proven to fit the specification so the first premise holds.

The second and third premises state that the block and feedback functions (for encryption) must be invertible with regards to the IVs or the feedback blocks. These premises are also premises of the corollary we are proving so they hold.

What remains to be shown is that the premise of the `general_IV_change_propagation_stream_version` predicate in the conclusion of the applied theorem holds: we must show that the IVs input to the predicate are different. However, the IVs input to the predicate are exactly those IVs introduced by the `general_IV_change_propagation_list_version` predicate in the conclusion of the corollary we are proving. Hence, since this predicate asserts that the IVs introduced by it are different, the premise of the `general_IV_change_propagation_stream_version` predicate holds.

6.8.3.5 The General MAC Propagation Theorem

The plaintext change propagating modes of operation can be used to construct MAC (Message Authentication Code) functions: the output of such a MAC function, the MAC value, is defined to be the last block of the ciphertext resulting from encrypting the message consisting of blocks of length n using the encryption function of the mode of operation. If the encryption function propagates plaintext changes, a change in one of the blocks of the message implies a change of the output MAC value since all ciphertext blocks from that point forward and, in particular, the last ciphertext block are changed, too. This property is desirable as we want to be able to use the MAC values to check data integrity: recalculating the MAC value of a message and comparing it to an already computed one can give some measure of confidence in the integrity of the message. However, we cannot avoid collisions: the codomain is only the set of different blocks of length n while the domain is the infinite set of all messages consisting of blocks of length n. Therefore, the MAC value not changing does not perfectly imply that the message has not been changed: two or more changes (in different blocks) of the message could result in the MAC value being the same as before these changes.

In this section, we consider the general MAC propagation theorem stating that the general MAC function propagates changes in the blocks of the input messages under certain assumptions. First, we state the theorem.

```
Corollary general_MAC_changed_block_implies_different_MAC_value :
  forall (general_MAC : general_MAC_Type)
         (block_enc : block_enc_dec_Type)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MAC general_MAC →
    feedback_fun_preserves_length_n feedback_fun block_enc →
    block_fun_invertible_wrt_plaintext block_fun block_enc →
    block_fun_invertible_wrt_ivs_feedback block_fun block_enc →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc →
```

```
    feedback_fun_invertible_wrt_ivs_feedback feedback_fun block_enc →
    general_MAC_propagation
      general_MAC block_enc block_fun feedback_fun.
```

The premises concerning the block and feedback functions are identical in this corollary and the general plaintext change propagation theorems in Section 6.8.3.2. This relation between the two theorems is no coincidence: if the encryption function of a mode of operation propagates plaintext changes, then the corresponding MAC function is propagating, too, in the sense that a change of a single block of the message results in the MAC value changing also.

The `general_MAC_propagation` predicate is used in the conclusion of the corollary.

```
Definition general_MAC_propagation
           (general_MAC_fun : general_MAC_fun_Type)
           (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    general_MAC_fun xss_1 ks block_enc block_fun feedback_fun <>
    general_MAC_fun xss_2 ks block_enc block_fun feedback_fun.
```

If this predicate is true, then if any two messages of equal lengths (a change of a block does not alter the number of blocks) are different at some index `i` only, the corresponding MAC values are unequal. We require that the plaintexts are only different in one blocks since more than one block changed might cancel out the effect of propagation to the last block and result in the MAC value remaining unchanged.

We prove the theorem above as a corollary of the general plaintext change propagation theorem using lists in Section 6.8.3.2. We need to show that the last blocks of the ciphertext lists corresponding to the two input plaintext lists, i.e., the MAC values, are different. Here, the general plaintext change propagation theorem using lists comes to our aid: if any change of a block is propagated to the end of the ciphertext, it is, in particular, propagated to the the last block changing the MAC value.

In the proof, we unfold the general MAC function via its specification. To do so, we need to provide an implementation of the general encryption function using lists: we use the `general_MoO_enc_lv_v0` function defined in Section 6.8.2. After the unfolding, we need to show the goal below.

```
  last_block
    (general_MoO_enc_lv_v0 xss_1 ks zero_block_n block_enc block_fun
       feedback_fun) <>
  last_block
    (general_MoO_enc_lv_v0 xss_2 ks zero_block_n block_enc block_fun
       feedback_fun)
```

We now apply the `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma in Section 6.7.2. This lemma states that if two lists (of blocks of length `n`) of equal lengths are different from some in-bounds index `i` and onwards (and equal before), then the last blocks of the lists are different.

After applying the lemma, we have to show that the premises of the lemma hold. The goal corresponding to the first of them is shown below.

```
  lists_different_from_index_and_equal_before
    (general_MoO_enc_lv_v0 xss_1 ks zero_block_n block_enc block_fun
       feedback_fun)
    (general_MoO_enc_lv_v0 xss_2 ks zero_block_n block_enc block_fun
       feedback_fun) i
```

To solve this goal, we apply the general plaintext change propagation theorem using lists from Section 6.8.3.2 directly: all premises in that theorem (including the ones in the predicate in the conclusion of

the theorem) are also premises in the corollary we are proving except the premise saying that the input general encryption function using lists fits the corresponding specification. This premise, however, holds since the `general_MoO_enc_lv_v0` function has been proven to have this property.

The second premise of the applied `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma says that the lengths of the ciphertext lists corresponding to the plaintext lists `xss_1` and `xss_2` are equal—remember, the lemma is applied to the ciphertext lists. Here, we use the lemma below.

```
Lemma
  general_MoO_encryption_function_list_version_preserves_number_of_blocks :
  forall general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version,
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
      length (general_MoO_enc_lv xss ks IV block_enc block_fun feedback_fun) =
      length xss.
```

This lemma states that encrypting a plaintext list with the general encryption function using lists results in a ciphertext list of the same length as the plaintext list, i.e., the number of blocks is preserved. The lemma is proven by induction in the plaintext list `xss`. After using the lemma, we just have to show that the plaintext lists have equal lengths: that, however, is given as a premise in the `general_MAC_propagation` predicate in the conclusion of the corollary we are proving.

The third and last premise of the `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma states that the `i` index is less than the length of the ciphertext corresponding to the plaintext list `xss_1`. Again, we use the `general_MoO_encryption_function_list_version_preserves_number_of_blocks` lemma above: we now have to show that the index is less than the length of the plaintext list itself. To show that, we use the `lists_of_equal_lengths_different_at_index_only_implies_index_inbounds` lemma in Section 6.7.2. This lemma says that any index at which two lists (of blocks of length `n`) of equal lengths are different must be in-bounds. The lemma is proven by contradiction: the index cannot be out of bounds. Here, we use the `False_ind` lemma from the standard library. This lemma states that the `False` proposition implies any proposition.

We apply the `lists_of_equal_lengths_different_at_index_only_implies_index_inbounds` lemma to the `xss_1` and `xss_2` plaintext lists and the `i` index from the `general_MAC_propagation` predicate in the conclusion of the corollary we are proving. Thus, we have to show—according to the premises of that lemma—that the `xss_1` and `xss_2` plaintext lists have equal lengths and that they are different at index `i` only: both of these restrictions are included in the premises of the `general_MAC_propagation` predicate.

We have shown that all premises of the applied `lists_of_equal_lengths_different_from_index_and_equal_before_implies_last_blocks_different` lemma are satisfied, and the proof is done.

6.9 Procedure for Each Mode

In Section 6.10, we prove the validity and propagation theorems for each mode of operation and, for some of the modes, the propagation theorems for the corresponding MAC functions. In doing so, we follow a procedure which we describe in this section.

In Section 6.9.1, we look at the procedure as a whole while, in Sections 6.9.2 to 6.9.6, we describe how each theorem for each mode is proven as a corollary of the corresponding general theorem. The sections consider, in order, the validity theorems in Section 6.9.2, the plaintext change propagation theorems in Section 6.9.3, the plaintext change non-propagation theorems in Section 6.9.4, the IV change propagation theorems in Section 6.9.5, and, lastly, the MAC propagation theorem in Section 6.9.6.

6.9.1 The Procedure as a Whole

In this section, we describe the procedure that we follow when specifying and proving theorems about each mode of operation in Section 6.10. The descriptions of the proofs of the theorems are deferred to Sections 6.9.2 to 6.9.6.

The first part of the procedure consists of specifying the block and feedback functions of the particular mode. Here, we make use of the auxiliary functions in Section 6.6, especially the XOR-blocks-n function in Section 6.6.1. Additionally, we implement functions that satisfy these specifications. Additionally, we specify, in terms of the specifications of the block and feedback functions and the specifications of the general encryption and decryption functions in Sections 6.8.1 and 6.8.2, both the encryption and decryption functions of the mode and, if applicable, the corresponding MAC function: the encryption, decryption, and MAC functions of the modes are specified to be externally equal to the corresponding general functions applied to the block and feedback functions of the modes. For the encryption and decryption functions, both stream and list versions are specified.

Secondly, we prove a number of properties about the block and feedback functions. Doing so allows us to prove the theorems for the mode as corollaries of the general theorems in Section 6.8.3: each of these general theorems assumes a subset of the properties to hold for the functions.

- The general validity theorems in Section 6.8.3.1 assume that the block functions are inverse and that the feedback functions output equal values.
- The general plaintext change propagation theorems in Section 6.8.3.2 and the general MAC propagation theorem in Section 6.8.3.5 assume that the block and feedback functions for encryption are invertible with regards to both the plaintext blocks and the IVs or feedback blocks.
- The general plaintext change non-propagation theorems in Section 6.8.3.3 assume that the block function for encryption is invertible with regards to the plaintext block (even if non-propagating, the ciphertext block corresponding to the changed plaintext block is changed) and that the output of the feedback function for encryption is independent of the input plaintext blocks.
- Finally, the general IV change propagation theorems in Section 6.8.3.4 assume that the block and feedback functions for encryption are invertible with regards to the IVs or feedback blocks.

If the mode of operation propagates plaintext changes when encrypting, we prove the properties required about the block and feedback functions required to apply all the general theorems except for the plaintext change non-propagation theorems. If the mode does not propagate plaintext changes when encrypting, we prove the properties required for all the general theorems except for the plaintext change propagation theorems and the MAC propagation theorem—in this case, no corresponding MAC function is specified or defined since changing a block in a message (except for the last block) does not alter the MAC value output by a MAC function built upon such a mode.

We show that the above properties hold for the implementations of the block and feedback functions. Since these functions satisfy the specifications, we can prove that these properties hold as a consequence of the properties holding for functions fitting the specifications. We do so since these implementations are used when applying the general theorems in the proofs of the theorems for each mode.

Having followed the procedure so far, we can now prove the theorems for each mode of operation. We prove that the mode is valid and propagates IV changes when encryption. If the mode propagates plaintext changes when encrypting, we prove that it does so and that the corresponding MAC function propagates changes in the messages input to it. Likewise, if the mode does not propagates plaintext changes when encryption, we show that that property holds. The proofs of these theorems are described in the following Sections 6.9.2 to 6.9.6.

In Section 6.10.4, where we specify and prove theorems about the f mode, we also include the BENC mode which is a special case of the f mode. For that mode, we specify the block and feedback functions

in terms of the block and feedback functions of the f mode. Also, we do not prove any properties about these functions directly, and the theorems concerning the BENC mode are proven as corollaries of the corresponding theorems for the f mode.

6.9.2 Procedure for the Validity Proofs

In this section, we describe the procedure we follow in Section 6.10 when proving that each mode of operation is valid.

Before proving that a particular mode is valid, we have already shown that the block functions of the mode are inverse and that the feedback functions output equal values. Also, we have shown that the implementations of these functions have these properties. Often, however, these properties are shown as consequences of assumptions about the underlying block encryption scheme or function, e.g., when proving that the properties above hold for the IFB block and feedback functions in Section 6.10.3.1, we assume that the underlying block encryption scheme is valid. Such assumptions about the block encryption scheme or function are premises of the specific validity theorems being proved.

Having shown that the properties abovementioned hold for the block and feedback functions (and their implementations), we can apply the general validity theorems in the proofs: we prove the validity theorem for the mode using streams as a corollary of the general validity theorem using streams from Section 6.8.3.1 and the validity theorem for the mode using lists as a corollary of the general validity theorem using lists from Section 6.8.3.1.

Procedure for the Validity Proofs Using Streams All the validity theorems for the specific modes using streams use the `MoO_valid_stream_version` predicate defined in Section 6.7.4 to capture the concept of the mode being valid. It is shown below.

```
Definition MoO_valid_stream_version
           (MoO_enc MoO_dec : MoO_enc_dec_Type_stream_version)
           (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss css : Stream block_n) (ks : block_k) (IV : block_n),
    bisimilar_Stream_block_n css (MoO_enc xss ks IV block_enc) →
    bisimilar_Stream_block_n (MoO_dec css ks IV block_enc_dec) xss.
```

The predicate states that if some stream `css` is bisimilar to the `xss` stream encrypted, then `xss` is bisimilar to `css` decrypted. In other words, encrypting and then decrypting any plaintext stream yields the plaintext stream.

All the specific validity theorems using streams have conclusions which state that the predicate above holds for the encryption and decryption functions using streams for the particular modes and the underlying block encryption scheme or function—in case no block decryption function is used, the block encryption function is input twice to the predicate. Hence, in the proofs of these theorems, the goal is to show that the premise of the predicate holds under the assumption that the premise of the predicate holds in addition to other premises of the theorems—notably, premises stating that the various functions satisfy the corresponding specifications and, potentially, premises about the underlying block encryption scheme or function.

In the proofs of the specific validity theorems using streams, we apply the general validity theorem using streams. This theorem uses the `general_MoO_valid_stream_version` predicate defined in Section 6.8.3.1 in its conclusion to state that the general encryption scheme using streams is valid. It is shown below.

```
Definition general_MoO_valid_stream_version
           (general_MoO_enc_sv general_MoO_dec_sv :
              general_MoO_enc_dec_Type_stream_version)
           (block_fun_enc block_fun_dec : block_fun_Type)
           (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
```

```
          (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss css : Stream block_n) (ks : block_k) (IV : block_n),
    bisimilar_Stream_block_n
      css
      (general_MoO_enc_sv
         xss ks IV block_enc block_fun_enc feedback_fun_enc) →
    bisimilar_Stream_block_n
      (general_MoO_dec_sv
         css ks IV block_enc_dec block_fun_dec feedback_fun_dec)
      xss.
```

This predicate is similar to the `MoO_valid_stream_version` predicate above. Now, however, we use the general encryption and decryption functions applied to the input block and feedback functions.

To be able to apply the general theorem, we have to transform the goals in the proofs so that they match the conclusion of the `general_MoO_valid_stream_version` predicate. Here, we exploit that the encryption and decryption functions are specified to be externally equal to the general encryption and decryption functions using streams (from Section6.8.1) applied to the block and feedback functions of the particular modes of operation. As a consequence, we can unfold the decryption functions in the goals of the proofs to get goals that match the conclusion of the `general_MoO_valid_stream_version` predicate above: we can apply the general theorem. In this process, the `xss`, `css`, `ks`, and `IV` parameters of the two predicates above line up.

Now, what remains to be shown in the proofs is that the premises of the applied, general theorem are satisfied. First, we consider the premise of the `general_MoO_valid_stream_version` predicate: we have to show that the `css` stream is bisimilar to `xss` encrypted under the general encryption function using streams. Now, we exploit that the specific encryption function for the modes are specified to be externally equal to the general encryption function using streams when applied to the corresponding block and feedback functions for encryption: we unfold the specific encryption function in the premise of the `MoO_valid_stream_version` predicate in the conclusion of the theorems we are proving. After this transformation, that premise matches the premise of the `general_MoO_valid_stream_version` predicate so we just apply it.

When we unfold the encryption and decryption functions above, we use the implementations of the general encryption and decryption functions (using streams) defined in Section 6.8.1 and the already defined implementations of the block and feedback functions of each mode we consider. Hence, the rest of the premises of the general theorem hold: the general functions fit the corresponding specifications (as proven in Section 6.8.1), the block functions are inverse, and the feedback functions output equal values. Remember, we have already shown the two latter properties for the implementations of the block and feedback functions—potentially as consequences of particular properties of the underlying block encryption scheme or function which, as mentioned above, are premises of the theorems being proven.

Unfolding the encryption and decryption functions, we use the equivalence described in Section 3.4.2 between bisimilarity and the `equal_values_at_equal_indices` predicate. Doing so eases the process of unfolding. Instead, we could have exploited that bisimilarity is a transitive relation, e.g., if the `xss` stream is bisimilar to the general decryption of `css`, and the general decryption of `css` is bisimilar to the decryption of `css` using the decryption function of the mode of operation (via the specification of the decryption function in terms of the general decryption function), then `xss` is bisimilar to this latter decryption in which case the goals in the proofs described above have been shown.

Procedure for the Validity Proofs Using Lists The proofs of the validity theorems for the specific modes using lists are similar to the proofs outlined above. Now, the theorems use the `MoO_valid_list_version` predicate defined in Section 6.7.4. It is shown below for convenience.

```
Definition MoO_valid_list_version
```

```
          (MoO_enc_lv MoO_dec_lv : MoO_enc_dec_Type_list_version)
          (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV : block_n),
    MoO_dec_lv (MoO_enc_lv xss ks IV block_enc) ks IV block_enc_dec = xss.
```

The predicate states that if any plaintext list is encrypted and then decrypted under the same key, the plaintext list is output (for any key and any IV): the mode is valid—in conjunction with the underlying block encryption scheme or function.

All non-general validity theorems using lists use the above predicate in their conclusion: in the proofs of the theorems, we have to show that the predicate holds. In this regard, we use the general validity theorem using lists in Section 6.8.3.1. This theorem uses the `general_MoO_valid_list_version` predicate defined in that section in its conclusion. It is shown below.

```
Definition general_MoO_valid_list_version
          (general_MoO_enc_lv general_MoO_dec_lv :
             general_MoO_enc_dec_Type_list_version)
          (block_fun_enc block_fun_dec : block_fun_Type)
          (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
          (block_enc block_enc_dec : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV : block_n),
    general_MoO_dec_lv
      (general_MoO_enc_lv
         xss ks IV block_enc block_fun_enc feedback_fun_enc)
      ks IV block_enc_dec block_fun_dec feedback_fun_dec =
    xss.
```

The predicate states that first encrypting and then decrypting under the same key any plaintext list using the general encryption scheme using lists yields the plaintext list (for any key and any IV).

In the proofs of the theorems, we have to show that the `MoO_valid_list_version` predicate holds for the specific encryption and decryption functions in conjunction with the underlying block encryption scheme or function. We do so by applying the general theorem. First, we unfold the encryption and decryption functions of each mode considered: these functions are specified in terms of the general encryption and decryption functions (using lists) applied to the block and feedback functions of the modes. Here, we use the implementations of the general functions defined in Section 6.8.2 and the implementations of the block and feedback functions already defined. After this transformation of the goal, we can apply the general theorem.

Applying the general theorem, we have to show that the premises of the it are satisfied. As seen in Section 6.8.3.1, these premises state that the input general encryption and decryption functions using lists fit the corresponding specifications, that the input block functions are inverse, and that the input feedback functions output equal values. The premises concerning the general functions are satisfied since we use implementations of the functions proved, in Section 6.8.2, to fit the specifications. The mentioned properties about the block and feedback functions are proven to hold for the implementations of these functions and, thus, these last premises hold, too. As for the validity theorems using streams, these attributes of the block and feedback functions are potentially proven as consequences of properties about the underlying block encryption scheme or function; in this case, these assumptions about the block encryption scheme or functions are included as premises of the theorems being proven.

6.9.3 Procedure for the Plaintext Change Propagation Proofs

In this section, we outline the procedure we follow when we prove in Section 6.10 that some modes of operation propagate changes in the plaintexts when encrypting.

Before we prove that any particular mode of operation propagates plaintext changes when encrypting, we prove that the corresponding block and feedback functions for encryption are invertible with regards to both the plaintext blocks and the IVs or feedback blocks input to them. Also, we prove that the

implementations of the functions have this property. Having done so, we can apply the general plaintext change propagation theorems (from Section 6.8.3.2) in conjunction with these implementations.

Often, we prove that a block or feedback function is invertible as a consequence of the underlying block encryption function being itself invertible with regards to the in-the-context-of-the-block-encryption-function plaintext block, e.g., when proving that the CBC block functions for encryption is invertible with regards to the plaintext block in Section 6.10.1.2, we assume this property about the underlying block encryption function to hold. For each mode for which we prove plaintext change propagation theorems, at least some of the lemmas stating that the block and feedback functions for encryption are invertible assume that the block encryption function is invertible. Hence, all non-general plaintext change propagation theorems have this property about the block encryption function as premises.

Since the above properties about the block and feedback functions have been proved, we can apply the general plaintext change propagation theorems from Section 6.8.3.2 in the proofs of the non-general analogues: as in the section above, we prove the non-general theorems as corollaries of the general theorems. More specifically, we prove the non-general theorems using streams as corollaries of the general theorem using streams in Section 6.8.3.2 and the non-general theorems using lists as corollaries of the general theorem using lists in Section 6.8.3.2.

Procedure for the Plaintext Change Propagation Proofs Using Streams The plaintext change propagation theorems for the modes of operation using streams use the `plaintext_change_propagation_stream_version` predicate (defined in Section 6.7.4) in their conclusions. It is shown below.

```
Definition plaintext_change_propagation_stream_version
           (MoO_enc : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_from_index_and_equal_before
      (MoO_enc xss_1 ks IV block_enc)
      (MoO_enc xss_2 ks IV block_enc)
      i.
```

The predicate states that, for any key and any IV, if any two plaintext streams are different at some index `i` only, i.e., only the $\texttt{i}^{\text{th}}$ plaintext blocks are different, then the corresponding ciphertext streams resulting from encryption using the input encryption function (in conjunction with the input block encryption function) under the same key and using the same IV are different at index `i` and all indices after that point—and equal at indices before that point. In other words, changing a plaintext stream at any index results in all ciphertext blocks from that point forward being changed, too: plaintext changes are propagated when encrypting.

All the non-general plaintext change propagation theorems using streams say that the encryption function of the mode using streams and the underlying block encryption function introduced in the theorems satisfy the predicate above. Hence, the goals of the theorems are to show that the conclusion of the predicate holds, i.e., that the encrypted streams are different from index `i` onwards. Here, we use the general plaintext change propagation theorem using streams in Section 6.8.3.2. This theorem states that the general encryption function using streams propagates changes of the plaintexts when encrypting if a number of premises hold for the block and feedback functions (in conjunction with the underlying block encryption function) to which the general function is applied. The theorem uses the `general_plaintext_change_propagation_stream_version` predicate defined in Section 6.8.3.2 and shown below to state that the general encryption function propagates.

```
Definition general_plaintext_change_propagation_stream_version
           (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
```

```
        (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV : block_n) (i : nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_from_index_and_equal_before
      (general_MoO_enc xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

To be able to apply the general theorem, we need that the conclusion of the theorem matches the goals of the proofs. Therefore, we unfold the encryption function in the goals: this function is specified in terms of the general encryption function applied to block and feedback functions (for encryption) of the particular modes so we unfold the general functions in terms of implementations of these latter functions. Now, the goals and the conclusion of the general theorem match, and we can apply the theorem.

When we apply the general theorem, we have to show that the premises of it are satisfied. First, we consider the premise of the `general_plaintext_change_propagation_stream_version` predicate above: when we apply the general theorem, the parameters of the predicate and the `plaintext_change_propagation_stream_version` predicate in the conclusion of the theorems we are proving line up. Hence, the premises of the two predicates are equal: the `xss_1` and `xss_2` streams are different at index `i` only. As a result, this premise of the applied theorem holds.

Additionally, the general theorem has several premises outside of the predicate in its conclusion: the implementations used when unfolding the encryption functions using streams of the particular modes must satisfy a number of properties: the implementation of the general encryption function using streams must fit the corresponding specification, and the implementations of the block and feedback functions must be invertible with regards to both the plaintext blocks and the IVs or feedback blocks.

We use the implementation of the general function defined in Section 6.8.1. This implementation is proved to satisfy the corresponding specification: that premise holds. Before proving the non-general plaintext change propagation theorems, we prove that the block and feedback functions for encryption are invertible and that the defined implementations also have this property. We use these implementations when unfolding the encryption function. As mentioned above, we often assume that the underlying block encryption function is invertible with regards to the plaintext blocks (in the context of the block encryption function) when proving that the block and feedback functions for encryption are invertible: this latter property about the block encryption function is a premise of the theorems being proved. Thus, the premises concerning the block and feedback functions hold, too, and the proofs are done.

Procedure for the Plaintext Change Propagation Proofs Using Lists The `plaintext_change_propagation_list_version` predicate defined in Section 6.7.4 and shown below is used in the conclusions of the non-general plaintext change propagation theorems using lists to state that the encryption functions (using lists) considered propagate changes in plaintexts.

```
Definition plaintext_change_propagation_list_version
        (MoO_enc_lv : MoO_enc_dec_Type_list_version)
        (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_from_index_and_equal_before
      (MoO_enc_lv xss_1 ks IV block_enc)
      (MoO_enc_lv xss_2 ks IV block_enc)
      i.
```

This predicate is similar to the `plaintext_change_propagation_stream_version` predicate above except that the streams have been replaced by lists. The idea remains the same: if the i^{th} plaintext block is

changed, then the corresponding ciphertext is changed at all indices greater than or equal to `i`. However, an extra premise has been added that states that the plaintext lists introduced by the predicate have equal lengths: changing a block in a plaintext does not alter its number of blocks.

The goals of the list versions of the non-general plaintext change propagation proofs is to show that the conclusion of the predicate above holds, i.e., that the ciphertext lists are different from index `i` and equal before. In this regard, we use the list version of the general plaintext change propagation theorem in Section 6.8.3.2. This theorem uses the `general_plaintext_change_propagation_list_version` predicate, which is similar to the predicate above, defined in Section 6.8.3.2 and shown below for convenience.

```
Definition general_plaintext_change_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_from_index_and_equal_before
      (general_MoO_enc_lv xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc_lv xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

Using this predicate in its conclusion, the general theorem says that the general encryption function using lists propagates changes in the plaintexts.

