Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Verifiable Security of Prefix-free Merkle-Damgård

submitted by

Malte Horst Arthur Skoruppa

on August 3, 2012

Supervisor

Prof. Dr. Michael Backes

Advisor

M.Sc. Matthias Berg

Reviewers

Prof. Dr. Michael Backes
Dr. Matteo Maffei

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, August 3, 2012

Malte Skoruppa

Acknowledgments

This thesis forms part of a project to carry out a formal verification of the security of some modern cryptographic hash designs in view of the ongoing SHA-3 competition. Several people were involved in this project, and an associated paper was published at the 25th IEEE Computer Security Foundations Symposium, *CSF 2012* [8].

I would like to thank Prof. Dr. Michael Backes for integrating me into such a fascinating and cutting-edge project on formal verification, as well as for allowing me to travel to Madrid (to work on the project) and to Boston (where the conference took place). I also want to express my gratitude to my advisor Matthias Berg, whose counsel and advice has always been of great value, and who proofread several drafts of this thesis. Further I would like to sincerely thank all my co-authors of aforementioned paper. Beside the two people just mentioned, these are Prof. Dr. Gilles Barthe, who initiated the project with Michael and welcomed us at Madrid; Dr. Benjamin Grégoire, one of the main developers of EasyCrypt who also performed a big part of the indifferentiability proof described in our paper; Dr. César Kunz, who implemented support for loops in EasyCrypt without which our work would not have been possible; and of course Dr. Santiago Zanella Béguelin, another main developer of EasyCrypt, whose intelligent advice and thorough work on the paper has been invaluable to all of us. I also thank Dr. Matteo Maffei, who kindly agreed to be the second reviewer of my thesis.

Finally, I want to thank my fellow students and good friends who made my studies so very enjoyable. Last, but not least, I deeply thank my family, who has faithfully supported me throughout all these years.

Abstract

In recent years, an increasingly popular approach to the game-playing technique in cryptographic proofs is to rigorously specify games as pieces of well-defined program code; this irons out potential ambiguities in their specification, and enables a formal analysis of those games and proofs with the help of automated tools, as envisioned by Halevi [49]. Barthe et al. recently developed EasyCrypt [13], a tool which comes with a fully-specified programming language suitable for the formalization of cryptographic games, as well as an associated probabilistic relational Hoare logic and built-in program verification techniques. EasyCrypt can automatically generate proof obligations arising within a game-playing proof, and solve these mechanically using state-of-the-art automated tools.

In this thesis, we use EasyCrypt to formally verify the indifferentiability of the prefix-free Merkle-Damgård construction, following a seminal proof by Coron et al. [39]. Merkle-Damgård is a cryptographic construction ubiquitously used to implement hash functions: These have received considerable attention from the cryptographic community in the last few years, motivated by the ongoing SHA-3 competition. Indifferentiability is a powerful and non-trivial security notion that yields many implications, and certainly constitutes a desirable security property to achieve when designing a modern cryptographic hash function.

More concretely, we specify a sensible sequence of games in EasyCrypt's language, and discuss the arguments that were needed for machine-checking the validity of the transitions relating those games. We focus in particular on the underlying axiomatization and derived lemmas used to justify the validity of side-conditions that arose when proving invariants of those games. Our results provide a first, but significant step towards a machine-checked verification of the indifferentiability of the finalists of the SHA-3 competition.

Contents

Notations and Conventions

Naming conventions.

x – Usually a block, i.e. a bitstring of length k (see Section 3.1)

y – Usually a chaining value, i.e. a bitstring of length n (see Section 3.1)

m – Usually a message, i.e. a bitstring of arbitrary length (see Section 3.1)

p – Usually a padding, i.e. a list of blocks (see Section 3.1)

IV – A fixed chaining value used as initialization vector in the Merkle-Damgård iteration (see Section 3.1)

pRHL judgments.

$\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi$ – pRHL judgment correlating commands c_1 and c_2 under the pre-condition Ψ and the post-condition Φ (see Section 4.3)

$e\langle 1\rangle$ – Within a pre-condition or a post-condition of a pRHL judgment, the interpretation of e in the left command of that judgment (see Section 4.3)

$e\langle 2\rangle$ – Within a pre-condition or a post-condition of a pRHL judgment, the interpretation of e in the right command of that judgment (see Section 4.3)

$=\{x\}$ – Short-hand notation for $x\langle 1\rangle = x\langle 2\rangle$ for some variable x (see Section 4.3)

Probabilities.

$\Pr[c, m : A]$ – Probability of event A after the execution of command c under initial memory m (see Section 4.4)

$\Pr[c : A]$ – Probability of event A after the execution of command c under an arbitrary initial memory (see Section 4.4)

Maps.

$\mathsf{upd}(T, xy, z)$ – The map resulting from the update of map T with the pair (xy, z) (see Section 6.1.1)

Lists.

nil – The empty list

$x::xs$ – Construction of a list

$xs_\ell \parallel xs_r$ – Concatenation of lists

Options.

$\mathsf{Some}(x)$ – The option associated to object x

None – Empty option

π_o – The projection of option o, i.e. if $o = \mathsf{Some}(x)$ for some object x, then $\pi_o = x$

Types.

τ list – The type of a list over elements of type τ

τ option – The option type over elements of type τ

1 Introduction

THE GAME-PLAYING TECHNIQUE is a salient technique to prove security properties of cryptographic constructions in modern cryptography, advocated by Bellare and Rogaway [23] and Shoup [68]. In a game-based proof, one typically formulates a desired security property of a cryptographic construction as a probabilistic experiment, called the *initial game*. In this game, a *challenger* uses or initializes the construction in question and communicates with an *adversary*, described by a black-box function. The goal of the adversary is to break some security property of the construction, denoted in the initial game by some event S. Then, the probability of event S is the probability of the adversary to break the security property. This probability is usually expressed in terms of a *security parameter*, and the goal is to show that the probability of S is *negligible* in terms of this security parameter. For instance, the probability for some adversary to determine which of two ciphertexts is the encryption of a chosen plaintext may be negligible in the length of the key.

To derive such a bound in the game-based setting, one defines a sequence of intermediate games and proves that the probability of S in each two consecutive games is equal, or differs only by a negligible amount. The last game usually has a form where it is easy to see the probability of S, or where such a probability is assumed or already known. Finally, by summing up over all the differences of the probabilities of S of any two consecutive games, one obtains an upper bound on the difference of the probabilities of event S in the initial and final games, yielding a bound for any adversary to break the desired security property. That is, since the adversary cannot distinguish with more than negligible probability between the initial and final games, and furthermore it is assumed or clear that the adversary cannot perform a successful attack on the final game with more than negligible probability, then it cannot do so for the initial game either.

Whenever two consecutive games do not have the same probability of some event S (typically the result of the game), but whose respective probabilities of event S differ only by a negligible amount, one introduces *failure events* capturing low-probability events that may lead to the games differing in event S. Then, one shows that the games have the same behavior with respect to event S unless one of these failure events occurs. Finally, the Fundamental Lemma of Game-Playing [23, 68] is used to conclude that the difference of the probabilities of event S in the two games is upper-bounded by the probability of the failure events, and the probability of the failure events is concretely calculated.

A CRISIS OF RIGOR. However, in practice the formal arguments used to justify the validity of such transformations can, more often than not, leave many things to be desired. Typically, games may be outlined in natural language at worst, or specified in some kind of ad-hoc pseudocode at best, and their precise mathematical definitions and formal inner workings left unclear. The proofs are then performed in a rather semi-formal way, along with a lot of "handwaving". Yet, even though some transformation may be easily understood at an intuitive level, it may in fact require rigorous mathematical arguments to formally justify its validity. In [23], Bellare and Rogaway speak of a *"crisis of rigor"* concerning cryptographic proofs in general, and endorse the usage of the game-playing technique along with a pseudocode-based descriptive language to formalize the games, but they still assume the operational semantics of the pseudocode to be implicitly clear. In [49], Halevi argues that the problem arises, for the most part, from the technical difficulties of fully formal proofs, especially since they consist in great parts of "mundane" arguments (e.g., checking that a transformation is truly valid), as opposed to the creative part of the proof, such as describing a simulator or a reduction. Indeed, cryptographers can hardly be expected to manually verify each and every one of these mundane parts down to their last detail; on the other hand, this would be necessary to obtain a fully verified proof that can be believed without reservation. In Halevi's words,

> *"The problem is that as a community, we generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect). I became acutely aware of this when I wrote my EME* paper [48]. After spending a considerable effort trying to simplify the proof, I ended up with a 23-page proof of security for a — err — marginally useful mode-of- operation for block ciphers. Needless to say, I do not expect anyone in his right mind to even read the proof, let alone carefully verify it. (On the other hand I was compelled to write the proof, since it was the only way that I could convince myself that the mode was indeed secure.)"*
>
> — Shai Halevi, [49]

Therefore, Halevi takes the idea of code-based game-playing proofs even further, and advocates the definition of a fully-specified programming language used to describe games; further, he advocates the creation of a tool tuned to this language that helps to automate the verification of the mundane parts of the proof, using formal verification methods.

A POSSIBLE SOLUTION: EASYCRYPT. In the years that followed, several tools addressing this situation saw the light of day, outlined in greater detail in Section 2. Recently, a new tool has been developed, called EasyCrypt [13], which defines an imperative programming language with random assignments, suitable for formalizing games in cryptographic proofs, and implements an associated probabilistic relational Hoare logic and program verification techniques. In this way, EasyCrypt enables an automated mechanical verification of game transformations with the help of state-of-the-art SMT solvers and automated theorem provers interacting with EasyCrypt. The cryptographer needs only concentrate on the essence of a proof by specifying a sequence of games, written as small programs, along with Hoare judgments specifying how the games are related to each other, and logical statements expressing properties of the operators used in the games. Then, the reduction can be mechanically verified, significantly increasing the trust that can be placed in the proof. EasyCrypt also delivers tools to support the reasoning with failure events, in particular a mechanical implementation of the Fundamental Lemma of Game-Playing as well as a lemma to bound the probability of failure events (see Section 4.4). Therefore, EasyCrypt seems to be an excellent candidate for achieving the goals described by Halevi's ambitious program.

Hash functions. Motivated by the ongoing *SHA-3 competition*, discussed in more detail in Section 3.3, hash functions and their security properties have received considerable attention from the cryptographic community in recent years. Most hash functions, including all SHA-3 finalists, implement the Merkle-Damgård construction (or a variant thereof), put forward by the pioneering works of Merkle [60] and Damgård [42] in 1990. The plain Merkle-Damgård construction hashes a message as follows. First, a message is split into blocks of the same length using an appropriate *padding function*. Then, a *compression function* is iterated over the blocks, generating an intermediate *chaining value* in each step. The compression function takes as arguments a block and a chaining value. The chaining value used for the compression of the first block is a fixed value called the *initialization vector*; the last chaining value output by the compression function is the final hash. Variants of Merkle-Damgård include the chopping of trailing bits of the last chaining value before outputting the hash, or imposing certain restrictions on the padding function. Several classic notions for the security of hash functions exist, such as *collision resistance*, *preimage resistance* and *second preimage resistance* (see discussion in Section 3.1). Merkle and Damgård showed independently that when used in conjunction with a *suffix-free* padding function, then the Merkle-Damgård iteration yields a collision-resistant hash function [42, 60] – a suffix-free padding function is a padding function with the property that no padding of a message is a suffix of the padding of another message. Both authors considered a concrete suffix-free padding function (in which a message is appended with as many 0's as necessary to obtain a multiple of the block size, and an additional block encodes the length of the original message), although their proofs are applicable more generally for any suffix-free padding function, a fact which at least Merkle also observed [60].

A new security notion that arose relatively recently is the notion of *indifferentiability*, introduced by Maurer et al. in 2004 [59]. Shortly thereafter, Coron et al. applied this notion to hash functions [39]. Essentially, a hash function that is indifferentiable from a random oracle behaves like a variable input length random oracle [22], i.e. an oracle which takes as input a bitstring of variable length and that returns a value chosen uniformly at random from its range for each fresh query that it receives; multiple identical queries are answered with the same response. The proof is performed in an idealized model: For instance, it may be assumed that the underlying compression function is a fixed input length random oracle, or even that an underlying building block of the compression function, such as a block cipher or a permutation, is an ideal primitive. Therefore, an indifferentiability proof gives confidence in the higher level design of the hash function: If an underlying building block is assumed to behave like an ideal primitive, then the hash function as a whole also behaves like an ideal primitive. Coron et al. furthermore showed that the plain Merkle-Damgård construction is indifferentiable from a random oracle when the employed padding function is *prefix-free*, and the compression function is assumed to be a fixed input length random oracle (a prefix-free padding function is defined analogously to a suffix-free one). Indifferentiability of a hash function is a powerful notion, and yields many other security properties including collision resistance, as well as preimage and second preimage resistance of the hash function [4], however the proof is not in the standard model, and may yield worse bounds than a direct proof of a specific security property. Still, indifferentiability is an interesting property to consider and is certainly a desirable security property when designing a state-of-the-art cryptographic hash function.

Contributions. We set out to verify in EasyCrypt the security property of indifferentiability for the finalists of the SHA-3 competition. Our first step was to provide a mechanically verified proof of the indifferentiability of the prefix-free Merkle-Damgård construction, closely following the proof by Coron et al. [39]. The resulting paper was published at the 25th IEEE Computer Security Foundations Symposium (CSF 2012) [8]. My main involvement in this project laid in the abstraction, generalization and formal verification of the side conditions arising within the EasyCrypt proof, in the Coq proof assistant [70]. The resulting EasyCrypt proof consisted of over 3700 lines of code, and the accompanying Coq file had over 4200 lines of code. These are available at:

`http://easycrypt.gforge.inria.fr/csf12/`

The main contributions of this thesis are a deeper look into the indifferentiability proof performed in EasyCrypt than it was done in [8], detailed in Chapter 5, and the development of the aforementioned Coq file, whose proofs are described in Chapter 6.

Outline. The layout of this thesis is as follows. We begin with a look at related work in Chapter 2. Then, we review the background of the thesis by discussing hash functions and their security properties, formally defining the Merkle-Damgård construction, as well as discussing the SHA-3 competition in Chapter 3. Next, we investigate the EasyCrypt tool in some more detail in Chapter 4. Our new results are outlined in Chapters 5 and 6. The high-level EasyCrypt proof is described in Chapter 5; it goes into deeper detail than the corresponding section in [8], and can be considered as an extended version thereof. Chapter 6 details the operators, predicates and axioms that the EasyCrypt proof is based on. Then, it discusses the side conditions that needed to be proven manually to enable EasyCrypt to verify the indifferentiability proof, and further provides formal proofs for all of those side conditions, based solely on the axiomatization presented in the same chapter.

Finally, Chapter 7 concludes with a discussion of the applicability of our results to the finalists of the SHA-3 competition, and future work.

2 Related Work

Pursuing a code-based approach to the game-playing technique to perform cryptographic proofs dates back to early work by Kilian and Rogaway in 1996 [53], and was applied in many subsequent papers by Rogaway, e.g. [21, 29, 30, 50, 51, 66]. Bellare and Rogaway argue in [23] that regarding games as code benefits the technique of game-based proofs since it is less error-prone and imposes some discipline on the process, and describe a sample programming language suitable for formalizing games. In [49], Halevi advocates a rigorous specification of such a programming language and furthermore the creation of an automated tool to support the verification of code-based game-playing proofs. Thus, cryptographers could concentrate on the creative parts of a proof, leaving machines to verify the more mundane parts (checking that some transformations are actually sound).

CryptoVerif was the first tool aiming at the automatic verification of game-based proofs in cryptographic schemes. In [33], Blanchet and Pointcheval used it to prove the unforgeability of the Full-Domain Hash signature scheme under the trapdoor-one-wayness of some permutations. Additionally, it is also suitable to prove security of cryptographic protocols: It has been applied to prove authentication and key secrecy properties of Kerberos [32]. Games in CryptoVerif are specified in a probabilistic process calculus, inspired by the pi-calculus [31], and transformations are captured by probabilistic bisimulation relations. Its focus is on automation: According to Blanchet, security assumptions on cryptographic primitives are applied in a fairly direct way and the tool has limitations as it comes to deeper mathematical reasoning. Therefore it is mainly suited for the analysis of security protocols using high-level primitives (e.g. encryptions or signatures) [31].

Several authors set out to develop frameworks to formalize game-based cryptographic proofs in proof assistants:

- In [62, 63], Nowak reports on a framework to develop game-based cryptographic proofs in Coq using a shallow embedding, in which games are implemented directly as Coq functions. He uses it to give security proofs for the semantic security of ElGamal [62], and for the security of two other cryptographic primitives based on number theory [63]. However, his framework lacks a formal theory for syntactic program transformations, which considerably hampers the possibility for proof automation in game transitions, and it cannot deal with complexity notions such as polynomial-time programs; Nowak and Zhang address these problems in [64] by using a typed probabilistic lambda-calculus, but do not implement the latter in a tool.

- In [1], Affeldt et al. provide a formal proof of the PRP/PRF switching lemma in the proof assistant Coq. In contrast to Nowak's framework, their framework uses a deep embedding; they define a simple programming language following the game-playing framework by Bellare and Rogaway [23], and a specialized version of the Fundamental Lemma. However, their framework is tuned to their particular example, and lacks a general applicability.
- In [9], Backes et al. developed a framework for game-based proofs in the proof assistant Isabelle/HOL [61]. It formalizes a simply-typed probabilistic higher-order functional language with references and support for events fit to express all constructs used in cryptographic games, including random oracles, polynomial-time programs, and sampling from possibly continuous probability measures. It comes along with a theory for observational equivalence and computational indistinguishability, and a definition of the Fundamental Lemma. Since the language is higher-order, oracles can be dealt with in a very natural fashion, as they can simply be passed as arguments to procedures. However, its high level of complexity is costly, and no working examples have been completed. In [44, 69], Driedger and Skoruppa contemplate in their respective Bachelor's theses pen-and-paper proofs of syntactic program transformations in that framework, resulting in example proofs concerning one-way permutations and the semantic security of ElGamal, yet these are not implemented in Isabelle/HOL.
- In [15], Barthe et al. presented CertiCrypt, a fully verified framework built on top of the Coq proof assistant that enables specifying and machine-checking game-based cryptographic proofs. They formalize an imperative programming language with random assignments, structured datatypes and procedure calls, and provide a set of certified tools to reason about equivalence of probabilistic programs, including a probabilistic relational Hoare logic, a theory of observational equivalence, verified program transformations and cryptographic techniques such as reasoning about failure events and lazy/eager sampling. A fair amount of proofs has been carried out in CertiCrypt, such as: the PRP/PRF switching lemma [15, 16]; existential unforgeability under adaptive chosen-message attacks of the FDH signature scheme [15, 76]; semantic security of ElGamal [12, 15] and of OAEP [15]; adaptive IND-CCA security of OAEP [14]. The authors also formalized a class of zero-knowledge proofs, known as Σ-protocols [41], and formally proved the security of many instances thereof from the literature [17]. Furthermore, they verified the semantic security of the Boneh-Franklin Identity-based encryption scheme [34] in [19]. More recently, they machine-checked a proof of the indifferentiability of a construction for hashing into elliptic curves by Brier et al. [35] from a random oracle in [11], and developed an extension of CertiCrypt, called CertiPriv, to reason about differential privacy [18]. The only noteworthy disadvantage of CertiCrypt is that the fully formally carrying out of proofs requires a high level of expertise and can be fairly time consuming, which is why the authors came up with EasyCrypt [13] to help in the automation of this task.

Courant et al. investigate a different approach for automatically proving several security notions such as IND-CPA and adaptive IND-CCA security in [40]: They define a Hoare logic for a simple programming language, and use it to implement an automated verification procedure for asymmetric encryption schemes in the random oracle model. Their focus is on full automation; while their logic is sound, it is not complete (i.e. it may fail to prove a secure encryption scheme), however they do achieve security proofs for several encryption schemes in the literature. Later, Barthe et al. design a general logic which captures typical reasoning patterns common in provable security [10], such as reductions, simulations, random oracles, adaptive adversaries, failure events, and lazy/eager sampling. The logic is called CIL (Computational Indistinguishability Logic) and subsumes the former logic. The authors applied it to prove the security of probabilistic signature schemes, the FDH signature scheme, and adaptive IND-CCA security of OAEP [10]. The rules of the logic have been formalized in Coq by Corbineau et al. [38]. Shortly ago, Daubignard et al. extended this logic such as to capture domain extenders, and presented a generic reduction theorem to prove the indifferentiability of several hash function constructions [43], applicable to four of the five SHA-3 finalists – a work closely related to the one presented in this thesis. However, their method has not yet been implemented.

In stark contrast to all the above work, Koblitz casts a radically different view at the concept of automated theorem-proving in reductionist security proofs [54, 55] and speaks of a "metaphorical race between human and machine". He criticizes inherent weaknesses of this concept, e.g. that mechanically verified proofs are not, or only in a limited way, understandable and replicable by humans – citing Manin's "a good proof is one which makes us wiser" [57] – and submits that computer-assisted theorem-proving has not yet reached a wide audience amongst mathematicians. However, he duly admits to an inchoate knowledge in the field of automated theorem-proving [55], having never constructed a game-based proof himself, and many of his claims are easily refuted. While he does raise a few valid concerns, he also seems to miss the point that machine-checked proofs intend to complete, rather than to replace, human-made proofs; and to increase, rather than to give in the first place, confidence in the correctness of these proofs.

Concerning indifferentiability, the notion was first introduced by Maurer et al. [59] and applied to hash functions by Coron et al. [39], who in the same paper performed a proof of the indifferentiability of the prefix-free Merkle-Damgård construction (as well as other variants of Merkle-Damgård), which this thesis heavily relies on. The ongoing SHA-3 competition has motivated further research on the security of hash functions, including indifferentiability for all the SHA-3 finalists [2, 3, 20, 24, 26, 37]. We discuss these in more detail in later sections.