When proving the non-general theorems, we apply the general theorem. First, however, we unfold the encryption functions of the modes we consider in the goals so that the goals match the conclusion of the `general_plaintext_change_propagation_list_version` predicate above: the encryption functions are specified in terms of the general encryption function (using lists) applied to the particular block and feedback functions for encryption. Therefore, we can unfold the encryption functions to get goals that involve the implementations of the general encryption function using lists and the block and feedback functions: the goals now match the conclusion of the predicate, and we can apply the general theorem.

Applying the general theorem, we have to prove that the premises of it hold. First, we look at the premises of the `general_plaintext_change_propagation_list_version` predicate: the plaintext lists introduced in the predicate must be of equal lengths and different at the index `i` introduced by the predicate, too. However, when we apply the general theorem, the parameters of the predicate align with the parameters of the `plaintext_change_propagation_list_version` predicate in the conclusion of the theorems we are proving. In that predicate, the mentioned properties about the plaintext lists are assumed to hold, and, thus, the premises of the general predicate hold also.

Similarly to the stream version of the general theorem, the rest of the premises of the list version of the general theorem, which we apply in the proofs currently being described, state that the used implementation of the general encryption function using lists must fit the corresponding specification and that the used implementations of the block and feedback functions for encryption must be invertible with regards to both the plaintext blocks and the IVs or feedback blocks.

The premises hold: we use an implementation of the general function defined in Section 6.8.2 which is proven to satisfy the specification and implementations of the block and feedback functions that have already been shown to be invertible. In some of the lemmas stating that these latter implementations are invertible, we assume that the underlying block encryption functions are invertible with regards to the in-the-context-of-the-block-encryption-functions plaintext blocks. This property about the block encryption functions, however, is assumed to hold in the theorems we are proving. Hence, all the used implementations of the block and feedback functions are invertible: the rest of the premises of the applied theorem are satisfied, and the proofs are done.

6.9.4 Procedure for the Plaintext Change Non-Propagation Proofs

In this section, we describe the procedure we follow when proving the non-general plaintext change non-propagation theorems in Section 6.10.

We prove the non-general plaintext change non-propagation theorems as corollaries of the corresponding, general theorems in Section 6.8.3.3. These general theorems assume that the implementations we use of the block and feedback functions for encryption satisfy the following properties: the implementations of the block functions are assumed to be invertible with regards to the plaintext blocks, and the implementations of the feedback functions are assumed to ignore the plaintext blocks, i.e., their outputs are assumed to be independent of the plaintext blocks. None of the modes for which we prove non-propagation require that any special properties about the underlying block encryption functions for the above properties to hold for their block and feedback functions for encryption so we do not need extra assumptions about the block encryption functions in the non-propagation theorems.

Having proven these properties about the block and feedback functions for encryption and their implementations, we can prove the non-general theorems by applying the corresponding, general theorems: the stream versions of the non-general theorems are proven as corollaries of the stream version of the general theorem in Section 6.8.3.2, and the list versions of the non-general theorems are proven as corollaries of the list version of the general theorem in Section 6.8.3.2.

Procedure for the Plaintext Change Non-Propagation Proofs Using Streams The stream versions of the non-general plaintext change non-propagation theorems use in their conclusions the `plaintext_change_non_propagation_stream_version` predicate defined in Section 6.7.4 and shown below.

```
Definition plaintext_change_non_propagation_stream_version
           (MoO_enc : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV :  block_n) (i :  nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only
      (MoO_enc xss_1 ks IV block_enc)
      (MoO_enc xss_2 ks IV block_enc)
      i.
```

The predicate holds for an encryption function using streams (in conjunction with an underlying block encryption function) if the it does not propagate changes in the plaintext blocks, i.e., if a plaintext block is changed, only the ciphertext block at the same index is changed also.

We prove the non-general theorems as corollaries of the corresponding, general theorem. That theorem uses in its conclusion the `general_plaintext_change_non_propagation_stream_version` predicate defined in Section 6.8.3.3 and shown below.

```
Definition general_plaintext_change_non_propagation_stream_version
           (general_MoO_enc : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : Stream block_n) (ks : block_k) (IV :  block_n) (i :  nat),
    streams_different_at_index_only xss_1 xss_2 i →
    streams_different_at_index_only
      (general_MoO_enc xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

This predicate is analogous to the `plaintext_change_non_propagation_stream_version` predicate above. Now, however, we consider the stream version of the general encryption function applied to block and feedback functions (for encryption).

The goals of the non-general proofs are to show that the conclusion of the `plaintext_change_non_propagation_stream_version` predicate in their conclusions holds, i.e., we have to show that the ciphertext streams output by the encryption functions of the considered modes of operation are only different at the `i` index introduced by the predicate. Here, we exploit that the encryption functions are specified in terms of the stream version of the general encryption function applied to the particular block and feedback functions for encryption: we unfold the encryption function using implementations of the general function and the block and feedback functions. Doing so transforms the goals into goals that match the conclusion in the `general_plaintext_change_non_propagation_stream_version` predicate in the conclusion of the general theorem: we apply this general theorem.

When we apply the general theorem, we have to show that the premises of it hold. First, we consider the premise of the `general_plaintext_change_non_ propagation_stream_version` predicate: the predicate introduces two plaintext streams and an index, and the streams are only different at that index only. However, when we apply the general theorem, the parameters of this predicate and the `plaintext_change_non_propagation_stream_version` predicate in the conclusions of the theorems we are proving line up. Therefore, the premise holds: it is already assumed in the `plaintext_change_non_propagation_stream_version` predicate.

In addition, the applied, general theorem has premises stating that the used implementation of the general encryption function using streams fits the corresponding specification, that the used implementations of the block function for encryption are invertible with regards to the plaintext blocks, and that the used implementations of the feedback functions for encryption ignore the plaintext blocks.

The first of these premises holds since we use the implementation of the general function defined in Section 6.8.1: that implementation is proved to satisfy the specification. The second and third premises hold since we use the implementations of the block and feedback functions for encryption about which we have already proven the properties: the implementations of the block functions have been proven to be invertible with regards to the plaintext blocks, and the implementations of the feedback functions have been proven to ignore the plaintext blocks. Those properties have been shown to hold with no special assumptions about the underlying block encryption functions: the last premises of the general theorem are satisfied, and the proofs are done.

Procedure for the Plaintext Change Non-Propagation Proofs Using Lists Now, we consider the proofs of the non-general plaintext change non-propagation theorems using streams. These theorems use the `plaintext_change_non_propagation_list_version` predicate defined in Section 6.7.4 and shown below in their conclusions to state that the encryption functions (using lists) of the particular modes of operation do not propagate changes in the plaintext.

```
Definition plaintext_change_non_propagation_list_version
           (MoO_enc_lv : MoO_enc_dec_Type_list_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_at_index_only
      (MoO_enc_lv xss_1 ks IV block_enc)
      (MoO_enc_lv xss_2 ks IV block_enc)
      i.
```

This predicate is similar to the `plaintext_change_non_propagation_stream_version` predicate. Now, however, we look at lists instead of streams, and we assume that the plaintext lists have equal lengths: changing a block does not change the number of blocks.

In the proofs of the non-general non-propagation theorems using lists, we apply the corresponding, general theorem. That theorem uses the `general_plaintext_change_non_propagation_list_version` predicate in its conclusion. This predicate is defined in Section 6.8.3.3 and shown below.

```
Definition general_plaintext_change_non_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (IV : block_n) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    lists_different_at_index_only
      (general_MoO_enc_lv xss_1 ks IV block_enc block_fun feedback_fun)
      (general_MoO_enc_lv xss_2 ks IV block_enc block_fun feedback_fun)
      i.
```

This predicate resembles the predicate above. Now, though, we consider the general encryption function (using lists) applied to block and feedback functions for encryption.

The goals of the proofs of the non-general theorems are to show that the conclusion of the `plaintext_change_non_propagation_list_version` predicate in the conclusions of the theorems holds, i.e., we have to show that the ciphertext lists output by the particular encryption functions are different at index `i` only. Here, the general theorem comes to our aid: we apply that theorem.

However, the conclusion of the general theorem does not match the goals of the proofs. This problem is remedied by unfolding the encryption functions in the goals using implementations of the general encryption function using lists and the particular block and feedback functions for encryption. This unfolding is possible since these encryption functions of the modes of operations are specified in terms of the general encryption function applied to the block and feedback functions (for encryption) of the modes. After the unfolding of the encryption functions and, hence, transformation of the goals, we apply the general theorem.

Now, what remains to be done is showing that the premises of the applied, general theorem are satisfied. First, we consider the premises of the `general_plaintext_change_non_propagation_list_version` predicate: the predicate introduces two plaintext lists and an index, and it is assumed that the lists are of equal lengths and different at the index only. These premises hold: when we apply the general theorem, the parameters of the predicate and the `plaintext_change_non_propagation_list_version` predicate align, and the latter predicate contains premises equal to the premises we are showing to hold.

The applied, general theorem has a few additional premises about the implementations of the general encryption function using lists and the block and feedback functions input to which the theorem is applied: the implementation of the general function must fit the corresponding specification, the implementations of the block functions must be invertible with regards to the plaintext blocks, and the implementations of the feedback functions must ignore the plaintext blocks.

The premise concerning the general function is satisfied since we use the implementation of it defined in Section 6.8.2: this implementation is proven to fit the specification. The premises pertaining to the implementations of the block and feedback functions hold, too, since when unfolding the encryption functions of the particular modes, we use the already defined implementations of the block and feedback functions: the implementations of the blocks functions have been proven to be invertible, and the implementations of the feedback functions have been proven to ignore the plaintext blocks. All premises have now been shown to hold, and the proofs are done.

6.9.5 Procedure for the IV Change Propagation Proofs

In this section, we outline the procedure we follow when we prove the non-general IV change propagation theorems in Section 6.10.

The non-general IV change propagation theorems are proven as corollaries of the corresponding general theorems in Section 6.8.3.4. These general theorems, however, assume that implementation input to the theorems of the block and feedback functions for encryption are invertible with regards to the IVs

or feedback blocks. Here, we exploit that before proving the non-general theorems, the relevant block and feedback functions and the implementations we use of the functions have already been shown to be invertible. Often, this property is proven for an implementation as a consequence of the underlying block encryption function being invertible with regards to the in-the-context-of-the-block-encryption-function plaintext block: we assume this latter attribute of the block encryption function to hold for at least either the block or feedback function for each mode of operation, and, hence, the non-general IV change propagation theorems all include premises stating this attribute to hold.

Since the abovementioned properties about the block and feedback functions and their implementations have been shown, we can apply the general theorems in the proofs of the non-general theorems: we apply the stream version of the general IV change propagation theorem from Section 6.8.3.4 in the non-general IV change propagation proofs using streams, and we apply the list version of the general IV change propagation theorem from Section 6.8.3.4 in the non-general IV change propagation proofs using lists.

When applying the general theorems, we have to show the premises of them are satisfied. The theorems assume that the implementations of the general encryption functions to which the theorems are applied fit the corresponding specifications: here, we use the implementations defined in Sections 6.8.1 and 6.8.2 of the general functions. These are proven to satisfy the specifications. Additionally, the theorems assume that the implementations of the block and feedback functions to which the theorems are applied are invertible with regards to the IVs or feedback blocks. Since we use the already defined and proven-to-be-invertible implementations of the block and feedback functions for encryption, this last assumption is satisfied, too.

The following paragraphs describe in further details the proofs of the non-general theorems using streams respectively lists.

Procedure for the IV Change Propagation Proofs Using Streams In the conclusions of the non-general IV change propagation theorems using streams, we use the `IV_change_propagation_stream_version` predicate defined in Section 6.7.4 and shown below to say that the encryption functions of the modes considered propagate changes of the IVs.

```
Definition IV_change_propagation_stream_version
           (MoO_enc_sv : MoO_enc_dec_Type_stream_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : Stream block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    streams_different_at_all_indices
      (MoO_enc_sv xss ks IV_1 block_enc) (MoO_enc_sv xss ks IV_2 block_enc).
```

The predicate states that if two IVs are different, then, every other input being the same, the corresponding ciphertext streams are blockwise different at all indices.

In the proofs of non-general theorems, the goals are to show that the conclusion of the predicate above holds, i.e., that the ciphertext streams are different at all indices. To show these goals, we apply the general IV change propagation theorem using streams. This theorem uses the `general_IV_change_propagation_stream_version` predicate defined in Section 6.8.3.4. It is shown below.

```
Definition general_IV_change_propagation_stream_version
           (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : Stream block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    streams_different_at_all_indices
      (general_MoO_enc_sv xss ks IV_1 block_enc block_fun feedback_fun)
      (general_MoO_enc_sv xss ks IV_2 block_enc block_fun feedback_fun).
```

This predicate is similar to the IV_change_propagation_stream_version predicate above except that now we consider the general encryption function (using streams) applied to block and feedback functions.

We be able to apply the general theorem in the non-general proofs, we have to transform the goals so that they match the conclusion of the predicate above. In this regard, we unfold the encryption functions in the goals: here, we exploit that these functions are specified in terms of the stream version of the general encryption function applied to the block and feedback functions (for encryption) for the particular modes of operation. After this unfolding, the goals and the conclusion match, and we apply the general theorem.

Applying the general theorem, we have to show that the premises of it hold. First, we consider the premise of the general_IV_change_propagation_stream_version predicate: the IVs introduced by the predicate are different. This premise is satisfied since when we apply the general theorem, the parameters of the two predicates above align: the premise of the IV_change_propagation_stream_version predicate states exactly that the IVs are different.

When unfolding the encryption functions, we use the implementation of the stream version of the general encryption function defined in Section 6.8.1 and the implementations of the block and feedback functions defined beforehand. As mentioned in above, these implementations are proved to have the properties assumed by the applied, general theorem: the implementation of the general function fits the corresponding specification, and the implementations of the block and feedback functions are invertible with regards to the IVs or feedback blocks. Hence, the rest of the premises of the general theorem are satisfied, and the proofs are done.

Procedure for the IV Change Propagation Proofs Using Lists The non-general IV change propagation theorems using lists use the IV_change_propagation_list_version predicate defined in Section 6.7.4 and shown below.

```
Definition IV_change_propagation_list_version
           (MoO_enc_lv : MoO_enc_dec_Type_list_version)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    lists_different_at_all_indices
      (MoO_enc_lv xss ks IV_1 block_enc) (MoO_enc_lv xss ks IV_2 block_enc).
```

This predicate is similar to the IV_change_propagation_stream_version predicate in the paragraph above except that we now consider lists instead of streams.

The list version of the IV change propagation theorem, of which we prove these non-general theorems as corollaries, use the general version of the predicate above, the general_IV_change_propagation_list_version predicate defined in Section 6.8.3.4 and shown below, in their conclusions.

```
Definition general_IV_change_propagation_list_version
           (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type)
           (block_enc : block_enc_dec_Type) :=
  forall (xss : list block_n) (ks : block_k) (IV_1 IV_2 : block_n),
    IV_1 <> IV_2 →
    lists_different_at_all_indices
      (general_MoO_enc_lv xss ks IV_1 block_enc block_fun feedback_fun)
      (general_MoO_enc_lv xss ks IV_2 block_enc block_fun feedback_fun).
```

This predicate resembles the general_IV_change_propagation_stream_version predicate in the paragraph above.

The situation in the proofs of the non-general IV change propagation theorems using lists is equal to the situation in the proofs of the corresponding theorems using streams in the paragraph above: the only

differences are that we now use an implementation of the general encryption function using lists, and that we now consider plaintext lists instead of streams. Hence, the same procedure as in the paragraph above is followed: after unfolding the encryption functions in the goals of the proofs, we apply the corresponding general theorem whose premises are proven to hold.

6.9.6 Procedure for the MAC Propagation Proofs

In this section, we describe the procedure we follow in the proofs of the non-general MAC propagation theorems in Section 6.10.

The non-general MAC propagation theorems are proven as corollaries of the general MAC propagation theorem in Section 6.8.3.5. That theorem assumes the same properties to hold for the implementations used of the particular block and feedback function for encryption as the general plaintext change propagations theorems in Section 6.8.3.2: the implementations are assumed to be invertible with regards to both the plaintext blocks and the IVs or feedback blocks. However, these properties have already been shown before we prove the MAC propagation theorems: we have already defined implementations of the block and feedback functions and proven that they are invertible as a consequence of them fitting the corresponding specifications.

As mentioned in Section 6.9.3, we often prove that the implementations are invertible as consequences of the underlying block encryption functions being invertible with regards to the in-the-context-of-the-block-encryption-functions plaintext blocks: all the non-general MAC propagation theorems assume that this attribute of the underlying block encryption functions holds since for each proof of these theorems at least some of the implementations require this attribute to hold.

The non-general MAC propagation theorems use the `MAC_propagation` predicate defined in Section 6.7.4, and shown below, in their conclusions to state that the particular MAC functions are propagating.

```
Definition MAC_propagation
           (MAC_fun : MAC_fun_Type) (block_enc : block_enc_dec_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    MAC_fun xss_1 ks block_enc <> MAC_fun xss_2 ks block_enc.
```

The predicate says that if two messages (of blocks) of equal lengths are different at some index `i` only, then the corresponding MAC values are different. In other words, changing a block of a messages changes the corresponding MAC value. We assume that the messages have equal lengths since changing a block does not alter the number of blocks in the message.

The goals in the proofs of the theorems are to show that the premise of the predicate above holds, i.e., we have to show that the MAC values are different. To this end, we apply the general MAC propagation theorem of which the theorems are corollaries. This general theorem uses the general counterpart of the predicate above: the `general_MAC_propagation` predicate defined in Section 6.8.3.5 and shown below.

```
Definition general_MAC_propagation
           (general_MAC_fun : general_MAC_fun_Type)
           (block_enc : block_enc_dec_Type)
           (block_fun : block_fun_Type)
           (feedback_fun : feedback_fun_Type) :=
  forall (xss_1 xss_2 : list block_n) (ks : block_k) (i : nat),
    length xss_1 = length xss_2 →
    lists_different_at_index_only xss_1 xss_2 i →
    general_MAC_fun xss_1 ks block_enc block_fun feedback_fun <>
    general_MAC_fun xss_2 ks block_enc block_fun feedback_fun.
```

This predicate resembles the non-general predicate. Now, however, we instead consider the general MAC function (specified in Section 6.8.2) applied to block and feedback functions for encryption. The

general theorem uses the general predicate above in its conclusion to state that the general MAC function propagates changes in the input messages.

To be able to apply the general theorem in the proofs of the non-general theorems, we have to transform the goals so that they match the conclusion of the `general_MAC_propagation` predicate above. Here, we exploit that the non-general MAC functions are specified in terms of the general MAC functions applied to block and feedback functions (for encryption) of the particular modes of operation: before proving the non-general theorems, implementations of the functions have been defined and proven to fit the corresponding specifications—both for the general MAC function and the block and feedback functions—and we use these implementations to unfold the general MAC function in the goals of the proofs.

After the unfolding, the goals of the proofs match the conclusion of the general predicate above: we can apply the general theorem. Now, the remainder of the proofs consists of showing that the premises of the applied, general theorem are satisfied. First, we consider the premises of the general predicate: the predicate introduces two message lists and an index `i`, and it is assumed that the lists are of equal lengths and that they are different at the index `i` only. These premises hold since when we apply the general theorem, the parameters of the general predicate and the non-general predicate used in the conclusion of the theorems we are proving line up: the non-general predicate has premises equal to the ones of the general predicate, and, hence, they hold.

The general theorem also has several premises outside the predicate in its conclusion: the used implementation of the general MAC function must fit the corresponding specification, and the used implementations of the particular block and feedback functions for encryption must be invertible with regards to both the plaintext blocks and the IVs or feedback blocks. We use the implementation of the general MAC function defined in Section 6.8.2. That implementation is proven to fit the specification so the premise concerning the general MAC function holds. Likewise, we use the implementations of the block and feedback functions already defined before proving the respective non-general MAC theorems: these implementations have already been proven to be invertible—potentially as a consequence of the underlying block encryption functions being invertible themselves.

6.10 Theorems and Proofs for Each Mode of Operation

In this section, we prove all the non-general theorems concerning the modes of operation and MAC schemes built on top of them. We prove these theorems as corollaries of the general theorems in Section 6.8.3 by following the procedures described in Section 6.9.

In the following sections, we specify and prove properties about the modes of operation and associated MAC schemes in this order: the CBC mode, the CFB mode, the IFB mode, and, finally, the `f`, CTR, and BENC modes.

6.10.1 The CBC Mode

The first mode of operation we consider is the Cipher Block Chaining (CBC) mode. In this section, we follow the procedure outlined in Section 6.9.1.

6.10.1.1 Specifications

In this section, we take the first step of the procedure: we specify the CBC block and feedback functions and, in terms of these specifications, the CBC encryption and decryption functions (both the stream and list versions) and the CBC-MAC function. In the following, we use the XOR-blocks-`n` function specified in Section 6.6.1: this function is used to compute the bitwise XORs of blocks of length `n`.

We follow the procedure described in Section 6.9.1 so we start by specifying the CBC block and feedback functions. First, we specify the block function for encryption below.

```
Definition specification_of_CBC_block_function_for_encryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      block_fun xs ks IV block_enc = block_enc (XOR_blocks_n_fun xs IV) ks.
```

Applying the CBC block function for encryption is extensionally equal to first bitwise XOR'ing the plaintext block and the IV or feedback block and then encrypting the result under the given key using the underlying block encryption function, `block_enc`.

Next, we consider the block function for decryption.

```
Definition specification_of_CBC_block_function_for_decryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (cs : block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      block_fun cs ks IV block_dec = XOR_blocks_n_fun (block_dec cs ks) IV.
```

The block function for decryption is the exact opposite of the block function for encryption—they will be proven to be inverse in Section 6.10.1.2 after all: applying the block function for decryption is extensionally equal to first decrypting the ciphertext block under the given key using the underlying block decryption function, `block_dec`, and then bitwise XOR'ing the result with the IV or feedback block.

We now specify the feedback functions. First, the feedback function for encryption is specified.

```
Definition specification_of_CBC_feedback_function_for_encryption
           (feedback_fun : feedback_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      feedback_fun xs ks IV block_enc =
      block_enc (XOR_blocks_n_fun xs IV) ks.
```

The feedback function for encryption is specified similarly to the block functions for encryption: their output behaviour are equal—the output of both functions is the corresponding ciphertext block. Likewise, the output of the feedback function for decryption is also the ciphertext block; here, however, the ciphertext block is given directly as input to the function.

```
Definition specification_of_CBC_block_function_for_decryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (cs : block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      block_fun cs ks IV block_dec = XOR_blocks_n_fun (block_dec cs ks) IV.
```

We define functions satisfying the specifications above: the `CBC_block_fun_enc_v0` function respectively the `CBC_block_fun_dec_v0` functions fits the specification of the CBC block function for encryption respectively decryption, and the `CBC_feedback_fun_enc_v0` function respectively the `CBC_feedback_fun_dec_v0` function fits the specification of the CBC feedback function for encryption respectively decryption.

With the CBC block and feedback functions specified, we can now turn to the specifications of the actual CBC encryption and decryption functions that use these block and feedback functions. First, we specify the stream version of the CBC encryption function.

```
Definition specification_of_CBC_encryption_function_stream_version
```

```
          (CBC_enc_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_enc_MoO : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_enc_MoO →
    specification_of_CBC_block_function_for_encryption block_fun →
    specification_of_CBC_feedback_function_for_encryption
      feedback_fun →
    forall (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CBC_enc_sv xss ks IV block_enc)
        (general_enc_MoO xss ks IV block_enc block_fun feedback_fun).
```

Applying the CBC encryption function using streams is bisimilar to applying the general encryption function (using streams) specified in Section 6.8.1 with the CBC block and feedback functions.

The stream version of the CBC decryption function is specified likewise.

```
Definition specification_of_CBC_decryption_function_stream_version
          (CBC_dec_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_dec_MoO : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_decryption_function_stream_version
      general_dec_MoO →
    specification_of_CBC_block_function_for_decryption block_fun →
    specification_of_CBC_feedback_function_for_decryption
      feedback_fun →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CBC_dec_sv css ks IV block_dec)
        (general_dec_MoO css ks IV block_dec block_fun feedback_fun).
```

This specification follows the same structure as the above specification for the CBC encryption function using streams: encryption functions (including the underlying block encryption function) and specifications are just replaced by their decryption counterparts, and the plaintext blocks are replaced by the ciphertext blocks.

We specify the list versions of the CBC encryption and decryption functions.

```
Definition specification_of_CBC_encryption_function_list_version
          (CBC_enc_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    specification_of_CBC_block_function_for_encryption block_fun →
    specification_of_CBC_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      CBC_enc_lv xss ks IV block_enc =
      general_MoO_enc_lv xss ks IV block_enc block_fun feedback_fun.

Definition specification_of_CBC_decryption_function_list_version
          (CBC_dec_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_MoO_dec_lv : general_MoO_enc_dec_Type_list_version)
```

```
      (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
  specification_of_general_MoO_decryption_function_list_version
    general_MoO_dec_lv →
  specification_of_CBC_block_function_for_decryption block_fun →
  specification_of_CBC_feedback_function_for_decryption
    feedback_fun →
  forall (css : list block_n) (ks : block_k) (IV : block_n)
      (block_dec : block_enc_dec_Type),
    CBC_dec_lv css ks IV block_dec =
    general_MoO_dec_lv
      css ks IV block_dec block_fun feedback_fun.
```

These specification are analogous to the corresponding stream versions above. Now, however, we use the list versions of the general encryption and decryption functions specified in Section 6.8.2. Also, since we use lists instead of streams, we use the normal notion of equality and not bisimilarity.

In Section 6.10.1.6, we prove the CBC-MAC theorem. To be able to do so, we specify the CBC-MAC function.