The notion of indifferentiability is not uncontroversial [65], nor is the random oracle methodology [36]. Nonetheless, indifferentiability from a random oracle represents an increasingly accepted notion for hash functions in the cryptographic community, and remains a good heuristics to assess their security, since an indifferentiability proof demonstrates that a hash function's high-level design preserves an ideal functionality, and implies bounds on all the classical security properties of hash functions [4].

3 Background

THE purpose of this chapter is to recapitulate the basic notions that this work is based on. First, we give some background on hash functions: How they are usually built, and classical security properties of cryptographic hash functions. Then, we introduce the more recent security notion of indifferentiability (and some related results), and finally we discuss the ongoing SHA-3 competition.

3.1 Hash Functions

Generally speaking, hash functions are functions that take inputs of variable length (e.g., large messages) and map them onto small bitstrings of a fixed length. We call such an output a *hash value*, or simply a *hash* for short.

Cryptographic hash functions are hash functions that aim to achieve certain security properties. In practice, they are designed such that their output is seemingly random and unique for each input message, even though in reality it clearly is not, since the domain of a hash function is necessarily much larger than its range. For instance, it should be hard for an adversary to generate a message that has a given hash, or generate a message that has the same hash as another message, or find two messages that have the same hash. Such functions are frequently used in many cryptographic constructions, in particular for (but not limited to) providing authenticity, such as in digital signature schemes or message authentication codes. The output of a cryptographic hash function is also called the *digest* of a message; depending on its usage, one may also speak e.g. of a *fingerprint* or a *checksum*.

At the heart of most cryptographic hash functions realized today lies the so-called Merkle-Damgård construction (or a variant thereof), which was proposed independently at the same time by Ralph Merkle [60] and Ivan Damgård [42]. It works by first splitting a message into *blocks* (bitstrings of a fixed length), using an appropriate *padding function*. Then, a *compression function* is iterated on those blocks. The compression function takes two arguments: a block and a *chaining value* (also a bitstring of a fixed length), and outputs a new chaining value to be used in the next iteration. The final chaining value is the digest of the message. The chaining value to be used in the first iteration is a fixed value called the *initialization vector*. Throughout this thesis, we use the following conventions:

1. A *block* is a bitstring of length k, and is usually denoted by x;
2. a *chaining value* is a bitstring of length n, and is usually denoted by y;
3. a *message* is a bitstring of arbitrary length, and is usually denoted by m;
4. a *padding* is a list of blocks, and is usually denoted by p; and
5. the *initialization vector* is an n-bit constant IV.

Furthermore, a padding function pad is a function of type $\{0,1\}^* \to \{0,1\}^k\ \mathsf{list}$, that maps a message to a padding. A compression function f is a function of type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, that compresses a pair of a block and a chaining value to a new chaining value. Now, we formally define the Merkle-Damgård construction.

Definition 3.1 (Merkle-Damgård)**.** Let pad be a padding function, f be a compression function, and IV be a (public) initialization vector. The Merkle-Damgård hash function $\mathsf{MD}_{\mathsf{pad},f}$ is defined as follows:

$$\begin{array}{lcl} \mathsf{MD}_{\mathsf{pad},f} & : & \{0,1\}^* \to \{0,1\}^n \\ \mathsf{MD}_{\mathsf{pad},f}(m) & := & f^*(\mathsf{pad}(m), \mathsf{IV}) \end{array}$$

where $f^* : \{0,1\}^k\ \mathsf{list} \times \{0,1\}^n \to \{0,1\}^n$ is recursively defined by the equations

$$\begin{array}{lcl} f^*(\mathsf{nil}, y) & := & y \\ f^*(x{::}xs, y) & := & f^*(xs, f(x,y)) \end{array}$$

Later on, we will often use the notion of an *MD-chain* to ease explanations. An MD-chain is a list $[(x_1, \mathsf{IV}), (x_2, y_2), \ldots, (x_j, y_j)]$ of pairs of blocks and chaining values, where the blocks are associated to the padding of a given message, and the chaining values are the intermediate values generated by a given compression function as used in the Merkle-Damgård iteration. We give the formal definition below.

Definition 3.2 (MD-chain)**.** Let pad be a padding function, f be a compression function, and IV be a (public) initialization vector. Then an MD-chain is a list $[(x_1, y_1), \ldots, (x_j, y_j)]$ such that

1. $y_1 = \mathsf{IV}$,
2. $\forall i \in \{1, \ldots, j-1\}.\ f(x_i, y_i) = y_{i+1}$, and
3. $[x_1, \ldots, x_j]$ is in the range of pad.

Remark. In the above definition we obtain $f(x_j, y_j) = \mathsf{MD}_{\mathsf{pad},f}(\mathsf{pad}^{-1}([x_1, \ldots, x_j]))$.

Many variants of the Merkle-Damgård construction exist. For instance, the HAIFA construction [27] uses a compression function of the form $f(x, y, b, s)$, where the additional parameters b and s denote a counter and a salt, respectively. The counter ensures that each block is processed with a unique variant of its compression function, while the usage of a salt value means that the HAIFA construction defines a *family* of hash functions, rather than a single hash function. A popular variant of Merkle-Damgård is the so-called *chop-MD* construction, which chops some trailing bits off the final hash; very similarly, the wide-pipe hash construction [56] prescribes to use larger internal chaining values than the final output, using a different compression function for the last iteration. Using a different compression function in the last iteration is also a widely used paradigm in its own right, and usually referred to as a *final transformation* (note that the chopping of trailing bits is typically not considered part of a final transformation).

All of these variants can be understood as heuristics aimed at improving the security properties of the plain Merkle-Damgård construction. Prominent classical security properties that cryptographic hash functions aim to achieve include the following.

Collision resistance. It is hard to find two distinct messages m_1, m_2 such that $H(m_1) = H(m_2)$.

Preimage resistance. For a given digest h, it is hard to find a message m such that $H(m) = h$.

Second preimage resistance. For a given message m_1, it is hard to find a message $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$.

Resistance to length extension attacks. Given a digest $H(m_1)$, it is hard to compute $H(m_1 \parallel m_2)$ for any non-empty message m_2.

An extensive survey of these (and related) security properties and their relationships can be found in [67]; an overview of several iteration modes and their property-preserving traits appears in [6].

We point out here the following basic result, a special case of which was already described by both Merkle and Damgård in [42, 60]: if a padding function pad is suffix-free (see below), then the plain Merkle-Damgård-construction $\mathsf{MD}_{\mathsf{pad},f}$ is at least as collision-resistant as the compression function f. More precisely, they show that if a collision can be found for $\mathsf{MD}_{\mathsf{pad},f}$, then a collision can also be found for f.

Lastly, we formally define the notion of a suffix-free (resp. prefix-free) padding function.

Definition 3.3 (Prefix- and suffix-free padding)**.** A padding function pad is *prefix-free* (resp. *suffix-free*) iff for any distinct messages m_1, m_2 and any padding p, it holds that $\mathsf{pad}(m_1) \neq \mathsf{pad}(m_2) \parallel p$ (resp. $\mathsf{pad}(m_1) \neq p \parallel \mathsf{pad}(m_2)$).

3.2 Indifferentiability

The notion of indifferentiability of two systems was introduced by Maurer et al. in [59], and constitutes a generalization of the well-established notion of indistinguishability. Similarly as for indistinguishability, it yields an elegant composition theorem; the indifferentiability of two components $\mathcal{H}$ and $\mathcal{F}$ implies that the security of any cryptosystem $\mathcal{C}(\mathcal{F})$ based on the component $\mathcal{F}$ is preserved when replacing the component $\mathcal{F}$ by the component $\mathcal{H}$ in the cryptosystem $\mathcal{C}(\cdot)$, i.e. $\mathcal{C}(\mathcal{H})$ is as secure as $\mathcal{C}(\mathcal{F})$. The difference is that indifferentiability may be applied to more general settings, namely where a distinguisher has access to some internal functions or variables of the considered components.

In [39], Coron et al. applied this notion to hash functions, considering in particular the Merkle-Damgård construction. Broadly speaking, if $\mathcal{H}^{\mathcal{G}}$ is a hash function based on an internal resource $\mathcal{G}$, and $\mathcal{F}$ is a variable input length random oracle, then we say that $\mathcal{H}^{\mathcal{G}}$ is indifferentiable from $\mathcal{F}$ (i.e., an ideal hash function) if there exists a simulator $\mathcal{S}$ such that no adversary can distinguish the scenario where it is given access to the procedures $\mathcal{H}^{\mathcal{G}}$ and $\mathcal{G}$, and the scenario where it is given access to the procedures $\mathcal{F}$ and $\mathcal{S}^{\mathcal{F}}$, with more than negligible probability (see Figure 3.1). The crucial difference to indistinguishability is that the adversary is given access to the internal resource $\mathcal{G}$ that the hash function is based on; this makes it possible to perform a proof of the statement in the ideal model, where $\mathcal{G}$ is assumed to be an ideal primitive, by constructing an appropriate simulator.

Definition 3.4 (Indifferentiability)**.** A procedure $\mathcal{H}$ with oracle access to an ideal primitive $\mathcal{G}$ is $(t_{\mathcal{S}}, q, \epsilon)$-indifferentiable from $\mathcal{F}$ if there exists a simulator $\mathcal{S}$ with oracle access to $\mathcal{F}$ and executing within time $t_{\mathcal{S}}$, such that for any distinguisher $\mathcal{D}$ that makes at most q oracle queries, the following inequality holds:

$$\left|\Pr\left[b \leftarrow \mathcal{D}^{\mathcal{H},\mathcal{G}}(\,) : b\right] - \Pr\left[b \leftarrow \mathcal{D}^{\mathcal{F},\mathcal{S}}(\,) : b\right]\right| \leq \epsilon$$

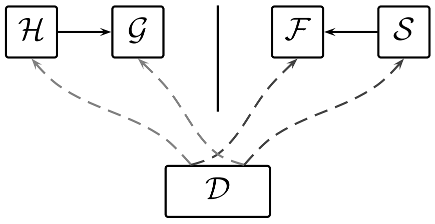

Figure 3.1: Indifferentiability of $\mathcal{H}$ from an ideal functionality $\mathcal{F}$

In the remainder of this thesis, the procedure $\mathcal{H}$ that we consider is the Merkle-Damgård construction, and the component $\mathcal{G}$ is the underlying idealized compression function, implemented as a fixed input length random oracle. The procedure $\mathcal{F}$ represents an ideal hash function, implemented as a variable input length random oracle. Therefore, the role of the simulator $\mathcal{S}$ is to simulate the compression function, i.e. it should behave towards the ideal hash function $\mathcal{F}$ similarly as the ideal compression function $\mathcal{G}$ behaves towards the Merkle-Damgård construction $\mathcal{H}$.

Coron et al. showed in [39] that when used with a prefix-free padding function, the plain Merkle-Damgård construction is indifferentiable from a random oracle. In this thesis, we focus on the formal verification of this statement using EasyCrypt and Coq. Coron et al. also showed the indifferentiability of similar constructions, such as chop-MD (see previous section), for a non-trivial number of output bits chopped off, which however are of no concern in the remainder.

It has been shown in [65] that the indifferentiability composition theorem outlined above does not apply unconditionally, since it does not capture cryptosystems that involve multiple, disjoint adversarial stages. Nevertheless, indifferentiability from a random oracle remains a very strong property for a hash function. Indeed, it shows that an ideal functionality is preserved, rather than a specific property such as collision-resistance. In fact, it implies bounds on the properties of collision resistance, preimage resistance, second preimage resistance and resistance to length extension attacks (see [4]). However, the proof is performed not in the standard model, but in an ideal model (since an underlying primitive is assumed to be ideal), and may deliver looser bounds than a direct proof. The proof presented in this thesis assumes that the compression function used in the Merkle-Damgård construction is ideal; generally, indifferentiability proofs for similar hash function constructions may go deeper than that by assuming that some underlying building block of the compression function is ideal, but not necessarily the compression function itself. Note that the assumption that some primitive is ideal is usually not realistic; however, an indifferentiability proof ensures that the higher level design of the hash function has no structural weaknesses, and therefore gives confidence in this design. In other words, if a vulnerability is found, then the problem should be looked for in the underlying primitive that was considered ideal in the indifferentiability proof. The notion of indifferentiability continues to grow in its acceptance by the cryptographic community, and may very well be considered a must-have for any cryptographic hash function designed in the future.

3.3 The SHA-3 Competition

The U.S. National Institute for Standards and Technology (NIST) specifies seven – supposedly – secure hash algorithms in the Federal Information Processing Standard (FIPS) 180-4, one of them called SHA-1, and six algorithms that belong to the SHA-2 family of hash functions, which differ significantly from SHA-1 (SHA stands for Secure Hash Algorithm). In the light of theoretical attacks found against SHA-1 in 2005 [71, 72], NIST announced that they planned to phase out SHA-1 in favor of the SHA-2 family of hash functions, and later on, in November 2007, announced a public competition to develop a new secure hash algorithm, the winner of which shall be adopted in the FIPS standard and named SHA-3. This competition is commonly known as the *SHA-3 competition*. As of today, there are still no practical collisions found for the full SHA-1, but in 2011 theoretical attacks with an estimated complexity of 2^{51} hash function calls have been published [58].

The cryptographic community answered enthusiastically to NIST's call for a new SHA algorithm and NIST received 64 submissions by the end of October 2008. 51 of those algorithms were accepted for the first round of the competition. In July 2009, 14 were selected for the second round. After further analysis and discussion supported by the cryptographic community, NIST narrowed down the list of candidates to 5 finalists in December 2010.

The five SHA-3 finalists are called BLAKE [7], Grøstl [47], JH [74], Keccak [25], and Skein [45]. All finalists can be considered as variants of the Merkle-Damgård construction:

BLAKE is a HAIFA [27] construction (see Section 3.1). JH, Keccak, and Skein can be seen as a chop-MD construction (or a slight variant thereof in the case of Skein), and Grøstl can be seen as a chop-MD construction with a final transformation before chopping. The compression functions of all finalists are built either from a block cipher (BLAKE, Skein) or from a permutation (Grøstl, JH, Keccak).

The original NIST requirements for the candidates of the SHA-3 competition included the classical security properties of collision resistance, preimage resistance and second preimage resistance. All of the finalists satisfy these properties, and in most cases, they even achieve optimal bounds for them when the underlying block ciphers or permutations are assumed to be ideal [5]. The design and security of all candidates has been surveyed and discussed in depth in [4, 5] and we refer the reader there for more details on this subject.

Although the original requirements did not include the property of indifferentiability from a random oracle, this notion has also been considered in the literature and is achieved by all five finalists [2, 3, 20, 24, 26, 37]. Indeed, as discussed in the previous section, it seems an important property to consider. The proofs of indifferentiability of all five finalists are quite involved and complex. Our original motivation and main aim is the formal verification of those proofs. For several reasons, our proof of the indifferentiability of the prefix-free Merkle-Damgård construction is not immediately applicable to any of the SHA-3 finalists. In particular, only two of them use a prefix-free padding function, and the compression function cannot be assumed ideal for any of the finalists. We discuss this, and how our proof would have to be adapted to fit the SHA-3 finalists in more detail in the conclusion. Nevertheless, our mechanically verified proof of prefix-free Merkle-Damgård provides an excellent starting point to tackle the proofs of those finalists.

4 EasyCrypt

EasyCrypt is a tool that aims to support cryptographers in performing fully formalized and mechanically verified game-based proofs. It was quite recently presented at *CRYPTO 2011*, where it won the Best Paper Award [13]. Using EasyCrypt, it is possible to specify games and the relations used to justify transitions between them in a strictly formal way; more importantly, EasyCrypt helps conducting game-based proofs by automatically generating the necessary verification conditions to derive the validity of desired claims – and solving these with the help of automated tools. It has been developed by the same people who previously published the tool CertiCrypt [15], and builds upon the same programming language pWhile and its related probabilistic relational Hoare logic, called pRHL. The most notable difference to CertiCrypt is that EasyCrypt provides automation to solve proof obligations, i.e. it actively helps the user to verify the "mundane" parts of a proof, so that the user can focus on the more interesting creative parts. Hence, EasyCrypt makes a tremendous step towards the realization of Halevi's program (see Introduction). In this chapter, we condense the most important information from [8] and [13] concerning EasyCrypt relevant for a better understanding of the thesis, including a description of the language pWhile, the logic pRHL, and how to use pRHL judgments to derive probability claims. We also go into two specific techniques that will be important later on, namely how to model random oracles and how to deal with lazy/eager sampling. More details can be found in [8, 13]; many lemmas and techniques that apply to the EasyCrypt framework are also discussed in previous publications on CertiCrypt, such as [12, 15, 16, 75].

4.1 Organization of a Proof

The basic idea of an EasyCrypt proof is that the user needs only to provide a *proof sketch* representing the essence of a security proof. In a typical EasyCrypt proof, a user introduces the types and operators manipulated in a construction, states associated axioms and lemmas, provides a sequence of games, and writes pRHL judgments and probability bounds that relate these games. When parsing such a proof sketch, EasyCrypt automatically generates verification conditions to derive the validity of the pRHL judgments, and solves these with the help of automated theorem provers and SMT solvers. In a final step, these pRHL judgments can be used to derive claims about the probability of certain events in the games.

More in detail, a proof in EasyCrypt is composed of 5 parts:

1. **Declaration of types, constants, operators, and predicates.** Typically, one first declares the basic types and constants used in a scheme. Additionally to the built-in types, a user may declare arbitrary other types (e.g. group elements), or type aliases (e.g. variable-length bitstrings as lists of Booleans). Operators may also be declared or defined and can be used within the definitions of the games. Furthermore, predicates may be declared, providing an elegant way to describe invariants in a compact fashion.

2. **Axioms and lemmas.** The second step is to specify axioms and lemmas that describe properties of objects manipulated by the scheme. These are used by the SMT solvers and automated theorem provers to check the validity of arising proof obligations. They are written as first-order formulae involving expressions over variables, constants, operators and predicates (declared previously). For instance, one may specify as an axiom that some constant is positive, or declare that an operator which decides a predicate is sound and/or complete. The difference between an axiom and a lemma in EasyCrypt is that the validity of a lemma is derived automatically, using either built-in axioms (e.g., properties of lists), or user-defined axioms declared previously. By contrast, the validity of an axiom is simply assumed. Thus, in order to believe the validity of an EasyCrypt proof, one only has to believe the axioms stated in this proof (and the soundness of the tool itself, obviously). In practice, when developing complex proofs, some claims intended as lemmas may be non-trivial, and lie out of the scope of the supported automated tools. Different automated tools may be tried, as EasyCrypt formats proof obligations in the intermediate format of the Why tool [46]; when all of these automated tools fail, it is still possible to export the corresponding statements to the Coq proof assistant and prove them manually. In Chapter 6, we detail the axioms and lemmas used in our proof of indifferentiability, and give human-readable proofs for all of the statements that needed to be derived from our underlying axiomatization.

3. **Definition of a game sequence.** A game is specified as a collection of procedures, global variables manipulated by these procedures, and adversary specifications. Procedures are defined in the language pWhile, defined in Section 4.2. Adversaries are specified as abstract procedures with oracle access; these oracles must be instantiated by other procedures defined within a game. An abstract procedure representing an adversary in a game may only call those procedures which it is given access to. Additionally, a game typically (though not necessarily) implements a main procedure describing the overall game.

4. **pRHL judgments.** The next step is to provide logical judgments in the probabilistic relational Hoare logic implemented by EasyCrypt. These judgments establish logical equivalences between the games; for instance, the return value of a procedure P_1 in a game G_1 may always be equal to the return value of a procedure P_2 in a game G_2. The form of a pRHL judgment in EasyCrypt is $\vdash G_1.P_1 \sim G_2.P_2 : \Psi \Longrightarrow \Phi$, where G_1, G_2 are games, P_1, P_2 are procedures, and Ψ, Φ are relational first-order formulae representing pre-conditions (resp. post-conditions) that must hold before (resp. after) the execution of both P_1 and P_2. This logic is briefly outlined in Section 4.3.

5. **Probability claims.** Finally, one writes probability claims relating the probabilities of events in one or more games using arithmetic operators and mathematical relations. For instance, one may want to derive that $\Pr[\mathsf{G}_1 : b] = \Pr[\mathsf{G}_2 : b]$, i.e. the probability that the games G_1 and G_2 output the same bit b is equal. The validity of such claims can be automatically derived by EasyCrypt, using either previously verified pRHL judgments, or by applying some hard-coded rules to compute bounds on the probability of elementary events. The final claim is typically a bound on the probability of an adversary performing a successful attack on the cryptosystem in consideration. We investigate reasoning about probabilities in more detail in Section 4.4.

4.2 The Language

Games in EasyCrypt are defined in the language pWhile, a strongly-typed, probabilistic, procedural, imperative programming language. The language pWhile includes types ranging over Booleans, integers, fixed-length bitstrings, lists, finite maps, as well as product, option and sum types. The grammar of the language is defined inductively by the following commands:

$$
\begin{array}{rcll}
\mathcal{C} & ::= & \mathsf{skip} & \text{nop} \\
 & | & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\
 & | & \mathcal{V} \xleftarrow{\$} \mathcal{DE} & \text{probabilistic assignment} \\
 & | & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
 & | & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{loop} \\
 & | & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \dots, \mathcal{E}) & \text{procedure call} \\
 & | & \mathcal{C};\ \mathcal{C} & \text{sequence}
\end{array}
$$

where $\mathcal{V}$ is a set of variables, $\mathcal{P}$ is a set of procedures, $\mathcal{E}$ is a set of expressions, and $\mathcal{DE}$ is a set of distribution expressions. Expressions are built from operators and predicates as usual. Distribution expressions represent distributions from which values can be randomly sampled. In this thesis, we consider only the distribution expression $\{0,1\}^n$, which denotes the uniform distribution on bitstrings of length n. That is, the command $y \xleftarrow{\$} \{0,1\}^n$ samples uniformly at random a bitstring of length n and assigns it to the variable y.

A *program memory* maps global and local variables to values. We denote by $\mathcal{M}$ the set of program memories; then, the semantics of a command c is defined as a function $[\![c]\!] : \mathcal{M} \to \mathcal{D}(\mathcal{M})$ from program memories to sub-distributions on memories (if a command does not always terminate, it generates a sub-distribution with total probability less than 1). The precise definition of the semantics is given in [15], but is not relevant in the remainder of the thesis.

4.3 The Logic

At the heart of EasyCrypt lies a probabilistic relational Hoare logic, called pRHL for short. Using this logic, one can express that a relation Φ over program memories must hold after the execution of two commands c_1 and c_2, under the assumption that a relation Ψ over program memories held before their execution; the general form a pRHL judgment is a quadruple of the form

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi,$$

where c_1, c_2 are commands, and Ψ and Φ are first-order formulae built with the usual connectives:

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \Rightarrow \Phi \mid \forall x.\ \Phi \mid \exists x.\ \Phi$$

Here, e is a Boolean expression built from constants and variables that occur in the commands c_1 or c_2 (possibly involving previously defined predicates). These expressions are tagged with $\langle 1\rangle$ or $\langle 2\rangle$, denoting their interpretation in the left or right command, respectively. Additionally, an expression e may contain the keyword res, which denotes the return value of a procedure. Furthermore the short-hand notation $=\{x\}$ may be used to denote $x\langle 1\rangle = x\langle 2\rangle$.

The first-order formulae Ψ and Φ are interpreted as relations over memories: For example, the formula $x\langle 1\rangle \le y\langle 2\rangle$ is interpreted as the relation $R = \{(m_1, m_2) \mid m_1(x) \le m_2(y)\}$. Then, the validity of a pRHL judgment is defined as follows. For any memories m_1, m_2 that satisfy the pre-condition Ψ, the sub-distributions $[\![c_1]\!](m_1)$ and $[\![c_2]\!](m_2)$ must satisfy the lifting of post-condition Φ, i.e. it must hold $[\![c_1]\!](m_1)\, \mathcal{L}(\Phi)\, [\![c_2]\!](m_2)$ (recall that since pWhile is probabilistic, its semantics maps program memories to *sub-distributions* over program memories). The lifting $\mu_1\, \mathcal{L}(R)\, \mu_2$ of a relation $R \subseteq A \times B$ (for discrete A and B) to μ_1 and μ_2 is defined by

$$\exists \mu : \mathcal{D}(A{\times}B).\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \forall (a,b) : A{\times}B.\ \mu(a,b) > 0 \Rightarrow a\, R\, b,$$

where $\pi_1(\mu)$ and $\pi_2(\mu)$ denote respectively the projection of μ on its first and second components. We refer the reader to [13] for more details.

Finally, the above definition can be used to derive rules for valid pRHL judgments for commands, in the style of

$$\frac{\vdash c_1 \sim c_2 : \Phi \Longrightarrow \Phi' \quad \vdash c_1' \sim c_2' : \Phi' \Longrightarrow \Phi''}{\vdash c_1; c_1' \sim c_2; c_2' : \Phi \Longrightarrow \Phi''}\,[\text{Seq}]$$

We refer the reader to [8] or [15] for an ample selection of pRHL rules, and comments about them. These rules are hard-coded into EasyCrypt, but have been verified in the CertiCrypt framework (improving the base of trust by making it possible to export an EasyCrypt proof into CertiCrypt).

To support the verification of pRHL judgments, these rules are implemented as tactics in EasyCrypt that can be applied in a goal-oriented fashion. These tactics may generate additional proof obligations that are passed to SMT solvers and automated theorem provers. Additionally, EasyCrypt implements a weakest pre-condition calculus that computes for a (deterministic loop-free) program and a set of post-conditions a set of sufficient verification conditions (see [13]), which must be implied by the pre-conditions. We conclude by observing that for the verification of a pRHL judgment, some preparation by the user is still required (specifying invariants, axioms and lemmas, sensible application of tactics, ...), but the involved effort is minimal when compared to a tool such as CertiCrypt.

4.4 Reasoning about Probabilities

As mentioned earlier, the last step in an EasyCrypt proof is to derive probability claims about the cryptosystem in question. These probabilities can be derived from previous pRHL judgments. First, we denote by $\Pr[c, m : A]$ the probability of event A after the execution of command c with initial memory m. We simply write $\Pr[c : A]$ when the initial memory is not relevant (i.e., the claim holds for any memory). The two basic rules applicable in EasyCrypt to derive probability claims from pRHL judgments are:

$$\frac{m_1\,\Psi\,m_2 \quad \vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi \quad \Phi \Rightarrow (A\langle 1\rangle \Leftrightarrow B\langle 2\rangle)}{\Pr[c_1, m_1 : A] = \Pr[c_2, m_2 : B]}\,[\text{PrEq}]$$

$$\frac{m_1\,\Psi\,m_2 \quad \vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi \quad \Phi \Rightarrow (A\langle 1\rangle \Rightarrow B\langle 2\rangle)}{\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]}\,[\text{PrLe}]$$

In particular, for $\Psi = \mathsf{true}$ (interpreted as the total relation on memories), we obtain $\Pr[c_1 : A] = \Pr[c_2 : B]$ and $\Pr[c_1 : A] \leq \Pr[c_2 : B]$, respectively.

However, these rules alone would not be sufficient in many instances. Indeed, often cryptographic proofs make use of *failure events*; typically, one argues that unless some specific failure event F is triggered in a program c_1 and in a program c_2, an event A occurs in c_1 if and only if an event B occurs in c_2. Then, the difference in probability of event A in c_1 and event B in c_2 is bounded by the probability of this specific event F. The lemma which formalizes this argument is known as the *Fundamental Lemma of Game-Playing* [23, 68]. EasyCrypt implements this lemma using its probabilistic relational Hoare logic as follows [13].

Lemma 4.1 (Fundamental Lemma). *Let c_1 and c_2 be two terminating commands, m_1 and m_2 be two initial memories, and A, B, F events such that*

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow (F\langle 1\rangle \Leftrightarrow F\langle 2\rangle) \wedge (\neg F\langle 1\rangle \Rightarrow (A\langle 1\rangle \Leftrightarrow B\langle 2\rangle))$$

Then, if $m_1\,\Psi\,m_2$,

1. $\Pr[c_1, m_1 : A \wedge \neg F] = \Pr[c_2, m_2 : B \wedge \neg F]$, *and*
2. $|\Pr[c_1, m_1 : A] - \Pr[c_2, m_2 : B]| \leq \Pr[\mathsf{c}_1, m_1 : F] = \Pr[\mathsf{c}_2, m_2 : F]$.

It still remains to compute a concrete bound on the probability of the failure event F. Typically, failure events are only triggered by oracle calls made by an adversary; when the adversary can only make a known bounded number of oracle queries, the following lemma can be used in EasyCrypt to compute such a bound [8].

Lemma 4.2 (Failure event lemma)**.** *Consider a program $c_1; c_2$, an integer expression i, an event F, and $u \in \mathbb{R}$. Assume the following:*
1. *Free variables in F and i are only modified by c_1 or oracles in some set O;*
2. *After executing c_1, F does not hold and $0 \leq i$;*
3. *Oracles $\mathcal{O} \in O$ do not decrease i and strictly increase i when F is triggered;*
4. *For every oracle $\mathcal{O}$ in O, $\neg F \Rightarrow \Pr[\mathcal{O} : F] \leq u$.*

Then, $\Pr[c_1; c_2 : F \wedge i \leq q] \leq q \cdot u$.

Lastly, we note that EasyCrypt can also derive probability claims using previously proven probability claims using standard arithmetics, and furthermore provides mechanisms to compute bounds on elementary events. For instance, it is possible to derive mechanically that the probability that a value v sampled uniformly at random from a set X belongs to a list of n values which are elements of X and independent of v is at most $^n/_{|X|}$.

4.5 Selected Techniques

Two techniques will be of particular interest later on. Since we will use random oracles, we will first discuss how these can be formalized in EasyCrypt. Furthermore, for a transition described in Section 5.4, we explain how to justify equivalences between games that sample random values across different procedures.

4.5.1 Random Oracles

It is easy to formalize random oracles in EasyCrypt. Indeed, a random oracle can simply be implemented as a stateful procedure that maintains a map storing answers for previous queries. Upon a fresh query, a random oracle samples uniformly at random a value from its output domain, stores this value in its map, and returns it; for a repeated query, the random oracle simply retrieves the old answer from its map, and returns this value.

Figure 4.1 shows an implementation of a random oracle $\mathcal{O} : X \to Y$ in EasyCrypt, where the variable $\boldsymbol{T}$ is a globally defined and initially empty finite map, usually modified exclusively by the procedure $\mathcal{O}$.

In Chapter 5, we will consider *fixed input length* and *variable input length* random oracles. The only difference is that the input domain of $\mathcal{O}$ is either the set of bitstrings of a fixed length, or the set of bitstrings of variable length. Concretely, we will use a fixed input length random oracle of type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, which can equivalently be seen as a fixed input length random oracle that takes bitstrings of length $k + n$.

```
**Oracle** $\mathcal{O}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{T})$ then
   $\boldsymbol{T}[x] \xleftarrow{\$} Y$
 return $\boldsymbol{T}[x]$
```

Figure 4.1: A random oracle in EasyCrypt

4.5.2 Lazy and Eager Sampling

When considering in a transformation two games that both sample random values, the actual sampling of these values in the games may happen across different procedures, meaning that they may be sampled at different points in time. In particular, we distinguish scenarios where values are sampled *lazily*, and scenarios where values are sampled *eagerly*. Lazy sampling means that values are sampled only at that point in time where they are first needed, e.g. when an adversary queries a random oracle on a particular input for the first time. By contrast, eager sampling means that such values may be sampled much earlier, for instance at the beginning of the main procedure of a game, where the variables of the game are initialized. This is best explained by an example: Figure 4.2 illustrates the difference between both scenarios (a similar example is discussed in [16, 75]).

```
Game $\mathsf{G}_{\mathsf{lazy}}$ :
 $\boldsymbol{T} \leftarrow \emptyset$;
 $b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{lazy}}}()$;
 return $b$

Oracle $\mathcal{O}_{\mathsf{lazy}}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{T})$ then
   $\boldsymbol{T}[x] \xleftarrow{\$} Y$
 return $\boldsymbol{T}[x]$
```

```
Game $\mathsf{G}_{\mathsf{eager}}$ :
 $\boldsymbol{T} \leftarrow \emptyset$; $\hat{\boldsymbol{y}} \xleftarrow{\$} Y$;
 $b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{eager}}}()$;
 return $b$

Oracle $\mathcal{O}_{\mathsf{eager}}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{T})$ then
   if $x = \hat{\mathsf{x}}$ then $\boldsymbol{T}[x] \leftarrow \hat{\boldsymbol{y}}$ else $\boldsymbol{T}[x] \xleftarrow{\$} Y$
 return $\boldsymbol{T}[x]$
```

Figure 4.2: An example illustrating lazy and eager sampling

In both games, a distinguisher is given access to some random oracle, and returns a bit which is the output of the game. The only difference between the two games is the following. In the eager game, for some constant $\hat{\mathsf{x}}$, the return value $\hat{\boldsymbol{y}}$ of oracle $\mathcal{O}_{\mathsf{eager}}$ is sampled eagerly, in this case in the initialization phase of the game. By contrast, in the lazy game, oracle $\mathcal{O}_{\mathsf{lazy}}$ samples this value lazily, i.e. only upon the call $\mathcal{O}_{\mathsf{lazy}}(\hat{\mathsf{x}})$. We expect the two games to behave identically, since the value $\hat{\boldsymbol{y}}$ is *uniformly distributed and independent of the adversary's view* – that is, it remains unknown to the adversary until it makes the corresponding call, at which point it is sampled uniformly at random in the lazy scenario.

One of the main motivations for switching from an eager sampling scenario to a lazy sampling scenario is precisely to capture this kind of informal argument. Since the sampling of the value $\hat{\boldsymbol{y}}$ takes place in the main procedure of game $\mathsf{G}_{\mathsf{eager}}$, but in the random oracle

$\mathcal{O}_{\mathsf{lazy}}$ in the game $\mathsf{G}_{\mathsf{lazy}}$, the problem of switching between these two scenarios can be seen as a problem of *inter-procedural* code motion; see [16] for further details.

To formally deal with this kind of a situation, EasyCrypt reduces this problem to a problem of swapping statements. Generally speaking (but continuing with the same names as in the above example for clarity), let $\mathcal{O}_{\mathsf{eager}}$ be an oracle that uses a value $\hat{\boldsymbol{y}}$ that has been sampled eagerly, and $\mathcal{O}_{\mathsf{lazy}}$ the same as $\mathcal{O}_{\mathsf{eager}}$, except that it samples $\hat{\boldsymbol{y}}$ when first needed. To formalize in EasyCrypt the argument that $\hat{\boldsymbol{y}}$ is uniformly distributed and independent of the adversary's view, we show that if the adversary has obtained no information about $\hat{\boldsymbol{y}}$, then re-sampling the value $\hat{\boldsymbol{y}}$ preserves the semantics of the program. More precisely, we identify some condition used that holds whenever the adversary has obtained some information about $\hat{\boldsymbol{y}}$ (e.g. in the above example, a sensible condition would be the condition $\hat{\mathsf{x}} \in \mathsf{dom}(\boldsymbol{T})$). Then, we show that the piece of code $S_{\hat{\boldsymbol{y}}} := \mathsf{if}\ \neg\mathsf{used}\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} Y$, which re-samples the value $\hat{\boldsymbol{y}}$ if the adversary has obtained no information about it, swaps with calls to oracles $\mathcal{O}_{\mathsf{lazy}}$ and $\mathcal{O}_{\mathsf{eager}}$, namely

$$\vdash y \leftarrow \mathcal{O}_{\mathsf{lazy}}(\vec{e}); S_{\hat{\boldsymbol{y}}} \sim S_{\hat{\boldsymbol{y}}}; y \leftarrow \mathcal{O}_{\mathsf{eager}}(\vec{e}) : =\!\{\vec{e}\} \wedge \Phi \Longrightarrow =\!\{y\} \wedge \Phi,$$

where Φ implies equality over all global variables.

Continuing with our above example, to show that the games $\mathsf{G}_{\mathsf{lazy}}$ and $\mathsf{G}_{\mathsf{eager}}$ behave identically, we would introduce the intermediate games shown in Figure 4.3.

```
Game $\mathsf{G}_{\mathsf{lazy'}}$ :
 $\boldsymbol{T} \leftarrow \emptyset$;
 $b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{lazy}}}()$;
 if $\hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})$ then $\hat{\boldsymbol{y}} \xleftarrow{\$} Y$
 else $\hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}]$;
 return $b$

Oracle $\mathcal{O}_{\mathsf{lazy}}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{T})$ then
   $\boldsymbol{T}[x] \xleftarrow{\$} Y$
 return $\boldsymbol{T}[x]$
```

```
Game $\mathsf{G}_{\mathsf{eager'}}$ :
 $\boldsymbol{T} \leftarrow \emptyset$;
 if $\hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})$ then $\hat{\boldsymbol{y}} \xleftarrow{\$} Y$
 else $\hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}]$;
 $b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{eager}}}()$;
 return $b$

Oracle $\mathcal{O}_{\mathsf{eager}}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{T})$ then
   if $x = \hat{\mathsf{x}}$ then $\boldsymbol{T}[x] \leftarrow \hat{\boldsymbol{y}}$ else $\boldsymbol{T}[x] \xleftarrow{\$} Y$
 return $\boldsymbol{T}[x]$
```

Figure 4.3: An example illustrating lazy and eager sampling, continued

Showing in EasyCrypt that $\Pr[\mathsf{G}_{\mathsf{lazy}} : b] = \Pr[\mathsf{G}_{\mathsf{lazy'}} : b]$ and $\Pr[\mathsf{G}_{\mathsf{eager}} : b] = \Pr[\mathsf{G}_{\mathsf{eager'}} : b]$ is easy and is not further detailed here. The step from a lazy sampling scenario to an eager sampling scenario happens in the transition between the games $\mathsf{G}_{\mathsf{lazy'}}$ and $\mathsf{G}_{\mathsf{eager'}}$. As outlined above, we show that

$$\vdash \begin{array}{l} y \leftarrow \mathcal{O}_{\mathsf{lazy}}(x); \\ \mathsf{if}\ \hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} Y \\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}] \end{array} \sim \begin{array}{l} \mathsf{if}\ \hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} Y \\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}]; \\ y \leftarrow \mathcal{O}_{\mathsf{eager}}(x) \end{array} \quad : \; =\!\{x, \boldsymbol{T}, \hat{\boldsymbol{y}}\} \Longrightarrow =\!\{y, \boldsymbol{T}, \hat{\boldsymbol{y}}\}.$$

By using a logic for swapping statements that is built into pRHL [16], this judgment then extends to the following judgment, which EasyCrypt can conclude automatically:

$$\vdash \begin{array}{l} b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{lazy}}}(); \\ \mathsf{if}\ \hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} Y \\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}] \end{array} \sim \begin{array}{l} \mathsf{if}\ \hat{\mathsf{x}} \notin \mathsf{dom}(\boldsymbol{T})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} Y \\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{T}[\hat{\mathsf{x}}]; \\ b \leftarrow \mathcal{D}^{\mathcal{O}_{\mathsf{eager}}}() \end{array} \quad : \quad =\!\{\boldsymbol{T}, \hat{\boldsymbol{y}}\} \Longrightarrow =\!\{b, \boldsymbol{T}, \hat{\boldsymbol{y}}\}$$

The latter then trivially implies

$$\vdash \mathsf{G}_{\mathsf{lazy}'} \sim \mathsf{G}_{\mathsf{eager}'} : \mathsf{true} \Longrightarrow =\!\{b\}.$$

We conclude $\Pr[\mathsf{G}_{\mathsf{lazy}'} : b] = \Pr[\mathsf{G}_{\mathsf{eager}'} : b]$ (see rule [PrEq] in Section 4.4) and hence $\Pr[\mathsf{G}_{\mathsf{lazy}} : b] = \Pr[\mathsf{G}_{\mathsf{eager}} : b]$. We refer the reader to [16] for a formal discussion of this topic.

5 High-level proof

INDIFFERENTIABILITY from a random oracle for the prefix-free Merkle-Damgård construction was first shown by Coron et al. [39] (see Section 3.2), as a pen-and-paper proof. We closely followed that proof in our EasyCrypt formalization, which we will describe in detail in this chapter. Our proof slightly differs from that of Coron in that the latter is based on a Davies-Meyer compression function [73] and is performed in the ideal cipher model (where the underlying block cipher is assumed to be ideal), whereas our proof sees the entire compression function as an ideal primitive and is therefore in the random oracle model — that is, we assume that the compression function is a fixed-input-length random oracle. More precisely, we claim the following.

Theorem 5.1 (Indifferentiability of pfMD)**.** *The Merkle-Damgård construction* MD *with an ideal compression function f, prefix-free padding function* pad*, and initialization vector* IV *is $(t_{\mathcal{S}}, q_{\mathcal{D}}, \epsilon)$-indifferentiable from a variable-input-length random oracle $F_q : \{0,1\}^* \to \{0,1\}^n$, where*

$$\epsilon = \frac{3\ell^2 \cdot q_D^2}{2^n} \qquad t_{\mathcal{S}} = \mathcal{O}(\ell \cdot q_D^2)$$

and ℓ is an upper bound on the block length of pad(m) *for any message m appearing in a query of the distinguisher.*

The precise axiomatization of the function pad (and its inverse unpad) is given in Section 6.1.2.

To perform the proof in EasyCrypt, we first define two games G_{real} and $\mathsf{G}_{\text{ideal}}$. The first game represents the scenario where the distinguisher $\mathcal{D}$ is given access to the Merkle-Damgård construction, implemented by the procedure $F_q : \{0,1\}^* \to \{0,1\}^n$, and the compression function, implemented as a fixed input length random oracle by the procedure $f_q : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. We call this scenario the *real world* since it models the actual MD-construction (based on an idealized compression function). The second game represents the scenario where the hash function behaves like a random oracle itself. Therefore, here the procedure F_q implements a variable-input-length random oracle. Since this hash function is idealized, we call this scenario the *ideal world*. The games G_{real} and

```
Game $\mathsf{G}_{\text{real}}$ :
  $\mathbf{q}_f \leftarrow 0$;
  $\boldsymbol{T} \leftarrow \emptyset$;
  $b \leftarrow \mathcal{D}^{F_q, f_q}()$;
  return $b$

Oracle $F_q(m)$ :
  $xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
  if $\mathbf{q}_f + |xs| \leq q$ then
    $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
    while $xs \neq \mathsf{nil}$ do
      $y \leftarrow f(\mathsf{hd}(xs), y)$;
      $xs \leftarrow \mathsf{tl}(xs)$
  return $y$

Oracle $f(x, y)$ :
  if $(x, y) \notin \mathsf{dom}(\boldsymbol{T})$ then
    $z \xleftarrow{\$} \{0,1\}^n$;
    $\boldsymbol{T}[x, y] \leftarrow z$
  return $\boldsymbol{T}[x, y]$

Oracle $f_q(x, y)$ :
  if $\mathbf{q}_f + 1 \leq q$ then
    $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$;
    $z \leftarrow f(x, y)$
  else $z \leftarrow \mathsf{IV}$;
  return $z$
```

```
Game $\mathsf{G}_{\mathsf{ideal}}$ :
  $\mathbf{q}_f \leftarrow 0$;
  $\boldsymbol{R}, \boldsymbol{T'} \leftarrow \emptyset$;
  $b \leftarrow \mathcal{D}^{F_q, f_q}()$;
  return $b$

Oracle $F_q(m)$ :
  $xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
  if $\mathbf{q}_f + |xs| \leq q$ then
    $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
    $z \leftarrow F(m)$
  else $z \leftarrow \mathsf{IV}$
  return $z$

Oracle $F(m)$ :
  if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
    $z \xleftarrow{\$} \{0,1\}^n$;
    $\boldsymbol{R}[m] \leftarrow z$
  return $\boldsymbol{R}[m]$

Oracle $f_q(x, y)$ :
  if $\mathbf{q}_f + 1 \leq q$ then
    if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
      $xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$;
      if $xs \neq \mathsf{None}$ then
        $m \leftarrow \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{xs}) \| [x])}$;
        $\boldsymbol{T'}[x, y] \leftarrow F(m)$
      else
        $\boldsymbol{T'}[x, y] \xleftarrow{\$} \{0,1\}^n$;
    $z \leftarrow \boldsymbol{T'}[x, y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
  else $z \leftarrow \mathsf{IV}$
  return $z$
```

Figure 5.1: The games G_{real} and $\mathsf{G}_{\text{ideal}}$

$\mathsf{G}_{\text{ideal}}$ are defined in Figure 5.1. Our goal is to show that the probabilities of the outcomes of the games G_{real} and $\mathsf{G}_{\text{ideal}}$ differ at most by a negligible amount, i.e. the advantage of the distinguisher in differentiating the two games is negligible, which proves the claim.