```
Definition specification_of_CBC_MAC_function (CBC_MAC : MAC_fun_Type) :=
  forall (general_MAC : general_MAC_fun_Type)
      (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MAC_function general_MAC →
    specification_of_CBC_block_function_for_encryption block_fun →
    specification_of_CBC_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k)
        (block_enc : block_enc_dec_Type),
      CBC_MAC xss ks block_enc =
      general_MAC xss ks block_enc block_fun feedback_fun.
```

The specification is straightforward: the CBC-MAC function is extensionally equal to the general MAC function applied to the CBC block and feedback functions for encryption.

6.10.1.2 Properties about the CBC Block and Feedback Functions

The next step of the procedure in Section 6.9.1 is to prove several properties about the block and feedback functions and their implementations.

First, we prove that the CBC block functions are inverse and that the CBC feedback functions output equal values. We use these properties when proving the CBC validity theorems in Section 6.10.1.3. When proving that the CBC block functions are inverse, we assume that the underlying block encryption scheme is valid. The lemma below states this implication.

```
Lemma CBC_block_functions_inverse :
  forall (block_fun_enc block_fun_dec : block_fun_Type)
      (block_enc block_dec : block_enc_dec_Type),
    block_enc_scheme_valid block_enc block_dec →
    specification_of_CBC_block_function_for_encryption block_fun_enc →
    specification_of_CBC_block_function_for_decryption block_fun_dec →
    block_functions_inverse
      block_fun_enc block_fun_dec
      block_enc block_dec.
```

Recall from the section above, the output of the CBC block function for encryption is equal to the block encryption of the bitwise XOR of the plaintext block and the IV or feedback block under the given key. Likewise, the output of the CBC block function decryption is equal to the bitwise XOR of the block decryption of the ciphertext block (being the output of the block function for encryption) under the given key and the IV or feedback block. Hence, since the goal of the proof of lemma above is to show

that first applying the block function for encryption and then the block function for decryption to any plaintext block yields the plaintext block, we have to show that this order of function applications have the same effect: the XOR-blocks-n function with the IV or feedback block as the second input, the block encryption function under the given key, the block decryption function under the same given key, and, finally, the XOR-blocks-n function again with the IV or feedback block as the second input.

The second and third function applications are cancelled by exploiting that the underlying block encryption scheme is valid: applying first the block encryption function and then the corresponding decryption function preserves the input block. The first and last function applications are cancelled by applying the `XOR_blocks_n_fun_applied_twice_to_same_second_input_yields_first_input` lemma in Section 6.6.1: bitwise XOR'ing any block, `xs`, with any block, `ys`, twice yields the `xs` block.

Next, we prove that the CBC feedback functions output equal values.

```
Lemma CBC_feedback_functions_output_equal_values :
  forall (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
         (block_fun_enc : block_fun_Type)
         (block_enc block_dec : block_enc_dec_Type),
    specification_of_CBC_feedback_function_for_encryption
      feedback_fun_enc →
    specification_of_CBC_feedback_function_for_decryption
      feedback_fun_dec →
    specification_of_CBC_block_function_for_encryption block_fun_enc →
    feedback_functions_output_equal_values
      feedback_fun_enc feedback_fun_dec
      block_fun_enc
      block_enc block_dec.
```

The lemma above is proven by unfolding the block and feedback functions via their specifications.

When proving the plaintext change, IV change, and MAC propagation theorems for the CBC mode, we exploit that the CBC block and feedback functions for encryption are invertible with regards to the plaintext blocks and the IVs or feedback blocks. First, we prove that the CBC block function for encryption is invertible with regards to the plaintext block if the underlying block encryption function is invertible with regards to the in-the-context-that-function plaintext block—which it is in reality, being an encryption function with domain and codomain of equal sizes.

```
Lemma CBC_block_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_CBC_block_function_for_encryption block_fun →
      block_fun_invertible_wrt_plaintext block_fun block_enc.
```

When proving the lemma above, we use the `XOR_blocks_n_fun_invertible_wrt_first_input` lemma from Section 6.6.1. That lemma states that the XOR-blocks-n function is invertible with regards to its first input. As seen in Section 6.10.1.1, the output of the CBC block function for encryption is equal to the ciphertext block resulting from encrypting the bitwise XOR of the plaintext block and the IV or feedback block under the given key using the underlying block encryption function. Hence, since the underlying block encryption function is assumed to be invertible with regards to the in-the-context-of-that-function plaintext block, the block function is a composite function of two invertible functions (we consider the other inputs to the functions fixed): this fact implies that the block function is invertible itself and is used to prove the lemma.

The CBC block function for encryption is also invertible with regards to the feedback block if the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block.

```
Lemma CBC_block_fun_enc_invertible_wrt_IV_feedback :
```

```
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_CBC_block_function_for_encryption block_fun →
      block_fun_invertible_wrt_IV_feedback block_fun block_enc.
```

The proof of this lemma is similar to the proof above. Now, however, we consider the second input to the XOR-blocks-n function: the feedback block. Hence, we use the `XOR_blocks_n_fun_invertible_wrt_second_input` lemma from Section 6.6.1 instead of the `XOR_blocks_n_fun_invertible_wrt_first_input` lemma: the XOR-blocks-n function is invertible with regards to its second input. Apart from this replacement, the proofs resemble each other.

Finally, we prove the same properties about the CBC feedback function for encryption as consequences of the underlying block encryption function being invertible with regards to the plaintext blocks (in its context).

```
Lemma CBC_feedback_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall feedback_fun : feedback_fun_Type,
      specification_of_CBC_feedback_function_for_encryption
        feedback_fun →
      feedback_fun_invertible_wrt_plaintext feedback_fun block_enc.

Lemma CBC_feedback_fun_enc_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall feedback_fun : feedback_fun_Type,
      specification_of_CBC_feedback_function_for_encryption
        feedback_fun →
      feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc.
```

As seen in Section 6.10.1.1, the specifications of the CBC block and feedback functions for encryption are similar: the plaintext block is bitwise XOR'ed with the IV or feedback block, and the resulting block is encrypted with the underlying block encryption function under the given key. Hence, the proofs of the two latter lemmas are very similar to the corresponding proofs for the CBC block function: after the unfolding of the block function respectively feedback function, the proofs are alike.

When proving the CBC theorems in Sections 6.10.1.3 to 6.10.1.6, we assume that the properties in this section hold for the implementations of the block and feedback functions we use. Hence, we prove lemmas stating that the implementations mentioned in the section above satisfy these properties. These lemmas are proven as corollaries of the lemmas in this section.

6.10.1.3 The CBC Validity Theorems

We prove that the stream and list versions of the CBC mode of operation scheme are valid. When doing so, we follow the procedure described in Section 6.9.2.

First, we state the stream version of the CBC validity theorem.

```
Corollary CBC_validity_theorem_stream_version :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall CBC_enc_sv CBC_dec_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CBC_encryption_function_stream_version
        CBC_enc_sv →
      specification_of_CBC_decryption_function_stream_version
        CBC_dec_sv →
      MoO_valid_stream_version
```

```
    CBC_enc_sv CBC_dec_sv
    block_enc block_dec.
```

This theorem is a corollary of the corresponding general validity theorem using streams in Section 6.8.3.1. It states that if the underlying block encryption scheme is valid, then the CBC scheme is valid, too.

The proof of the theorem is done as described in Section 6.9.2. Doing so, we use the implementations of the CBC block and feedback functions mentioned in Section 6.10.1.1: these implementations have been proven to have the properties required to apply the general theorem. Hence, the proof as described in Section 6.9.2 works.

Likewise, we prove the validity theorem for the CBC mode of operation using lists.

```
Corollary CBC_validity_theorem_list_version :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall CBC_enc_lv CBC_dec_lv : MoO_enc_dec_Type_list_version,
      specification_of_CBC_encryption_function_list_version
        CBC_enc_lv →
      specification_of_CBC_decryption_function_list_version
        CBC_dec_lv →
      MoO_valid_list_version
        CBC_enc_lv CBC_dec_lv
        block_enc block_dec.
```

This theorem is similar to the theorem above using streams. Now, however, we consider the list versions of the CBC encryption and decryption functions. The theorem is a corollary of the general validity theorem using lists in Section 6.8.3.1.

The proof of the theorem follows the procedure described in Section 6.9.2. As in the proof of the stream version of the theorem, we use the implementations of the CBC block and feedback functions mentioned in Section 6.10.1.1: again, these implementations have been proved to have the attributes required to apply the general validity theorem using lists, and, hence, the procedure works.

6.10.1.4 The CBC Plaintext Change Propagation Theorems

The CBC mode propagates a change of any plaintext block when encrypting. We prove this property for both the stream and list versions of the CBC mode by following the procedure outlined in Section 6.9.3.

First, we state the stream version of the CBC plaintext change propagation theorem.

```
Corollary CBC_plaintext_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CBC_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CBC_encryption_function_stream_version
        CBC_enc_sv →
      plaintext_change_propagation_stream_version CBC_enc_sv block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block (which any encryption function is: the corresponding decryption function must be able to tell the original plaintext block), then the CBC encryption function using streams propagates plaintext changes. The theorem is a corollary of the general plaintext change propagation theorem using streams in Section 6.8.3.2.

The proof of the theorem above follows the procedure described in Section 6.9.3. When doing so, we use the implementations of the CBC block and feedback functions for encryption mentioned in Section 6.10.1.1, and since these implementations have been proven to have the properties required to apply the general theorem, the procedure works.

The list version of the CBC plaintext change propagation theorem is similar.

```
Corollary CBC_plaintext_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CBC_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_CBC_encryption_function_list_version
        CBC_enc_lv →
      plaintext_change_propagation_list_version
        CBC_enc_lv block_enc.
```

Again, the underlying block encryption function being invertible with regards to the in-the-context-of-that-function plaintext block implies that the CBC encryption function propagates plaintext changes. The theorem is a corollary of the corresponding general theorem: the general plaintext change propagation theorem using lists in Section 6.8.3.2.

In the proof of the theorem above, we follow the procedure in Section 6.9.3. We use the implementations of the CBC block and feedback functions for encryption mentioned in Section 6.10.1.1: the properties required to apply the general theorem have been proved to hold for these implementations, and, therefore, the procedure works.

6.10.1.5 The CBC IV Change Propagation Theorems

The CBC mode of operation propagates changes of the IV when encrypting. We prove this property for both the stream and list versions of the mode by following the procedure in Section 6.9.5. We use the implementations mentioned in Section 6.10.1.1 of the CBC block and feedback functions for encryption. Since these implementations have been proven to have the attributes required to apply the general IV change propagation theorems in Section 6.8.3.4, the procedure works.

We state the CBC IV change propagation theorem using streams.

```
Corollary CBC_IV_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CBC_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CBC_encryption_function_stream_version
        CBC_enc_sv →
      IV_change_propagation_stream_version CBC_enc_sv block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block, then the CBC encryption function using streams propagates IV changes. The theorem is proven as a corollary of the general IV change propagation theorem using streams in Section 6.8.3.4.

Likewise, we state the CBC IV change propagation theorem using lists.

```
Corollary CBC_IV_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CBC_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_CBC_encryption_function_list_version
        CBC_enc_lv →
      IV_change_propagation_list_version CBC_enc_lv block_enc.
```

If the underlying block encryption function is invertible, then also the CBC encryption function using lists propagates IV changes. The theorem is a corollary of the general IV change propagation theorem using lists in Section 6.8.3.4.

6.10.1.6 The CBC-MAC Propagation Theorem

In Section 6.10.1.1, the CBC-MAC function is specified. This function has the property that if a block of the input message is changed, then the output MAC value is changed, too, if the underlying block

encryption is invertible with regards to the in-the-context-of-that-function plaintext block.

```
Corollary CBC_MAC_propagation_theorem :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CBC_MAC : MAC_fun_Type,
      specification_of_CBC_MAC_function CBC_MAC →
      MAC_propagation CBC_MAC block_enc.
```

The theorem above is a corollary of the general MAC theorem in Section 6.8.3.5.

We follow the procedure in Section 6.9.6. When doing so, we use the implementations mentioned in Section 6.10.1.1 of the CBC block and feedback functions for encryption: these implementations have been proven to be invertible with regards to both the plaintext blocks and the IVs or feedback blocks, and, hence, the procedure works.

6.10.2 The CFB Mode

The second mode of operation we consider is the Cipher Feedback (CFB) mode. Again, we follow the procedure described in Section 6.9.1.

6.10.2.1 Specifications

In this section, we take the first step of the procedure described in Section 6.9.1: first, we define the specifications of the CFB block and feedback functions, and, then, we specify the stream and list versions of the CFB encryption and decryption functions and the CFB-MAC function in terms of these specifications. In the following, we use the specification of the XOR-blocks-n function in Section 6.6.1. This function is used to compute the bitwise XORs of blocks of length n.

The specifications below reflect that both of the block functions for the CFB mode bitwise XOR the encryption of the IV or feedback block using the underlying block encryption function under the given key and the plain- or ciphertext block. The CFB mode does not use any underlying block *decryption* function.

```
Definition specification_of_CFB_block_function_for_encryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      block_fun xs ks IV block_enc = XOR_blocks_n_fun xs (block_enc IV ks).

Definition specification_of_CFB_block_function_for_decryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (cs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      block_fun cs ks IV block_enc = XOR_blocks_n_fun cs (block_enc IV ks).
```

Like for the CBC mode, the CFB feedback functions just output the ciphertext blocks.

```
Definition specification_of_CFB_feedback_function_for_encryption
           (feedback_fun : feedback_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      feedback_fun xs ks IV block_enc =
```

```
    XOR_blocks_n_fun xs (block_enc IV ks).
```

The CFB feedback function for encryption is not given the ciphertext block directly, but it is given the same inputs as the CFB block function for encryption. Hence, these functions are extensionally equal which is reflected by their similar specifications.

```
Definition specification_of_CFB_feedback_function_for_decryption
           (feedback_fun : feedback_fun_Type) :=
  forall (cs : block_n) (ks : block_k) (IV : block_n)
         (block_enc : block_enc_dec_Type),
    feedback_fun cs ks IV block_enc = cs.
```

The CFB feedback function for decryption is given the ciphertext block directly. Thus, the function can just pass on the block.

We implement block and feedback functions for the CFB mode: `CFB_block_fun_enc_v0` and `CFB_feedback_fun_enc_v0` are CFB block and feedback functions for encryption, and `CFB_block_fun_dec_v0` and `CFB_feedback_fun_dec_v0` are CFB block and feedback functions for decryption. These functions have been proven to satisfy the above specifications.

We now specify the CFB encryption and decryption functions using streams in terms of the specifications of the general encryption and decryption functions using streams in Section 6.8.1 and the specifications of the CFB block and feedback functions defined above.

```
Definition specification_of_CFB_encryption_function_stream_version
           (CFB_enc_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
    specification_of_CFB_block_function_for_encryption block_fun →
    specification_of_CFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CFB_enc_sv xss ks IV block_enc)
        (general_MoO_enc_sv xss ks IV block_enc block_fun feedback_fun).

Definition specification_of_CFB_decryption_function_stream_version
           (CFB_dec_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_decryption_function_stream_version
      general_MoO_dec_sv →
    specification_of_CFB_block_function_for_decryption block_fun →
    specification_of_CFB_feedback_function_for_decryption
      feedback_fun →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CFB_dec_sv css ks IV block_enc)
        (general_MoO_dec_sv css ks IV block_enc block_fun feedback_fun).
```

We specify outputs of CFB encryption and decryption functions using streams to be bisimilar to outputs of general encryption and decryption functions applied to CFB block and feedback functions.

Next, we specify the CFB encryption and decryption functions using lists also in terms of the specifications of the general encryption and decryption functions using lists in Section 6.8.2 and the specifications of the CFB block and feedback functions.

```
Definition specification_of_CFB_encryption_function_list_version
           (CFB_enc_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    specification_of_CFB_block_function_for_encryption block_fun →
    specification_of_CFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      CFB_enc_lv xss ks IV block_enc =
      general_MoO_enc_lv xss ks IV block_enc block_fun feedback_fun.

Definition specification_of_CFB_decryption_function_list_version
           (CFB_dec_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_dec_list_version :
            general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_decryption_function_list_version
      general_dec_list_version →
    specification_of_CFB_block_function_for_decryption block_fun →
    specification_of_CFB_feedback_function_for_decryption
      feedback_fun →
    forall (css : list block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      CFB_dec_lv css ks IV block_dec =
      general_dec_list_version css ks IV block_dec block_fun feedback_fun.
```

Outputs of the CFB encryption and decryption functions using lists are specified to be equal to outputs of the general encryption and decryption functions using lists applied to CFB block and feedback functions.

In Section 6.10.2.6, we prove the propagating property of the CFB-MAC function. Below, we specify the function.

```
Definition specification_of_CFB_MAC_function (CFB_MAC : MAC_fun_Type) :=
  forall (general_MAC : general_MAC_fun_Type)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MAC_function general_MAC →
    specification_of_CFB_block_function_for_encryption block_fun →
    specification_of_CFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k)
           (block_enc : block_enc_dec_Type),
      CFB_MAC xss ks block_enc =
      general_MAC xss ks block_enc block_fun feedback_fun.
```

The CFB-MAC function is specified similarly to the CBC-MAC function specified in Section 6.10.1.1: the CFB-MAC function is extensionally equal to the general MAC function (specified in Section 6.8.2) applied to CFB block and feedback functions for encryption.

6.10.2.2 Properties about the CFB Block and Feedback Functions

Taking the next step of the procedure in Section 6.9.1, we prove lemmas concerning a number of properties about the CFB block and feedback functions and their implementations. These lemmas facilitate proving the CFB theorems in Sections 6.10.2.3 to 6.10.2.6.

The general validity theorems in Section 6.8.3.1 assume that the block functions are inverse and that the feedback functions output equal values. Therefore, we prove these properties for the CFB block and feedback functions.

First, we prove that the CFB block functions are inverse.

```
Lemma CFB_block_functions_inverse :
  forall block_enc : block_enc_dec_Type,
  forall block_fun_enc block_fun_dec : block_fun_Type,
    specification_of_CFB_block_function_for_encryption block_fun_enc →
    specification_of_CFB_block_function_for_decryption block_fun_dec →
    block_functions_inverse
      block_fun_enc block_fun_dec
      block_enc block_enc.
```

The lemma above states that, for any underlying block encryption function, the CFB block functions are inverse.

In the section above, the outputs of these functions are specified to be equal to the bitwise XOR of the plaintext or ciphertext block (dependent on which block function is considered) and the encryption of the IV or feedback block under the given key using the underlying block encryption function. Thus, applying the block function for encryption and then the block function for decryption to a plaintext block amounts to bitwise XOR'ing the block twice with the encryption of the IV or feedback block. Proving that the block functions are inverse is equivalent to showing that the plaintext block results from applying the XOR-blocks-n function (used to compute the bitwise XOR) twice with the second input being the IV or feedback block.

In this regard, we apply the `XOR_blocks_n_fun_applied_twice_to_same_second_input_yields_first_input` lemma in Section 6.6.1: the first input to the XOR-blocks-n function is output if the function is applied twice with the same second input for both applications. Hence, the plaintext block is output above, and the block functions are inverse.

The CFB feedback functions output equal values regardlessly of the chosen underlying block encryption function.

```
Lemma CFB_feedback_functions_output_equal_values :
  forall block_enc : block_enc_dec_Type,
  forall (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
         (block_fun_enc : block_fun_Type),
    specification_of_CFB_feedback_function_for_encryption
      feedback_fun_enc →
    specification_of_CFB_feedback_function_for_decryption
      feedback_fun_dec →
    specification_of_CFB_block_function_for_encryption block_fun_enc →
    feedback_functions_output_equal_values
      feedback_fun_enc feedback_fun_dec
      block_fun_enc
      block_enc block_enc.
```

When proving the lemma above, we exploit that the outputs of the CFB feedback functions are specified in the section above to be equal to the ciphertext block: the feedback function for encryption is extensionally equal to the CFB block function for encryption and, hence, output a block equal to the ciphertext block, and the feedback function for decryption is given the ciphertext block as input and just pass it on. Therefore, the proof of the lemma essentially merely consists of unfolding the CFB feedback functions and the CFB block function for encryption and applying the `reflexivity` tactic.

The general plaintext change, IV change, and MAC propagation theorems in Sections 6.8.3.2, 6.8.3.4, and 6.8.3.5 assume that the block and feedback functions for encryption are invertible: they all assume that the functions are invertible with regards to the IVs or feedback blocks while the plaintext change and MAC propagation theorems assume, in addition, that the functions are invertible with regards to

the plaintext blocks, too. We apply these general theorems in the corresponding theorems for the CFB mode. Hence, we show that the CFB block and feedback functions for encryption are invertible.

First, we show that the CFB block and feedback functions for encryption are invertible with regards to the plaintext blocks for any underlying block encryption function.

```
Lemma CFB_block_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
  forall block_fun : block_fun_Type,
    specification_of_CFB_block_function_for_encryption block_fun →
    block_fun_invertible_wrt_plaintext block_fun block_enc.

Lemma CFB_feedback_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
  forall feedback_fun : feedback_fun_Type,
    specification_of_CFB_feedback_function_for_encryption
      feedback_fun →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc.
```

In Section 6.10.2.1, the outputs of the CFB block and feedback functions for encryption are specified to be equal to the bitwise XOR of the plaintext block and the IV or feedback block encrypted under the given key with the underlying block encryption function.

Since the plaintext block is given as the first input to the XOR-blocks-n function, which we use to compute the bitwise XOR, we have to show that the XOR-blocks-n function is invertible with regards to the first input. Here, we use the `XOR_blocks_n_fun_invertible_wrt_first_input` lemma in Section 6.6.1: that lemma states that the XOR-blocks-n is invertible with regards to the first input.

Finally, we show that the CFB block and feedback functions for encryption are invertible with regards to the feedback block if the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block.

```
Lemma CFB_block_fun_enc_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_CFB_block_function_for_encryption block_fun →
      block_fun_invertible_wrt_IV_feedback block_fun block_enc.

Lemma CFB_feedback_fun_enc_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall feedback_fun : feedback_fun_Type,
      specification_of_CFB_feedback_function_for_encryption
        feedback_fun →
      feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc.
```

As mentioned above, the outputs of the CFB block and feedback functions for encryption are both specified to be equal to the bitwise XOR of the plaintext block and the IV or feedback block encrypted under the given key using the underlying block encryption function. Hence, the functions are invertible with regards to the feedback block if the the XOR-blocks-n function, which we use to compute the bitwise XOR, is invertible with regards to its second input. Here, we use the `XOR_blocks_n_fun_invertible_wrt_second_input` lemma in Section 6.6.1: this lemma states that the XOR-blocks-n function *is* invertible with regards to its second input.

We prove that the properties in this section also hold for the implementations of the CFB block and feedback functions mentioned in the section above. Doing so allows us to use these implementations in the proofs of the CFB theorems in the following sections when we apply the corresponding general theorems. That the implementations have these properties is proven as a consequence of the lemmas in this section and that the implementations satisfy the corresponding specifications.

6.10.2.3 The CFB Validity Theorems

Both the stream and list versions of the CFB encryption scheme are valid. We prove this fact by following the procedure described in Section 6.9.2. When doing so, we use the implementations of the CFB block and feedback functions mentioned in Section 6.10.2.1: these implementations have been proven to have the properties required to apply the general validity theorems in Section 6.8.3.1—the implementations of the block functions are inverse, and the implementations of the feedback functions output equal values. Thus, the procedure works.

First, we state the stream version of the CFB validity theorem.

```
Corollary CFB_validity_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
  forall CFB_enc_sv CFB_dec_sv : MoO_enc_dec_Type_stream_version,
    specification_of_CFB_encryption_function_stream_version
      CFB_enc_sv →
    specification_of_CFB_decryption_function_stream_version
      CFB_dec_sv →
    MoO_valid_stream_version
      CFB_enc_sv CFB_dec_sv
      block_enc block_enc.
```

This theorem says that the CFB encryption scheme using streams is valid for any underlying block encryption function (remember, no block decryption function is used in the CFB mode). The theorem is a corollary of the general validity theorem using streams in Section 6.8.3.1.

Likewise, we state the list version of the CFB validity theorem.

```
Corollary CFB_validity_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
  forall CFB_enc_lv CFB_dec_lv : MoO_enc_dec_Type_list_version,
    specification_of_CFB_encryption_function_list_version CFB_enc_lv →
    specification_of_CFB_decryption_function_list_version CFB_dec_lv →
    MoO_valid_list_version
      CFB_enc_lv CFB_dec_lv
      block_enc block_enc.
```

Also the CFB mode using lists is valid for any underlying block encryption function. The theorem above is a corollary of the general validity theorem using lists in Section 6.8.3.1.

6.10.2.4 The CFB Plaintext Change Propagation Theorems

The CFB encryption function propagates plaintext changes. We prove this property for both the stream and list versions of the function by following the procedure in Section 6.9.3. The procedure calls for the use of implementations of the CFB block and feedback functions for encryption which have been proven to be invertible with regards to both the plaintext blocks and the IVs or feedback blocks: this fact is required to apply the general plaintext change propagation theorems (in Section 6.8.3.2) of which the theorems in this section are corollaries. Since we use the implementations mentioned in Section 6.10.2.1 for which we have proven invertibility, the procedure works.

First, we consider the stream version of the CFB plaintext change propagation theorem.

```
Corollary CFB_plaintext_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CFB_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CFB_encryption_function_stream_version
        CFB_enc_sv →
      plaintext_change_propagation_stream_version CFB_enc_sv block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block, then the CFB encryption function using streams propagates plaintext changes. The theorem is a corollary of the general plaintext change propagation theorem using streams in Section 6.8.3.2.

Next, we state the list version of the CFB plaintext change propagation theorem.

```
Corollary CFB_plaintext_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CFB_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_CFB_encryption_function_list_version
        CFB_enc_lv →
      plaintext_change_propagation_list_version
        CFB_enc_lv block_enc.
```

As the stream version of the CFB encryption function, the list version propagates plaintext changes if the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block. The theorem is a corollary of the general plaintext change propagation theorem using lists in Section 6.8.3.2.

6.10.2.5 The CFB IV Change Propagation Theorems

We prove the IV change propagation property for both the stream and list versions of the CFB encryption function. We do so by following the procedure in Section 6.9.5. Here, we use implementations of block and feedback functions for encryption that have been proven to be invertible with regards to the IVs or feedback blocks so that the general IV change propagation theorems in Section 6.8.3.4 can be applied. In this regard, we use the implementations mentioned in Section 6.10.2.1 of the CFB block and feedback functions for encryption: these implementations have been proven to be invertible, and, thus, the procedure works.

We state the stream version of the CFB IV change propagation theorem.

```
Corollary CFB_IV_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CFB_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CFB_encryption_function_stream_version
        CFB_enc_sv →
      IV_change_propagation_stream_version CFB_enc_sv block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block, then the CFB encryption function using streams propagates IV changes. The theorem is a corollary of the general IV change propagation theorem using streams in Section 6.8.3.4.

Likewise, we state the list version of the CFB IV change propagation theorem.

```
Corollary CFB_IV_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CFB_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_CFB_encryption_function_list_version
        CFB_enc_lv →
      IV_change_propagation_list_version CFB_enc_lv block_enc.
```

Also the CFB encryption function using lists propagates IV changes if the underlying block encryption function is invertible. This latter theorem is a corollary of the general IV change propagation theorem using lists in Section 6.8.3.4.

6.10.2.6 The CFB-MAC Propagation Theorem

We prove that the CFB-MAC function specified in Section 6.10.2.1 propagates changes of the input messages if the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block. This property is stated in the CFB-MAC propagation theorem below.

```
Corollary CFB_MAC_propagation_theorem :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CFB_MAC : MAC_fun_Type,
      specification_of_CFB_MAC_function CFB_MAC →
      MAC_propagation CFB_MAC block_enc.
```

This theorem is a corollary of the general MAC propagation theorem in Section 6.8.3.5.

When proving the theorem above, we follow the procedure in Section 6.9.6. Following this procedure, we need implementations of the CFB block and feedback functions for encryption that are invertible with regards to both the plaintext blocks and the IVs or feedback blocks: here, we use the implementations mentioned in Section 6.10.2.1 for which these properties have been proved. Hence, the procedure works.

6.10.3 The IFB Mode

In this section, we specify and prove theorems about the Input Feedback (IFB) mode defined in Section 6.4.3. We follow the procedure outlined in Section 6.9.1.

6.10.3.1 Specifications

The first step of the procedure in Section 6.9.1 consists of specifying and implementing the block and feedback functions and, subsequently, specify the encryption and decryption functions using streams or list and, potentially, the MAC function in terms of the former functions. For the IFB mode, we specify the corresponding MAC function, the IFB-MAC function, since the mode propagates plaintext changes when encrypting. In this section, we take this first step. In doing so, we use the XOR-blocks-n function specified in Section 6.6.1 to compute the bitwise XORs of blocks of length n.

First, we consider the specification of the IFB block function for encryption.

```
Definition specification_of_IFB_block_function_for_encryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      block_fun xs ks IV block_enc = block_enc (XOR_blocks_n_fun xs IV) ks.
```

The IFB block function for encryption is extensionally equal to the CBC block function for encryption (specified in Section 6.10.1.1): the bitwise XOR of the plaintext block and the IV or feedback block is encrypted under the given key forming the output of the function.

Next, we specify the IFB feedback function for encryption.

```
Definition specification_of_IFB_feedback_function_for_encryption
           (feedback_fun : feedback_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs : block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      feedback_fun xs ks IV block_enc = XOR_blocks_n_fun xs IV.
```

The feedback block is computed as the bitwise XOR of the plaintext block and the prior feedback block—which is the input to the underlying block encryption function used in the block function for encryption defined above: hence the name of the mode.

We specify the IFB block function for decryption.

```
Definition specification_of_IFB_block_function_for_decryption
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (cs : block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      block_fun cs ks IV block_dec = XOR_blocks_n_fun (block_dec cs ks) IV.
```

Similarly to the IFB block function for encryption, the block function for decryption is extensionally equal to the CBC block function for decryption: the output of the block function is specified to be equal to the bitwise XOR of the decryption of the ciphertext block under the given key (using the underlying block decryption function) and the IV or feedback block.

Lastly, we consider the IFB feedback function for decryption.

```
Definition specification_of_IFB_feedback_function_for_decryption
           (feedback_fun : feedback_fun_Type) :=
  forall (cs : block_n) (ks : block_k) (IV : block_n)
         (block_dec : block_enc_dec_Type),
    feedback_fun cs ks IV block_dec = block_dec cs ks.
```

The output of the feedback function for decryption is specified to be equal to the decryption of the ciphertext block using the underlying block decryption function under the given key.

We implement functions that are proven to satisfy the specifications above: the `IFB_block_fun_enc_v0` respectively `IFB_feedback_fun_enc_v0` function is an implementation of the IFB block respectively feedback function for encryption while the `IFB_block_fun_dec_v0` respectively `IFB_feedback_fun_dec_v0` function is an implementation of the IFB block respectively feedback function for decryption. These implementations are used when we apply the general theorems in Section 6.8.3 in the proofs of the IFB theorems in Sections 6.10.3.3 to 6.10.3.6.

With the specifications of the IFB block and feedback functions defined, we can specify the IFB encryption and decryption functions in terms of these specifications and the specifications of the general encryption and decryption functions in Sections 6.8.1 and 6.8.2.

First, we consider the stream version of the IFB encryption function.

```
Definition specification_of_IFB_encryption_function_stream_version
           (IFB_enc_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
    specification_of_IFB_block_function_for_encryption block_fun →
    specification_of_IFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (IFB_enc_sv xss ks IV block_enc)
        (general_MoO_enc_sv xss ks IV block_enc block_fun feedback_fun).
```

The output of the IFB encryption function using streams is specified to be bisimilar to the output of the general encryption function (using streams) when applied to the same inputs and the IFB block and feedback functions for encryption.

Next, we consider the stream version of the IFB decryption function.

```
Definition specification_of_IFB_decryption_function_stream_version
           (IFB_dec_sv : MoO_enc_dec_Type_stream_version) :=
  forall (general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_decryption_function_stream_version
      general_MoO_dec_sv →
    specification_of_IFB_block_function_for_decryption block_fun →
    specification_of_IFB_feedback_function_for_decryption
      feedback_fun →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (IFB_dec_sv css ks IV block_dec)
        (general_MoO_dec_sv css ks IV block_dec block_fun feedback_fun).
```

This function is specified analogously to the IFB encryption function using streams above: the output of the function is specified to be bisimilar to the output of the general decryption function (using streams) when applied to the same inputs and, in addition, the IFB block and feedback functions for decryption.

Similarly, we specify the list versions of the IFB encryption and decryption functions.

```
Definition specification_of_IFB_encryption_function_list_version
           (IFB_enc_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_MoO_enc_lv : general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_MoO_enc_lv →
    specification_of_IFB_block_function_for_encryption block_fun →
    specification_of_IFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      IFB_enc_lv xss ks IV block_enc =
      general_MoO_enc_lv xss ks IV block_enc block_fun feedback_fun.

Definition specification_of_IFB_decryption_function_list_version
           (IFB_dec_lv : MoO_enc_dec_Type_list_version) :=
  forall (general_MoO_dec_lv : general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MoO_decryption_function_list_version
      general_MoO_dec_lv →
    specification_of_IFB_block_function_for_decryption block_fun →
    specification_of_IFB_feedback_function_for_decryption
      feedback_fun →
    forall (css : list block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      IFB_dec_lv css ks IV block_dec =
      general_MoO_dec_lv
        css ks IV block_dec block_fun feedback_fun.
```

Now, we use the list versions of the general encryption and decryption functions and the normal notion of equality instead of bisimilarity: the outputs of the IFB encryption respectively decryption function is specified to be equal to the output of the list version of the general encryption respectively decryption function using lists applied to the same inputs and the IFB block and feedback functions for encryption respectively decryption.

Finally, we specify the IFB-MAC function.

```
Definition specification_of_IFB_MAC_function (IFB_MAC : MAC_fun_Type) :=
  forall (general_MAC : general_MAC_fun_Type)
         (block_fun : block_fun_Type) (feedback_fun : feedback_fun_Type),
    specification_of_general_MAC_function general_MAC →
    specification_of_IFB_block_function_for_encryption block_fun →
    specification_of_IFB_feedback_function_for_encryption
      feedback_fun →
    forall (xss : list block_n) (ks : block_k)
           (block_enc : block_enc_dec_Type),
      IFB_MAC xss ks block_enc =
      general_MAC xss ks block_enc block_fun feedback_fun.
```

The MAC value output by the IFB-MAC function is specified to be equal to the output of the general MAC function specified in Section 6.8.2 when applied to the same inputs and the IFB block and feedback functions for encryption.

6.10.3.2 Properties about the Block and Feedback Functions

The next step of the procedure in Section 6.9.1 is to prove several properties about the block and feedback functions and their implementations. Doing so allows us to apply the general theorems in Section 6.8.3 when proving the IFB theorems in Sections 6.10.3.3 to 6.10.3.6: the general theorems assumes that these properties hold for the used implementations.

First, we prove that the IFB block functions are inverse and that the IFB feedback functions output equal values: these properties are assumed by the general validity theorems in Section 6.8.3.1.

The IFB block functions are inverse if the underlying block encryption scheme is valid.

```
Lemma IFB_block_functions_inverse :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall block_fun_enc block_fun_dec : block_fun_Type,
      specification_of_IFB_block_function_for_encryption
        block_fun_enc →
      specification_of_IFB_block_function_for_decryption
        block_fun_dec →
      block_functions_inverse
        block_fun_enc block_fun_dec
        block_enc block_dec.
```

We use the `block_functions_inverse` predicate defined in Section 6.7.3 in the conclusion of the lemma above: we have to show that applying first the block function for encryption and then the block function for decryption on any plaintext block outputs again the plaintext block.

In the section above, the output of the block function for encryption is specified to be equal to encryption of the bitwise XOR of the plaintext block and the IV or feedback block using the underlying block encryption function, and the output of the block function for decryption is specified to be equal to bitwise XOR of the decryption of the ciphertext block (in the context of the block function) using the underlying block decryption function and the IV or feedback block. Hence, applying the block function for encryption and then the block function for decryption on a plaintext block is extensionally equal to bitwise XOR'ing the plaintext block with the IV or feedback block, then applying first the underlying block encryption function and then the underlying block decryption function, and, finally, bitwise XOR'ing with the IV or feedback block again.

The proof consists of showing that this order of operations outputs the plaintext block. First, we use the premise stating that the underlying block encryption scheme is valid: the applications of the underlying block encryption and decryption functions cancel each other. Hence, the rest of the proof amounts to showing that bitwise XOR'ing twice with the IV or feedback block is extensionally equal to doing

nothing. Here, we use the `XOR_blocks_n_fun_applied_twice_to_same_second_input_yields_first_input` lemma (in Section 6.6.1) that states that bitwise XOR'ing any block (of length `n`) twice with some block (also of length `n`) yields the first block: the last two operations consisting of XOR'ing with the IV or feedback block cancel out, and the plaintext block is output.

The underlying block encryption scheme being valid also implies that the IFB feedback functions output equal values.

```
Lemma IFB_feedback_functions_output_equal_values :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall (feedback_fun_enc feedback_fun_dec : feedback_fun_Type)
           (block_fun_enc : block_fun_Type),
      specification_of_IFB_feedback_function_for_encryption
        feedback_fun_enc →
      specification_of_IFB_feedback_function_for_decryption
        feedback_fun_dec →
      specification_of_IFB_block_function_for_encryption
        block_fun_enc →
      feedback_functions_output_equal_values
        feedback_fun_enc feedback_fun_dec
        block_fun_enc
        block_enc block_dec.
```

In the conclusion of the lemma, we use the `feedback_functions_output_same` predicate defined in Section 6.7.3: in the proof of the lemma, we have to show that the output of the feedback function for encryption applied to a plaintext block is equal to the output of the feedback function for decryption applied to the corresponding ciphertext block (computed via the block function for encryption).

In the section above,

- the output of the IFB feedback function for encryption is specified to be equal to the bitwise XOR of the plaintext block and the IV or feedback block,
- the output of the IFB feedback function for decryption is specified to be equal to the decryption of the ciphertext block using the underlying block decryption function under the given key,
- and the output of the IFB block function for encryption is specified to be equal to the encryption of the bitwise XOR of the plaintext block and the IV or feedback block using the underlying block encryption function under the given key.

Putting these specifications together, the goal of the proof of the lemma above is to show that that first encrypting and then decrypting a block equal to the bitwise XOR of the plaintext block and the IV or feedback block using the underlying block encryption scheme under the given key yields the block. This property, however, follows directly from the premise (of the lemma) stating that the underlying block encryption scheme is valid: the operations of encryption and decryption cancel out, and the input block is output.

In the proofs of the IFB propagation theorems in Sections 6.10.3.4 to 6.10.3.6, we need that the IFB block and feedback functions for encryption are invertible with regards to the plaintext block and the IV or feedback block: in the proofs of the IFB plaintext change propagation theorems in Section 6.10.3.4 and the IFB-MAC propagation theorem in Section 6.10.3.6, we need that the block and feedback functions for encryption are invertible with regards to *both* the plaintext block and the IV or feedback block while we only need the functions to be invertible with regards to the IV or feedback block in the proof of the IV change propagation theorems in Section 6.10.3.5. Hence, in the following, we prove that these properties hold for the IFB block and feedback functions for encryption—possibly as a consequence of the underlying block encryption function being invertible with regards to the in-the-context-of-that-function plaintext block.

The IFB block function for encryption is invertible with regards to the plaintext block and the IV or feedback block if the underlying block encryption function is invertible.

```
Lemma IFB_block_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_IFB_block_function_for_encryption block_fun →
      block_fun_invertible_wrt_plaintext block_fun block_enc.


Lemma IFB_block_fun_enc_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_IFB_block_function_for_encryption block_fun →
      block_fun_invertible_wrt_IV_feedback block_fun block_enc.
```

In the section above, the output of the IFB block function for encryption is specified to be equal to the encryption of the bitwise XOR of the plaintext block and the IV or feedback block using the underlying block encryption function under the given key. Hence, in the proofs of the lemmas above, we just have to show that the underlying block encryption function is invertible and that the XOR-blocks-n function is invertible with regards to its first input in the proof of the first lemma above and invertible with regards to its second input in the proof of the second lemma above when the other input is fixed.

The first requirement, that the underlying block encryption function is invertible, is a premise of the two lemmas. The latter requirement, that the XOR-blocks-n function is invertible with regards to the first or the second input follows from the `XOR_blocks_n_fun_invertible_wrt_first_input` and `XOR_blocks_n_fun_invertible_wrt_second_input` lemmas in Section 6.6.1: these lemmas state exactly these properties about the XOR-blocks-n function.

The IFB feedback function for encryption is invertible with regards to the plaintext block and the IV or feedback block—regardlessly of the choice of underlying block encryption function.

```
Lemma IFB_feedback_fun_enc_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
  forall feedback_fun : feedback_fun_Type,
    specification_of_IFB_feedback_function_for_encryption
      feedback_fun →
    feedback_fun_invertible_wrt_plaintext feedback_fun block_enc.


Lemma IFB_feedback_fun_enc_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
  forall feedback_fun : feedback_fun_Type,
    specification_of_IFB_feedback_function_for_encryption
      feedback_fun →
    feedback_fun_invertible_wrt_IV_feedback feedback_fun block_enc.
```

In the section above, the output of the feedback function for encryption is specified to be equal to the bitwise XOR of the plaintext block and the IV or feedback block. Thus, again, the goals of the lemmas are to show that the XOR-blocks-n function is invertible with regards to the first input or the second input: as above, we apply the `XOR_blocks_n_fun_invertible_wrt_first_input` and `XOR_blocks_n_fun_invertible_wrt_second_input` lemmas in this regard.

The properties in this section have been proved for the implementations of the IFB block and feedback functions mentioned in the section above since we use these implementations when applying the general theorems in the proofs of the IFB theorems: the general theorems assume the properties to hold for the input implementations.

6.10.3.3 The IFB Validity Theorems

The IFB encryption scheme is valid—both when using streams and using lists. We prove this attribute by following the procedure in Section 6.9.2. The procedure makes use of implementations of the block and feedback functions for which it assumes that two properties hold: the implementations of the block functions are assumed to be inverse, and the implementations of the feedback functions are assumed to output equal values. However, we use the implementations of the IFB block and feedback functions mentioned in Section 6.10.3.1, and, since these implementations have been proven to satisfy the properties, the procedure works.

We state the stream version of the IFB validity theorem.

```
Corollary IFB_validity_theorem_stream_version :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall IFB_enc_sv IFB_dec_sv : MoO_enc_dec_Type_stream_version,
      specification_of_IFB_encryption_function_stream_version
        IFB_enc_sv →
      specification_of_IFB_decryption_function_stream_version
        IFB_dec_sv →
      MoO_valid_stream_version
        IFB_enc_sv IFB_dec_sv
        block_enc block_dec.
```

The stream version of the IFB mode of operation is valid if the underlying block encryption scheme is valid. The theorem is a corollary of the general validity theorem using streams in Section 6.8.3.1.

Likewise, we state the list version of the IFB validity theorem.

```
Corollary IFB_validity_theorem_list_version :
  forall block_enc block_dec : block_enc_dec_Type,
    block_enc_scheme_valid block_enc block_dec →
    forall IFB_enc_lv IFB_dec_lv : MoO_enc_dec_Type_list_version,
      specification_of_IFB_encryption_function_list_version
        IFB_enc_lv →
      specification_of_IFB_decryption_function_list_version
        IFB_dec_lv →
      MoO_valid_list_version
        IFB_enc_lv IFB_dec_lv
        block_enc block_dec.
```

Also the IFB mode of operation is valid if the underlying block encryption scheme is so. The theorem is a corollary of the general validity theorem using lists in Section 6.8.3.1.

6.10.3.4 The IFB Plaintext Change Propagation Theorems

The stream and list versions of the IFB encryption function propagate plaintext changes. When proving this property, we follow the procedure described in Section 6.9.3. When doing so, we use the implementations of the IFB block and feedback functions for encryption mentioned in Section 6.10.3.1. The procedure assumes that these implementations are invertible with regards to the plaintext blocks and IVs or feedback blocks: this property has been proven, and, therefore, the procedure works.

We state the IFB plaintext change propagation theorem using streams.

```
Corollary IFB_plaintext_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall IFB_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_IFB_encryption_function_stream_version
        IFB_enc_sv →
```

```
    plaintext_change_propagation_stream_version IFB_enc_sv block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block, then the IFB encryption function using streams propagates plaintext changes. The theorem is a corollary of the general plaintext change propagation theorem using streams in Section 6.8.3.2.

Similarly, we state the list version of the IFB plaintext change propagation theorem.

```
Corollary IFB_plaintext_change_propagation_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext block_enc →
    forall IFB_enc_list_version : MoO_Type_list_version,
      specification_of_IFB_encryption_list_version
        IFB_enc_list_version →
      MoO_propagation_list_version
        IFB_enc_list_version block_enc.
```

If the underlying block encryption function is invertible, then the list version of the IFB encryption function, too, propagates plaintext changes. The theorem is proven as a corollary of the general plaintext change propagation theorem using lists in Section 6.8.3.2.

6.10.3.5 The IFB IV Change Propagation Theorems

The stream and list versions of the IFB encryption functions propagates IV changes. We prove this property by following the procedure in Section 6.9.5.

The procedure makes use of implementations of the block and feedback functions for encryption and assumes that they are invertible with regards to the IVs or feedback blocks. When following the procedure, we use the implementations mentioned in Section 6.10.3.1 of the IFB block and feedback functions for encryption. Since these implementations have been proven to be invertible, the procedure works. However, when proving that the implementation of the block function is invertible, we assume that the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block. Hence, this assumption is included as premises of the theorems in this section.

First, we state the IFB IV change propagation theorem using streams.

```
Corollary IFB_IV_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall IFB_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_IFB_encryption_function_stream_version
        IFB_enc_sv →
      IV_change_propagation_stream_version IFB_enc_sv block_enc.
```

If the underlying block encryption function is invertible, then the stream version of the IFB encryption function propagates IV changes. The theorem is a corollary of the stream version of the general IV change propagation theorem in Section 6.8.3.4.

Next, we state the IFB IV change propagation theorem using lists.

```
Corollary IFB_IV_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall IFB_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_IFB_encryption_function_list_version
        IFB_enc_lv →
      IV_change_propagation_list_version IFB_enc_lv block_enc.
```

Similarly to the stream version of the IFB encryption function, the list version propagates IV changes if the underlying block encryption function is invertible. The theorem is proven as a corollary of the general IV change propagation theorem using lists in Section 6.8.3.4.

6.10.3.6 The IFB-MAC Propagation Theorem

The IFB-MAC function propagates changes in the messages input to it. We state this fact in the IFB-MAC propagation theorem below.

```
Corollary IFB_MAC_propagation_theorem :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall IFB_MAC : MAC_fun_Type,
      specification_of_IFB_MAC_function IFB_MAC →
      MAC_propagation IFB_MAC block_enc.
```

If the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block, then the IFB-MAC function propagates changes in the messages. The theorem is a corollary of the general MAC propagation theorem in Section 6.8.3.5.

When proving the theorem above, we follow the procedure in Section 6.9.6. That procedure uses implementations of the block and feedback functions for encryption. In this regard, we use the implementations of the IFB block and feedback functions for encryption mentioned in Section 6.10.3.1. The procedure assumes that the used implementations are invertible with regards to both the plaintext blocks and the IVs or feedback blocks: this property has been proven for the used implementations in Section 6.10.3.2, and, hence, the procedure works. However, when proving that the implementation of the IFB block function for encryption is invertible, we assume that the underlying block encryption function is invertible. Therefore, this latter property is a premise in the theorem above.

6.10.4 The f, CTR, and BENC Modes

In this section, we specify the f, CTR, and BENC modes defined in Section 6.4.4. The CTR and BENC modes are special cases of the f mode: the f function of the CTR mode is the counter-n function specified in Section 6.6.2, and the f function of the BENC mode is the underlying block encryption function with the key already provided—applying the f function of the BENC mode is extensionally equal to encrypting the input block under the key.

In large parts, we follow the procedure in Section 6.9.1. However, a few deviations from the procedure occur as the f mode is different from the other modes in that its encryption and decryption functions take an extra input, the f function, and the CTR and BENC modes are special cases of the f mode.

We use the XOR-blocks-n function, in this section, to compute the bitwise XORs of blocks of length n and the make-block_enc-unary function to turn the underlying block encryption function into a unary function.

6.10.4.1 Extra Types and Predicate

The feedback function for the f mode takes an extra input compared to normal feedback functions: an f function. This function is applied to the input IV or feedback block to form the output of the feedback function. Likewise, the encryption and decryption functions (both the stream and list versions) take as input an f function which is passed on to the feedback function.

The domain and codomain of any f function is the set of blocks of length n: the type of any f function is block_n → block_n. Therefore, the type of the feedback function for the f mode is as defined below.

```
Definition feedback_fun_w_f_Type : Type :=
  (block_n → block_n) → feedback_fun_Type.
```

Likewise, the types of the encryption and decryption function using streams respectively lists are defined below.

```
Definition MoO_enc_dec_w_f_Type_stream_version : Type :=
  (block_n → block_n) → MoO_enc_dec_Type_stream_version.
```

```
Definition MoO_enc_dec_w_f_Type_list_version : Type :=
  (block_n → block_n) → MoO_enc_dec_Type_list_version.
```

When proving that the feedback function of the f mode is invertible with regards to the IV or feedback block in Section 6.10.4.2 and that the stream and list versions of the encryption function of the f mode propagates IV changes, we assume that the f function is invertible. In these cases, we use the `f_invertible` predicate defined below.

```
Definition f_invertible (f : block_n → block_n) :=
  forall IV_1 IV_2 : block_n,
    f IV_1 = f IV_2 →
    IV_1 = IV_2.
```

6.10.4.2 Specifications

Following the procedure in Section 6.9.1, we first specify and implement the f mode block and feedback functions. Then, we specify the stream and list versions of the f mode encryption and decryption functions in terms of the specifications of the block and feedback functions and the specifications of the general encryption and decryption functions in Sections 6.8.1 and 6.8.2. Since the f, CTR, and BENC modes do not propagate plaintext changes, we do not specify any corresponding MAC functions. Deviating from the procedure, we specify the CTR and BENC encryption and decryption functions in terms of the f mode counterparts.

First, we specify the f mode block function for both encryption and decryption.

```
Definition specification_of_f_mode_block_function
           (block_fun : block_fun_Type) :=
  forall XOR_blocks_n_fun : block_n → block_n → block_n,
    specification_of_XOR_blocks_n_function XOR_blocks_n_fun →
    forall (xs_cs : block_n) (ks :  block_k) (IV :  block_n)
           (block_enc : block_enc_dec_Type),
      block_fun xs_cs ks IV block_enc =
      XOR_blocks_n_fun xs_cs (block_enc IV ks).
```

The input plaintext or ciphertext block is bitwise XOR'ed with the encryption of the input IV or feedback block under the given key using the underlying block encryption function—no underlying block decryption function is used in the f mode (and, consequently, neither in the BENC mode).

Likewise, we specify the f mode feedback function for both encryption and decryption.

```
Definition specification_of_f_mode_feedback_function_w_f
           (feedback_fun_w_f : feedback_fun_w_f_Type) :=
  forall (f :  block_n → block_n)
         (xs_cs :  block_n) (ks :  block_k) (IV :  block_n)
         (block_enc : block_enc_dec_Type),
    feedback_fun_w_f f xs_cs ks IV block_enc = f IV.
```

The f function is applied to the IV or feedback block.

We define functions, the `f_mode_block_fun_v0` and `f_mode_feedback_fun_w_f_v0` functions, that satisfy the specifications above.