Note that in order to limit the number of oracle queries allowed to the simulator, we increase a global counter $\mathbf{q}_f$ in both the procedures f_q and F_q. A call to the procedure f_q increments the value $\mathbf{q}_f$. A call to the procedure F_q increases the value of $\mathbf{q}_f$ by the number of blocks associated with the padding of the parameter m – the idea is that one call to the Merkle-Damgård iteration results in that many calls to the compression function. Thus, the counter $\mathbf{q}_f$ counts the number of evaluations of the FIL-RO f in game $\mathsf{G}_{\mathsf{real}}$. This is more permissive than the proof of Coron et al. [39], which uses separate counters for the two oracles accessible to the distinguisher. Indeed, our formalization gives the distinguisher the freedom to decide how to distribute his queries among F_q and f_q. The bound which we enforce on this counter is $q := \ell \cdot q_{\mathcal{D}}$; if the value of $\mathbf{q}_f$ exceeds q, then all oracles accessible to the distinguisher in all games always simply return the value IV. This effectively means that if the distinguisher makes n_f queries to f_q and n_F queries to F_q, we require

$$\mathbf{q}_f \leq n_f + \ell \cdot n_F \leq \ell \cdot (n_f + n_F) = \ell \cdot q_{\mathcal{D}} = q.$$

For instance, if the distinguisher makes only queries to f_q but none to F_q, we obtain $\mathbf{q}_f = n_f = q_{\mathcal{D}} = q$. On the other hand, it it makes only queries to F_q (with a maximum query length of ℓ) but none to f_q, we get $\mathbf{q}_f \leq \ell \cdot n_F = \ell \cdot q_{\mathcal{D}} = q$. Therefore, the distinguisher is always allowed at least $q_{\mathcal{D}}$ queries in the two games, as required in Theorem 5.1.

To enable us to perform the proof, the simulator f_q implemented in the game $\mathsf{G}_{\mathsf{ideal}}$ must sensibly simulate the role of the idealized compression function f_q in the game $\mathsf{G}_{\mathsf{real}}$. That is, the simulator $\mathsf{G}_{\mathsf{ideal}}.f_q$ has to behave towards the oracle $\mathsf{G}_{\mathsf{ideal}}.F_q$ similarly as the idealized compression function $\mathsf{G}_{\mathsf{real}}.f_q$ behaves towards the MD-construction $\mathsf{G}_{\mathsf{real}}.F_q$.

The connection between procedures F_q and f_q in the real world is the following. Let m be a message with $\mathsf{pad}(m) = [x_1, \ldots, x_j]$. Then, if the distinguisher makes the query $F_q(m)$ in the game $\mathsf{G}_{\mathsf{real}}$, the result will be the return value of $f(x_j, y_j)$, where y_j is some chaining value sampled during the Merkle-Damgård iteration. Thus, the idealized compression function f_q will return the same value upon the query $f_q(x_j, y_j)$, since it relays to the FIL-RO f. The simulator needs to achieve the same thing in the ideal world. Whenever the distinguisher queries $F_q(m)$ in the game $\mathsf{G}_{\mathsf{ideal}}$ and then reconstructs the MD-chain associated with m using queries to simulator f_q, the simulator should return the same answer as $F_q(m)$ upon the query $f_q(x_j, y_j)$. Conversely, if the distinguisher first builds an MD-chain using queries to f_q and later queries F_q for the associated message of this chain, the answer of F_q should be consistent with the last query to f_q. To achieve this in the ideal world, the simulator must detect when a fresh query it receives may be the *last* element of an MD-chain, and relay to the VIL-RO F in this case. We will see later that up to some negligible probability, the distinguisher can only construct an MD-chain using calls to f_q by querying all its elements in the correct order, because it cannot guess the intermediate chaining values. Therefore, if the distinguisher constructs an MD-chain using calls to f_q, when it queries for the last element, it must already have queried f_q for all preceding elements of this chain. Hence, the simulator is able to detect that a chain has been completed (as it maintains a table of its previous queries), and it can reconstruct the entire message m and forward it to oracle F to maintain consistency with F_q.

To better express this idea, we introduce the notion of a *complete chain*. Intuitively, a complete chain is an MD-chain whose elements are all in the domain of the map maintained by the simulator, except for the ultimate element of this chain. Formally, it is defined as follows.

Definition 5.2 (Complete chain)**.** A complete chain in a map $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ is a list $[(x_1, y_1), \ldots, (x_j, y_j)]$ such that

1. $y_1 = \mathsf{IV}$,
2. $\forall i \in \{1, \ldots, j-1\}.\ (x_i, y_i) \in \mathsf{dom}(T) \wedge T[x_i, y_i] = y_{i+1}$, and
3. $[x_1, \ldots, x_j]$ is in the domain of unpad.

To detect complete chains, the simulator uses the function $\mathsf{findseq}$ (see Section 6.1.4 for its axiomatization). The function $\mathsf{findseq}((x,y), \boldsymbol{T'})$ searches in the map $\boldsymbol{T'}$ for a complete chain of the form $[(x_1, y_1), \ldots, (x_{j-1}, y_{j-1}), (x, y)]$ and returns $[(x_1, y_1), \ldots, (x_{j-1}, y_{j-1})]$, or None if no such chain is found. Thus, upon a query $f_q(x, y)$ in game $\mathsf{G}_{\mathsf{ideal}}$, the simulator relays to F whenever there is a complete chain of the form $[(x_1, \mathsf{IV}), \ldots, (x_{j-1}, y_{j-1}), (x, y)]$ in the simulator's map $\boldsymbol{T'}$. Otherwise, the simulator answers with a uniformly distributed random value.

An example illustrating how the simulator works is depicted in Figure 5.2. Here, the simulator receives a sequence of queries

$$y_2 \leftarrow f_q(x_1, \mathsf{IV});\ y_3 \leftarrow f_q(x_2, y_2);\ y_4 \leftarrow f_q(x_3, y_3)$$

where $[x_1, x_2, x_3] = \mathsf{pad}(m)$ for some message m. The first two queries do not complete a chain – note that because of the prefix-freeness of pad there is no message m' such that $\mathsf{pad}(m') = [x1]$ or $\mathsf{pad}(m') = [x_1, x_2]$. Therefore the simulator samples random values for these queries; indeed, in both worlds the corresponding chaining values are unknown to the distinguisher until it has queried f_q for them. The third query completes a chain, and so the simulator answers it by querying F for the hash of $\mathsf{unpad}([x_1, x_2, x_3])$. Hence, the simulator's answer to query $f_q(x_3, y_3)$ is consistent with the random oracle's answer to query $F_q(m)$, exactly as in the real world.

$$\begin{array}{ccc}
\underbrace{\boxed{\quad (x_1, IV) \quad}}_{\boldsymbol{T'}[x_1,IV]\leftarrow y_2} & \underbrace{\boxed{\quad (x_2, y_2) \quad}}_{\boldsymbol{T'}[x_2,y_2]\leftarrow y_3} & \underbrace{\boxed{\quad (x_3, y_3) \quad}}_{\boldsymbol{T'}[x_3,y_3]\leftarrow y_4} \\
\text{incomplete chain} & \text{incomplete chain} & \text{complete chain} \\
y_2 \xleftarrow{\$} \{0,1\}^n & y_3 \xleftarrow{\$} \{0,1\}^n & y_4 \leftarrow F(m)
\end{array}$$

Figure 5.2: An example illustrating how the simulator works

To show that the probabilities of the outcomes of the two games G_{real} and $\mathsf{G}_{\text{ideal}}$ differ only by a negligible amount, we introduce a sequence of hybrid games. We then upper-bound the probability by which each two consecutive games may differ; by summing up over these probabilities, we obtain a concrete bound for the advantage of the distinguisher in telling apart the initial and final games. Specifically, we prove:

$$|\Pr[\mathsf{G}_{\text{real}} : b] - \Pr[\mathsf{G}_{\text{ideal}} : b]| \leq \frac{3q^2}{2^n} \tag{5.1}$$

Recall that $q = \ell \cdot q_{\mathcal{D}}$, which justifies the bound stated in Theorem 5.1. The overall running time of the simulator is bounded by $\mathcal{O}(\ell \cdot q_D^2)$, as the function $\mathsf{findseq}$ runs in time $\mathcal{O}(\ell \cdot q_D)$ and the simulator may be called up to $q_{\mathcal{D}}$ times. The running time of the function $\mathsf{findseq}$ is justified since the length of any possible chain in $\boldsymbol{T}$ is upper-bounded by q; see Section 6.1.4 for a pseudocode implementation of $\mathsf{findseq}$. The functions unpad and mapfst (see Sections 6.1.2 and 6.1.3 respectively for their definitions) are easily seen to run in time $\mathcal{O}(\ell \cdot q_D)$ as well.

To bound the amount by which the probabilities of the outcomes of two consecutive games may differ, we may introduce failure events in the hybrid games. We then apply the Fundamental Lemma of Game-Playing (see Theorem 4.1) to conclude that the probability of some event in the two games is bounded by the probability of this failure event. The event which we are interested in is usually the outcome of the game, but we may also be interested e.g. in relating the probability of some failure events in two games with each other.

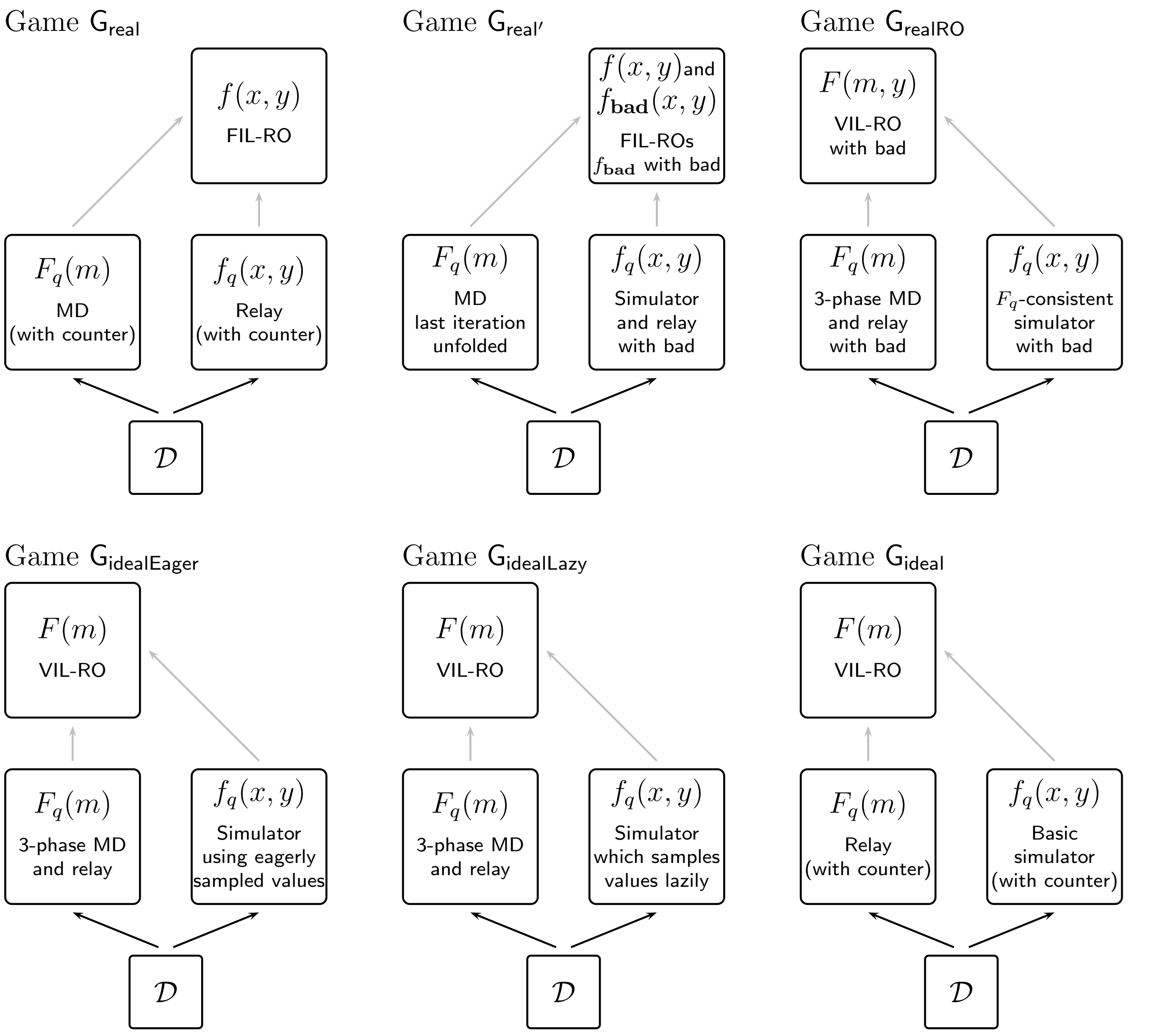

Figure 5.3: High-level overview of the games

Figure 5.3 shows a high-level overview over the sequence of games which we use. The arrows show in which way the procedures may call each other; the distinguisher is always given access to the two procedures $F_q : \{0,1\}^* \to \{0,1\}^n$ and $f_q : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, as depicted by the black arrows in the figure. In all cases, these two procedures may call further internal procedures, as depicted by the gray arrows. Note that the distinguisher has no knowledge of the existence of these internal procedures, or of the calls made by F_q and f_q.

The two most difficult steps are those from $\mathsf{G}_{\mathsf{real}'}$ to $\mathsf{G}_{\mathsf{realRO}}$, and from $\mathsf{G}_{\mathsf{idealEager}}$ to $\mathsf{G}_{\mathsf{idealLazy}}$. All other steps are easy and essentially concern the introduction or removing of some failure events, or some code cleanup. In the step from $\mathsf{G}_{\mathsf{real}'}$ to $\mathsf{G}_{\mathsf{realRO}}$, we perform the step from the actual Merkle-Damgård iteration using an idealized compression function to a variable-

```
Game $\mathsf{G}_{\mathsf{real}'}$ :
  $\mathbf{q}_f \leftarrow 0$;
  $\boldsymbol{T}, \boldsymbol{T'} \leftarrow \emptyset$;
  $\boldsymbol{Y} \leftarrow$ nil;
  $\boldsymbol{Z} \leftarrow$ IV::nil;
  $\mathbf{bad}_1 \leftarrow$ false;
  $\mathbf{bad}_2 \leftarrow$ false;
  $\mathbf{bad}_3 \leftarrow$ false;
  $b \leftarrow \mathcal{D}^{F_q, f_q}()$;
  return $b$
```

```
Oracle $F_q(m)$ :
  $xs \leftarrow \mathsf{pad}(m)$;  $y \leftarrow$ IV;
  if $\mathbf{q}_f + |xs| \leq q$ then
    $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
    while $|xs| > 1$ do
      $y \leftarrow f_{\mathbf{bad}}(\mathsf{hd}(xs), y)$;
      $xs \leftarrow \mathsf{tl}(xs)$
    $y \leftarrow f_{\mathbf{bad}}(\mathsf{hd}(xs), y)$
  return $y$
```

```
Oracle $f(x, y)$ :
  if $(x, y) \notin \mathsf{dom}(\boldsymbol{T})$ then
    $z \xleftarrow{\$} \{0,1\}^n$;
    $\boldsymbol{Z} \leftarrow z{::}\boldsymbol{Z}$;  $\boldsymbol{Y} \leftarrow y{::}\boldsymbol{Y}$;
    $\boldsymbol{T}[x, y] \leftarrow z$
  return $\boldsymbol{T}[x, y]$

Oracle $f_{\mathbf{bad}}(x, y)$ :
  if $(x, y) \notin \mathsf{dom}(\boldsymbol{T})$ then
    $z \xleftarrow{\$} \{0,1\}^n$;
    $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
    $\boldsymbol{Z} \leftarrow z{::}\boldsymbol{Z}$;  $\boldsymbol{Y} \leftarrow y{::}\boldsymbol{Y}$;
    $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
    $\boldsymbol{T}[x, y] \leftarrow z$
  return $\boldsymbol{T}[x, y]$
```

```
Oracle $f_q(x, y)$ :
  if $\mathbf{q}_f + 1 \leq q$ then
    if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
      $xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$
      if $xs \neq$ None then
        $\boldsymbol{T'}[x, y] \leftarrow f_{\mathbf{bad}}(x, y)$
      else
        if set_bad3$(y, \boldsymbol{T'}, \boldsymbol{T})$ then
          $\mathbf{bad}_3 \leftarrow$ true;
          $\boldsymbol{T'}[x, y] \leftarrow f(x, y)$
        else
          $\boldsymbol{T'}[x, y] \leftarrow f_{\mathbf{bad}}(x, y)$
    $z \leftarrow \boldsymbol{T'}[x, y]$;  $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
  else $z \leftarrow$ IV
  return $z$
```

Figure 5.4: The game $\mathsf{G}_{\mathsf{real}'}$

input-length random oracle and a consistent simulator. The step from $\mathsf{G}_{\mathsf{idealEager}}$ to $\mathsf{G}_{\mathsf{idealLazy}}$ concerns the step from eager sampling to lazy sampling, which we will address in more detail in the concerned section. For all the transformations except the one from $\mathsf{G}_{\mathsf{real}'}$ to $\mathsf{G}_{\mathsf{realRO}}$, we have that the probabilities of the outcomes are equal; the difference of the probabilities of the outcomes of $\mathsf{G}_{\mathsf{real}'}$ and $\mathsf{G}_{\mathsf{realRO}}$ is bounded by $3q^2/2^n$, which leads to the final bound stated in Equation 5.1. In the following sections, we will consider each of these transformations in detail.

5.1 From $\mathsf{G}_{\mathsf{real}}$ to $\mathsf{G}_{\mathsf{real}'}$

The first hybrid game that we consider is the game $\mathsf{G}_{\mathsf{real}'}$, defined in Figure 5.4. In this game, three easy but essential modifications to the game $\mathsf{G}_{\mathsf{real}}$ are being made to prepare for the next transformation:

1. The events $\mathbf{bad}_1$, $\mathbf{bad}_2$ and $\mathbf{bad}_3$ and some related changes are introduced;
2. the relay procedure f_q is transformed into a simulator that still behaves as a relay;
3. the last loop iteration in F_q is unrolled.

The first change means that we instrument the code such that under certain conditions, a failure event is raised. We will explain the precise meaning of those events in the next transformation, where they are needed. We also introduce a copy of oracle f, called $f_{\mathbf{bad}}$, which behaves like f except that it may set the flags $\mathbf{bad}_1$ and $\mathbf{bad}_2$. The lists $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are introduced to allow us to appropriately detect these events. We note that modifying the global variables $\boldsymbol{Y}$, $\boldsymbol{Z}$, $\mathbf{bad}_1$, $\mathbf{bad}_2$ or $\mathbf{bad}_3$, all unknown to the distinguisher, does not change the observable behavior of the procedures F_q and f_q.

The second change, concerning procedure f_q, introduces a new map $\boldsymbol{T'}$ of queries known to the distinguisher. Observe that $\boldsymbol{T'} \subseteq \boldsymbol{T}$, because queries to F_q result in entries being added only to $\boldsymbol{T}$, whereas queries to f_q result in the same entries being added to both $\boldsymbol{T}$ and $\boldsymbol{T'}$. Additionally, the procedure f_q now behaves in two ways depending on whether $\mathsf{findseq}((x, y), \boldsymbol{T'}) \neq \mathsf{None}$; this distinction is a central element of the simulator. Indeed, if this condition holds, then there is a complete chain in map $\boldsymbol{T'}$ ending in (x, y). This means that the simulator should call the VIL-RO F to maintain consistency with the oracle F_q in the ideal world. If it does not hold, then the simulator should simply sample a fresh random value, i.e. behave like a FIL-RO. In this game, oracle f_q still returns the same answer in both cases, and only sets $\mathbf{bad}_{\{1,2,3\}}$ accordingly.

The unrolling of the last loop iteration will also be important in the next transformation, where the last call to the procedure $f_{\mathbf{bad}}$ will be replaced by a call to the VIL-RO introduced there. Here, to be able to show that the unrolling of the last iteration is sound, EasyCrypt needs to know that after the assignment $xs \leftarrow \mathsf{pad}(m)$, the variable xs contains at least one element, which follows immediately from Lemma 6.17.

Since none of the transformations change the behavior of the procedures F_q or f_q accessible to the distinguisher, it is easy to conclude in EasyCrypt that

$$\vdash \mathsf{G}_{\mathsf{real}} \sim \mathsf{G}_{\mathsf{real}'} : \mathsf{true} \Longrightarrow =\{b\},$$

which implies

$$\Pr\left[\mathsf{G}_{\text{real}} : b\right] = \Pr\left[\mathsf{G}_{\text{real}'} : b\right]. \tag{5.2}$$

5.2 From $\mathsf{G}_{\mathsf{real}'}$ to $\mathsf{G}_{\mathsf{realRO}}$

This transformation lies at the heart of our proof. Intuitively, we replace the FIL-ROs f and $f_{\mathbf{bad}}$ by a VIL-RO F, and modify the oracle F_q such that it behaves as a relay for F, while the simulator now maintains consistency with F_q by calling F whenever a new query completes a chain in $\boldsymbol{T'}$. To prove the validity of this transformation, we have to show that the simulator indeed behaves consistently with F_q.

Broadly speaking, we introduce the following changes:

1. The VIL-RO F is substituted for the FIL-ROs f and $f_{\mathbf{bad}}$;
2. the Merkle-Damgård iteration in F_q is split into three phases;
3. the last call to the procedure $f_{\mathbf{bad}}$ in oracle F_q of the game $\mathsf{G}_{\mathsf{real}'}$ is replaced by a call to the procedure F in the game $\mathsf{G}_{\mathsf{realRO}}$;
4. the simulator also calls oracle F in case it gets a new query which completes a chain, so as to be consistent with oracle F_q;
5. the new maps $\boldsymbol{I}$ and $\boldsymbol{T'_i}$ are introduced, along with a modification to map $\boldsymbol{T}$ and further related changes, to allow for a more powerful book-keeping.

```
Game $\mathsf{G}_{\mathrm{realRO}}$ :
 $\mathbf{q}_f \leftarrow 0$;
 $\mathbf{q}'_f \leftarrow 1$;
 $\boldsymbol{T}, \boldsymbol{T'}, \boldsymbol{T'_i}, \boldsymbol{R}, \boldsymbol{I} \leftarrow \emptyset$;
 $\boldsymbol{I}[0] \leftarrow (\mathsf{IV}, \mathsf{false})$;
 $\boldsymbol{Y} \leftarrow \mathsf{nil}$;
 $\boldsymbol{Z} \leftarrow \mathsf{IV}::\mathsf{nil}$;
 $\mathbf{bad}_1 \leftarrow \mathsf{false}$;
 $\mathbf{bad}_2 \leftarrow \mathsf{false}$;
 $\mathbf{bad}_3 \leftarrow \mathsf{false}$;
 $b \leftarrow \mathcal{D}^{F_q, f_q}()$;
 return $b$
```

```
Oracle $F_q(m)$ :
 $xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
 $i \leftarrow 0$;
 if $\mathbf{q}_f + |xs| \leq q$ then
   $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
   while $|xs| > 1 \wedge$
       $(\mathsf{hd}(xs), y) \in \mathsf{dom}(\boldsymbol{T'})$ do
     $i \leftarrow \boldsymbol{T'_i}[\mathsf{hd}(xs), y]$;
     $y \leftarrow \boldsymbol{T'}[\mathsf{hd}(xs), y]$;
     $xs \leftarrow \mathsf{tl}(xs)$;
   while $|xs| > 1 \wedge$
       $(\mathsf{hd}(xs), i) \in \mathsf{dom}(\boldsymbol{T})$ do
     $i \leftarrow \boldsymbol{T}[\mathsf{hd}(xs), i]$;
     $y \leftarrow \mathsf{fst}(\boldsymbol{I}[i])$;
     $xs \leftarrow \mathsf{tl}(xs)$;
   while $|xs| > 1$ do
     $z \overset{\$}{\leftarrow} \{0,1\}^n$;
     $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
     $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
     $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
     $\boldsymbol{T}[\mathsf{hd}(xs), i] \leftarrow \mathbf{q}'_f$;
     $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{true})$;
     $i \leftarrow \mathbf{q}'_f$;
     $y \leftarrow z$;
     $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
     $xs \leftarrow \mathsf{tl}(xs)$
   $y \leftarrow \mathsf{fst}(F(m, y))$
 return $y$
```

```
Oracle $F(m, y)$ :
 if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
   $z \overset{\$}{\leftarrow} \{0,1\}^n$;
   $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
   $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
   $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
   $\boldsymbol{R}[m] \leftarrow (z, \mathbf{q}'_f)$
   $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$
   $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
 return $\boldsymbol{R}[m]$
```

```
Oracle $f_q(x, y)$ :
 if $\mathbf{q}_f + 1 \leq q$ then
   if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
     $xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$
     if $xs \neq \mathsf{None}$ then
       $m \leftarrow \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{xs}) \| [x])}$;
       $(z, j) \leftarrow F(m, y)$;
       $\boldsymbol{T'}[x, y] \leftarrow z$; $\boldsymbol{T'_i}[x, y] \leftarrow j$;
     else
       $found, found\_bad3 \leftarrow \mathsf{false}$;
       $j, k' \leftarrow 0$;
       while $k' < \mathbf{q}'_f$ do
         if $\mathsf{snd}(\boldsymbol{I}[k'])$ then
           $found\_bad3 \leftarrow (\mathsf{fst}(\boldsymbol{I}[k']) = y)$;
         else if $\neg found \wedge \mathsf{fst}(\boldsymbol{I}[k']) = y \wedge$
           $(x, k') \in \mathsf{dom}(\boldsymbol{T}) \wedge$
           $\mathsf{snd}(\boldsymbol{I}[\boldsymbol{T}[x, k']])$ then
           $found \leftarrow \mathsf{true}$; $j \leftarrow \boldsymbol{T}[x, k']$;
         $k' \leftarrow k' + 1$;
       if $found$ then
         $z \leftarrow \mathsf{fst}(\boldsymbol{I}[j])$; $\boldsymbol{I}[j] \leftarrow (z, \mathsf{false})$;
         $\boldsymbol{T'}[x, y] \leftarrow z$; $\boldsymbol{T'_i}[x, y] \leftarrow j$;
       else
         if $found\_bad3$ then
           $\mathbf{bad}_3 \leftarrow \mathsf{true}$;
           $z \overset{\$}{\leftarrow} \{0,1\}^n$;
           $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$;
           $\boldsymbol{T'}[x, y] \leftarrow z$;
           $\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
           $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
         else
           $z \overset{\$}{\leftarrow} \{0,1\}^n$;
           $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
           $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
           $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
           $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$;
           $\boldsymbol{T'}[x, y] \leftarrow z$;
           $\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
           $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
   $z \leftarrow \boldsymbol{T'}[x, y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
 else $z \leftarrow \mathsf{IV}$
 return $z$
```

Figure 5.5: The game $\mathsf{G}_{\mathrm{realRO}}$

Replacing the FIL-ROs by a VIL-RO is a significant modification. Indeed, now the oracle F_q does not use a FIL-RO to perform a normal Merkle-Damgård iteration any longer. Instead, the last call to $f_{\mathbf{bad}}$ is replaced by a call to the new VIL-RO F. However, F_q still performs an iteration similar to Merkle-Damgård in order to sample values eagerly, as was also the case in game $\mathsf{G}_{\mathsf{real'}}$ – we will further discuss these loops later. As for the simulator f_q, it now behaves differently depending on the condition $\mathsf{findseq}((x, y), \boldsymbol{T'}) \neq \mathsf{None}$; specifically, whenever a query (x, y) to f_q completes a chain in $\boldsymbol{T'}$, the simulator now calls F, to maintain consistency with F_q from the distinguisher's perspective.

We introduce several new maps in this game. Firstly, we introduce a map $\boldsymbol{I} : \mathbb{N} \to \{0,1\}^n \times \mathbb{B}$ which enumerates all values randomly sampled so far in game $\mathsf{G}_{\mathsf{realRO}}$ through means of the counter $\mathbf{q}'_f$. Each time when a new value is sampled at random, this newly sampled value is written to $\boldsymbol{I}[\mathbf{q}'_f]$, with an additional *tainted* flag that keeps track of values known to the distinguisher (here false means that a value is known to the distinguisher, and true means it is not). Then, the counter $\mathbf{q}'_f$ is incremented. As before, the simulator maintains a map $\boldsymbol{T'} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ of previously answered queries; we additionally introduce the map $\boldsymbol{T'_i} : \{0,1\}^k \times \{0,1\}^n \to \mathbb{N}$, also maintained by the simulator, to be able to relate the maps $\boldsymbol{T'}$ and $\boldsymbol{I}$. The VIL-RO F maintains its previous queries in a table $\boldsymbol{R} : \{0,1\}^* \to \{0,1\}^n \times \mathbb{N}$, where the integer indicates the value of counter $\mathbf{q}'_f$ at the point at which the corresponding value was sampled. Finally, the map $\boldsymbol{T} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ from game $\mathsf{G}_{\mathsf{real}'}$ is transformed into a map $\boldsymbol{T} : \{0,1\}^k \times \mathbb{N} \to \mathbb{N}$ in game $\mathsf{G}_{\mathsf{realRO}}$. Now, instead of immediately storing the query-answer pairs mapping pairs of blocks and chaining values to chaining values, it simply stores the indices of those chaining values enumerated in table $\boldsymbol{I}$. All of these modifications allow us to precisely keep track of the order in which queries were made, and of which values are known to the distinguisher. It is necessary to formulate explicitly several invariants about the relation of these maps first so as to allow EasyCrypt to derive more powerful invariants. We list them here for reference, and the purpose of completeness. We start off with very basic invariants:

1. $\boldsymbol{T'}\langle 1\rangle \subseteq \boldsymbol{T}\langle 1\rangle$
2. $1 \leq \mathbf{q}'_f\langle 2\rangle$
3. $\mathbf{q}'_f\langle 2\rangle \notin \mathsf{dom}(\boldsymbol{I}\langle 2\rangle)$
4. $\forall i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle).\ 0 \leq i < \mathbf{q}'_f\langle 2\rangle$
5. $\forall i.\ 0 \leq i < \mathbf{q}'_f\langle 2\rangle \Rightarrow i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle)$
6. $\mathsf{fst}(\boldsymbol{I}\langle 2\rangle[0]) = \mathsf{IV}$

Further, relating the new maps $\boldsymbol{I}\langle 2\rangle$, $\boldsymbol{T'_i}\langle 2\rangle$, $\boldsymbol{R}\langle 2\rangle$ and $\boldsymbol{T}\langle 2\rangle$ between each other and with the old map $\boldsymbol{T'}\langle 1\rangle$, we derive

7. $\forall (x,y) \in \mathsf{dom}(\boldsymbol{T'}\langle 1\rangle).\ (x,y) \in \mathsf{dom}(\boldsymbol{T'_i}\langle 2\rangle)$
8. $\forall (x,y) \in \mathsf{dom}(\boldsymbol{T'_i}\langle 2\rangle).\ (x,y) \in \mathsf{dom}(\boldsymbol{T'}\langle 1\rangle) \wedge \boldsymbol{T'_i}\langle 2\rangle[(x,y)] \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge$
   $\mathsf{fst}(\boldsymbol{I}\langle 2\rangle[\boldsymbol{T'_i}\langle 2\rangle[(x,y)]]) = \boldsymbol{T'}\langle 1\rangle[(x,y)]$
9. $\forall m \in \mathsf{dom}(\boldsymbol{R}\langle 2\rangle).\ \mathsf{snd}(\boldsymbol{R}\langle 2\rangle[m]) \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge$
   $\mathsf{fst}(\boldsymbol{I}\langle 2\rangle[\mathsf{snd}(\boldsymbol{R}\langle 2\rangle[m])]) = \mathsf{fst}(\boldsymbol{R}\langle 2\rangle[m])$
10. $\forall (x,i) \in \mathsf{dom}(\boldsymbol{T}\langle 2\rangle).\ i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge (x, \mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i])) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle) \wedge$
    $\boldsymbol{T}\langle 2\rangle[(x,i)] \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge \mathsf{fst}(\boldsymbol{I}\langle 2\rangle[\boldsymbol{T}\langle 2\rangle[(x,i)]]) = \boldsymbol{T}\langle 1\rangle[(x, \mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i]))]$

Certain (highly improbable) events may occur that may lead to the games $\mathsf{G}_{\text{real}'}$ and $\mathsf{G}_{\text{realRO}}$ being distinguishable; we capture these using appropriate failure events, and prove that the difference of the probabilities of the outcomes of the games is bounded by the probability that these events occur. We use three different failure events $\mathbf{bad}_1$, $\mathbf{bad}_2$ and $\mathbf{bad}_3$.

1. $\mathbf{bad}_1$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that is either IV or has already been sampled for a distinct query before. The role of this event is twofold: on the one hand, if IV is sampled as a random value, then there could exist a complete chain in $\boldsymbol{T}$ that is a suffix of another complete chain in $\boldsymbol{T}$ as illustrated in the first example of Figure 5.6 (here $\boldsymbol{T}[x_2, y_2] = \mathsf{IV}$). The problem is that oracle F_q in the game G_{real} will generate the same values for the two messages corresponding to those two chains, while F_q in the game $\mathsf{G}_{\text{ideal}}$ most likely will not. On the other hand, if a sampled value has been sampled for another query before, then there could exist two complete chains in $\boldsymbol{T}$ that collide at some point and are identical from that point on as illustrated in the second example of Figure 5.6. Again the two corresponding messages would yield the same answer in G_{real} but most likely not in $\mathsf{G}_{\text{ideal}}$ on queries to F_q.

2. $\mathbf{bad}_2$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that has already been used as a chaining value in a previous query. This means that this query may be part of an MD-chain of which the distinguisher has already queried later points in the chain, which should never happen. The event also captures that no fixed-points (i.e. entries of the form $\boldsymbol{T}[x, y] = y$) should be sampled.

3. $\mathbf{bad}_3$ is triggered whenever a chaining value y in a query has already been sampled as a random value and is in the range of $\boldsymbol{T}$ for some previous query (x', y'), but (x', y') does not appear in the domain of $\boldsymbol{T'}$ and (x', y') is not the last element of a complete chain in $\boldsymbol{T}$. Intuitively, this means that y was never returned by f_q or F_q and hence the distinguisher managed to guess a previously randomly sampled value which should be unknown to it.

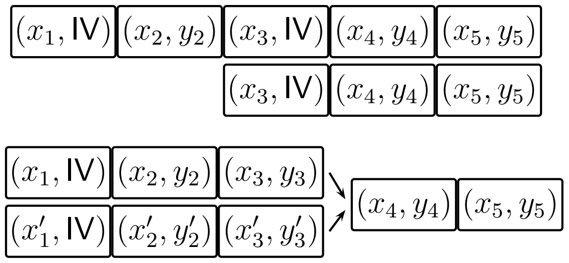

Figure 5.6: Two examples illustrating the necessity of event $\mathbf{bad}_1$

Event $\mathbf{bad}_1$ gives us two important invariants, namely

1. $\mathsf{Injective}(\boldsymbol{T}\langle 1\rangle)$ (see Definition 6.1 for the straightforward definition of this predicate)
2. $\mathsf{IV} \notin \boldsymbol{T}\langle 1\rangle$

These two conditions are the *well-formedness* conditions of table $\boldsymbol{T}\langle 1\rangle$, and they play an important role for many lemmas described in Chapter 6. Before we get to the more involved invariants, we need to derive a few more rather technical invariants which hold up to some failure event, or are related in some other way to the failure events.

3. $\mathsf{Injective\_fst}(\boldsymbol{I}\langle 2\rangle)$
4. $\forall i.\ (\mathsf{IV}, i) \notin \boldsymbol{R}\langle 2\rangle$
5. $\forall (x, y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle).\ y \in \boldsymbol{Y}\langle 1\rangle$
6. $\forall z \in \mathsf{ran}(\boldsymbol{T}\langle 1\rangle).\ z \in \boldsymbol{Z}\langle 1\rangle$
7. $\forall (z, i) \in \mathsf{ran}(\boldsymbol{R}\langle 2\rangle).\ z \in \boldsymbol{Z}\langle 1\rangle$
8. $\forall i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle).\ \mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i]) \in \boldsymbol{Z}\langle 1\rangle$
9. $\forall z \in \boldsymbol{Z}\langle 1\rangle.\ \exists i.\ i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge \mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i]) = z$
10. $\forall (x, y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle).\ (x, y) \in \mathsf{dom}(\boldsymbol{T'}\langle 1\rangle) \vee y = \mathsf{IV} \vee y \in \mathsf{ran}(\boldsymbol{T}\langle 1\rangle)$
11. $\forall x, y, i.\ (x, y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle) \wedge (x, y) \notin \mathsf{dom}(\boldsymbol{T'}\langle 1\rangle) \wedge i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle) \wedge$
    $\mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i]) = y \wedge (x, i) \notin \mathsf{dom}(\boldsymbol{T}\langle 2\rangle) \Rightarrow \mathsf{findseq}(x, y, \boldsymbol{T}\langle 1\rangle) \neq \mathsf{None}$
12. $\forall i \in \mathsf{dom}(\boldsymbol{I}\langle 2\rangle).\ \mathsf{set\_bad3}(\mathsf{fst}(\boldsymbol{I}\langle 2\rangle[i]), \boldsymbol{T'}\langle 1\rangle, \boldsymbol{T}\langle 1\rangle) \Leftrightarrow \mathsf{snd}(\boldsymbol{I}\langle 2\rangle[i])$

The predicate Injective_fst is defined as

$$\mathsf{Injective\_fst}(\boldsymbol{I}) := \forall i, j.\ i \in \mathsf{dom}(\boldsymbol{I}) \Rightarrow j \in \mathsf{dom}(\boldsymbol{I}) \Rightarrow \mathsf{fst}(\boldsymbol{I}[i]) = \mathsf{fst}(\boldsymbol{I}[j]) \Rightarrow i = j.$$

Now, we show in EasyCrypt an invariant of elementary importance. Intuitively, we want to say that the distinguisher can only make queries to f_q "in order". That is, it can only make queries associated with an MD-chain by first querying for the first chaining value, then for the second, and so on, since the intermediate chaining values are always unknown to it until it has made a query to f_q for the preceding element in the MD-chain. The simulator checks for this ordering in game $\mathsf{G}_{\mathsf{real'}}$ by using the predicate set_bad3 (see Definition 6.16); in game $\mathsf{G}_{\mathsf{realRO}}$, it uses the map $\boldsymbol{I}$ to iterate over all preceding chaining values. Furthermore the failure event $\mathbf{bad}_2$ in both games ensures that a randomly sampled value was not accidentally already used as a chaining value in a previous query from the distinguisher to f_q, since this may otherwise destroy the correct ordering of a possible chain (which the simulator might not notice through the use of event $\mathbf{bad}_3$).

Knowing this, we now turn our attention to the three loops in oracle F_q. Indeed, although oracle F_q eventually behaves as a simple relay for procedure F, it still samples random values eagerly as was the case in game $\mathsf{G}_{\mathsf{real'}}$. The first loop in F_q reconstructs chaining values that may already have been sampled by previous queries of the distinguisher to f_q and are therefore in the domain of $\boldsymbol{T'}$. Since the distinguisher can only make queries that belong to an MD-chain in order, these values can only occur in the prefix of such a chain. The second and third loops then take care of the rest of the MD-chain, i.e. the suffix not yet queried by the distinguisher, except for the very last element (which is always taken care of by a call to F at the end). The purpose of the second loop is to re-use chaining values that might already have been sampled by a previous call to F_q, e.g. because the padding of the input from a previous call and the padding of the current input have the same prefix, or simply because the same query was made more than once. Finally, the third loop samples fresh chaining values if needed. Lastly, observe that in the case that *found* is set to true in the procedure f_q of game $\mathsf{G}_{\mathsf{realRO}}$, the corresponding chaining value pre-sampled eagerly by the third loop in F_q is re-used by reading it from the map $\boldsymbol{I}$, and the *tainted* flag is updated in map $\boldsymbol{I}$ to reflect that the current value is no longer unknown to the distinguisher. Thus f_q and F_q use consistent intermediate chaining values, as is the case in game $\mathsf{G}_{\mathsf{real'}}$, where both procedures call oracle $f_{\mathbf{bad}}$ (resp. f).

The invariant we derive to express that the simulator can only make queries in order is

$$\mathsf{Claim5}(\boldsymbol{T'}\langle 1\rangle, \boldsymbol{T}\langle 1\rangle).$$

The predicate Claim5, named thus in reminiscence of the *Claim 5* from the proof of indifferentiability of the prefix-free Merkle-Damgård construction appearing in the extended version of Coron's paper [39], is defined as follows.

Definition 5.3 (Predicate Claim5). Let $\boldsymbol{T}$ and $\boldsymbol{T'}$ be two maps of type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. Then

$$\begin{aligned}\mathsf{Claim5}(\boldsymbol{T'}, \boldsymbol{T}) := \forall (x,y),(x',y').\ & (x',y') \in \mathsf{dom}(\boldsymbol{T}) \wedge (x,y) \in \mathsf{dom}(\boldsymbol{T'}) \wedge \\ & \boldsymbol{T}[(x',y')] = y \wedge \mathsf{findseq}(x',y',\boldsymbol{T}) = \mathsf{None} \Rightarrow (x',y') \in \mathsf{dom}(\boldsymbol{T'}).\end{aligned}$$

Intuitively, this means that whenever we have some element (x,y) in the domain of map $\boldsymbol{T'}$ that succeeds an element (x',y') of an MD-chain in $\boldsymbol{T}$, this preceding element (x',y') must also be in the domain of $\boldsymbol{T'}$, except if (x',y') is already the last element of a complete chain in $\boldsymbol{T}$ (in that case, y is known to the distinguisher). The latter restriction is expressed by the premise $\mathsf{findseq}(x,y,\boldsymbol{T}) = \mathsf{None}$. We refer to Section 6.1.6 for a more detailed discussion of this predicate.