We specify the f mode encryption and decryption functions using streams.

```
Definition specification_of_f_mode_encryption_function_w_f_stream_version
           (f_mode_enc_w_f_sv : MoO_enc_dec_w_f_Type_stream_version) :=
  forall (general_MoO_enc_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun_w_f : feedback_fun_w_f_Type),
    specification_of_general_MoO_encryption_function_stream_version
      general_MoO_enc_sv →
```

```
    specification_of_f_mode_block_function block_fun →
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    forall (f : block_n → block_n)
           (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (f_mode_enc_w_f_sv f xss ks IV block_enc)
        (general_MoO_enc_sv
           xss ks IV block_enc block_fun (feedback_fun_w_f f)).

Definition specification_of_f_mode_decryption_function_w_f_stream_version
           (f_mode_dec_w_f : MoO_enc_dec_w_f_Type_stream_version) :=
  forall (general_MoO_dec_sv : general_MoO_enc_dec_Type_stream_version)
         (block_fun : block_fun_Type)
         (feedback_fun_w_f : feedback_fun_w_f_Type),
    specification_of_general_MoO_decryption_function_stream_version
      general_MoO_dec_sv →
    specification_of_f_mode_block_function block_fun →
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    forall (f : block_n → block_n)
           (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (f_mode_dec_w_f f css ks IV block_enc)
        (general_MoO_dec_sv
           css ks IV block_enc block_fun (feedback_fun_w_f f)).
```

The output of the f mode encryption respectively decryption function using streams is specified to be bisimilar to the output of the general encryption respectively decryption function (using streams) applied to the f mode block function and the f mode feedback function applied to the input f function.

Likewise, we specify the f mode encryption and decryption functions using lists.

```
Definition specification_of_f_mode_encryption_function_w_f_list_version
           (f_mode_enc_w_f_lv : MoO_enc_dec_w_f_Type_list_version) :=
  forall (general_enc_list_version :
            general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type)
         (feedback_fun_w_f : feedback_fun_w_f_Type),
    specification_of_general_MoO_encryption_function_list_version
      general_enc_list_version →
    specification_of_f_mode_block_function block_fun →
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    forall (f : block_n → block_n)
           (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      f_mode_enc_w_f_lv f xss ks IV block_enc =
      general_enc_list_version
        xss ks IV block_enc block_fun (feedback_fun_w_f f).

Definition specification_of_f_mode_decryption_function_w_f_list_version
           (f_mode_dec_w_f_lv : MoO_enc_dec_w_f_Type_list_version) :=
  forall (general_dec_list_version :
            general_MoO_enc_dec_Type_list_version)
         (block_fun : block_fun_Type)
         (feedback_fun_w_f : feedback_fun_w_f_Type),
    specification_of_general_MoO_decryption_function_list_version
      general_dec_list_version →
```

```
    specification_of_f_mode_block_function block_fun →
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    forall (f : block_n → block_n)
           (css : list block_n) (ks : block_k) (IV : block_n)
           (block_dec : block_enc_dec_Type),
      f_mode_dec_w_f_lv f css ks IV block_dec =
      general_dec_list_version
        css ks IV block_dec block_fun (feedback_fun_w_f f).
```

Similarly to above, the output of the f mode encryption respective decryption function using lists is specified to be equal to the output of the general encryption respectively decryption function (using lists) applied to the block and feedback functions of the f mode where the feedback function is applied to the input f function.

The theorems concerning the CTR and BENC modes in Sections 6.10.4.4 to 6.10.4.6 are corollaries of the f mode theorems (in the same sections). We implement the encryption and decryption functions of the f mode to facilitate applying the f mode theorems in the proofs of the CTR and BENC theorems: the `f_mode_enc_w_f_sv_v0` and `f_mode_dec_w_f_sv_v0` functions implement the f mode encryption and decryption functions using streams, and the `f_mode_enc_w_f_lv_v0` and `f_mode_dec_w_f_lv_v0` functions implement the f mode encryption and decryption functions using lists.

We specify the CTR and BENC encryption and decryption functions in terms of the f mode analogues.

First, we specify the CTR functions; the CTR mode is a special case of the f mode where the f function is the counter-n function specified in Section 6.6.2. Hence, the CTR encryption and decryption functions are extensionally bisimilar or equal to the f mode encryption and decryption functions applied to the counter-n function as the f function.

We specify the CTR encryption and decryption functions using streams.

```
Definition specification_of_CTR_encryption_function_stream_version
           (CTR_enc_sv : MoO_enc_dec_Type_stream_version) :=
  forall (f_mode_enc_w_f_sv : MoO_enc_dec_w_f_Type_stream_version)
         (counter_n_fun : block_n → block_n),
    specification_of_f_mode_encryption_function_w_f_stream_version
      f_mode_enc_w_f_sv →
    specification_of_counter_n_function counter_n_fun →
    forall (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CTR_enc_sv xss ks IV block_enc)
        (f_mode_enc_w_f_sv counter_n_fun xss ks IV block_enc).

Definition specification_of_CTR_decryption_function_stream_version
           (CTR_dec_sv : MoO_enc_dec_Type_stream_version) :=
  forall (f_mode_dec_w_f_sv : MoO_enc_dec_w_f_Type_stream_version)
         (counter_n_fun : block_n → block_n),
    specification_of_f_mode_decryption_function_w_f_stream_version
      f_mode_dec_w_f_sv →
    specification_of_counter_n_function counter_n_fun →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (CTR_dec_sv css ks IV block_enc)
        (f_mode_dec_w_f_sv counter_n_fun css ks IV block_enc).
```

Similarly, we specify the CTR encryption and decryption functions using lists.

```
Definition specification_of_CTR_encryption_function_list_version
```

```
          (CTR_enc_lv : MoO_enc_dec_Type_list_version) :=
  forall (f_mode_enc_w_f_lv : MoO_enc_dec_w_f_Type_list_version)
         (counter_n_fun : block_n → block_n),
    specification_of_f_mode_encryption_function_w_f_list_version
      f_mode_enc_w_f_lv →
    specification_of_counter_n_function counter_n_fun →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      CTR_enc_lv xss ks IV block_enc =
      f_mode_enc_w_f_lv counter_n_fun xss ks IV block_enc.

Definition specification_of_CTR_decryption_function_list_version
          (CTR_dec_lv : MoO_enc_dec_Type_list_version) :=
  forall (f_mode_dec_w_f_lv : MoO_enc_dec_w_f_Type_list_version)
         (counter_n_fun : block_n → block_n),
    specification_of_f_mode_decryption_function_w_f_list_version
      f_mode_dec_w_f_lv →
    specification_of_counter_n_function counter_n_fun →
    forall (css : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      CTR_dec_lv css ks IV block_enc =
      f_mode_dec_w_f_lv counter_n_fun css ks IV block_enc.
```

Next, we specify the BENC functions; the BENC mode is a special case of the f mode where the f function is the underlying block encryption function with the key already provided. Thus, the f function for the BENC mode is the function output by the make-block_enc-unary function when applied to the underlying block encryption function and the key. This f function is used for both encryption and decryption, as enforced by the f mode: no underlying block decryption function is used.

First, we specify the BENC functions using streams.

```
Definition specification_of_BENC_encryption_function_stream_version
          (BENC_enc_sv : MoO_enc_dec_Type_stream_version) :=
  forall (f_mode_enc_w_f_sv : MoO_enc_dec_w_f_Type_stream_version),
    specification_of_f_mode_encryption_function_w_f_stream_version
      f_mode_enc_w_f_sv →
    forall (xss : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (BENC_enc_sv xss ks IV block_enc)
        (f_mode_enc_w_f_sv
           (make_block_enc_unary block_enc ks) xss ks IV block_enc).

Definition specification_of_BENC_decryption_function_stream_version
          (BENC_dec_sv : MoO_enc_dec_Type_stream_version) :=
  forall (f_mode_dec_w_f_sv : MoO_enc_dec_w_f_Type_stream_version),
    specification_of_f_mode_decryption_function_w_f_stream_version
      f_mode_dec_w_f_sv →
    forall (css : Stream block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      bisimilar_Stream_block_n
        (BENC_dec_sv css ks IV block_enc)
        (f_mode_dec_w_f_sv
           (make_block_enc_unary block_enc ks) css ks IV block_enc).
```

The output of the BENC encryption respectively decryption function using streams is specified to be bisimilar to the output of the f mode encryption respectively decryption function (using streams) applied to the f function for the BENC mode.

Likewise, we consider the BENC functions using lists.

```
Definition specification_of_BENC_encryption_function_list_version
           (BENC_enc_lv : MoO_enc_dec_Type_list_version) :=
  forall (f_mode_enc_w_f_lv : MoO_enc_dec_w_f_Type_list_version),
    specification_of_f_mode_encryption_function_w_f_list_version
      f_mode_enc_w_f_lv →
    forall (xss : list block_n) (ks : block_k) (IV : block_n)
           s(block_enc : block_enc_dec_Type),
      BENC_enc_lv xss ks IV block_enc =
      f_mode_enc_w_f_lv
        (make_block_enc_unary block_enc ks) xss ks IV block_enc.

Definition specification_of_BENC_decryption_function_list_version
           (BENC_dec_lv : MoO_enc_dec_Type_list_version) :=
  forall (f_mode_dec_w_f_lv : MoO_enc_dec_w_f_Type_list_version),
    specification_of_f_mode_decryption_function_w_f_list_version
      f_mode_dec_w_f_lv →
    forall (css : list block_n) (ks : block_k) (IV : block_n)
           (block_enc : block_enc_dec_Type),
      BENC_dec_lv css ks IV block_enc =
      f_mode_dec_w_f_lv
        (make_block_enc_unary block_enc ks) css ks IV block_enc.
```

The outputs of the BENC functions using lists are specified to be equal to the outputs of the f mode functions using lists when applied to the f function for the BENC mode.

6.10.4.3 Properties about the Block and Feedback Functions

The next step of the procedure in Section 6.9.1 consists of proving some properties about the block and feedback functions and their implementations.

The encryption function of the f mode does not propagate plaintext changes. Hence, for the f mode, we prove the properties assumed about the block and feedback functions by the general validity, plaintext change non-propagation, and IV change propagation theorems in Section 6.8.3.

We do not specify the block and feedback functions for the CTR and BENC modes so, naturally, we do not consider these functions in this section.

First, we prove the properties assumed by the general validity theorems in Section 6.8.3.1: we prove that the block function of the f mode is an involution, i.e., the inverse of itself, and that the feedback function of the f mode outputs equal values whether it is used for encryption or decryption. Recall, we use the same block function and the same feedback function for both f mode encryption and f mode decryption.

The block function of the f mode is an involution for any underlying block encryption function.

```
Lemma f_mode_block_function_own_inverse :
  forall block_enc : block_enc_dec_Type,
  forall block_fun : block_fun_Type,
    specification_of_f_mode_block_function block_fun →
    block_functions_inverse
      block_fun block_fun
      block_enc block_enc.
```

As specified in the section above, the output of the block function is the bitwise XOR of the plaintext or ciphertext block and the block encryption of the IV or feedback block under the key. Hence, showing that the block function is an involution amounts to showing, according to the definition of the `block_functions_inverse` predicate in Section 6.7.3, that bitwise XOR'ing a block twice with the same block yields the first block. Here, we use the `XOR_blocks_n_fun_applied_twice_to_same_second_input_`

`yields_first_input` lemma in Section 6.6.1: the XOR-blocks-n function, with which we compute the bitwise XORs, has the property we want to show.

The feedback function of the f mode outputs equal values whether it is used for encryption or decryption—the f mode uses the same feedback function for both encryption and decryption—regardlessly of the choices of the f function and the underlying block encryption function.

```
Lemma f_mode_feedback_function_w_f_output_equal_values :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall (feedback_fun_w_f : feedback_fun_w_f_Type)
         (block_fun : block_fun_Type),
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    specification_of_f_mode_block_function block_fun →
    feedback_functions_output_equal_values
      (feedback_fun_w_f f) (feedback_fun_w_f f)
      block_fun
      block_enc block_enc.
```

The proof of the lemma above is straightforward: according to the `feedback_functions_output_equal_values` predicate defined in Section 6.7.3, we have to show that the feedback function outputs equal values when applied to two potentially different sets of inputs. However, the same IV or feedback block and the same f function are input to the function for both applications. The output of the feedback function is specified in the section above to be equal to the output of the input f function when applied to the input IV or feedback block. Hence, the output of the feedback function only depends on the f function and IV or feedback block input to it. Since these inputs are the same for the two applications of the feedback function, the corresponding outputs are equal.

We prove that the f mode (and, in extension, the CTR and BENC modes) does not propagate plaintext changes when encrypting: in Section 6.10.4.5, we prove the f mode plaintext change non-propagation theorems by applying the corresponding general theorems in Section 6.8.3.3. These general theorems assume that the block function for encryption is invertible with regards to the plaintext block (even if the mode is non-propagating when encrypting, the ciphertext block corresponding to the changed plaintext block is altered) and that the feedback function for encryption ignores the plaintext block.

We prove first that the f mode block function, which is used for encryption, is invertible with regards to the plaintext block for any underlying block encryption function.

```
Lemma f_mode_block_fun_invertible_wrt_plaintext :
  forall block_enc : block_enc_dec_Type,
  forall block_fun : block_fun_Type,
    specification_of_f_mode_block_function block_fun →
    block_fun_invertible_wrt_plaintext block_fun block_enc.
```

As specified in the section above, the output of the feedback function is specified to be equal to the bitwise XOR of the plaintext block and the block encryption of the IV or feedback block under the key. Hence, we prove the lemma above by showing that the XOR-blocks-n function, which is used to compute the bitwise XORs, is invertible with regards to the plaintext block which is the first input to it. In this regard, we use the `XOR_blocks_n_fun_invertible_wrt_first_input` lemma proven in Section 6.6.1. This lemma says exactly that the XOR-blocks-n function is invertible with regards to the first input.

Secondly, we prove that the f mode feedback function, which is used for encryption, ignores the plaintext block regardlessly of the f function and the underlying block encryption function.

```
Lemma f_mode_feedback_fun_w_f_ignores_plaintext :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall feedback_fun_w_f : feedback_fun_w_f_Type,
    specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
    feedback_fun_ignores_plaintext (feedback_fun_w_f f) block_enc.
```

The goal of the lemma above is to show that the output of the feedback function is independent of the input plaintext block, i.e., changing the input plaintext block—and keeping the other inputs unchanged—does not result in the output changing also. This property holds for the f mode feedback function: as specified in the section above, the output of the feedback function is equal to the output of the input f function applied to the input IV or feedback block.

In Section 6.10.4.6, we prove that the f mode propagates IV changes when encrypting. When proving the theorems stating this property, we apply the general IV change propagation theorems in Section 6.8.3.4: these general theorems assume that the block and feedback functions for encryption are invertible with regards to the IVs or feedback blocks.

We prove that the f mode block function, which, again, is used for encryption, is invertible with regards to the IV or feedback block if the underlying block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block.

```
Lemma f_mode_block_fun_invertible_wrt_IV_feedback :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall block_fun : block_fun_Type,
      specification_of_f_mode_block_function block_fun →
      block_fun_invertible_wrt_IV_feedback block_fun block_enc.
```

As specified in the section above, when encrypting, the output of the block function is equal to the bitwise XOR of the input plaintext block and the block encryption of the input IV or feedback block under the key—the IV or feedback block is given as the second input to the XOR-blocks-n function which we use to compute the bitwise XORs. Hence, the block function is invertible with regards to the IV or feedback block if the XOR-blocks-n function is invertible with regards to the second input and the block encryption function is invertible. The first property follows from the `XOR_blocks_n_fun_invertible_wrt_second_input` lemma in Section 6.6.1: the XOR-blocks-n *is* invertible with regards to the second input. The last property about the block encryption function is a premise of the lemma we are proving.

Likewise, we prove that the f mode feedback function, used now for encryption, is invertible with regards to the IV or feedback block. Now, however, any underlying block encryption function suffices, but the f function is assumed to be invertible.

```
Lemma f_mode_feedback_fun_w_f_invertible_wrt_IV_feedback :
  forall f : block_n → block_n,
    f_invertible f →
    forall block_enc : block_enc_dec_Type,
    forall feedback_fun_w_f : feedback_fun_w_f_Type,
      specification_of_f_mode_feedback_function_w_f feedback_fun_w_f →
      feedback_fun_invertible_wrt_IV_feedback
        (feedback_fun_w_f f) block_enc.
```

As specified in the section above, the output of the f mode feedback function is equal to the output of the input f function when the f function is applied to the IV or feedback block input to the feedback function. Hence, the feedback function being invertible with regards to the IV or feedback block is equivalent to the f function being invertible: this property of the f function is a premise of the lemma.

We prove the properties in this section for the implementations mentioned in the section above of the f mode block and feedback functions. Doing so enables us to apply the general theorems in the proofs of the f mode theorems in the following sections with these implementations.

6.10.4.4 The f Mode, CTR, and BENC Validity Theorems

The f, CTR, and BENC modes are valid—both the stream and list versions.

When proving that the f mode is valid, we follow the procedure in Section 6.9.2. We deviate to a lesser degree from the procedure since the encryption, decryption, and feedback functions of the f mode

take an extra input, the f function. Thus, when unfolding the encryption and decryption functions of the f mode, inputting the implementation of the feedback function to the general validity theorems, and using the lemma stating that the feedback function outputs equal values, we input the f function introduced by the particular theorems we prove.

The procedure uses implementations of the block and feedback functions for encryption respectively decryption. Here, we deviate from the procedure since the f mode only has one block function and one feedback function for both encryption and decryption. We use the implementations mentioned in Section 6.10.4.2 of these functions: as mentioned in the section above, we have proven that the implementation of the f mode block function is an involution, i.e., its own inverse, and that the implementation of the f mode feedback function outputs equal values. Hence, the procedure works.

The properties of the implementations of the f mode block and feedback functions are proven without any assumptions about the underlying block encryption function and the f function. Thus, the f mode validity theorems do not assume anything about these functions.

We state first the f mode validity theorem using streams.

```
Corollary f_mode_validity_theorem_stream_version :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall f_mode_enc_w_f_sv f_mode_dec_w_f_sv :
         MoO_enc_dec_w_f_Type_stream_version,
    specification_of_f_mode_encryption_function_w_f_stream_version
      f_mode_enc_w_f_sv →
    specification_of_f_mode_decryption_function_w_f_stream_version
      f_mode_dec_w_f_sv →
    MoO_valid_stream_version
      (f_mode_enc_w_f_sv f) (f_mode_dec_w_f_sv f)
      block_enc block_enc.
```

The stream version of the f mode of operation is valid for any f function and any underlying block encryption function. The theorem is a corollary of the general validity theorem using streams in Section 6.8.3.1.

Similarly, we state the f mode validity theorem using lists.

```
Corollary f_mode_validity_theorem_list_version :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall f_mode_enc_w_f_lv f_mode_dec_w_f_lv :
          MoO_enc_dec_w_f_Type_list_version,
    specification_of_f_mode_encryption_function_w_f_list_version
      f_mode_enc_w_f_lv →
    specification_of_f_mode_decryption_function_w_f_list_version
      f_mode_dec_w_f_lv →
    MoO_valid_list_version
      (f_mode_enc_w_f_lv f) (f_mode_dec_w_f_lv f)
      block_enc block_enc.
```

The list version of the f mode is valid for any f function and any underlying block encryption function, too. The theorem is a corollary of the general validity theorem using lists in Section 6.8.3.1.

We prove that the CTR mode using streams respectively lists is valid. We state the CTR validity theorem using streams.

```
Corollary CTR_validity_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
  forall CTR_enc_sv CTR_dec_sv : MoO_enc_dec_Type_stream_version,
    specification_of_CTR_encryption_function_stream_version
      CTR_enc_sv →
```

```
    specification_of_CTR_decryption_function_stream_version
      CTR_dec_sv →
    MoO_valid_stream_version
      CTR_enc_sv CTR_dec_sv
      block_enc block_enc.
```

The CTR mode using streams is valid for any underlying block encryption function. The theorem is proven as a corollary of the above f mode validity theorem using streams.

Resemblingly, we state the CTR validity theorem using lists.

```
Corollary CTR_validity_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
  forall CTR_enc_lv CTR_dec_lv : MoO_enc_dec_Type_list_version,
    specification_of_CTR_encryption_function_list_version CTR_enc_lv →
    specification_of_CTR_decryption_function_list_version CTR_dec_lv →
    MoO_valid_list_version
      CTR_enc_lv CTR_dec_lv
      block_enc block_enc.
```

The CTR mode using lists is valid for any underlying block encryption function, too. The theorem is a corollary of the list version of the f mode validity theorem above.

We prove the CTR validity theorems above by applying the corresponding f mode theorems. In the proofs, we unfold the CTR encryption and decryption functions via the specifications of them in Section 6.10.4.2. In this way, we face problems involving the f mode encryption and decryption functions applied to the counter-n function (the f function of the CTR mode): we can apply the f mode theorems.

When applying the f mode theorems, we use implementations of the f mode encryption and decryption functions and the counter-n function. Here, we use the implementations mentioned in Section 6.10.4.2 of the f mode functions and the `counter_n_fun_v0` implementation (defined in Section 6.6.2) of the counter-n function.

The applied f theorems do not have any special assumptions about the f function and the underlying block encryption function. Hence, the f function being the counter-n function poses no problems, and we can prove that the CTR mode is valid for any underlying block encryption function.

Similarly, we prove that the stream and list versions of the BENC mode of operation is valid. First, we consider the BENC validity theorem using streams.

```
Corollary BENC_validity_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
  forall BENC_enc_sv BENC_dec_sv : MoO_enc_dec_Type_stream_version,
    specification_of_BENC_encryption_function_stream_version
      BENC_enc_sv →
    specification_of_BENC_decryption_function_stream_version
      BENC_dec_sv →
    MoO_valid_stream_version
      BENC_enc_sv BENC_dec_sv
      block_enc block_enc.
```

The BENC mode using streams is valid for any underlying block encryption function. The theorem is a corollary of the stream version of the f mode validity theorem above.

Likewise, we state the BENC validity theorem using lists.

```
Corollary BENC_validity_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
  forall BENC_enc_lv BENC_dec_lv : MoO_enc_dec_Type_list_version,
    specification_of_BENC_encryption_function_list_version
      BENC_enc_lv →
    specification_of_BENC_decryption_function_list_version
      BENC_dec_lv →
```

```
MoO_valid_list_version
  BENC_enc_lv BENC_dec_lv
  block_enc block_enc.
```

Also the BENC mode using lists is valid for any underlying block encryption function. The theorem is a corollary of the above f mode validity theorem using lists.

In Section 6.10.4.2, we specify that the BENC encryption respectively decryption functions are extensionally bisimilar or equal to the f mode encryption respectively decryption functions applied to the f function of the BENC mode: the underlying block encryption function made unary via the make-block_enc-unary function specified in Section 6.6.3 and the key. Hence, we prove the BENC validity theorems by applying the f mode validity theorems above with the input f function being `make_block_enc_unary block_enc ks`—the make-block_enc-unary function applied to the underlying block encryption function and the key introduced by the `MoO_valid_stream_version` or `MoO_valid_list_version` predicate.

When unfolding the BENC encryption and decryption functions in the proofs of the theorems above, we use the implementations of the f encryption and decryption functions mentioned in Section 6.10.4.2.

Again, the applied f mode theorems do not assume any special properties about the f and underlying block encryption functions. Hence, we can prove that the BENC mode is valid for any underlying block encryption function.

6.10.4.5 The f Mode, CTR, and BENC Plaintext Change Non-Propagation Theorems

Both the stream and list versions of the encryption functions of the f, CTR, and BENC modes do not propagate plaintext changes.

We prove that the f mode encryption function using streams respectively lists does not propagate plaintext changes by following the procedure in Section 6.9.4. As in the section above, we deviate from the procedure to some degree: the f mode encryption and feedback functions take the f function as an extra input compared to regular encryption and feedback functions. Thus, when we unfold the encryption function, input the feedback function to the general plaintext change non-propagation theorems in Section 6.8.3.3, and use the lemma stating that the feedback function ignores the plaintext block, we input the f function introduced in the proofs.

We use the implementations of the f mode block and feedback functions mentioned in Section 6.10.4.2: the implementation of the block function has been proven to be invertible with regards to the plaintext block, and the implementation of the feedback function has been proven to ignore the plaintext block. Hence, we can apply the general plaintext change non-propagation theorems with these implementations, and the procedure works.

The attributes of the implementations of the block and feedback functions of the f mode are proven to hold for any underlying block encryption function and any f function. Therefore, the f mode plaintext change non-propagation theorems do not assume any attributes of these functions. Likewise, the corresponding BENC theorems do not assume any attributes of the underlying block encryption function.

First, we state the f mode plaintext change non-propagation theorem using streams.

```
Corollary
  f_mode_plaintext_change_non_propagation_theorem_stream_version :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall f_mode_enc_w_f_sv : MoO_enc_dec_w_f_Type_stream_version,
    specification_of_f_mode_encryption_function_w_f_stream_version
      f_mode_enc_w_f_sv →
    plaintext_change_non_propagation_stream_version
      (f_mode_enc_w_f_sv f) block_enc.
```

The f mode encryption function using streams does not propagate plaintext changes for any f function or any underlying block encryption function. The theorem is a corollary of the stream version of the

general plaintext change non-propagation theorem in Section 6.8.3.3.

Next, we state the f mode plaintext change non-propagation theorem using lists.

```
Corollary f_mode_plaintext_change_non_propagation_theorem_list_version :
  forall f : block_n → block_n,
  forall block_enc : block_enc_dec_Type,
  forall f_mode_enc_w_f_lv : MoO_enc_dec_w_f_Type_list_version,
    specification_of_f_mode_encryption_function_w_f_list_version
      f_mode_enc_w_f_lv →
    plaintext_change_non_propagation_list_version
      (f_mode_enc_w_f_lv f) block_enc.
```

The f mode encryption function using lists does not propagate plaintext changes—again, regardlessly of the choices of the f function and underlying block encryption function. The theorem is a corollary of the general plaintext change non-propagation theorem using lists in Section 6.8.3.3.

We show that the CTR encryption functions do not propagate plaintext changes for any underlying block encryption function: we state the CTR plaintext change non-propagation theorem using streams respectively lists.

```
Corollary CTR_plaintext_change_non_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
  forall CTR_enc_sv : MoO_enc_dec_Type_stream_version,
    specification_of_CTR_encryption_function_stream_version
      CTR_enc_sv →
    plaintext_change_non_propagation_stream_version CTR_enc_sv block_enc.


Corollary CTR_plaintext_change_non_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
  forall CTR_enc_lv : MoO_enc_dec_Type_list_version,
    specification_of_CTR_encryption_function_list_version CTR_enc_lv →
    plaintext_change_non_propagation_list_version
      CTR_enc_lv block_enc.
```

These theorems are proven as corollaries of the f mode plaintext change non-propagation theorems above.