The restriction that the distinguisher may only query chaining values in order has one essential implication. Namely, it means that any chain in the table $\boldsymbol{T'}$ maintained by the simulator will always be completed by the last element of that chain (both in the games $\mathsf{G}_{\mathsf{real'}}$ and $\mathsf{G}_{\mathsf{realRO}}$). Therefore, in the game $\mathsf{G}_{\mathsf{realRO}}$, upon a query $f_q(x,y)$ where (x,y) is the last element of an MD-chain, the simulator will always detect that this query corresponds to the last element of this MD-chain (by the completeness of the function findseq, see Axiom 6.14). Hence, for any query $F_q(m)$ that the distinguisher could make, the simulator will always be consistent with F_q's answer whenever the distinguisher tries to construct the MD-chain associated to m by repeatedly calling f_q. We formalize this in a final invariant:

$$\mathsf{RinT}(\boldsymbol{R}\langle 2\rangle, \boldsymbol{T}\langle 1\rangle),$$

and we define the predicate RinT as follows.

Definition 5.4 (Predicate RinT). Let $\boldsymbol{T} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ and $\boldsymbol{R} : \{0,1\}^* \to \{0,1\}^n \times \mathbb{N}$ be two maps. Then

$$\begin{aligned}
\mathsf{RinT}(\boldsymbol{R}, \boldsymbol{T}) := \big(&\forall m \in \mathsf{dom}(\boldsymbol{R}).\ \exists (x,y).\ \mathsf{findseq}(x,y,\boldsymbol{T}) \neq \mathsf{None} \wedge \\
&\qquad m = \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\boldsymbol{T})}) \| [x])} \wedge \\
&\qquad (x,y) \in \mathsf{dom}(\boldsymbol{T})\big) \wedge \\
\big(&\forall (x,y).\ \mathsf{findseq}(x,y,\boldsymbol{T}) \neq \mathsf{None} \Rightarrow \\
&\qquad \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\boldsymbol{T})}) \| [x])} \in \mathsf{dom}(\boldsymbol{R}) \Rightarrow \\
&\qquad (x,y) \in \mathsf{dom}(\boldsymbol{T})\big) \wedge \\
\big(&\forall (x,y).\ \mathsf{findseq}(x,y,\boldsymbol{T}) \neq \mathsf{None} \Rightarrow \\
&\qquad (x,y) \in \mathsf{dom}(\boldsymbol{T}) \Rightarrow \\
&\qquad \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\boldsymbol{T})}) \| [x])} \in \mathsf{dom}(\boldsymbol{R}) \wedge \\
&\qquad \mathsf{fst}(\boldsymbol{R}[\pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\boldsymbol{T})}) \| [x])}]) = \boldsymbol{T}[(x,y)]\big).
\end{aligned}$$

Intuitively, the invariant says that the map $\boldsymbol{R}\langle 2\rangle$ contains every message associated to every complete chain in map $\boldsymbol{T}\langle 1\rangle$, and nothing else. More precisely, for every message m in the domain of $\boldsymbol{R}\langle 2\rangle$, there is a corresponding complete chain of the form $c \parallel [(x,y)]$ in the map $\boldsymbol{T}\langle 1\rangle$ with $(x,y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle)$. Conversely, for every complete chain of the form $c \parallel [(x,y)]$ in the map $\boldsymbol{T}\langle 1\rangle$ with $(x,y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle)$, its associated message m is in the domain of $\boldsymbol{R}\langle 2\rangle$. Thus, in oracle f_q, in the case where $\mathsf{findseq}((x,y), \boldsymbol{T'}) \neq \mathsf{None}$, the transformation can be proven valid because $\mathsf{EasyCrypt}$ can infer from the above invariant that $(x,y) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle)$ in game $\mathsf{G}_{\mathsf{real'}}$ if and only if $m \in \mathsf{dom}(\boldsymbol{R}\langle 2\rangle)$ in game $\mathsf{G}_{\mathsf{realRO}}$.

Finally, we can show in $\mathsf{EasyCrypt}$ the following pRHL judgment:

$$\vdash \mathsf{G}_{\mathsf{real'}} \sim \mathsf{G}_{\mathsf{realRO}} : \mathsf{true} \Longrightarrow =\!\{\mathbf{bad}_1, \mathbf{bad}_2, \mathbf{bad}_3\} \wedge \neg(\mathbf{bad}_1 \vee \mathbf{bad}_2 \vee \mathbf{bad}_3)\langle 1\rangle \Rightarrow =\!\{b\}$$

We now apply Theorem 4.1 to conclude that the advantage of the distinguisher in differentiating between the games $\mathsf{G}_{\mathsf{real'}}$ and $\mathsf{G}_{\mathsf{realRO}}$ is upper bounded by the probability of any of the failure events occurring in game $\mathsf{G}_{\mathsf{realRO}}$, i.e.

$$|\Pr[\mathsf{G}_{\mathrm{real'}} : b] - \Pr[\mathsf{G}_{\mathrm{realRO}} : b]| \leq \Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_1 \vee \mathbf{bad}_2 \vee \mathbf{bad}_3]\,. \tag{5.3}$$

We are left to bound the probability of the failure events $\mathbf{bad}_1$, $\mathbf{bad}_2$ and $\mathbf{bad}_3$. Here, the bounds for $\mathbf{bad}_1$ and $\mathbf{bad}_2$ can be computed in the same fashion. Both events are triggered when a freshly sampled random value is already in a list of size at most q (list $\boldsymbol{Z}$ for event $\mathbf{bad}_1$, and list $\boldsymbol{Y}$ for event $\mathbf{bad}_2$). Therefore, the probability of either event is upper-bounded by $q/2^n$, for each of the possible q queries. To derive this in EasyCrypt, we first observe that the code which samples random values and raises the events $\mathbf{bad}_1$ and $\mathbf{bad}_2$ is the same in the oracles f_q, F_q and F of game $\mathsf{G}_{\mathsf{realRO}}$, namely

```
$z \xleftarrow{\$} \{0,1\}^n$;
$\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
$\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
```

In a meta-game $\hat{\mathsf{G}}_{\mathsf{realRO}}$, we outline this code fragment in all three procedures into a new procedure sample_z. We can then apply Lemma 4.2 in this game by taking $u = q/2^n$ and $i = |\boldsymbol{Z}| - 1$ (resp. $i = |\boldsymbol{Y}|$) to bound the probability of event $\mathbf{bad}_1$ (resp. $\mathbf{bad}_2$). We obtain

$$\Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_1] = \Pr\left[\hat{\mathsf{G}}_{\mathrm{realRO}} : \mathbf{bad}_1\right] \leq \frac{q^2}{2^n} \tag{5.4}$$

$$\Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_2] = \Pr\left[\hat{\mathsf{G}}_{\mathrm{realRO}} : \mathbf{bad}_2\right] \leq \frac{q^2}{2^n} \tag{5.5}$$

The bounding of event $\mathbf{bad}_3$ is more complex, since it relates to values that have already been sampled eagerly, and therefore the distribution of those values is not locally known. Hence, we leave the bounding of this event to the next sections, until after the step to a scenario where values are sampled lazily. Nevertheless, we already reveal here that the upper bound that we will ultimately obtain for event $\mathbf{bad}_3$ in game $\mathsf{G}_{\mathsf{realRO}}$ is equal to those for $\mathbf{bad}_1$ and $\mathbf{bad}_2$, namely $q^2/2^n$. Summing up over Equations 5.3, 5.4, 5.5 and 5.13, we finally get:

$$|\Pr[\mathsf{G}_{\mathrm{real}'} : b] - \Pr[\mathsf{G}_{\mathrm{realRO}} : b]| \leq \frac{3q^2}{2^n} \tag{5.6}$$

5.3 From $\mathsf{G}_{\mathsf{realRO}}$ to $\mathsf{G}_{\mathsf{idealEager}}$

The transformation considered in this section is an easy one, since it simply serves as a preparation for the next transformation, which will be concerned with the step from an eager sampling scenario to a lazy sampling scenario. In the game $\mathsf{G}_{\mathsf{idealEager}}$, defined in Figure 5.7, the following changes are introduced:

1. The main body of the game $\mathsf{G}_{\mathsf{realRO}}$ is instrumented with a loop which re-samples chaining values that are unknown to the adversary;

2. the failure events $\mathbf{bad}_1$, $\mathbf{bad}_2$ and $\mathbf{bad}_3$ are removed, and the resulting code is simplified;

3. the oracle $F(m, y)$ is transformed into an oracle $F(m)$;

4. a new event $\mathbf{bad}_4$ is introduced that will be used to bound the probability of event $\mathbf{bad}_3$ in game $\mathsf{G}_{\mathsf{realRO}}$.

```
$\boxed{\textbf{Game}\ \ \mathsf{G}_{\mathrm{idealEager}}:}$
$\textbf{Game}\ \ \mathsf{G}_{\mathrm{idealLazy}}:$
$\mathbf{q}_f \leftarrow 0;$
$\mathbf{q}'_f \leftarrow 1;$
$\boldsymbol{T}, \boldsymbol{T'}, \boldsymbol{T'_i}, \boldsymbol{R}, \boldsymbol{I} \leftarrow \emptyset;$
$\boldsymbol{I}[0] \leftarrow (\mathsf{IV}, \mathsf{false});$
$\boldsymbol{Y'} \leftarrow \mathsf{nil};$
$\mathbf{bad}_4 \leftarrow \mathsf{false};$
$l \leftarrow 0;$
  $l \leftarrow 0;$
  while $l < \mathbf{q}'_f$ do
    if $\mathsf{snd}(\boldsymbol{I}[l])$ then
      $z \xleftarrow{\$} \{0,1\}^n;$
      $\boldsymbol{I}[l] \leftarrow (z, \mathsf{true});$
    $l \leftarrow l + 1;$
$b \leftarrow \mathcal{D}^{F_q, f_q}();$
  $l \leftarrow 0;$
  while $l < \mathbf{q}'_f$ do
    if $\mathsf{snd}(\boldsymbol{I}[l])$ then
      $z \xleftarrow{\$} \{0,1\}^n;$
      $\boldsymbol{I}[l] \leftarrow (z, \mathsf{true});$
    $l \leftarrow l + 1;$
return $b$
```

```
$\textbf{Oracle}\ F_q(m):$
$xs \leftarrow \mathsf{pad}(m);\ y \leftarrow \mathsf{IV};$
$i \leftarrow 0;$
if $(0 < \mathbf{q}'_f \wedge$
  $\mathbf{q}_f + |xs| \leq q)$ then
  $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|;$
  while $|xs| > 1 \wedge$
      $(\mathsf{hd}(xs), y) \in \mathsf{dom}(\boldsymbol{T'})$ do
      $i \leftarrow \boldsymbol{T'_i}[\mathsf{hd}(xs), y];$
      $y \leftarrow \boldsymbol{T'}[\mathsf{hd}(xs), y];$
      $xs \leftarrow \mathsf{tl}(xs);$
  while $|xs| > 1 \wedge$
      $(\mathsf{hd}(xs), i) \in \mathsf{dom}(\boldsymbol{T})$ do
      $i \leftarrow \boldsymbol{T}[\mathsf{hd}(xs), i];$
      $xs \leftarrow \mathsf{tl}(xs);$
  while $|xs| > 1$ do
      $z \xleftarrow{\$} \{0,1\}^n;$
      $\boldsymbol{T}[\mathsf{hd}(xs), i] \leftarrow \mathbf{q}'_f;$
      $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{true});$
      $i \leftarrow \mathbf{q}'_f;$
      $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1;$
      $xs \leftarrow \mathsf{tl}(xs);$
  $y \leftarrow \mathsf{fst}(F(m));$
return $y$
```

```
$\textbf{Oracle}\ F(m):$
if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
  $z \xleftarrow{\$} \{0,1\}^n;$
  $\boldsymbol{R}[m] \leftarrow (z, \mathbf{q}'_f)$
  $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$
  $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1;$
return $\boldsymbol{R}[m]$
```

```
$\textbf{Oracle}\ f_q(x,y):$
if $\mathbf{q}_f + 1 \leq q$ then
  if $(0 < \mathbf{q}'_f \wedge$
      $(x,y) \notin \mathsf{dom}(\boldsymbol{T'}))$ then
      $xs \leftarrow \mathsf{findseq}((x,y), \boldsymbol{T'})$
      if $xs \neq \mathsf{None}$ then
        $m \leftarrow \pi_{\mathsf{unpad}(\mathsf{mapfst}(\pi_{xs}) \| [x])};$
        $(z,j) \leftarrow F(m);$
        $\boldsymbol{T'}[x,y] \leftarrow z;\ \boldsymbol{T'_i}[x,y] \leftarrow j;$
      else
        $found \leftarrow \mathsf{false};\ j, k' \leftarrow 0;$
        while $(k' < \mathbf{q}'_f \wedge \neg found)$ do
          if $(\boldsymbol{I}[k'] = (y, \mathsf{false}) \wedge$
            $(x, k') \in \mathsf{dom}(\boldsymbol{T}) \wedge$
            $\mathsf{snd}(\boldsymbol{I}[\boldsymbol{T}[x,k']]) \wedge$
            $k' < \boldsymbol{T}[x,k'] \wedge$
            $\boldsymbol{T}[x,k'] < \mathbf{q}'_f)$ then
            $found \leftarrow \mathsf{true};\ j \leftarrow \boldsymbol{T}[x,k'];$
          else
            $k' \leftarrow k' + 1;$
        if $found$ then
          $\boxed{z \leftarrow \mathsf{fst}(\boldsymbol{I}[j]);}$    $z \xleftarrow{\$} \{0,1\}^n;$
          $\mathbf{bad}_4 \leftarrow \mathbf{bad}_4 \vee z \in \boldsymbol{Y'};$
          $\boldsymbol{I}[j] \leftarrow (z, \mathsf{false});$
          $\boldsymbol{T'}[x,y] \leftarrow z;\ \boldsymbol{T'_i}[x,y] \leftarrow j;$
        else
          $z \xleftarrow{\$} \{0,1\}^n;$
          $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false});$
          $\boldsymbol{T'}[x,y] \leftarrow z;$
          $\boldsymbol{T'_i}[x,y] \leftarrow \mathbf{q}'_f;$
          $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1;$
      $\boldsymbol{Y'} \leftarrow y{::}\boldsymbol{Y'};$
  $z \leftarrow \boldsymbol{T'}[x,y];\ \mathbf{q}_f \leftarrow \mathbf{q}_f + 1;$
else
  $z \leftarrow \mathsf{IV};$
return $z$
```

Figure 5.7: The games $\mathsf{G}_{\mathrm{idealEager}}$ and $\mathsf{G}_{\mathrm{idealLazy}}$

The loop which re-samples chaining values in the main body of game $\mathsf{G}_{\mathsf{idealEager}}$ will be discussed in more depth in the next section, since it is important for the transition from eager to lazy sampling. Here, simply observe that the loop has no effect in this game, except that it increments the value of l (unknown to the distinguisher) once: at the point where the loop is executed, we have $\mathbf{q}'_f = 1$ and $\mathsf{snd}(\boldsymbol{I}[0]) = \mathsf{false}$. Therefore there is only one iteration and the guard of the if-statement is not fulfilled.

Next, we remove events $\mathbf{bad}_{\{1,2,3\}}$ and their related lists $\boldsymbol{Y}$ and $\boldsymbol{Z}$. Because of this, we can also merge the two branches under the "if *found_bad3* ..."-statement of the simulator f_q in game $\mathsf{G}_{\mathsf{realRO}}$. Additionally, we transform the oracle $F(m,y)$ into an oracle $F(m)$. Indeed, the only reason why we needed to pass y as an argument to F in the previous game was so that we could appropriately set event $\mathbf{bad}_2$ – by removing this event we can now get rid of the superfluous parameter y. Lastly, because F does not need this parameter any longer, we do not need to keep track of the intermediate chaining values y in the second and third loops of oracle F_q; it becomes even more clear now that F_q behaves as a relay and that the three loops have no effect on its return value.

Similarly to the transformation described in Section 5.1, we observe that the removal or introduction of failure events (unknown to the distinguisher) does not otherwise influence the behavior of either the procedures f_q or F_q.

Finally, because we still have to bound the probability of event $\mathbf{bad}_3$ in game $\mathsf{G}_{\mathsf{realRO}}$, we introduce a new event $\mathbf{bad}_4$ along with an associated list $\boldsymbol{Y'}$ (named so to distinguish it from the list $\boldsymbol{Y}$). This event plays a similar role to event $\mathbf{bad}_3$ used in the previous game, albeit not the same. It ensures that a random value pre-sampled by procedure F_q and re-used by the distinguisher has not already been used as a chaining value in a previous query to f_q; event $\mathbf{bad}_3$ did the same in game $\mathsf{G}_{\mathsf{realRO}}$, but for random values sampled by the simulator itself, not for those pre-sampled by F_q. Therefore, event $\mathbf{bad}_4$ on its own is not enough to bound the probability of event $\mathbf{bad}_3$. We need it because we want to bound the probability of $\mathbf{bad}_3$ by the probability of a more general event in game $\mathsf{G}_{\mathsf{idealEager}}$: We show that if $\mathbf{bad}_3$ occurs in game $\mathsf{G}_{\mathsf{realRO}}$, then either $\mathbf{bad}_4$ is is triggered in game $\mathsf{G}_{\mathsf{idealEager}}$, or there exists a positive value $i \leq \mathbf{q}'_f$ such that $\boldsymbol{I}\langle 2\rangle[i] = (y, \mathsf{true})$ and $y \in \boldsymbol{Y}\langle 2\rangle$. To achieve this, we first define the following predicate.

Definition 5.5 (Predicate $\mathrm{I}_\exists$)**.** Let $\mathbf{q}'_f \in \mathbb{N}$ be an integer, $\boldsymbol{I} : \mathbb{N} \to \{0,1\}^n \times \mathbb{B}$ a map and $\boldsymbol{Y'} : \{0,1\}^n$ list a list of chaining values. Then

$$\mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'}) := \exists i.\ 0 \leq i \leq \mathbf{q}'_f \wedge \mathsf{snd}(\boldsymbol{I}[i]) \wedge \mathsf{fst}(\boldsymbol{I}[i]) \in \boldsymbol{Y'}.$$

Intuitively, this means that there is a chaining value appearing in map $\boldsymbol{I}$ which should be unknown to the distinguisher, but the distinguisher used it as part of a query to f_q – this should be impossible. We now show in EasyCrypt:

$$\vdash \mathsf{G}_{\mathsf{realRO}} \sim \mathsf{G}_{\mathsf{idealEager}} : \mathsf{true} \Longrightarrow =\!\{b\} \wedge (\mathbf{bad}_3\langle 1\rangle \Rightarrow (\mathbf{bad}_4\langle 2\rangle \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})\langle 2\rangle))$$

This yields the following (in-)equations:

$$\Pr\left[\mathsf{G}_{\mathrm{realRO}} : b\right] = \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : b\right] \tag{5.7}$$

$$\Pr\left[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_3\right] \leq \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})\right] \tag{5.8}$$

To derive the above pRHL judgment, for technical reasons we need to show a few generic invariants that hold in game $\mathsf{G}_{\mathsf{realRO}}$. We list them here for the sake of completeness.

1. $0 < \mathbf{q}'_f\langle 1\rangle$
2. $\forall m \in \mathsf{dom}(\boldsymbol{R}\langle 1\rangle).\ \mathsf{snd}(\boldsymbol{R}\langle 1\rangle[m]) < \mathbf{q}'_f$
3. $\forall (x,y) \in \mathsf{dom}(\boldsymbol{T'}\langle 1\rangle).\ (x,y) \in \mathsf{dom}(\boldsymbol{T'_i}\langle 1\rangle)$
4. $\forall (x,y) \in \mathsf{dom}(\boldsymbol{T'_i}\langle 1\rangle).\ \boldsymbol{T'_i}\langle 1\rangle[x,y] < \mathbf{q}'_f$
5. $\forall (x,i) \in \mathsf{dom}(\boldsymbol{T}\langle 1\rangle).\ i < \boldsymbol{T}\langle 1\rangle[x,i] < \mathbf{q}'_f$

We postpone the computation of the probability of $\mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ in game $\mathsf{G}_{\mathsf{idealEager}}$ to the next section, since again this will be easier when the values are sampled lazily and their distribution is locally known.

5.4 From $\mathsf{G}_{\mathsf{idealEager}}$ to $\mathsf{G}_{\mathsf{idealLazy}}$

In this step, we will make the transition from a simulator which re-uses eagerly sampled random values to a simulator which lazily samples its own random values upon fresh queries. In the new game $\mathsf{G}_{\mathsf{idealLazy}}$, defined in Figure 5.7, we introduce the following two changes:

1. The simulator f_q now samples a new random value in the case where *found* is true, instead of re-using the corresponding value in map $\boldsymbol{I}$;

2. the loop in the main body introduced in the last section is swapped with the call to the distinguisher $\mathcal{D}$.