We prove the theorems above by applying the corresponding f mode theorems: in the proofs, we unfold the CTR encryption functions via their specifications in Section 6.10.4.2. After the unfolding, the f mode theorems are directly applicable. Here, the f function input to the f mode theorems is the counter-n function which is the f function of the CTR mode.

When applying the f mode theorems, we use the `f_mode_enc_w_f_sv_v0` and `f_mode_enc_w_f_lv_v0` implementations mentioned in Section 6.10.4.2 of the stream and list versions of the f mode encryption function and the `counter_n_fun_v0` implementation defined in Section 6.6.2 of the counter-n function.

The applied f mode theorems are applicable to all f functions and underlying block encryption functions: the theorems can be applied to the f function of the CTR mode, and we do not have to assume any special properties about the underlying block encryption function in the CTR theorems we are proving.

Likewise, we prove that also the BENC encryption functions do not propagate plaintext changes for any underlying block encryption function: we state the stream and list versions of the BENC plaintext change non-propagation theorem.

```
Corollary BENC_plaintext_change_non_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
  forall BENC_enc_sv : MoO_enc_dec_Type_stream_version,
    specification_of_BENC_encryption_function_stream_version
      BENC_enc_sv →
    plaintext_change_non_propagation_stream_version
      BENC_enc_sv block_enc.
```

```
Corollary BENC_plaintext_change_non_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
  forall BENC_enc_lv : MoO_enc_dec_Type_list_version,
    specification_of_BENC_encryption_function_list_version
      BENC_enc_lv →
    plaintext_change_non_propagation_list_version
      BENC_enc_lv block_enc.
```

These theorems, too, are corollaries of the corresponding f mode theorems above.

In Section 6.10.4.2, the outputs of the BENC encryption functions are specified to be bisimilar or equal to the outputs of the f mode encryption functions applied to the f function of the BENC mode. Therefore, we prove the BENC theorems above by applying the corresponding f mode theorems. Thereby, we input the f function of the BENC mode to the f mode theorems. When applying the BENC mode theorems, we use the implementations mentioned in Section 6.10.4.2 of the f mode encryption functions.

Again, the f mode plaintext change non-propagation theorems do not assume any special properties about the f and block encryption functions: they can be applied to the f function of the BENC mode, and we can prove that the BENC encryption function does not propagate plaintext changes for any underlying block encryption function.

6.10.4.6 The f Mode, CTR, and BENC IV Change Propagation Theorems

The stream and list versions of the f mode encryption function propagates IV changes if the input f function is invertible and the underlying block encryption is invertible with regards to the in-the-context-of-that-function plaintext block. In extension, the CTR and BENC encryption functions propagate IV changes if the underlying block encryption function is invertible—recall that the f function of the CTR mode is the invertible counter-n function and that the f function of the BENC mode is the block encryption function made unary by fixing the key: this operation preserves invertibility.

We show that both the stream and list versions of the encryption function of the f mode propagate IV changes by following the procedure in Section 6.9.5. As in the sections above, we deviate from the procedure: the f mode encryption and feedback functions take an extra input, the f function. Therefore, we input the f function introduced by the theorems when unfolding the f mode encryption functions, when applying the general IV change propagation theorems in Section 6.8.3.4, and when using the lemma stating that the feedback function is invertible with regards to the IV or feedback block.

When following the procedure, we use the implementations in Section 6.10.4.2 of the f mode block and feedback functions: these implementations have been proven to be invertible with regards to the IV or feedback block. However, when proving that the block function is invertible, we assume that the underlying block encryption function is invertible, and when proving that the feedback function is invertible, we assume that the f function is invertible. Hence, the f mode IV change propagation theorems below assume that the block encryption function and the f function are invertible, and the corresponding CTR and BENC theorems below assume that the block encryption function is invertible.

We state the f mode IV change propagation theorem using streams.

```
Corollary f_mode_IV_change_propagation_theorem_stream_version :
  forall f : block_n → block_n,
    f_invertible f →
    forall block_enc : block_enc_dec_Type,
      block_enc_invertible_wrt_plaintext_in_context block_enc →
      forall f_mode_enc_w_f_sv : MoO_enc_dec_w_f_Type_stream_version,
        specification_of_f_mode_encryption_function_w_f_stream_version
          f_mode_enc_w_f_sv →
        IV_change_propagation_stream_version
          (f_mode_enc_w_f_sv f) block_enc.
```

If the f function and the underlying block encryption function are invertible, the f mode encryption function using streams propagates IV changes. The theorem is a corollary of the stream version of the general IV change propagation theorem in Section 6.8.3.4.

Likewise, we state the f mode IV change propagation theorem using lists.

```
Corollary f_mode_IV_change_propagation_theorem_list_version :
  forall f : block_n → block_n,
    f_invertible f →
    forall block_enc : block_enc_dec_Type,
      block_enc_invertible_wrt_plaintext_in_context block_enc →
      forall f_mode_enc_w_f_lv : MoO_enc_dec_w_f_Type_list_version,
        specification_of_f_mode_encryption_function_w_f_list_version
          f_mode_enc_w_f_lv →
        IV_change_propagation_list_version
          (f_mode_enc_w_f_lv f) block_enc.
```

Also the f mode encryption function using lists propagates IV changes if the f function and the underlying block encryption function are invertible. The theorem is a corollary of the list version of the general IV change propagation theorem in Section 6.8.3.4.

We prove that the CTR encryption functions propagate IV changes if the underlying block encryption functions are invertible. This property follows from the theorem above and the fact that the counter-n function is invertible as proven in Section 6.6.2. We state the CTR IV change propagation theorems.

```
Corollary CTR_IV_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CTR_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_CTR_encryption_function_stream_version
        CTR_enc_sv →
      IV_change_propagation_stream_version CTR_enc_sv block_enc.

Corollary CTR_IV_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall CTR_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_CTR_encryption_function_list_version
        CTR_enc_lv →
      IV_change_propagation_list_version CTR_enc_lv block_enc.
```

These theorems are corollaries of the corresponding f mode theorems above.

We prove the theorems above by unfolding the CTR encryption functions and applying the f mode theorems. We unfold the functions via their specifications in Section 6.10.4.2: the functions are extensionally equal to the f mode encryption functions applied to the f function of the CTR mode of operation being the counter-n function. Hence, we can apply the f mode theorems directly.

We use the `f_mode_enc_w_f_sv_v0` and `f_mode_enc_w_f_lv_v0` implementations mentioned in Section 6.10.4.2 of the stream and list versions of the f mode encryption function and the `counter_n_fun_v0` implementation defined in Section 6.6.2 of the counter-n function.

The applied f theorems assume that the input f function is invertible and that the input block encryption function is invertible with regards to the in-the-context-of-that-function plaintext block. The first assumption is satisfied for the CTR mode: in Section 6.6.2, we prove that the counter-n function invertible. In extension, the `counter_n_fun_v0` implementation is invertible, too. The second assumption is satisfied since it is a premise of the CTR theorems we are proving.

Finally, we state the BENC IV change propagation theorems.

```
Corollary BENC_IV_change_propagation_theorem_stream_version :
  forall block_enc : block_enc_dec_Type,
```

```
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall BENC_enc_sv : MoO_enc_dec_Type_stream_version,
      specification_of_BENC_encryption_function_stream_version
        BENC_enc_sv →
      IV_change_propagation_stream_version BENC_enc_sv block_enc.

Corollary BENC_IV_change_propagation_theorem_list_version :
  forall block_enc : block_enc_dec_Type,
    block_enc_invertible_wrt_plaintext_in_context block_enc →
    forall BENC_enc_lv : MoO_enc_dec_Type_list_version,
      specification_of_BENC_encryption_function_list_version
        BENC_enc_lv →
      IV_change_propagation_list_version BENC_enc_lv block_enc.
```

The BENC encryption function—both the stream and list versions—propagates IV changes if the underlying block encryption function is invertible.

As specified in Section 6.10.4.2, the outputs of the BENC encryption functions are bisimilar or equal to the outputs of the f mode encryption functions applied to the f function of the BENC mode. Thus, we prove the BENC IV change propagation theorems above by applying the corresponding f mode theorems with the input f function being the f function of the BENC mode. We input the implementations mentioned in Section 6.10.4.2 of the f mode encryption functions to the f mode theorems when applying them.

We use the `f_mode_enc_w_f_sv_v0` and `f_mode_enc_w_f_lv_v0` implementations mentioned in Section 6.10.4.2 of the f mode encryption functions using streams respectively lists.

As above, the applied f mode IV change propagation theorems assume that the input block encryption and f functions are invertible. We input the block encryption function introduced by the BENC theorems we are proving: this function is assumed to be invertible. The input f function we input is the f function of the BENC mode. This function is the underlying block encryption function made unary by fixing the key. Here, we exploit that, according to the `make_block_enc_unary_preserves_invertibility` lemma in Section 6.6.3, the make-`block_enc`-unary function, that we use to make the block encryption function unary, preserves invertibility, i.e., if the block encryption function input is invertible with regards to the in-the-context-of-that-function plaintext block, then the output function is invertible.

6.11 Summary and Conclusion

The goal of this chapter has been to first model modes of operation and MAC schemes built on top of them and then proving theorems about them. Additionally, we have set out to do so in an easily extendable way that facilitate modelling and proving theorems about other and, potentially, new modes of operation and MAC schemes.

Using the specifications of the general encryption, decryption, and MAC functions and the general theorems involving them as the foundation, we have been able to quickly prove several properties about the modes of operation we consider—both when used to encrypt or decrypt finite and infinite numbers of blocks—and the corresponding MAC schemes: first, the block and feedback functions of the (potentially new) mode have been specified after which the encryption and decryption functions (and the MAC function if relevant) of the mode have been specified in terms of the corresponding general functions applied to functions fitting the specifications of these block and feedback functions.

With this foundation or framework, we have straightforwardly specified the CBC, CFB, and CTR modes and the CBC- and CFB-MAC functions as well as the newly defined IFB, f, and BENC modes and IFB-MAC function after which we have proven properties about them.

We have proven that all the considered modes are valid and that the corresponding encryption functions propagate IV changes. In addition, we have proven that the encryption functions of the CBC, CFB,

and IFB modes propagate plaintext changes while the encryption functions of the `f`, CTR, and BENC modes do not. Lastly, we have proven that the CBC-, CFB-, and IFB-MAC functions propagate changes of the input messages. In conclusion, we have succeeded in showing these properties of the considered modes.

The foundation consisting of the specifications of the general functions and the general theorems can be used if we want to prove the above properties for additional, newly defined modes of operation (or already defined ones for that matter): we have succeeded in making a framework that facilitates proving theorems about such new modes of operation.

Performing the work in this chapter has been an enlightening experience. Not only have we succeeded in proving properties about a few already known modes of operation and MAC schemes—which was the initial goal—but we have also generalised the results, making it simple to extend our findings to other (maybe new) modes and MAC schemes.

In Chapter 5 about the one-time pad cryptosystem, we learned that using vectors instead of lists would have been beneficial. This lesson has proved to be very valuable in our work in this chapter: using the `block_n` and `block_k` types—built on top of the `t` type representing vectors—for the blocks, we have not had to consider other block lengths. In this way, we have been able to avoid thinking about cases that does not happen in real execution of the protocols without inserting lots of assumptions about lengths of input lists, e.g., we have not had to consider the case where stream or list of blocks of length `0` are encrypted: the result of that could conceivably be another stream or list of blocks of length `0`, but what about the IV change propagation property then? In short, using vectors instead of lists has spared us from considering such cases without having to include a high amount of length premises, thereby giving us more time to focus on the interesting aspects: the cryptographic properties.

Chapter 7

The RSA Encryption and Signature Schemes

In this chapter, we introduce, define, and prove theorems about the RSA encryption and signatures schemes.

In Section 7.1, we introduce the RSA encryption and signature schemes[1] and outline the goals of this chapter. The RSA schemes use a key pair consisting of a public and private key: we describe this key pair and prove lemmas about it in Section 7.3. Having done so, we are able to define the RSA cryptosystem in Section 7.4. Here, we also state and prove several theorems about it. In the same way, we define and prove theorems about the RSA signature scheme in Section 7.5. Finally, in Section 7.6, we summarise and conclude on the contributions of this chapter.

7.1 Introduction

The RSA[2] cryptosystem is a public-key system which means that, unlike for the symmetric-key systems we consider—the one-time pad in Chapter 5 and the modes of operation in Chapter 6—different keys are used for encryption and decryption. Conceptually, a public key known to everyone is used to encrypt messages that only one person can decrypt via his or her private key. In the corresponding signature scheme, the private key is used for signing messages while the public key is used to verify signatures. We refer to the public and private keys collectively as the RSA key pair.

The RSA cryptosystem and the corresponding signature scheme assume that it is hard to compute primes p and q when given the modulus $n = pq$ which is part of the RSA key pair, i.e., it is assumed that it is hard to find the prime factorisation of n.

The goal of this chapter is to model and prove several theorems about the RSA cryptosystem and signature scheme. To be able to do so, we first define and specify valid RSA key pairs in Section 7.3.

Having handled the RSA key pair, our goal in Section 7.4 is to define, specify, and prove theorems about the RSA cryptosystem. Here, we define and specify both the RSA encryption function and decryption function which facilitate proving several properties about these functions:

- We prove that the ciphertext resulting from encrypting any plaintext is a member of the designated ciphertext space. This space is defined in Section 7.4.1 as $\mathcal{C} = \mathbb{Z}_n$.
- We prove that the RSA encryption scheme is valid, i.e., for any proper plaintext and any valid key pair, first encrypting the plaintext under the public key and then decrypting it under the private

[1] A signature scheme consists of two functions: a signing function and a verification function. The signing function is used to sign messages: it outputs a signature on the input message. The verification function is applied to a message and a signature (and a key) and accepts them (in this case, outputs `true`) if and only if the signature is a signature for the message.

[2] The letters ("R", "S", and "A") are the initial letters of the last names of the inventors of the cryptosystem: Ron Rivest, Adi Shamir, and Leonard Adleman.

key yields the plaintext. This property is highly desirable since we want to be able to reveal the original plaintext by decrypting the ciphertext.

- Lastly, we show that the RSA encryption function is invertible regarding the plaintext, i.e., if the ciphertexts resulting from encrypting two proper plaintexts under the same valid public key are equal, then so are the plaintexts. This latter property follows from the scheme being valid: if encrypting two different plaintexts could result in the same ciphertext, then the decryption function would have to guess which plaintext was actually encrypted when given the ciphertext—it would not be able to do so with certainty, and, hence, the scheme could not possibly be valid for all plaintexts.

Likewise, our goal in Section 7.5 is to define, specify, and prove theorems about the RSA signature scheme. We define and specify the signing and verification functions after which we prove properties about them:

- We prove that any signature produced by the signing function is in the designated signature space which we define in Section 7.5.1 as $\mathcal{A} = \mathbb{Z}_n$.
- We show that the RSA signature scheme is valid, i.e., for any proper message and signature, the verification functions accepts (or outputs `true`) if and only if the signature is the signature of the message.

7.2 Technical Details

The code in this chapter is contained in the `RSA.v` file in the `Ole-Dalgaard-Lauridsen_code/RSA` directory in the archive file at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_code.tar.gz`.

7.3 The RSA Key Pair

In this section, we describe the RSA key pair and prove lemmas about it. The key pair is used for both the RSA cryptosystem in Section 7.4 and the RSA signature scheme in Section 7.5.

In Section 7.3.1, we define and describe all parts of the key pair. After that, in Section 7.3.2, we specify the key pair which enables us to finally state and prove properties about the key pair in Section 7.3.3. In that section, we additionally prove several properties of the key pair. These properties are exploited when we prove properties of the RSA encryption and signature schemes.

7.3.1 Definitions

The RSA cryptosystem and signature scheme use a natural number $n = pq$ called the modulus where p and q are distinct primes. Furthermore, two natural numbers e and d are defined such that $ed \equiv_{\phi(n)} 1$ where ϕ is Euler's totient function. The key pair consists of a public key and a private key. The public key consists of n and e while the private key consists of p, q, and d.

For the RSA cryptosystem, this division of keys enables any person to encrypt messages under the public key such that only people knowing the private key can decrypt it. Likewise for the RSA signature scheme, a person knowing the private key can sign a message. This signature can then be checked against the message by anyone knowing the public key.

7.3.2 Specifications

We start by specifying what it means for an RSA key pair to be valid. As mentioned above, the key pair includes a modulus n which is a product of two distinct prime p and q. This requirement is captured by

the valid_modulus_primes proposition using the prime predicate from the SSReflect library (see Section 4.2.3) to restrict p and q to be primes.

```
Definition valid_modulus_primes (n p q : nat) :=
  n = p * q
  /\
  prime p
  /\
  prime q
  /\
  p <> q.
```

However, we do not need to know the explicit prime factorisation of the modulus when proving properties about the RSA cryptosystem and signature scheme. Therefore, we define another proposition for the modulus only.

```
Definition valid_modulus (n : nat) :=
  exists p q : nat,
    valid_modulus_primes n p q.
```

Now, we just require that some primes exist for which (in addition to the modulus) the first proposition is true.

The key includes two more natural numbers: the public exponent e for encryption and verification and the private exponent d for decryption and signing. As mentioned above, the product of these exponents must be congruent to 1 modulo $\phi(n)$.

```
Definition valid_exponents (d e n : nat) :=
  e * d = 1 %[mod (totient n)].
```

Here, m = n %[mod d] is shorthand notation for m %% d = n %% d defined in the SSReflect library. m %% d in turn is infix notation for modn m d which denotes the remainder of the natural number m modulo the natural number d. See Section 4.2.2.

Having specified a valid modulus and valid exponents, we are ready to specify validity of the entire key (only implicitly the primes).

```
Definition valid_key_pair (d e n : nat) :=
  valid_exponents d e n
  /\
  valid_modulus n.
```

7.3.3 Properties and Proofs

We prove several properties of valid RSA key pairs. These properties are used in proofs for the RSA cryptosystem and the RSA signature scheme in sections below.

We state and prove properties of the modulus in Section 7.3.3.1 and the exponents in Section 7.3.3.2.

7.3.3.1 The Modulus

We prove several properties of the modulus n and $\phi(n)$.

We first prove that any modulus is positive.

```
Lemma zero_less_than_modulus :
  forall n : nat,
    valid_modulus n →
    0 < n.
```

This follows directly from the fact that any modulus is a product of primes. Since all primes are positive (see lemma `prime_gt0` in Section 4.2.3), then so is the modulus. This lemma is used in the proofs of the theorems stating that encrypted messages and signatures are in the designated ciphertext and signature spaces.

The RSA cryptosystem and signature scheme use properties of the Euler ϕ function. We prove that $\phi(p) = p - 1$ for any prime p.

```
Lemma totient_prime_minus_1 :
  forall p : nat,
    prime p →
    totient p = p − 1.
```

The SSReflect library lemma `totient_pfactor` is used. While this library lemma could have been used instead of `totient_prime_minus_1` in later proofs, the lemma above encapsulates exactly what we need in those situations—we avoid having to rewrite `p` to `p ^ 1` each time. It is used in the lemma below about $\phi(n)$ and in the validity proof for the RSA encryption scheme.

```
Lemma totient_modulus_n :
  forall n p q : nat,
    valid_modulus_primes n p q →
    totient n = (p − 1) ∗ (q − 1).
```

Here, we use the SSReflect lemma `totient_coprime` (see Section 4.2.3) that says that $\phi(pq) = \phi(p)\phi(q)$ when p and q are coprime—which they are in our case since where p and q are distinct primes. The above lemma is used in the validity theorem of the RSA encryption scheme and also in the following lemma.

Also, we show that $\phi(n)$ is always greater than 1.

```
Lemma one_less_than_totient_n :
  forall n : nat,
    valid_modulus n →
    1 < totient n.
```

We need this property in later proofs because we need to use the `discriminate` tactic on a hypothesis stating that `0 = 1 %[mod totient n]`: if `totient n` is 1, then the hypothesis is true. We could also have settled for showing that $\phi(n)$ is different from 1 since SSReflect defines $n \bmod 0 := n$ for any natural number n. However, we want to avoid this situation since in reality computing the remainder modulo 0 is an undefined operation.

The proof of the lemma is done by case analysis of the prime factors of `n` and by using the above lemma about the value of `totient n`. In most cases, the considered factors are too small to be primes so we use the `discriminate` tactic. In the case where both primes equal 2, we use the fact that they are distinct to prove the subgoal by contradiction via the `False_ind` standard lemma that states that the `False` proposition implies any proposition.

The proven lemma is used when showing that the `d` exponent is a positive natural number.

7.3.3.2 The Exponents

First, we prove several properties of the `e` and `d` exponents.

We show that the `d` exponent is a positive natural number.

```
Lemma zero_less_than_d :
  forall d e n : nat,
    valid_key_pair d e n →
    0 < d.
```

This lemma is used when showing that the `e` exponent, too, is a positive natural number and in the validity theorem for the RSA cryptosystem.

We prove that an RSA key pair remains valid when interchanging the exponents.

```
Lemma swapping_exponents_in_key_pair_preserves_validity :
  forall d e n : nat,
    valid_key_pair d e n →
    valid_key_pair e d n.
```

This is used in the proof of the validity of the RSA signature scheme. The reason that we need this lemma is that when we use the signature scheme, we first sign by lifting to the power of d and then we verify by lifting to the power of e: this is the opposite of the RSA cryptosystem where first e is used and then d when encrypting and then decrypting. The proof of this lemma is more or less just exploiting that multiplication between natural numbers is commutative.

We immediately get that the e exponent is a positive natural number from the two latter lemmas.

```
Lemma zero_less_than_e :
  forall d e n : nat,
    valid_key_pair d e n →
    0 < e.
```

This lemma is applied multiple times in the validity proof for the RSA encryption scheme.

7.4 The RSA Cryptosystem

In this section, we define the RSA cryptosystem and state and prove properties about it. We follow the definition in [3] modulo renaming of some variables.

In Section 7.4.1, we define the RSA cryptosystem state some properties about it. In Section 7.4.2, we specify the cryptosystem. These specifications, finally, enable us to state and prove theorems about it in Section 7.4.3.

7.4.1 Definitions and Properties

As described earlier, the RSA cryptosystem uses a natural number $n = pq$ called the modulus where p and q are distinct primes. This modulus determines the plaintext and ciphertext spaces of the scheme: $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$. Furthermore, two natural numbers e and d are defined such that $ed \equiv_{\phi(n)} 1$ where ϕ is Euler's totient function. Since the RSA cryptosystem is a public-key encryption scheme, we have both a public and a private key: the public key consists of n and e while the private key consists of p, q, and d. This division of keys enables any person to encrypt messages under the public key such that only people knowing the private key can decrypt it.

Now, if we define the key pair $K := (n, p, q, e, d)$, the RSA encryption function e_K and decryption function d_K corresponding to this key are—for any $x \in \mathcal{P}$ and $y \in \mathcal{C}$—

$$e_K(x) = x^e \bmod n \tag{7.1}$$

and

$$d_K(y) = y^d \bmod n. \tag{7.2}$$

The RSA encryption scheme is a cryptosystem. This implies that it is valid in the sense that if we first encrypt any plaintext and then decrypt it under corresponding public and private keys, we end up with the the original plaintext again. This follows the definition of validity seen in the above chapters except that we now use a key pair instead of a single key.

Showing validity of the scheme amounts to showing that $\forall x \in \mathcal{P} : m^{ed} \equiv_n m$. Since $n = pq$ for distinct primes p and q, this proposition is true if and only if both of the propositions $\forall x \in \mathcal{P} : m^{ed} \equiv_p m$ and $\forall x \in \mathcal{P} : m^{ed} \equiv_q m$ are true. This follows from the Chinese remainder theorem since the fact that p and q are distinct primes implies that they are coprime.

Each of the last two propositions are handled similarly. They are shown to be true by using Fermat's little theorem that says that if a is not a multiple of a prime p, then $a^{p-1} \equiv_p 1$. Also, we apply the rule that states that $\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$ when $n = pq$ and p and q are primes.

Now, to, e.g., show that the proposition $\forall x \in \mathcal{P} : m^{ed} \equiv_p m$ is true, we use case analysis on the message m: if $m \equiv_p 0$, then the proposition follows immediately; if $m \not\equiv_p 0$, we can use Fermat's little theorem since m is not a multiple of p and p is a prime. Now, since $ed \equiv_{\phi(n)} 1$ and e and d are positive integers (if any one them were 0, their product could not be congruent to 1), we have that a natural number t exists such that $ed - 1 = t\phi(n)$. This implies that

$$m^{ed} = m^{ed-1}m = m^{t\phi(n)}m = m^{t(p-1)(q-1)}m = (m^{(p-1)})^{t(q-1)}m \equiv_p 1^{t(q-1)}m = m. \tag{7.3}$$

Having shown $m^{ed} \equiv_p m$ for all $m \in \mathcal{P}$, we have proven this proposition. We do the same for q, and the validity of the cryptosystem follows via the Chinese remainder theorem.

We prove a corollary stating that the RSA encryption function is one-to-one. This follows directly from the above validity property: if two distinct messages were to result in equal ciphertexts, then the decryption function being given one of these ciphertexts could not possibly tell how to output the original message with full certainty—its choice of output would have to be independent of the choice between the two messages.

7.4.2 Specifications

In this section, we specify the RSA encryption scheme.

As explained above, encrypting a message using the RSA cryptosystem is done by computing the remainder modulo the modulus n of the message lifted to the power of the encryption exponent e.

```
Definition specification_of_RSA_encryption_function
           (encrypt : nat → nat → nat → nat) :=
  forall x e n : nat,
    encrypt x e n = x ^ e %% n.
```

Likewise, decrypting a ciphertext is done in the same way except that we use the d exponent instead. We write both in full for clarity.

```
Definition specification_of_RSA_decryption_function
           (decrypt : nat → nat → nat → nat) :=
  forall y d n : nat,
    decrypt y d n = y ^ d %% n.
```

We implement actual encryption and decryption functions that satisfy these specifications.

```
Definition RSA_enc_fun_v0 (x e n : nat) : nat :=
  x ^ e %% n.

Definition RSA_dec_fun_v0 (y d n : nat) : nat :=
  y ^ d %% n.
```

As seen, these implementations closely follow the structures of the specifications. Hence, the proofs that the are satisfactory only consist of unfolding the definitions and applying the `reflexivity` tactic. These implementations—or rather their existences—are used in some of the proofs below: the existence of an encryption function is used in the proof of validity of the RSA signature scheme in Section 7.5.3.2. The existence of the decryption function is used in the proof that RSA encryption is one-to-one.