We have investigated the difference between eager and lazy sampling in Section 4.5.2. At this point, we highlight that the initial game $\mathsf{G}_{\mathsf{real}}$ samples values eagerly, while the final game $\mathsf{G}_{\mathsf{ideal}}$ samples values lazily. To clarify this, let m be a message with $\mathsf{pad}(m) = [x_1, \ldots, x_n]$, and assume for simplicity that the distinguisher has made no calls to oracles f_q or F_q yet in the two games. Now, observe the following fundamental difference: If the distinguisher makes the query $F_q(m)$ in game $\mathsf{G}_{\mathsf{real}}$, the FIL-RO f is called repeatedly before an answer is returned, and therefore n random values are sampled. If the distinguisher makes the query $F_q(m)$ in game $\mathsf{G}_{\mathsf{ideal}}$, then F_q calls the VIL-RO F once, and therefore only a single random value is sampled. Now, if the distinguisher queries $f_q(x_1, \mathsf{IV})$ in game $\mathsf{G}_{\mathsf{real}}$, oracle f_q will query f and therefore the corresponding random value which was pre-sampled by the call $F_q(m)$ will be used. In game $\mathsf{G}_{\mathsf{ideal}}$, the simulator f_q will sample a new random value instead. That is, in game $\mathsf{G}_{\mathsf{real}}$, some random values may be sampled *earlier* than in game $\mathsf{G}_{\mathsf{ideal}}$. Additionally, some values which were randomly sampled in game $\mathsf{G}_{\mathsf{real}}$ may not be sampled at all in game $\mathsf{G}_{\mathsf{ideal}}$ – the distinguisher needs not reconstruct the entire MD-chain associated to m by queries to f_q. Notwithstanding, the delaying of this sampling of random values does not make any difference from the distinguisher's perspective; the cryptographical argument is that these values are *uniformly distributed and independent of the adversary's view.* Indeed, the intermediate chaining values generated in a Merkle-Damgård iteration remain unknown to the distinguisher until it queries the oracle f_q for them, at which point they are sampled uniformly at random if needed. Therefore we see that a sequence of games from $\mathsf{G}_{\mathsf{real}}$ to $\mathsf{G}_{\mathsf{ideal}}$ must involve a transition of lazy/eager sampling, as described in Section 4.5.2. The purpose of the present transformation is to tackle this particular issue.

The main difference between the games $\mathsf{G}_{\mathsf{idealEager}}$ and $\mathsf{G}_{\mathsf{idealLazy}}$ considered here is that the oracle f_q in game $\mathsf{G}_{\mathsf{idealEager}}$ re-uses values that were eagerly pre-sampled by the oracle F_q, while the oracle f_q in game $\mathsf{G}_{\mathsf{idealLazy}}$ re-samples those random values. Intuitively, this is sound because these eagerly sampled values have not been revealed to the distinguisher; we know this because at that point it holds that $\mathsf{snd}(\boldsymbol{I}[j]) = \mathsf{true}$ (the second component of the pairs in map $\boldsymbol{I}$ keeps track of whether a value is unknown to the distinguisher). Now, we would like to show that the re-sampling of these eagerly sampled values, unknown to the distinguisher, preserves the semantics of the program.

However, the values pre-sampled by F_q are not locally known in the context of the procedure f_q, such that it is difficult if not impossible to argue about them in the local context of procedure f_q. This is where the loop of the main body (depicted in the inner boxes of the two games in Figure 5.7) comes into play. This loop eagerly samples all random values for $\boldsymbol{I}$ in the game $\mathsf{G}_{\mathsf{idealEager}}$ (i.e., it is executed *before* the call to the distinguisher). In game $\mathsf{G}_{\mathsf{idealLazy}}$, this loop is swapped with the call to the distinguisher (i.e., it is executed *after* the call to the distinguisher). In game $\mathsf{G}_{\mathsf{idealLazy}}$, the purpose of the loop is to sample all random values that have not been lazily sampled during the execution of the distinguisher. For readability, let loop denote the code of the loop. Focusing now on the pieces of code $\mathsf{loop}; \mathsf{G}_{\mathsf{idealEager}}.F_q(m)$ and $\mathsf{G}_{\mathsf{idealLazy}}.F_q(m); \mathsf{loop}$ (respectively $\mathsf{loop}; \mathsf{G}_{\mathsf{idealEager}}.f_q(x, y)$ and $\mathsf{G}_{\mathsf{idealLazy}}.f_q(x, y); \mathsf{loop}$), we can address all randomly sampled variables locally, and by showing that this loop swaps with the calls to the eager and lazy oracles in a semantic-preserving way, we prove that the re-sampled values are independent of the view of an adversary with access to these oracles. This is the main trick in a transition of lazy/eager sampling; referring to the discussion in Section 4.5.2, the condition used here is the condition $\mathsf{snd}(\boldsymbol{I}[j]) = \mathsf{false}$ for any value j in the domain of $\boldsymbol{I}$ (meaning that the value $\mathsf{fst}(\boldsymbol{I}[j])$ has been revealed to the distinguisher), and the statement $S_{\hat{\boldsymbol{y}}}$ which re-samples values unknown to the distinguisher is loop.

Concretely, we show

$$\vdash \begin{array}{l} l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \\ r \leftarrow \mathsf{G}_{\mathsf{idealEager}}.F_q(m); \end{array} \sim \begin{array}{l} r \leftarrow \mathsf{G}_{\mathsf{idealLazy}}.F_q(m); \\ l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \end{array} \ :\ =\!\{m, l\} \wedge =\!\Phi \Longrightarrow =\!\{r, l\} \wedge =\!\Phi$$

and

$$\vdash \begin{array}{l} l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \\ r \leftarrow \mathsf{G}_{\mathsf{idealEager}}.f_q(x, y); \end{array} \sim \begin{array}{l} r \leftarrow \mathsf{G}_{\mathsf{idealLazy}}.f_q(x, y); \\ l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \end{array} \ :\ =\!\{x, y, l\} \wedge =\!\Phi \Longrightarrow =\!\{r, l\} \wedge =\!\Phi,$$

where $\Phi := \left\{\mathbf{q}_f, \mathbf{q}'_f, \boldsymbol{T}, \boldsymbol{T'}, \boldsymbol{T'_i}, \boldsymbol{R}, \boldsymbol{I}, \boldsymbol{Y'}, \mathbf{bad}_4\right\}$ is the set of all global variables. From this, we immediately conclude

$$\vdash \begin{array}{l} l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \\ b \leftarrow D^{\mathsf{G}_{\mathsf{idealEager}}.f_q, \mathsf{G}_{\mathsf{idealEager}}.F_q}(); \end{array} \sim \begin{array}{l} b \leftarrow D^{\mathsf{G}_{\mathsf{idealLazy}}.f_q, \mathsf{G}_{\mathsf{idealLazy}}.F_q}(); \\ l \leftarrow 0; \\ \mathsf{while}\ l < \mathbf{q}'_f\ \mathsf{do} \\ \quad \mathsf{if}\ \mathsf{snd}(\boldsymbol{I}[l])\ \mathsf{then} \\ \quad\quad z \xleftarrow{\$} \{0,1\}^n; \\ \quad\quad \boldsymbol{I}[l] \leftarrow (z, \mathsf{true}); \\ \quad l \leftarrow l + 1; \end{array} \ :\ =\!\{l\} \wedge =\!\Phi \Longrightarrow =\!\{b, l\} \wedge =\!\Phi.$$

An additional instruction $l \leftarrow 0$ situated before the loop was introduced along with said loop in $\mathsf{G}_{\mathsf{idealEager}}$ (see Figure 5.7). This is needed to satisfy the pre-condition $=\{l\}$ in the latter pRHL judgment, since we want to use this judgment to conclude equivalence of the two entire games. Indeed, using this it is now a trivial task for EasyCrypt to derive

$$\vdash \mathsf{G}_{\mathsf{idealEager}} \sim \mathsf{G}_{\mathsf{idealLazy}} : \mathsf{true} \Longrightarrow =\{b, \mathbf{bad}_4, \boldsymbol{I}, \mathbf{q}'_f, \boldsymbol{Y'}\}.$$

This implies that the games $\mathsf{G}_{\mathsf{idealEager}}$ and $\mathsf{G}_{\mathsf{idealLazy}}$ have the same probabilities of outcome, and the same probability of event $\mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$:

$$\Pr\left[\mathsf{G}_{\text{idealEager}} : b\right] = \Pr\left[\mathsf{G}_{\text{idealLazy}} : b\right] \tag{5.9}$$

$$\Pr\left[\mathsf{G}_{\text{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})\right] = \Pr\left[\mathsf{G}_{\text{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})\right] \tag{5.10}$$

The hardest part here is the proof of the above judgment for oracle f_q (i.e., the second judgment) in the case where *found* $=$ true, since here $\mathsf{G}_{\mathsf{idealEager}}.f_q$ uses the chaining value $\boldsymbol{I}[j]$, whereas $\mathsf{G}_{\mathsf{idealLazy}}.f_q$ re-samples a fresh random value. In this case, observe the following. In the lazy program (the one on the right-hand side), since we have *found* $=$ true, we know (from the while loop which was responsible for setting *found* $=$ true) that $j \in \mathsf{dom}(\boldsymbol{I})$ and $\mathsf{snd}(\boldsymbol{I}[j]) = \mathsf{true}$. Let us fix the value j at this point in the lazy program as $\hat{\jmath}$. Now, since $\boldsymbol{I}$ was not modified in this program before, we must already have had $\hat{\jmath} \in \mathsf{dom}(\boldsymbol{I})$ and $\mathsf{snd}(\boldsymbol{I}[\hat{\jmath}]) = \mathsf{true}$ at the start of the lazy program. In the pRHL judgment, we have as a pre-condition that all global variables are equal at the start of the two programs, in particular $\boldsymbol{I}$. So, we must also have $\hat{\jmath} \in \mathsf{dom}(\boldsymbol{I})$ and $\mathsf{snd}(\boldsymbol{I}[\hat{\jmath}]) = \mathsf{true}$ at the start of the eager program (the one on the left-hand side). Furthermore, we know – using an analogous argument – that $\hat{\jmath} < \mathbf{q}'_f$ at the start of the eager program. Thus, the value $\boldsymbol{I}[\hat{\jmath}]$ *will* be sampled at the start of the eager program by the loop which samples random values. We can "cancel out" this random sampling against the fresh random sampling performed in procedure f_q of the lazy program (the other random samplings in the loop at the start of the eager program can be canceled out against the random samplings in the loop at the end of the lazy program). Technically, we *derandomize* the two random samplings of $\boldsymbol{I}[\hat{\jmath}]$ on both sides, that is, we assume that $\hat{z}$ is a uniformly distributed random value and we replace these two samplings on both sides by deterministic assignments of the value $\hat{z}$. After the execution of both programs, we need to show that $\mathsf{upd}(\mathsf{upd}(\boldsymbol{I}\langle 1\rangle, \hat{\jmath}, (\hat{z}, \mathsf{true})), \hat{\jmath}, (\hat{z}, \mathsf{false})) = \mathsf{upd}(\boldsymbol{I}\langle 2\rangle, \hat{\jmath}, (\hat{z}, \mathsf{false}))$, as the value $\boldsymbol{I}[\hat{\jmath}]$ has been updated twice in the eager game, but only once in the lazy game. Note that we have $\boldsymbol{I}\langle 1\rangle = \boldsymbol{I}\langle 2\rangle$, so the only thing we need to do to finish this argument is to formalize an extensionality axiom:

Axiom 5.6 (Extensionality of $\boldsymbol{I}$)**.** *Let* $\boldsymbol{I}, \boldsymbol{I'} : \mathbb{N} \to \{0,1\}^n \times \mathbb{B}$ *be two maps. Then*

$$\boldsymbol{I} = \boldsymbol{I'} \Leftrightarrow \big((\forall i.\ i \in \mathsf{dom}(\boldsymbol{I}) \Leftrightarrow i \in \mathsf{dom}(\boldsymbol{I'})) \wedge \\ (\forall i \in \mathsf{dom}(\boldsymbol{I}).\ \boldsymbol{I}[i] = \boldsymbol{I'}[i])\big).$$

Lastly, we want to bound the probability of event $\mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ in game $\mathsf{G}_{\mathsf{idealLazy}}$. We define a meta-game $\mathsf{G}'_{\mathsf{idealLazy}}$, whose only modification is that the loop which re-samples values in the main body of the game now also sets an event $\mathbf{bad}_4$. More precisely, we replace the instruction $z \xleftarrow{\$} \{0,1\}^n$ in the loop of the main body with

$$z \xleftarrow{\$} \{0,1\}^n;\\ \mathbf{bad}_4 \leftarrow \mathbf{bad}_4 \vee z \in \boldsymbol{Y'}$$

Because of this modification, we can say the following: If $\mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ holds at the end of game $\mathsf{G}_{\mathsf{idealLazy}}$, then clearly $\mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ also holds at the end of $\mathsf{G}'_{\mathsf{idealLazy}}$. If $\mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ holds at the end of game $\mathsf{G}'_{\mathsf{idealLazy}}$, then necessarily $\mathbf{bad}_4$ must have been set to true in this game, either by the simulator or later by the loop in the main body. Therefore, the event $\mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})$ in game $\mathsf{G}_{\mathsf{idealLazy}}$ implies $\mathbf{bad}_4 = \mathsf{true}$ in game $\mathsf{G}'_{\mathsf{idealLazy}}$. Further, clearly the event $\mathbf{bad}_4 = \mathsf{true}$ in game $\mathsf{G}_{\mathsf{idealLazy}}$ also implies $\mathbf{bad}_4 = \mathsf{true}$ in game $\mathsf{G}'_{\mathsf{idealLazy}}$. Thus we can easily derive in EasyCrypt that

$$\Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists(\mathbf{q}'_f, \boldsymbol{I}, \boldsymbol{Y'})\right] \leq \Pr\left[\mathsf{G}'_{\mathrm{idealLazy}} : \mathbf{bad}_4\right]. \tag{5.11}$$

Finally, we use the same technique to bound the probability of event $\mathbf{bad}_4$ in game $\mathsf{G}'_{\mathrm{idealLazy}}$ that we also used to bound the probability of events $\mathbf{bad}_1$ and $\mathbf{bad}_2$ in game $\mathsf{G}_{\mathsf{realRO}}$. We outline the code fragment $z \xleftarrow{\$} \{0,1\}^n; \mathbf{bad}_4 \leftarrow \mathbf{bad}_4 \vee z \in \boldsymbol{Y'}$ in both the procedures f_q and the main body of $\mathsf{G}'_{\mathrm{idealLazy}}$ into an oracle *sample*$_z$ to obtain a new meta-game $\hat{\mathsf{G}}'_{\mathrm{idealLazy}}$. We apply Lemma 4.2 in this game by taking $u = q/2^n$ and $i = |\boldsymbol{Y'}|$, and obtain:

$$\Pr\left[\mathsf{G}'_{\mathrm{idealLazy}} : \mathbf{bad}_4\right] = \Pr\left[\hat{\mathsf{G}}'_{\mathrm{idealLazy}} : \mathbf{bad}_4\right] \leq \frac{q^2}{2^n} \tag{5.12}$$

Pulling Equations 5.8, 5.10, 5.11 and 5.12 together, we finally obtain the bound for the probability of event $\mathbf{bad}_3$ in the game $\mathsf{G}_{\mathsf{realRO}}$:

$$\Pr\left[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_3\right] \leq \Pr\left[\hat{\mathsf{G}}'_{\mathrm{idealLazy}} : \mathbf{bad}_4\right] \leq \frac{q^2}{2^n} \tag{5.13}$$

5.5 From $\mathsf{G}_{\mathsf{idealLazy}}$ to $\mathsf{G}_{\mathsf{ideal}}$

Finally, we are only left to show that the probabilities of the outcomes of the game $\mathsf{G}_{\mathsf{idealLazy}}$ and of the final game $\mathsf{G}_{\mathsf{ideal}}$ (see Figure 5.1) are equal. This only amounts to some rather simple code cleanup:

1. The loop which re-samples values in the main body of $\mathsf{G}_{\mathsf{idealLazy}}$ is removed;
2. the three loops in procedure F_q are also removed;
3. in the simulator, the loop which looks for values pre-sampled in map $\boldsymbol{I}$ and may set *found* to true is removed;
4. the global variables $\mathbf{bad}_4$, $\boldsymbol{Y'}$, $\mathbf{q}'_f$, $\boldsymbol{I}$, $\boldsymbol{T'_i}$, and $\boldsymbol{T}$ and all associated instructions are removed;

5. the two branches under the conditional statement "$\mathsf{if}\, \mathit{found} \ldots$" in the simulator are merged.

The modification which we introduced in the last section means that the simulator is no longer dependent of the pre-sampled values in map $\boldsymbol{I}$. Therefore, the simulator does not need to look for these pre-sampled values any longer, and we can remove the loop with the assignment $\mathit{found} \leftarrow \mathsf{true}$. Further, it is now easy to remove all loops in the main body and in procedure F_q. Indeed, the variables modified in these loops are disregarded by the procedures F_q and f_q; F_q always behaves as a relay for F, while the simulator f_q always forwards calls to F in the case where $\mathsf{findseq}((x, y), \boldsymbol{T'}) \neq \mathsf{None}$ and samples a uniformly distributed random value otherwise.

Next, we remove all global variables which are of no further use. Specifically, we remove the global variables $\mathbf{bad}_4$, $\boldsymbol{Y'}$, $\mathbf{q}'_f$, $\boldsymbol{I}$, $\boldsymbol{T'_i}$, and $\boldsymbol{T}$. Furthermore we remove all associated assignments from or to these variables. Since we removed $\mathbf{q}'_f$, we modify the types of procedure $F : \{0,1\}^* \to \{0,1\}^n \times \mathbb{N}$ and of map $\boldsymbol{R} : \{0,1\}^* \to \{0,1\}^n \times \mathbb{N}$ to simply $F : \{0,1\}^* \to \{0,1\}^n$ and $\boldsymbol{R} : \{0,1\}^* \to \{0,1\}^n$, and adapt the associated code accordingly. Finally, the code which remains under the two branches of the statement "$\mathsf{if}\, \mathit{found} \ldots$" in procedure f_q is only "$z \xleftarrow{\$} \{0,1\}^n$; $\boldsymbol{T'}[x, y] \leftarrow z$", and we merge these branches.

The resulting game is the game $\mathsf{G}_{\mathsf{ideal}}$. Because none of the above modifications influence the behavior of the oracles accessible to the distinguisher, we can show in EasyCrypt that

$$\vdash \mathsf{G}_{\mathsf{idealLazy}} \sim \mathsf{G}_{\mathsf{ideal}} : \mathsf{true} \Longrightarrow =\!\{b\}$$

and hence

$$\Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : b\right] = \Pr\left[\mathsf{G}_{\mathrm{ideal}} : b\right]. \tag{5.14}$$

Here we need the same generic invariants as those used to prove the equivalence in Section 5.3. Finally, using Equations 5.2, 5.6, 5.7, 5.9 and 5.14 we obtain Equation 5.1, which finishes the proof of Theorem 5.1.

6 The manual part

In this chapter, we will consider the axiomatization of the functions used in the games described in Chapter 5, and build increasingly complex lemmas about their properties. Indeed, to enable the SMT solvers and automated theorem provers used by EasyCrypt to successfully verify proof obligations arising within the proof, EasyCrypt needs to feed these automated tools with appropriate characteristics of those functions. As an obvious example, in the step from $\mathsf{G}_{\mathsf{real}'}$ to $\mathsf{G}_{\mathsf{realRO}}$, where the case distinction on $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$ plays a central role, it is often necessary to provide the automated tools with the knowledge that the function findseq is both sound and complete (we will formalize this in two axioms); in practice, the lemmas needed to prove the validity of logical side conditions can become much more complex than that.

As discussed in Section 4.1, those axioms and lemmas are stated in EasyCrypt before the proof of some pRHL judgment is performed; during the proof of this judgment, they can then be passed to the automated tools as needed. We normally start by postulating some basic axioms on which a proof relies; from there, we state more involved lemmas. EasyCrypt is often able to derive the validity of some easier lemmas automatically from those axioms and/or from prior lemmas. However, as the lemmas get more involved, this does not typically work; even the validity of some easier statements cannot normally be derived if their proof involves so much as a simple induction (inductive proofs seem to be hard for automated tools). In these cases, the concerned lemmas can be exported to the Coq proof assistant, and proven there. Then, they can simply be stated as axioms in EasyCrypt, as they are proven elsewhere.

In this chapter, we will describe the axioms that we needed to state for our proof of indifferentiability, and the lemmas that we derived from them. In many cases, these lemmas had to be proven manually in the Coq proof assistant: The corresponding Coq file contained over 4200 lines of code, and is available upon request, or from:

`http://easycrypt.gforge.inria.fr/csf12/`

Here, we additionally describe all of those proofs in a human-readable manner.

6.1 Predicates, Definitions, and Axioms

In this section, we outline the basic definitions and axioms that we will build on later in the chapter.

6.1.1 Maps

Since many lemmas in this chapter concern the maps maintained in the games described in Chapter 5, we introduce here a few notions to deal with them more easily. The type of maps that we are interested in these lemmas always have the type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, i.e. maps that map pairs of a block and a chaining value to a chaining value; these maps are used to keep track of the previous answers of oracle f_q (the simulator) in the games.

In the games $\mathsf{G}_{\mathsf{real}'}$ and $\mathsf{G}_{\mathsf{realRO}}$, whose equivalence is discussed in Section 5.2, we use several failure events that are raised whenever such a map may be updated with a value that exhibits certain dependencies to values already contained in this map. Therefore, the maps in these games possess certain properties (up to the failure events) which are essential for many lemmas.

Two of these properties are particularly important, and are both guaranteed by the failure event $\mathbf{bad}_1$. The first is that these maps are injective. We define a straight-forward predicate in EasyCrypt to express this as follows.

Definition 6.1 (Predicate Injective)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map. The predicate Injective is defined as

$$\mathsf{Injective}(T) := \forall xy, xy'.\ xy \in \mathsf{dom}(T) \Rightarrow xy' \in \mathsf{dom}(T) \Rightarrow T[xy] = T[xy'] \Rightarrow xy = xy'.$$

The second property is that the constant IV does not appear in the range of a map. Together, we call these properties the *well-formedness* conditions of a map T.

Definition 6.2 (Well-formedness)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map. T is called *well-formed* iff it holds that $\mathsf{Injective}(T)$ and $\mathsf{IV} \notin \mathsf{ran}(T)$.

Recall from Section 5.2 that in game $\mathsf{G}_{\mathsf{real}'}$, two maps T' and T are maintained such that it holds that $T' \subseteq T$, i.e. all elements in the domain of T' are also in the domain of T, and they map to the same values. We define the following predicate to express this property.