In the below sections, we often require that the plaintexts or ciphertexts being considered are in their respective spaces.

```
Definition in_plaintext_space (x n : nat) :=
  x < n.
```

```
Definition in_ciphertext_space (y n : nat) :=
  y < n.
```

Since $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, we just require the natural numbers representing the plain- and ciphertexts to be less than the modulus n.

7.4.3 Theorems and Proofs

In this section, we state and prove theorems about the RSA cryptosystem. In Section 7.4.3.1, we prove that any ciphertext produced by any RSA encryption function is in the designated ciphertext space. After that, in Section 7.4.3.2, we prove that the RSA encryption scheme is valid. Lastly, in Section 7.4.3.3, we prove that the RSA encryption function is invertible with regards to the plaintext.

7.4.3.1 Ciphertexts are in the Designated Ciphertext Space

First, we prove that RSA ciphertexts are in the designated ciphertext space.

```
Theorem RSA_ciphertexts_are_in_designated_ciphertext_space :
  forall encrypt : nat → nat → nat → nat,
    specification_of_RSA_encryption_function encrypt →
    forall x e n : nat,
      valid_modulus n →
      in_ciphertext_space (encrypt x e n) n.
```

This theorem follows directly from the specification of the RSA encryption function: the last step of the function is to find the remainder modulo n. Therefore, the resulting natural number is less than n and, hence, in the ciphertext space $\mathcal{C} = \mathbb{Z}_n$. Also, we use the `zero_less_than_modulus` lemma that states that any valid modulus is greater than 0 (see Section 7.3.3.1). This restriction on n is necessary as, as mentioned earlier, SSReflect defines $a \,\%\%\, 0$ to be equal to a for any natural number a (and, besides, we want to avoid this situation in reality, too, since computing the remainder modulo 0 is undefined).

7.4.3.2 Validity of the RSA Encryption Scheme

We prove that the RSA encryption scheme is valid, i.e., if any plaintext `x` in the plaintext space is encrypted and then decrypted under corresponding valid keys, then the result is `x` again.

```
Theorem RSA_encryption_scheme_valid :
  forall encrypt decrypt : nat → nat → nat → nat,
    specification_of_RSA_encryption_function encrypt →
    specification_of_RSA_decryption_function decrypt →
    forall d e n : nat,
      valid_key_pair d e n →
      forall x : nat,
        in_plaintext_space x n →
        decrypt (encrypt x e n) d n = x.
```

Unlike for the theorem in the section above, we now require a whole valid key pair and not just a valid modulus. The input message must be in the plaintext space. If it were not, however, we could have still shown the natural number representing the decrypted ciphertext to be equal to the remainder modulo n of the natural number representing the original plaintext.

The proof of the theorem follows the steps outlined in Section 7.4.1. First, we unfold the encryption and decryption functions in the conclusion of the theorem and do a few minor rewrites: we now need to show that `x ^ (e * d) %% n == x`. In order to use the SSReflect lemma `chinese_remainder` described in Section 4.2.2, we show that `x ^ (e * d) == x %[mod p]` and `x ^ (e * d) == x %[mod q]` where `p` and `q` are the two distinct primes whose product is `n`—such primes exist since `n` is a valid modulus (given by the `valid_key_pair` premise in the theorem).

That `x ^ (e * d) = x %[mod p]` is shown by case analysis of `x %% p`. In the case where `x %% p = 0`, we have to show that `x ^ (e * d) %% p == 0`. According to the SSReflect lemmas in Chapter 4, we can reduce `x` modulo `p` after which we can substitute it with 0 (given the case of `x %% p`). We now have to show that `0 ^ (e * d) %% p == 0`. Here, we use the `exp0n` SSReflect lemma in Section 4.2.1: the expression on the left side of == equals `0 %% p` if the `e * d` exponent is positive. In this regard, we use the `zero_less_than_e` and `zero_less_than_d` lemmas in Section 7.3.3.2 in conjunction with the `muln_gt0` lemma in Section 4.2.1: the product of two positive natural numbers is positive itself.

In the inductive case where `x %% p` equals the successor of some natural number `m'`, we have to show that `x ^ (e * d) %% p == m'.+1`. Here, we use that $ed \equiv_{\phi(n)} 1$ which implies that some natural number h exists such that $ed - 1 = h\phi(n)$ (remember that e and d are positive natural numbers): we can rewrite the goal of showing that `x ^ (e * d) %% p == m'.+1` to a goal of showing that `(x ^ (e * d).-1 * x) %% p == m'.+1` to a goal of showing that `(x ^ (h * totient n) * x) %% p == m'.+1`.

The `totient_modulus_n` lemma shown in Section 7.3.3.1 tells us that `totient n = (p - 1) * (q - 1)`. This fact, along with a rewrites using the arithmetic SSReflect lemmas in Chapter 4, we use to transform the subgoal into `((x ^ (p - 1)) ^ (h * (q - 1)) * x) %% p == m'.+1`. Since `totient p = p - 1` (proven in Section 7.3.3.1), we can—after some minor rewrites—apply the SSReflect lemma `Euler_exp_totient` (see Section 4.2.4) that states that `x ^ totient p = 1 %[mod p]` if `x` and `p` are coprime. `x` and `p` are coprime since `x` is not congruent to 0 modulo `p` and `p` is a prime.

We now have the subgoal `(1 ^ (h * (q - 1)) %% p * (x %% p)) %% p == m'.+1`. Via minor rewrites, we transform this to the subgoal `x %% p == m'.+1` which is given as a hypothesis in the inductive case of our case analysis of `x %% p`.

We follow the essentially same procedure (except an additional application of the `mulnC` SSReflect lemma in Section 4.2.1 to switch `(p - 1)` and `(q - 1)` when proving that `x ^ (e * d) == x %[mod q]`.

As described above, the rest of the proof is just using the `chinese_remainder` lemma encapsulating the Chinese remainder theorem. This procedure works since we can show that `p` and `q` are coprime via the `prime_coprime` SSReflect lemma in Section 4.2.3.

7.4.3.3 RSA Encryption Function is Invertible

The last property we prove about the RSA cryptosystem is that the RSA encryption function is invertible with regards to the plaintext.

```
Corollary RSA_encryption_invertible_wrt_plaintext :
  forall encrypt : nat → nat → nat → nat,
    specification_of_RSA_encryption_function encrypt →
    forall d e n : nat,
      valid_key_pair d e n →
      forall x_1 x_2 : nat,
        in_plaintext_space x_1 n →
        in_plaintext_space x_2 n →
        encrypt x_1 e n = encrypt x_2 e n →
        x_1 = x_2.
```

The theorem above states that if a valid key pair is used, then if the ciphertexts corresponding to two proper plaintexts are equal, then the plaintexts are equal, too: the encryption function is invertible with regards to the plaintext.

The theorem is a corollary of the validity theorem in the section above: we apply the validity theorem to the encryption function introduced by the corollary and the `RSA_dec_fun_v0` decryption function defined in Section 7.4.2.

We assert that

```
RSA_dec_fun_v0 (encrypt x_1 e n) d n = RSA_dec_fun_v0 (encrypt x_2 e n) d n
```

which follows from the last premise of the corollary. Now, we can rewrite with the validity twice in this assertion: encrypting and then decrypting under the same key gives the original plaintext input. Hence, we have that `x_1` = `x_2` which is exactly the conclusion of the corollary and the goal we have to show.

The validity theorem has a number of premises. The premises state that the input encryption and decryption functions must satisfy the corresponding specifications, the input key pair must be valid, and the input plaintext must be proper. These premises are all part of the corollary we are proving except for the premise saying that the decryption function must fit the corresponding specification. However, the `RSA_dec_fun_v0` function used as the decryption function has been proven to fit the specification: all premises of the applied theorem hold, and the proof is done.

7.5 The RSA Signature Scheme

In this section, we define the RSA signature scheme and state and prove properties about it. We follow the definition in [3] modulo renaming of some variables.

In Section 7.5.1, we define and state properties about the RSA signature scheme. Then, in Section 7.5.2, we specify the functionalities of the scheme. Lastly, we state and prove theorems involving the scheme in Section 7.5.3.

7.5.1 Definitions and Properties

The functions in the RSA encryption scheme can be used to implement a secure signature scheme.

As for the RSA cryptosystem, we use a modulus n where $n = pq$ for two distinct primes p and q. Likewise, we have two natural numbers e and d such that $ed \equiv_{\phi(n)} 1$. These five natural numbers comprise the key K for the signature scheme: the public key consists of n and e, and the private key consists of p, q, and d.

The message space $\mathcal{P}$ and the signature space $\mathcal{A}$ are both equal to $\mathbb{Z}_n$. Now, signing a message is done in the same way as decrypting a ciphertext in the cryptosystem: $\mathbf{sig}_K(x) = x^d \bmod n$ for any $x \in \mathcal{P}$.[3]

The verification function "encrypts" the signature and checks that it equals the message: $\mathbf{ver}_K(x, y) = \text{true} \Leftrightarrow y^e \bmod n = x$ for any $x \in \mathcal{P}$ and $y \in \mathcal{A}$.

We prove two properties about the RSA signature scheme: first, we prove that any signature is in the designated signature space. This follows from the fact that the computation of the signature ends with computing the remainder modulo n (remember that $\mathcal{A} = \mathbb{Z}_n$) and the fact that n is a positive natural number.

More importantly, we prove that the verification function outputs "true" if and only if the input signature is a signature on the input message. This is a direct consequence of the RSA encryption scheme being valid and of the commutativity of multiplication of natural numbers: the message is first "decrypted" when signing and then "encrypted" when verifying. However, because of the commutativity, this is the same as first encrypting and then decrypting. Now, since the RSA encryption scheme is valid, the original message is output, and in this case the verification outputs "true". If the input message and signature do not correspond, then the verification function will not output "true" since then the message will not be equal to the "encrypted" signature (again because of the validity of the RSA encryption scheme.

7.5.2 Specifications

In this section, we specify the functions used in the RSA signature scheme. First, we specify the RSA signing function.

[3] In this dissertation, $\mathbf{sig}_K : \mathcal{P} \to \mathcal{A}$ respectively $\mathbf{ver}_K : \mathcal{P} \times \mathcal{A} \to \{\text{true}, \text{false}\}$ denote the signing respectively verification function using the key K.

```
Definition specification_of_RSA_signing_function
           (signing_fun : nat → nat → nat → nat) :=
  forall x d n : nat,
    signing_fun x d n = x ^ d %% n.
```

We see that this is equal to the specification of the RSA decryption function in Section 7.4.2.

```
Lemma specifications_of_RSA_signing_and_decryption_functions_equal :
  specification_of_RSA_signing_function =
  specification_of_RSA_decryption_function.
```

We use this lemma when proving the validity of the signature scheme. We do this to be able to use the validity theorem of the RSA cryptosystem directly. The lemma is proven by merely unfolding the specifications and using the `reflexivity` tactic.

The specification of the verification function is a little more involved, but is closely related to the specification of the RSA encryption function also given in Section 7.4.2.

```
Definition specification_of_RSA_verification_function
           (verification_fun : nat → nat → nat → nat → bool) :=
  forall x y e n : nat,
    y ^ e %% n = x ↔ verification_fun x y e n = true.
```

Any function satisfying this specification outputs `true` if and only if the "encryption" of the signature yields the given message.

Similarly to the encryption scheme, we again operate with spaces. Both the message space $\mathcal{P}$ and the signature space $\mathcal{A}$ equal $\mathbb{Z}_n$.

```
Definition in_message_space (x n : nat) :=
  x < n.


Definition in_signature_space (y n : nat) :=
  y < n.
```

We prove that being in the message space is the same as being in the plaintext space for the RSA cryptosystem.

```
Lemma in_message_space_in_plaintext_space_equal :
  in_message_space = in_plaintext_space.
```

Also, we prove that being in the signature space, too, is the same as being in the plaintext space.

```
Lemma in_signature_space_in_plaintext_space_equal :
  in_signature_space = in_plaintext_space.
```

These lemmas, too, are proven by unfolding the specifications and applying the `reflexivity` tactic. They are used in the validity proof of the signature scheme. In this way, we can apply the validity theorem for the RSA cryptosystem directly: one of the premises of that theorem is that the plaintext input to the encryption function is in the plaintext space. Even though we input a message and a signature instead, this is not a problem because of the lemmas above.

7.5.3 Theorems and Proofs

In this section, we state and prove theorems about the RSA signature scheme. In Section 7.5.3.1, we prove that any signature produced by any RSA signing function is in the designated ciphertext space. After that, in Section 7.5.3.2, we prove the validity of the RSA signature scheme.

7.5.3.1 Signatures are in the Designated Signature Space

We prove that any signatures calculated via any signing function satisfying the appropriate specification from Section 7.5.2 are within the designated signature space $\mathcal{A} = \mathbb{Z}_n$.

```
Theorem y_always_in_designated_signature_space :
  forall signing_fun : nat → nat → nat → nat,
    specification_of_RSA_signing_function signing_fun →
    forall x d n : nat,
      valid_modulus n →
      in_signature_space (signing_fun x d n) n.
```

The only reason that we include the premise that the modulus is valid is to ensure that the modulus is positive. In the proof, we apply the `ltn_pmod` lemma that states that `m %% d < d` as long as `0 < d` (see Section 4.2.2). Also, we use our `zero_less_than_modulus` lemma from Section 7.3.3.1 that says that any valid modulus is greater than 0.

7.5.3.2 Validity of the RSA Signature Scheme

We prove that the RSA signature scheme is valid as a consequence of the validity theorem for the RSA encryption scheme in Section 7.4.3.2.

```
Corollary RSA_signature_scheme_valid :
  forall (signing_fun : nat → nat → nat → nat)
         (verification_fun : nat → nat → nat → nat → bool),
    specification_of_RSA_signing_function signing_fun →
    specification_of_RSA_verification_function verification_fun →
    forall x y e d n : nat,
      in_message_space x n →
      in_signature_space y n →
      valid_key_pair d e n →
      (signing_fun x d n = y ↔ verification_fun x y e n = true).
```

This corollary states that for any message in the message space, any signature in the signature space, and any valid key pair, any verification function outputs "true" if and only if the signature equals the signature of the message.

The proof of the corollary is naturally split into two parts: one for each direction of the biimplication in the conclusion. Each part includes an application of the validity theorem for the cryptosystem.

When going the right direction (as opposed to the left) of the biimplication, we apply the validity theorem with `e` and `d` interchanged: as mentioned above, we use these exponents in reverse order when compared to the encryption scheme. Therefore, we use the `swapping_exponents_in_key_pair_preserves_validity` lemma from Section 7.3.3.2: interchanging the exponents preserves the validity of the key pair.

Also, we use the `specifications_of_RSA_signing_and_decryption_functions_equal` lemma from Section 7.5.2. In this way, we can use the provided signing function as the decryption function when applying the validity theorem for the cryptosystem. We use the `RSA_enc_fun_v0` function from Section 7.4.2 as the encryption function.

Our `x` is now in the message space. We need it to be in the plaintext space for the encryption scheme: the `in_message_space_in_plaintext_space_equal` lemma from Section 7.5.2 helps us in this regard.

Applying the validity theorem, we get a hypothesis stating that `signing_fun (RSA_enc_fun_v0 x d n) e n = x`. This hypothesis is then used to show the conclusion of the right direction of the biimplication.

The left direction of the biimplication is shown in much the same way. Now, however, instead of interchanging the exponents, we use `y` as the plaintext instead of `x` when applying the validity theorem for the RSA encryption scheme. Hence, we use the `in_signature_space_in_plaintext_space_equal` lemma from Section 7.5.2. This lemma states that being in the signature space (as `y` is) is the same as being in the plaintext space (which is what we require of `y` when applying the theorem).

7.6 Summary and Conclusion

The goal of this chapter has been to define, model, and prove theorems about the RSA encryption and signature schemes: we have succeeded in doing so.

After having examined valid RSA key pairs, we have shown several properties about the RSA encryption scheme: we have proven that any ciphertext produced by it is a member of the designated ciphertext space, that the scheme is valid, and, as a consequence of the scheme being valid, that the encryption function of the scheme is invertible regarding the plaintext. Additionally, we have proven two properties about the RSA signature scheme: we have shown that any signature produced by it is in the designated signature space and that the scheme is valid.

The work of this chapter has been a very educational task: we have extended our capabilities in the art of Coq by venturing outside the borders of the standard libraries. This migration was slow in the beginning—learning to use the SSReflect lemmas was at first a somewhat confounding undertaking, especially regarding the `reflect` predicate and the hidden `is_true` coercion—but profoundly awarding in the end: we were able to focus on the cryptographic notions and properties instead of the underlying mathematical ones, e.g., having the Chinese remainder theorem already proven was a godsend when proving the validity of the RSA cryptosystem.

Having undergone this involved learning experience, we are ready to start pursuing the last goal of this work: proving properties about the ElGamal cryptosystem.

Chapter 8

The ElGamal Cryptosystem

In this chapter, we introduce and define the ElGamal cryptosystem and state and prove theorems about it.

We start by introducing the ElGamal cryptosystem in Section 8.1 where we also outline the goal of this chapter. Having done that, we define, specify, and prove theorems about the cryptosystem: first, in Section 8.3, we define and specify valid ElGamal keys. This enables us further define and specify the encryption and decryption functionalities of the cryptosystem in Section 8.4 where we also prove theorems about the scheme.

8.1 Introduction

The ElGamal cryptosystem is, as the RSA cryptosystem in Chapter 7, a public-key system: the cryptosystem uses a key pair consisting of a public key and a private key. The public key, which is known to everyone (or, at least, potentially), is used to encrypt plaintexts into ciphertexts that can only be decrypted by a person knowing the corresponding private key. In addition, the ElGamal cryptosystem uses randomness: a random number k, which is kept secret and which is not needed for decryption, is used when encrypting a plaintext. This number influences the output ciphertext.

The ElGamal cryptosystem is built on the assumption that the discrete logarithm problem is hard in the plaintext space being a cyclic group G, i.e., given an element $\alpha \in G$ with order n and a power of this element $\beta \in \langle\alpha\rangle$, it should be hard to find an integer a with $0 \leq a \leq n-1$ such that $\alpha^a = \beta$ (see [3]). In this chapter, we consider the case where the cyclic group G is $\mathbb{Z}_p^*$ for some prime p, and we assume that the discrete logarithm problem is hard in this group.

Our goal with this chapter is to model and prove theorems about the ElGamal cryptosystem in $\mathbb{Z}_p^*$. To do so, we first define and specify valid ElGamal key pairs in Section 8.3. Then we define the ElGamal encryption and decryption functions in Section 8.4.1. These definitions help us in specifying the functions in Section 8.4.2. Using these specifications, we prove the following properties about the ElGamal cryptosystem in Section 8.4.3:

- We show that the ciphertexts produced by the system are in the designated ciphertext space. This space is defined in Section 8.4.1 as $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$.
- We prove that the system is valid, i.e., for any proper plaintext and any valid key pair, first encrypting the plaintext under the public key of the key pair and then decrypting the resulting ciphertext under the private key of the key pair gives the plaintext as a result. This property is very desirable: the owner of the private key wishes to know the original plaintext.
- Finally, we prove that the ElGamal encryption function is invertible regarding the plaintext, i.e., if two proper plaintexts are encrypted into the same ciphertext (using the same public key of a

valid key pair), then they are equal. This property follows from the fact that the system is valid: were it possible to encrypt two different, proper plaintexts into the same ciphertext, the decryption function would not be able to output the actual plaintext with certainty, and the system would not be valid for all proper plaintexts.

8.2 Technical Details

The code in this chapter is contained in the "`ElGamal in Z_p*.v`" file in the `Ole-Dalgaard-Lauridsen_code/ElGamal` directory in the archive file at `http://cs.au.dk/~oledl/Ole-Dalgaard-Lauridsen_code.tar.gz`. Some of the names of the definitions in the code has been appended with an "_" to avoid coincidence of names with the code in Chapter 7.

8.3 The ElGamal Key

In this section, we describe and prove lemmas about the ElGamal key. This key is used in Section 8.4 about the ElGamal cryptosystem.

In Section 8.3.1, we define valid ElGamal keys. This allows us to specify valid keys in Coq: these specifications are defined in Section 8.3.2.

8.3.1 Definitions

The ElGamal cryptosystem assumes that a prime p exists such that the discrete logarithm problem is hard in the multiplicative group of units modulo p ($\mathbb{Z}_p^*$). The key includes such a prime p.

Since p is a prime, $\mathbb{Z}_p^*$ is cyclic. This means that a generator $\alpha \in \mathbb{Z}_p^*$ exists. This α, too, is in the key.

The last part of the key is the natural numbers a and β. Here, it holds that $\beta \equiv_p \alpha^a$. a and β could in principle be negative, but we restrict them to be natural numbers for simplicity (and to be able to use the `nat` type from Section 3.1): if a were negative, we could use $a \bmod (p-1)$ instead and get the same results since α is a generator of $\mathbb{Z}_p^*$ and hence has order $p-1$ (p is prime). Likewise, we could use $\beta \bmod p$ instead since β is used in a multiplication modulo p (using exponentiation). See Section 8.4.1 for the details of the cryptosystem.

As the ElGamal cryptosystem is a public-key system, the key is divided into a public key and a private key: p, α, and β comprise the public key while a comprises the private key.

8.3.2 Specifications

In the following, we use the correspondence between natural numbers and congruence classes where $a \in \mathbb{N}$ with $a < n$ corresponds to the congruence class $\bar{a}_n$ modulo n. This can lead to somewhat imprecise language, but is preferable for brevity. Besides, any confusion is negligible.

Before we specify valid ElGamal keys, we need to specify what is means for a natural number to be a member of $\mathbb{Z}_n$.

```
Definition in_Z_n (x n : nat) :=
  x < n.
```

Using this definition, we can specify what it means to be a member of $\mathbb{Z}_n^*$ (as opposed to its superset $\mathbb{Z}_n$).

```
Definition in_Z_n_star (x n : nat) :=
  in_Z_n x n
  ∧
  coprime x n.
```

A natural number is in $\mathbb{Z}_n^*$ if and only if it is a member of $\mathbb{Z}_n$ and coprime to n. Here, the `coprime` predicate (described in Section 4.2.2) from the SSReflect library is used.

For the ElGamal key to be valid, we require its α element to be a generator of $\mathbb{Z}_p^*$. Therefore, we specify what it means to be a generator of $\mathbb{Z}_n^*$ for some natural number n.

```
Definition generator_of_Z_n_star (g n : nat) :=
  forall x : nat,
    in_Z_n_star x n →
    (exists i : nat,
       g ^ i %% n = x).
```

Any element in $\mathbb{Z}_n^*$ can be expressed as a power of the generator (here, denoted `g`). This property of the α element is never used in our proofs, but is included here for completeness.

We are now ready to specify what it means for the entire ElGamal key to be valid.

```
Definition valid_ElGamal_key_pair (p alpha a beta : nat) :=
  prime p
  ∧
  in_Z_n_star alpha p
  ∧
  generator_of_Z_n_star alpha p
  ∧
  beta = alpha ^ a %[mod p].
```

We require `p` to be a prime. Here, we use the SSReflect predicate `prime` (see Section 4.2.3). `alpha` is a primitive element of the multiplicative group modulo `p`. Lastly, `beta` is congruent to `alpha ^ a` modulo `p`: we do not demand `beta` to be less than `p`.

Being a generator of $\mathbb{Z}_p^*$ implies being a member of it so the second clause of the definition is not necessary given the third clause. However, our definition of generators does not restrict them to be less than p (`n` in the definition): this stems from our use of the correspondence between natural numbers and congruence classes modulo p mention earlier in this section.

8.4 The ElGamal Cryptosystem

In this section, we define, specify, and prove theorems about the ElGamal cryptosystem.

First, we define the encryption and decryption functionalities of the scheme and state properties about them in Section 8.4.1. This lays the foundation for specify them in Section 8.4.2 where we also specify and prove lemmas about the `inverse_modulo_p` function that we use for the ElGamal decryption function. Finally, we state and prove theorems involving the encryption scheme in Section 8.4.3.

8.4.1 Definitions and Properties

As described in Section 8.3.1, the ElGamal key consists of a prime p, a primitive element of α of $\mathbb{Z}_p^*$, and natural numbers a and $beta$ such that $\beta \equiv_p \alpha^a$. a is the private key part while the rest of the key is the public key part. These values determine the spaces of the cryptosystem: the plaintext space is $\mathcal{P} = \mathbb{Z}_p^*$, the randomness space is $\mathcal{R} = \mathbb{Z}_{p-1}$, and the ciphertext space is $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$.

When encrypting using the ElGamal cryptosystem, we use a secret random natural number $k \in \mathcal{R}$. Then the encryption function corresponding to the key $K = (p, \alpha, a, \beta)$ is

$$e_K(x, k) = (y_1, y_2) \tag{8.1}$$

where $y_1 = \alpha^k \bmod p$ and $y_2 = x\beta^k \bmod p$ while the decryption function under the same key is

$$d_K(y_1, y_2) = y_2(y_1^a)^{-1} \bmod p. \tag{8.2}$$

The inverse of y_1^a modulo p exists since $\alpha \in \mathbb{Z}_p^*$ implies that $y_1 \in \mathbb{Z}_p^*$ which, in turn, implies that $y_1^a \in \mathbb{Z}_p^*$. Both y_1 and y_2 are members of $\mathbb{Z}_p^*$. Thus, the ciphertext is in the designated ciphertext space $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$; we prove this property in Section 8.4.3.1.

We prove in Section 8.4.3.2 that the ElGamal cryptosystem is valid in the sense that

$$\forall x \in \mathcal{P} \forall k \in \mathcal{R} : d_k(e_K(x, k)) = x \tag{8.3}$$

for any valid key K. This property follows from the fact that—for any $x \in \mathcal{P}$ and any $k \in \mathcal{R}$—

$$\begin{aligned} d_k(e_K(x, k)) &= d_K(\alpha^k \bmod p, x\beta^k \bmod p) \\ &= (x\beta^k \bmod p)((\alpha^k \bmod p)^a)^{-1} \bmod p \\ &= x\beta^k((\alpha^k)^a)^{-1} \bmod p \\ &= x\beta^k((\alpha^a)^k)^{-1} \bmod p \\ &= x\beta^k(\beta^k)^{-1} \bmod p \\ &= x \bmod p \\ &= x. \end{aligned} \tag{8.4}$$

Here, the last equality follows from the fact that x is in $\mathcal{P} = \mathbb{Z}_p^*$ (again, we use the correspondence mentioned earlier between natural numbers and congruence classes).

In Section 8.4.3.3, we prove, as a corollary to the validity theorem alluded to above, that the ElGamal encryption function is invertible with regards to the plaintext, i.e., any given ciphertext output by the encryption function can only be an encryption of one particular, proper plaintext. This property follows naturally from the cryptosystem being valid: the decryption function *must* be able to tell the plaintext from the ciphertext.

8.4.2 Specifications

In this section, we specify the ElGamal encryption scheme: first, we specify functions that compute modular inverses and implement a function satisfying this specification in Section 8.4.2.1. The specification is used when we specify the decryption functionality in Section 8.4.2.2, and the function is used when we implement a decryption function in the same section. Here, we also specify the encryption functionality.

8.4.2.1 The `inverse_modulo` Function

We specify a function that finds the inverse of a natural number `x` modulo some natural number `p`. We need this specification to be able to specify the decryption functionality of the ElGamal cryptosystem. Also, we need to implement a function satisfying this specification to be able to implement an ElGamal decryption function whose existence we need when proving the corollary in Section 8.4.3.3 stating that ElGamal encryption is invertible with regards to the plaintext.

We start by specifying what it means for two natural numbers to be each other's inverses modulo some natural number `p`.

```
Definition inverses_modulo (a b p : nat) :=
  a * b = 1 %[mod p].
```

This definition is used when we specify the behaviour of any function outputting the modular multiplicative inverse.

```
Definition specification_of_inverse_modulo_function
           (inverse_modulo_fun : nat → nat → option nat) :=
  (forall a p : nat,
     not (coprime a p) →
```

```
    inverse_modulo_fun a p = None)
  ∧
  (forall a : nat,
    inverse_modulo_fun a 0 = None)
  ∧
  (forall a p' : nat,
    coprime a (S p') →
    (exists b : nat,
      inverse_modulo_fun a (S p') = Some b
      ∧
      b < S p'
      ∧
      inverses_modulo a b (S p'))).
```

The specification captures three exhaustive situations: either `a` and `p` are not coprime, `p` is 0, or they are coprime and `p` is a successor of another natural number.

In the first case, `a` does not have any inverse modulo `p` since coprimality with `p` is necessary.

In the second case, we choose to output `None`: this works since `p` will always be positive—besides, computing the remainder modulo 0 is undefined. However, in the SSReflect library `a %% 0` is defined to be `a`. Thus, in fact, 1 is its own inverse modulo 0.

In the third case, `p = S p'` for some natural number `p'`, and `a` and `p` are coprime. We specify that the natural number (inside the `Some` constructor) output by any function satisfying the specification is inverse to `a` and is less than `p`. Also, we specify that the function actually outputs `Some b` for some natural number `b` and *not* `None`.

We choose to restrict the output natural number to be less than `p`. This restriction is necessary to ensure that the output of any function satisfying the specification is uniquely determined by it.

```
Lemma there_is_only_one_inverse_modulo_function :
  forall inverse_modulo_fun_1 inverse_modulo_fun_2 :
         nat → nat → option nat,
    specification_of_inverse_modulo_function inverse_modulo_fun_1 →
    specification_of_inverse_modulo_function inverse_modulo_fun_2 →
    forall a p : nat,
      inverse_modulo_fun_1 a p = inverse_modulo_fun_2 a p.
```

We use this lemma to prove that the implemented ElGamal decryption function in Section 8.4.2.2 satisfies the corresponding specification in the same section. This implementation we use in the proof that any ElGamal encryption function is invertible with regards to the plaintext. In the proof of the above lemma, we use a helping lemma stating that any two natural numbers that are inverses modulo `p` of `a` are equivalent to each other modulo `p`.

```
Lemma inverses_modulo_p_unique_modulo_p :
  forall a b_1 b_2 p : nat,
    inverses_modulo a b_1 p →
    inverses_modulo a b_2 p →
    b_1 = b_2 %[mod p].
```

The proof of this lemma uses another lemma saying that the existence of an inverse of `a` modulo `p` implies that `a` and `p` are coprime if `p` is positive.

```
Lemma coprime_if_inverse_exists_modulo_Sp' :
  forall a p' : nat,
    (exists b : nat, inverses_modulo a b (S p')) →
    coprime a (S p').
```

Below, we define a function proven to satisfy the specification of the `inverse_modulo` function: this function is used when implementing the ElGamal decryption function in Section 8.4.2.2. Also, it is used

in the proof that *that* decryption function satisfies the corresponding specification (also in Section 8.4.2.2) and in the proof of the validity theorem for the ElGamal cryptosystem (in Section 8.4.3.2).

```
Definition inverse_modulo_fun_v0 (a p : nat) : option nat :=
  match coprime a p with
    | true ⇒
      match p with
        | S _ ⇒ Some ((a ^ (totient p − 1)) %% p)
        | 0 ⇒ None
      end
    | false ⇒ None
  end.
```

Euler's totient theorem states that $a^{\phi(n)} \equiv_n 1$ when a and n are coprime. This fact implies that $aa^{\phi(n)-1} \equiv_n 1$. Therefore, $a^{\phi(n)-1}$, and hence also $a^{\phi(n)-1} \bmod p$, is an inverse modulo p of a.

```
Lemma about_coprime_inverses_modulo_Sp' :
  forall a p' : nat,
    coprime a (S p') →
    inverses_modulo a (a ^ (totient (S p') − 1)) (S p').
```

This lemma is used to prove that the above `inverse_modulo_fun_v0` function satisfies the corresponding specification. In addition, it is used in the validity theorem for the ElGamal cryptosystem in Section 8.4.3.2. The lemma is proved by applying the `Euler_exp_totient` SSReflect lemma that says that `a ^ totient n = 1 %[mod n]` if `a` and `n` are coprime (see Section 4.2.4).

8.4.2.2 The Scheme

In this section, we specify the encryption and decryption functionalities of the ElGamal cryptosystem.

We start by specifying the ElGamal encryption function.

```
Definition specification_of_ElGamal_in_Z_p_star_encryption_function
           (encrypt : enc_fun_Type) :=
  forall x k p alpha beta : nat,
    encrypt x k p alpha beta = ((alpha ^ k) %% p, (x ∗ beta ^ k) %% p).
```

This specification follows the formulae shown in Section 8.4.1. A wrapper type for the encryption function is defined for brevity.

```
Definition enc_fun_Type := nat → nat → nat → nat → nat → nat ∗ nat.
```

Likewise, we specify the ElGamal decryption function.

```
Definition specification_of_ElGamal_in_Z_p_star_decryption_function
           (decrypt : dec_fun_Type) :=
  forall inverse_modulo_fun : nat → nat → option nat,
    specification_of_inverse_modulo_function inverse_modulo_fun →
    forall y_1 y_2 p a : nat,
      coprime y_1 p →
      prime p →
      decrypt (y_1, y_2) p a =
      (y_2 ∗ (extract_value_nat (inverse_modulo_fun (y_1 ^ a) p))) %% p.
```

This specification, too, follows the formulae in Section 8.4.1. It makes use of an `inverse_modulo` function fitting the corresponding specification in Section 8.4.2.1. Additionally, it uses the `extract_value_nat` operator defined below. This operator outputs `m` of type `nat` when applied to `Some m` of type `option nat`. If it is applied to `None`, it outputs the natural number `0`, but this situation never occurs in this case: `inverse_modulo_fun (y_1 ^ a) p` equals `Some m` for some natural number `m` since `y_1` and `p` are coprime and `p`, being a prime, is positive—see the specification of the decryption function.

```
Definition extract_value_nat (x : option nat) :
  nat :=
  match x with
    | Some m ⇒ m
    | None ⇒ 0 (* Dummy value. *)
  end.
```

A wrapper type for the decryption function is defined for brevity.

```
Definition dec_fun_Type := (nat * nat) → nat → nat → nat.
```

We implement the ElGamal decryption function.

```
Definition ElGamal_in_Z_p_star_dec_fun_v0 (y : nat * nat) (p a : nat) : nat :=
  (( snd y) *
   (extract_value_nat (inverse_modulo_fun_v0 ((fst y) ^ a) p))) %% p.
```

We use the existence of such a function in the proof of the corollary (in Section 8.4.3.3) stating that the ElGamal encryption function is invertible with regards to the plaintext. In addition to the `fst` and `snd` operators that output the first respectively second values of pairs of values, the decryption function uses the `inverse_modulo_fun_v0` function defined in Section 8.4.2.1. If `inverse_modulo_fun_v0` outputs `None`, the `extract_value_nat` operator outputs the natural number 0 as a dummy value. However, this situation never occurs when using the ElGamal cryptosystem properly since, in this case, the ciphertext `y` is in the ciphertext space $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$: `fst y` and `p` are coprime, and `p`, being a prime, is positive which implies, according to the specification of the `inverse_modulo` function in Section 8.4.2.1, that `inverse_modulo_fun_v0 ((fst y) ^ a) p` equals `Some m` for some natural number `m`.

The decryption function is shown to fit the specification of ElGamal decryption functions above. Here, we use the `there_is_only_one_inverse_modulo_function` from Section 8.4.2.1.

8.4.3 Theorems and Proofs

In this section, we state and prove theorems about the ElGamal cryptosystem: in Section 8.4.3.1, we prove that the ciphertexts computed by the encryption function are members of the designated ciphertext space $\mathcal{C}$. In Section 8.4.3.2, we prove the validity theorem for the scheme, and, as a corollary to this theorem, we prove that the encryption function is invertible with regards to the plaintexts input in Section 8.4.3.3.

In the following, we use predicates to indicate that a natural number is in the plaintext space $\mathcal{P} = \mathbb{Z}_p^*$

```
Definition in_plaintext_space (x p : nat) :=
  in_Z_n_star x p.
```

or that a pair of natural numbers is in the ciphertext space $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$.

```
Definition in_ciphertext_space (y : nat * nat) (p : nat) :=
  in_Z_n_star (fst y) p
  ∧
  in_Z_n_star (snd y) p.
```

The `in_Z_n_star` predicate is defined in Section 8.3.2. The latter definition uses the `fst` and `snd` operators that output the first respectively the second value when applied to a pair of values.

8.4.3.1 Ciphertexts are in Designated Ciphertext Space

We prove that any ciphertext produced by an ElGamal encryption function is in the designated ciphertext space $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$ when applied to a plaintext in the plaintext space. Before doing that, however, we prove two helping lemmas concerning the `in_Z_n_star` predicate from Section 8.3.2.

We show that $\mathbb{Z}_n^*$ is closed under multiplication modulo n.

```
Lemma in_Z_n_star_closed_under_muln_modulo_n :
  forall x y n : nat,
    in_Z_n_star x n →
    in_Z_n_star y n →
    in_Z_n_star ((x * y) %% n) n.
```

The proof is split into two parts corresponding to the two clauses of the conjunction in the definition of the `in_Z_n_star` predicate in Section 8.3.2: first, we show that `in_Z_n ((x * y) %% n) n`. We do that by applying the `ltn_pmod` SSReflect lemma in Section 4.2.2: if `n` is positive, then any remainder modulo `n` is less than `n`. Here, we use the `in_Z_n_n_positive` lemma below: since `in_Z_n_star x n`, `n` is positive.

Secondly, we show that `((x * y) %% n)` and Coq `n` are coprime. Here, we use the `coprime_modl` and `coprime_mull` SSReflect lemmas in Section 4.2.2. These lemmas concern the preservation of coprimality when using the `modn` and `muln` operators.

We also show that $\mathbb{Z}_n^*$ is closed under exponentiation modulo n.

```
Lemma in_Z_n_star_closed_under_expn_modulo_n :
  forall x e n : nat,
    in_Z_n_star x n →
    in_Z_n_star (x ^ e %% n) n.
```

The proof of this lemma, too, is split into two parts: again, the first part is proven by using the `ltn_pmod` and `in_Z_n_n_positive` lemmas. The second subgoal now is `coprime (x ^ e %% n) n`. Hence, we use the `coprime_modl` and `coprime_expl` SSReflect lemmas in Section 4.2.2.

We prove that if the `in_Z_n` predicate holds for any two natural numbers, then the second number is positive.

```
Lemma in_Z_n_n_positive :
  forall n m : nat,
    in_Z_n m n →
    0 < n.
```

This lemma is proven by using the `leq_ltn_trans` and `leq0n` SSReflect lemmas in Section 4.2.1.

Using the two helping lemmas above, we prove that any ciphertext resulting from encrypting a plaintext in the plaintext space is in the designated ciphertext space $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$.

```
Theorem ElGamal_in_Z_p_star_ciphertexts_in_designated_ciphertext_space :
  forall encrypt : enc_fun_Type,
    specification_of_ElGamal_in_Z_p_star_encryption_function encrypt →
    forall p alpha a beta : nat,
      valid_ElGamal_key_pair p alpha a beta →
      forall x k : nat,
        in_plaintext_space x p →
        in_ciphertext_space (encrypt x k p alpha beta) p.
```

In the proof of the theorem, the `in_ciphertext_space` predicate in the conclusion is unfolded: we have to show that both values in the pair of natural numbers output by the encryption function are in $\mathbb{Z}_p^*$. Unfolding the `in_plaintext_space` predicate in the premises, we get that the plaintext is in $\mathcal{P} = \mathbb{Z}_p^*$ already. Likewise, by unfolding the `valid_ElGamal_key_pair` predicate, we get that `alpha` also is in $\mathbb{Z}_p^*$. Since the first value of the pair output by the encryption function is `alpha ^ k %% p` and the second value of the pair is `(x * beta ^ k) %% p`, we can use the `in_Z_n_star_closed_under_muln_modulo_n` and `in_Z_n_star_closed_under_expn_modulo_n` helping lemmas above to prove that the pair is a member of $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$ and, finally, prove the theorem. In doing so, we exploit that `beta = alpha ^ a %[mod p]` which follows from the key pair being valid.

8.4.3.2 Validity of the ElGamal Encryption Scheme

We prove that the ElGamal encryption scheme is valid: any valid plaintext is preserved if it is first encrypted and then decrypted under corresponding public and private keys (here, the two keys are merged into one key $K = (p, \alpha, a, \beta)$). This property is captured by the theorem below. The scheme is valid regardlessly of the randomness k used—the theorem only needs k to be a natural number: we do not require it to be in the randomness space $\mathcal{R} = \mathbb{Z}_{p-1}$.

```
Theorem ElGamal_in_Z_p_star_encryption_scheme_valid :
  forall (encrypt : enc_fun_Type) (decrypt : dec_fun_Type),
    specification_of_ElGamal_in_Z_p_star_encryption_function encrypt →
    specification_of_ElGamal_in_Z_p_star_decryption_function decrypt →
    forall p alpha a beta : nat,
      valid_ElGamal_key_pair p alpha a beta →
      forall x k :  nat,
        in_plaintext_space x p →
        decrypt (encrypt x k p alpha beta) p a = x.
```

In the proof of the theorem, we use the existence of an implementation of the `inverse_modulo` function specified in Section 8.4.2.1: we use the `inverse_modulo_fun_v0` implementation defined in that section.

In the beginning of the proof, we unfold the encryption and decryption functions in the conclusion of the theorem. Doing so, we arrive at the goal below.

```
(( x * beta ^ k) %% p *
 extract_value_nat
   (inverse_modulo_fun_v0 ((alpha ^ k %% p) ^ a) p)) %% p =
x
```

We solve this goal by case analysis of p.

When handling the base case of p, we use contradiction: p is a prime and, thus, cannot be 0.

Handling the inductive case of p is more involving: first, we unfold the `extract_value_nat` operator and the `inverse_modulo_fun_v0` function. Now, we have to show the goal below.

```
(x * (( beta ^ k * (beta ^ k) ^ (totient p'.+1 - 1))  %% p'.+1)) %% p'.+1 =
x
```

This goal is rewritten to the one below by using the arithmetic SSReflect lemmas in Chapter 4 and exploiting that `beta = alpha ^ a %[mod p]` as a consequence of the key pair being valid.

Now, we use the `about_coprime_inverses_modulo_Sp'` lemma in Section 8.4.2.1: `beta ^ k` and `(beta ^ k) ^ (totient p'.+1 - 1)` are inverses modulo `p = p.+1` if `beta ^ k` and p are coprime which they are since `beta = alpha ^ a %[mod p'.+1]` and p are coprime—again, as a consequence of the key pair being valid. Per the definition of inverses, we can substitute the product of the two inverses modulo p with 1 modulo p. Then, after a few additional rewrites with the arithmetic SSReflect lemmas, the goal is transformed into the goal saying that `x %% p'.+1 = x`.

This latter goal is solved by applying the `modn_small` SSReflect lemma in Section 4.2.2: the remainder of x modulo p is x if x is less than p. This requirement about x and p is satisfied since the theorem we are proving assumed that `in_plaintext_space x p`: the plaintext space consists of natural numbers coprime with p and, in particular, less than p. The proof is done.

8.4.3.3 ElGamal Encryption is Invertible

We show that any ElGamal encryption function is invertible regarding the input plaintext as a consequence of the ElGamal cryptosystem being valid.

```
Corollary
  ElGamal_in_Z_p_star_encryption_function_invertible_wrt_plaintext :
  forall encrypt : enc_fun_Type,
```

```
specification_of_ElGamal_in_Z_p_star_encryption_function encrypt →
forall p alpha a beta : nat,
  valid_ElGamal_key_pair p alpha a beta →
  forall x_1 x_2 k_1 k_2 : nat,
    in_plaintext_space x_1 p →
    in_plaintext_space x_2 p →
    encrypt x_1 k_1 p alpha beta = encrypt x_2 k_2 p alpha beta →
    x_1 = x_2.
```

The theorem above states that if encrypting two proper plaintexts under the same, valid key results in equal ciphertexts, then the plaintexts are equal.

The theorem is a corollary of the validity theorem in the section above: we apply the validity theorem to the ElGamal encryption function `encrypt` introduced by the corollary and the `ElGamal_in_Z_p_star_dec_fun_v0` implementation defined in Section 8.4.2.2 of the ElGamal decryption function.

We assert that

```
ElGamal_in_Z_p_star_dec_fun_v0 (encrypt x_1 k_1 p alpha beta) p a =
ElGamal_in_Z_p_star_dec_fun_v0 (encrypt x_2 k_2 p alpha beta) p a
```

which follows from the last premise of the corollary.

We can now rewrite the assertion above using the validity theorem: encrypting and then decrypting any proper plaintext under any valid key yields the plaintext. Hence, the assertion above becomes `x_1 = x_2` by rewriting twice with the validity theorem: the goal being the conclusion of the corollary can be solved.

However, the validity theorem has a few premises that must hold to be able to rewrite with it: the input encryption and decryption functions must satisfy the specifications, the input key pair must be valid, and the input plaintext must be proper. Here, the input encryption function is the one introduced by the corollary: by a premise of the corollary, this function fits the specification. Likewise, the input key pair is the one introduced by the corollary: it is valid. The input two plaintexts—recall that we rewrite twice with the validity theorem—are the `x_1` and `x_2` plaintexts introduced by the corollary: these plaintexts are assumed to be proper. Finally, the decryption function fits the specification: we use the `ElGamal_in_Z_p_star_dec_fun_v0` function proven to be an implementation of the ElGamal decryption function.

8.5 Summary and Conclusion

The goal of this chapter has been to model and prove properties about the ElGamal cryptosystem in $\mathbb{Z}_p^*$ where p is prime: we have succeeded in doing so.

After having defined and specified valid ElGamal key pairs, we have proven the following properties about the cryptosystem: we have shown that any ciphertext output by the ElGamal encryption function is in the designated ciphertext space, that the system is valid, and, as a consequence of the system being valid, that the encryption function is invertible regarding the input plaintext.

The work in this chapter has been educative: we have expanded our Coq capabilities. On the bedrock of the experience and knowledge gained in Chapter 7 about the RSA schemes, we have shown it possible to conduct the same treatment to the ElGamal cryptosystem: we have been able to prove properties about the system and, because of the SSReflect libraries, do so without having to prove all the underlying mathematical properties, e.g., we have not had to prove a lot of rules for arithmetics involving the modulo operator and properties of coprime, natural numbers.

The work of this chapter wraps up the main part of this dissertation.

Part IV

Conclusion

Chapter 9

Summary and Conclusion

Only a Sith deals in absolutes.

Obi-Wan Kenobi

In this chapter, we summarise and conclude on the work of this thesis.

Our goal has been to model and prove theorems about various cryptographic protocols using the Coq proof assistant and, to some extent, the SSReflect libraries, in an elementary fashion. We have succeeded in doing so.

Using the Coq proof assistant, we have proven several cryptographic properties with certainty: one of the main motivators of this work has been to do so. In this way, we have proven results while being sure that all special cases have been taken care of which contrasts with many proofs in the field of cryptography where often many details are left out. In addition, we have been able to do so with only an elementary experience with interactive proof assistants—although, unquestionably, lots of experience has been gained in the process.

Below, we summarise and conclude on each contribution of this work.

The One-Time Pad Cryptosystem Our first contribution is modelling the encryption and decryption functions of the one-time pad cryptosystem and proving properties about them.

We have specified the encryption and decryption functions of the cryptosystem. Using these specifications, we have proven that the cryptosystem is valid, that it is "one-time" in the sense that the ciphertexts resulting from encrypting two plaintexts with the same key reveal non-trivial information about the plaintexts, and that the encryption function is onto by which we mean that, for any plaintext and ciphertext of equal lengths, it is possible to choose a key such that the ciphertext equals the result of encrypting the plaintext under that key.

This contribution has, aside from producing results on the one-time pad cryptosystem, served as a stepping stone into the realm of interactively and formally proving cryptographic properties. Along the way, we have learned that vectors are preferable to lists when modelling blocks of bits. This lesson has helped us tremendously in achieving our second goal.

Modes of Operation Our second goal and contribution are modelling and proving properties about modes of operation and MAC schemes built on top of them.

We have specified generalised versions of the encryption and decryption functions of the modes and a general version of the MAC functions. Then we have specified the functions of each mode we consider and the corresponding MAC functions in terms of these general functions. By first proving theorems stating specific properties about the general functions, we have been able to easily prove properties about the (non-general) functions: these non-general theorems have been proven as corollaries of the former, general theorems.

This procedure of first proving general theorems and then using them to prove corresponding non-general theorems has helped us tremendously: we can easily extend our result to other and, potentially, new modes of operation by specifying their functions (and the corresponding MAC functions if relevant) in terms of the general functions enabling us to show properties about their functions by using the general theorems. We have done just that: we have proven several properties about the already known CBC, CFB, and CTR modes as well as the new IFB, f, and BENC modes we have defined ourselves.

We have proven several properties about the modes of operation and the MAC functions: we have shown that all the modes are valid, that the encryption function of each mode either does or does not propagate plaintext changes (not both), that all the encryption functions propagate changes in the initialisation vectors, and that all the MAC functions propagate changes in the input messages.

The initial goal of the work involving modes of operation was to specify a few existing modes and MAC schemes and proving properties about them. This goal has been surpassed immensely: by generalising the results, we have been able to prove properties about several modes and MAC schemes—including new ones.

Achieving the first goal of showing properties about the one-time pad cryptosystem taught us that vectors were favourable to lists when modelling blocks of bits. This lesson has been excessively helpful when working with modes of operation and MAC functions: using vectors, we have avoided having to consider cases where inputs have wrong lengths—cases that would never happen in real executions of the protocols—without having to include lots of premises about the input lengths. In turn, we have had more time to focus on the meat of the matter: the cryptographic notions and properties.

The RSA Encryption and Signature Schemes Our third contribution is modelling and proving theorems about the RSA encryption and signature schemes.

We have specified valid RSA key pairs and the functions of the RSA schemes. After doing so, we have been able to prove several properties about the schemes. For the encryption scheme, we have shown that ciphertexts produced by it are members of the designated ciphertext space, that the scheme is valid, and that its encryption function is invertible regarding the input plaintexts. For the signature scheme, we have shown that the signatures produced by it are in the designated signature space and that the scheme is valid.

We have used the SSReflect libraries when proving the properties about the RSA schemes. At first, this venture was slow—many SSReflect lemmas were different from anything else we had encountered before—but turned out to be a major benefit: instead of having to devote a lot of time on proving lemmas about the underlying mathematics, e.g., lemmas stating properties of primes and the Chinese remainder theorem, we had the luxury of being able to focus on the cryptographic details.

The work done in achieving the third goal has been a cultivating experience broadening our minds: using the SSReflect libraries, the spectrum of cryptographic properties we can prove has widened extensively.

The ElGamal Cryptosystem Our fourth and last contribution is modelling and proving properties about the ElGamal cryptosystem in $\mathbb{Z}_p^*$ where p is prime.

We have specified valid ElGamal key pairs and the encryption and decryption functions of the cryptosystem. Using these specifications, we have proven that the ciphertexts resulting from using the encryption function are members of the designated ciphertext space, that the system is valid, and that the encryption function is invertible regarding the input plaintexts.

As for the RSA schemes, we have used the SSReflect libraries when proving these properties. Again, doing so has been advantageous: we have been able to focus on proving the cryptographic properties instead of the properties of the underlying mathematics.

Bibliography

[1] The Coq Proof Assistant. `https://coq.inria.fr`. 11

[2] Mathematical Components resources. `http://ssr.msr-inria.inria.fr`. 25

[3] Douglas R. Stinson. *Cryptography: Theory and Practice*. Discrete Mathematics and Its Applications. Third edition, 2005. 39, 40, 42, 44, 163, 167, 171