Definition 6.3 (Predicate Inclusion)**.** Let T and T' be two maps of type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. Then the predicate Inclusion is defined as

$$\mathsf{Inclusion}(T', T) := \forall xy.\ xy \in \mathsf{dom}(T') \Rightarrow xy \in \mathsf{dom}(T) \wedge T[xy] = T'[xy].$$

We use the statements $T' \subseteq T$ and $\mathsf{Inclusion}(T', T)$ equivalently. EasyCrypt also provides a polymorphic update function upd such that

$$\mathsf{upd}(T, xy, z)[xy'] = \mathsf{if}\ xy = xy'\ \mathsf{then\ return}\, z\ \mathsf{else\ return}\ T[xy'],$$

which is internally defined by the elementary properties that one expects of such an operator. In the above statement we say that T gets *updated* with the pair (xy, z). It is easy to automatically derive additional properties related to our predicates, such as:

1. If $\mathsf{Inclusion}(T', T)$ and $\mathsf{Injective}(T)$, then $\mathsf{Injective}(T')$;
2. If $\mathsf{Inclusion}(T', T)$ and $xy \notin \mathsf{dom}(T)$, then $xy \notin \mathsf{dom}(T')$;
3. If $\mathsf{Inclusion}(T', T)$ and $z \notin \mathsf{ran}(T)$, then $z \notin \mathsf{ran}(T')$.

Later, we will often need such properties and we note them here once and for all; they will be implicit in the proofs performed in this chapter.

Lastly, we introduce the notion of a *benign* update. Intuitively, a benign update of a well-formed map T is an update which preserves the well-formedness of T, and does not modify values that were already present in T. Formally, we define it as follows.

Definition 6.4 (Benign update)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, xy be a pair of a block and a chaining value, and z be a chaining value. Assume that T is well-formed. The update of the map T with the pair (xy, z) is *benign* iff it holds that $T \subseteq \mathsf{upd}(T, xy, z)$, and $\mathsf{upd}(T, xy, z)$ is still well-formed.

Remark. The definition does not exclude the case $xy \in \mathsf{dom}(T)$. Indeed, in this case the update of T with the pair $(xy, T[xy])$ is benign.

In the games described in Section 5.2, all updates of T are benign up to the failure events described there. Similarly as above, we note that we can automatically derive certain related properties of the upd function.

1. If $xy \notin \mathsf{dom}(T)$, then $\forall z.\ \mathsf{Inclusion}(T, \mathsf{upd}(T, xy, z))$;
2. If $\mathsf{Injective}(T)$ and $z \notin \mathsf{ran}(T)$, then $\forall xy.\ \mathsf{Injective}(\mathsf{upd}(T, xy, z))$;
3. If $z' \notin \mathsf{ran}(T)$ and $z \neq z'$, then $\forall xy.\ z' \notin \mathsf{ran}(\mathsf{upd}(T, xy, z))$.

Thus it is easy to see that if a map T is well-formed and gets updated with a pair (xy, z) such that $xy \notin \mathsf{dom}(T)$, $z \notin \mathsf{ran}(T)$ and $z \neq \mathsf{IV}$, then this update is benign. Similarly, if we consider two well-formed maps $T' \subseteq T$ and some element $xy \in \mathsf{dom}(T)$, then the update of T' with the pair $(xy, T[xy])$ is benign.

6.1.2 Operators pad and unpad

The first operators that we define are the functions pad and unpad, which are used to decompose a message into a padding, respectively to reassemble a message from a padding.

Definition 6.5 (Operators pad and unpad)**.** The operator pad is a function which maps messages to paddings, i.e. a function of type $\{0,1\}^* \to \{0,1\}^k$ list. Conversely, the operator unpad is a function of type $\{0,1\}^k$ list $\to \{0,1\}^*$ option which maps paddings in the range of pad back to messages.

We characterize the desired behavior of the functions pad and unpad and their mutual relation using the following axioms.

Axiom 6.6. *It holds that* $\mathsf{unpad}(\mathsf{nil}) = \mathsf{None}$.

Axiom 6.7 (Soundness of unpad)**.** *Let p be a padding such that* $\mathsf{unpad}(p) \neq \mathsf{None}$. *Then there exists a message m such that* $\mathsf{pad}(m) = p$.

Axiom 6.8 (Completeness and correctness of unpad)**.** *Let m be a message. Then it holds true that* $\mathsf{unpad}(\mathsf{pad}(m)) = \mathsf{Some}(m)$.

Further, we need the obvious property that the function unpad is injective (in other words, no message has two distinct paddings).

Axiom 6.9 (Injectivity of unpad)**.** *Let p and p' be paddings. Assume that* $\mathsf{unpad}(p) \neq \mathsf{None}$ *and that* $\mathsf{unpad}(p) = \mathsf{unpad}(p')$. *Then it holds that $p = p'$.*

Lastly, we formalize our assumption that the padding function is prefix-free as follows.

Axiom 6.10 (Prefix-freeness)**.** *Let m and m' be two messages with $m \neq m'$. Then for all paddings p, it holds that* $\mathsf{pad}(m) \neq \mathsf{pad}(m') \parallel p$.

6.1.3 Operator mapfst

The operator mapfst is a simple operator which maps the operator fst to a list. It is used to recover a padding from a list returned by the function $\mathsf{findseq}$, since the function $\mathsf{findseq}$ returns a chain of the form $[(x_1, y_1), \ldots, (x_j, y_j)]$, consisting of pairs of blocks and chaining values.

Definition 6.11 (Operator mapfst)**.** The operator mapfst is a function of type $(\{0,1\}^k \times \{0,1\}^n)$ $\mathsf{list} \to \{0,1\}^k$ list defined by the following equations:

$$\begin{aligned} \mathsf{mapfst}(\mathsf{nil}) &:= \mathsf{nil} \\ \mathsf{mapfst}((x, y)::c) &:= x::\mathsf{mapfst}(c), \end{aligned}$$

where x is a block, y a chaining value and $c : (\{0,1\}^k \times \{0,1\}^n)$ list is a list of pairs of blocks and chaining values.

6.1.4 Operator findseq

As discussed in Chapter 5, the operator $\mathsf{findseq}$ is used to find a complete chain (see Definition 5.2) in a map T. That is, the function call $\mathsf{findseq}(x, y, T)$ searches in map T for a complete chain of the form $c \parallel [(x, y)]$. It returns $\mathsf{Some}(c)$, or None if there is no such chain. All elements of c are required to be in the domain of T, however the last element of the complete chain, namely (x, y), is not.

To define this operator, we first define an auxiliary predicate $\mathsf{ischain}$.

Definition 6.12 (Predicate $\mathsf{ischain}$)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, y and z be chaining values, and xy and xy' be pairs of blocks and chaining values. Furthermore let $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list. Then the predicate $\mathsf{ischain}$ is recursively defined by the following equations:

$$\begin{aligned}
\mathsf{ischain}(T, y, \mathsf{nil}, z) &:= (y = z) \\
\mathsf{ischain}(T, y, xy{::}\mathsf{nil}, z) &:= \big(xy \in \mathsf{dom}(T) \wedge y = \mathsf{snd}(xy) \wedge \\
&\qquad T[xy] = z\big) \\
\mathsf{ischain}(T, y, xy{::}xy'{::}c, z) &:= \big(xy \in \mathsf{dom}(T) \wedge y = \mathsf{snd}(xy) \wedge \\
&\qquad T[xy] = \mathsf{snd}(xy') \wedge \\
&\qquad \mathsf{ischain}(T, \mathsf{snd}(xy'), xy'{::}c, z)\big)
\end{aligned}$$

Now, instead of giving a concrete implementation for the function findseq, we simply define it by the properties we expect it to have. On the one hand, this yields a higher level of abstraction as we do not commit to a specific implementation; on the other, this also means that we have to believe that there indeed exists a function which exhibits these properties. Specifically, we characterize the function findseq by two axioms stating that it is sound and complete, respectively.

Axiom 6.13 (Soundness of findseq). *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* x *a block and* y *a chaining value. Assume that* $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$*. Then, it holds that* $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T)}, y)$ *and* $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T)}) \parallel [x]) \neq \mathsf{None}$.

Axiom 6.14 (Completeness of findseq). *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* x *a block,* y *a chaining value, and* $c : (\{0,1\}^k \times \{0,1\}^n)$ list *a list. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{ischain}(T, \mathsf{IV}, c, y)$
4. $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) \neq \mathsf{None}$

Then, it holds that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$.

Note that for the completeness, we require that T is injective and that $\mathsf{IV} \notin \mathsf{ran}(T)$, i.e. T is well-formed. Requiring the well-formedness of T for the completeness of findseq is not strictly necessary to perform our indifferentiability proof, but it makes the function findseq much easier to implement in practice, since findseq needs only be successful in well-formed maps. We will also see later that we can use these well-formedness conditions to prove that given the premises of the completeness axiom, we can not only say that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$, but indeed that $\mathsf{findseq}(x, y, T) = \mathsf{Some}(c)$ (see Lemma 6.34). A possible pseudocode implementation of findseq that fulfills the above axioms is given below.

```
function findseq($x, y, T$) =
 $c \leftarrow$ nil;
 $len \leftarrow 0$;
 while $len < q \wedge y \in \mathsf{ran}(T) \wedge y \neq \mathsf{IV}$ do
   $(x', y') \leftarrow T^{-1}[y]$;
   $c \leftarrow (x', y'){::}c$; $len \leftarrow len + 1$; $y \leftarrow y'$;
 if $y = \mathsf{IV} \wedge \mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) \neq \mathsf{None}$ then
   return $c$
 else return None
```

As in Chapter 5, we use $q := \ell \cdot q_{\mathcal{D}}$ to bound the maximal length of a chain (recall that the number of oracle queries that the distinguisher can perform is upper-bounded by q). This ensures termination even in the highly unlikely event where there are loops in a chain, and yields a running time of $\mathcal{O}(\ell \cdot q_D)$ for the function findseq.

Lastly, we also define a related predicate valid_chain. Intuitively, valid_chain(T, c) holds whenever c is a complete chain in T whose last element is also in the domain of T.

Definition 6.15 (Predicate valid_chain)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map and $c : (\{0,1\}^k \times \{0,1\}^n)$ list a list. Then the predicate valid_chain is defined by

$$\mathsf{valid\_chain}(T, c) := \exists z.\ \mathsf{ischain}(T, \mathsf{IV}, c, z) \wedge \mathsf{unpad}(\mathsf{mapfst}(c)) \neq \mathsf{None}$$

6.1.5 Predicate set_bad3

The predicate set_bad3 is used in game $\mathsf{G}_{\mathsf{real}'}$ to detect the failure event $\mathbf{bad}_3$, as discussed in Section 5.2. In this chapter, we will see several lemmas involving this predicate which are necessary to prove the validity of the transformation described in that section.

Definition 6.16 (Predicate set_bad3)**.** Let y be a chaining value and let T and T' be two maps of type $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. Then

$$\mathsf{set\_bad3}(y, T', T) := y \in \mathsf{ran}(T) \wedge T^{-1}[y] \notin \mathsf{dom}(T') \wedge \forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[y]], T).$$

The idea is the following. Whenever the distinguisher makes a query $f_q(x, y)$ in game $\mathsf{G}_{\mathsf{real}'}$ and there is some preceding element $(x', y') := T^{-1}[y]$ in the domain of T that maps to the given chaining value y, then the current query is a continuation of a chain in T. But if this preceding element (x', y') does not appear in the domain of the map T', then this means that the distinguisher has never made the query $f_q(x', y')$; now if additionally, (x', y') is not the last element of a complete chain in T, then y should be unknown to the distinguisher. Indeed, the distinguisher cannot guess the intermediate chaining values generated in a Merkle-Damgård iteration until it queries for them, which it can only do in the correct order; the distinguisher may only learn the last value generated in a Merkle-Damgård iteration beforehand, by querying the oracle implementing the hash function, since the last chaining value is the final hash. Thus if set_bad3(y, T', T) evaluates to true in the procedure $\mathsf{G}_{\mathsf{real}'}.f_q$, then the distinguisher has used in the current query a chaining value which should be unknown to it, and $\mathbf{bad}_3$ is set to true.

Note that we use a map T^{-1} in this predicate. This is possible because in the game $\mathsf{G}_{\mathsf{real}'}$ where this predicate is used, we have that the map T is injective. In our actual EasyCrypt implementation, we update a map T_{inv} along with the map T, prove as an invariant that T_{inv} is the inverse of T throughout the game $\mathsf{G}_{\mathsf{real}'}$, and pass T_{inv} as an additional parameter to set_bad3 – we omitted these technical details in the description of the games for the sake of the reader. In all the lemmas described in the remainder of this chapter that involve an inversion on T, we will have as a premise that T is injective, such that it makes sense to speak about the map T^{-1}.

6.1.6 Predicate Claim5

Lastly, we recall the definition of the predicate Claim5 as presented in Definition 5.3 of Section 5.2. We will see several lemmas concerning this predicate at the end of this chapter.

$$\begin{aligned}\mathsf{Claim5}(T', T) = \forall (x, y), (x', y').\ (x', y') \in \mathsf{dom}(T) \wedge (x, y) \in \mathsf{dom}(T') \wedge \\ T[(x', y')] = y \wedge \mathsf{findseq}(x', y', T) = \mathsf{None} \Rightarrow (x', y') \in \mathsf{dom}(T').\end{aligned}$$

Recall that in game $\mathsf{G}_{\mathsf{real}'}$, the simulator f_q maintains a map T' of previously answered queries, while the FIL-RO f (resp. $f_{\mathbf{bad}}$) maintains a map T, and we have that $T' \subseteq T$: Indeed, a fresh call of the distinguisher to procedure F_q generates an MD-chain whose elements are all contained in map T, while a fresh call to f_q generates a single element in the maps T' and T. Then, the statement $\mathsf{Claim5}(T', T)$ means that the distinguisher has only made queries to f_q associated to an MD-chain in the correct order; essentially, for all elements (x, y) that belong to a chain in map T, it holds that the preceding element $(x', y') \in \mathsf{dom}(T)$, for which $T[(x', y')] = y$, is also in the domain of T' whenever (x, y) is in the domain of T'. That is, for any element (x, y) that belongs to a chain in T', all preceding elements of the chain are also in the domain of T', meaning that the distinguisher has made the corresponding queries.

In the corner case where (x, y) is the *first* element of a chain in a well-formed map T, the statement is trivially fulfilled where we consider it. Indeed, in this case it holds that $y = \mathsf{IV}$, and therefore the premises $(x', y') \in \mathsf{dom}(T)$ and $T[(x', y')] = y$ imply that $\mathsf{IV} \in \mathsf{ran}(T)$, which is false if T is well-formed.

Similarly, the case where (x', y') is the *last* element of a chain in T also represents a corner case, since in this case the element $T[(x', y')] = y$ corresponds to the final hash of a message (instead of an intermediate chaining value) and is therefore known to the distinguisher. So, the distinguisher could *extend* this chain by making additional queries (consider e.g. a length extension attack). This is not a problem in our setup as we assume the padding function to be prefix-free; however, for the statement $\mathsf{Claim5}(T', T)$ to hold as an invariant, we need to explicitly exclude this case, since here the fact that an element (x, y) is in the domain of T' does not necessarily mean that all preceding elements of the original chain are also in the domain of T'. This is why the predicate uses the premise $\mathsf{findseq}(x', y', T) = \mathsf{None}$, meaning that (x', y') is not the end of a complete chain in T.

In the game $\mathsf{G}_{\mathsf{real}'}$ where this predicate is used, we always have the invariants that $T' \subseteq T$, and that T is well-formed. Therefore we are able to eventually obtain that $\mathsf{Claim5}(T', T)$ is also an invariant of that game.

6.2 Padding Lemmas

In this section, we will prove two simple lemmas concerning the padding function, based on the axioms stated in Section 6.1.2. The first lemma states that nil is not a valid padding. This is needed to be able to conclude that any padding contains at least one element, which is relevant e.g. in the transformation described in Section 5.1.

Lemma 6.17. *There is no message m such that* $\mathsf{pad}(m) = \mathsf{nil}$.

Proof. We prove the statement by contradiction. Let m be a message, and assume for contradiction that $\mathsf{pad}(m) = \mathsf{nil}$. By Axiom 6.8, we know that $\mathsf{unpad}(\mathsf{pad}(m)) = \mathsf{Some}(m)$ and hence we obtain $\mathsf{unpad}(\mathsf{nil}) = \mathsf{Some}(m)$, contradicting Axiom 6.6. □

The second padding lemma that we want to show is an alternative description of the prefix-freeness of the padding function, based on unpad instead of pad. This will later be useful as a helper lemma.

Lemma 6.18. *Let p_1 and p_2 be two paddings. Assume that* $\mathsf{unpad}(p_1 \parallel p_2) \neq \mathsf{None}$ *and that* $p_2 \neq \mathsf{nil}$. *Then it holds that* $\mathsf{unpad}(p_1) = \mathsf{None}$.

Proof. We know by Axiom 6.7 that there exists a message m such that $\mathsf{pad}(m) = p_1 \parallel p_2$. Now, assume for contradiction that $\mathsf{unpad}(p_1) \neq \mathsf{None}$. Then we also know that there exists a message m' such that $\mathsf{pad}(m') = p_1$. Therefore we get $\mathsf{pad}(m) = \mathsf{pad}(m') \parallel p_2$. We now perform a case distinction:

Case $m = m'$. If $m = m'$, then $\mathsf{pad}(m) = \mathsf{pad}(m')$, and thus $p_2 = \mathsf{nil}$, contradicting our assumptions.

Case $m \neq m'$. In this case, we obtain a contradiction to prefix-freeness: Using Axiom 6.10, we conclude that $\mathsf{pad}(m) \neq \mathsf{pad}(m') \parallel p_2$, which immediately yields the contradiction.

□

6.3 Lemmas on mapfst

We note here that the operator mapfst is distributive.

Lemma 6.19 (Distributivity of mapfst). *Let c_1 and c_2 be two lists of type* $(\{0,1\}^k \times \{0,1\}^n)$ list. *Then it holds that* $\mathsf{mapfst}(c_1 \parallel c_2) = \mathsf{mapfst}(c_1) \parallel \mathsf{mapfst}(c_2)$.

Proof. We prove the claim by structural induction over the list c_1.

Base case: $c_1 = \mathsf{nil}$. Trivial.

Induction step: $c_1 = a{::}al$.
Induction hypothesis:

$$\mathsf{mapfst}(al \parallel c_2) = \mathsf{mapfst}(al) \parallel \mathsf{mapfst}(c_2)$$

We need to show that $\mathsf{mapfst}(a{::}al \parallel c_2) = \mathsf{mapfst}(a{::}al) \parallel \mathsf{mapfst}(c_2)$. This is easy:

$$\begin{aligned}
\mathsf{mapfst}(a{::}al \parallel c_2) &= \mathsf{mapfst}(a{::}(al \parallel c_2)) \\
&= \mathsf{fst}(a){::}\mathsf{mapfst}(al \parallel c_2) && \text{(by Definition 6.11)} \\
&= \mathsf{fst}(a){::}(\mathsf{mapfst}(al) \parallel \mathsf{mapfst}(c_2)) && \text{(by IH)} \\
&= \mathsf{fst}(a){::}\mathsf{mapfst}(al) \parallel \mathsf{mapfst}(c_2) \\
&= \mathsf{mapfst}(a{::}al) \parallel \mathsf{mapfst}(c_2) && \text{(by Definition 6.11)}
\end{aligned}$$

□

In the remainder of this chapter, we will assume an intuitive understanding of this operator. For instance, we assume that it is clear that a list $\mathsf{mapfst}(c \parallel [(x,y)])$ can be rewritten as $\mathsf{mapfst}(c) \parallel [x]$. We strive to write such lists in their simplest possible form, except where it makes sense to do otherwise for the purpose of clarity.

6.4 Lemmas on ischain

In this section, we will discuss several lemmas concerning the predicate ischain. Many of these lemmas are not immediately needed for the proof of indifferentiability performed in EasyCrypt; rather, they serve as helper lemmas on which to build more complex lemmas later on.

We start with a very basic lemma that makes explicit a property of the statement $\mathsf{ischain}(T, y, c, z)$ where c is not empty: the value y is the first chaining value used in c.

Lemma 6.20. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list a list, x be a block, and y, y' and z be chaining values. Assume that $\mathsf{ischain}(T, y, (x, y')::c, z)$ holds. Then we have $y = y'$.*

Proof. We only need to distinguish the cases where $c = \mathsf{nil}$ and where c has the form $a::al$. In each case the statement follows immediately from Definition 6.12. □

The next lemma can be thought of as an alternative definition of the predicate ischain, in the case where the chain in question is not nil. It is an extremely helpful tool in many of the lemmas that follow.

Lemma 6.21. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list be lists, x' be a block, and y, y' and z be chaining values. Then it holds:*

$$\begin{aligned}\mathsf{ischain}(T, y, c_1 \parallel (x', y')::c_2, z) \Leftrightarrow\ & \mathsf{ischain}(T, y, c_1, y') \wedge \\ & (x', y') \in \mathsf{dom}(T) \wedge \\ & \mathsf{ischain}(T, T[(x', y')], c_2, z)\end{aligned}$$

Proof. To prove the claim, we prove both directions of the claim separately. In both cases we use a structural induction over the list c_1. In the respective base cases, we distinguish the cases where $c_2 = \mathsf{nil}$ and where c_2 has the form $b::bl$ and apply Definition 6.12. For the direction "$\Leftarrow$", we additionally need to to apply Lemma 6.20 to obtain $T[(x', y')] = \mathsf{snd}(b)$ in the case where $c_2 = b::bl$, since the definition of ischain requires that the second argument of the predicate ischain is equal to the second projection of the first element of the chain. In the respective induction steps, we distinguish the cases where $c_1 = \mathsf{nil}$ and where c_1 has the form $a::al$ and use Definition 6.12 on the induction premises to fulfill the premises of the respective induction hypotheses, and easily obtain the conclusion. □

The previous lemma gives us two corollaries. The first corollary permits the automated tools used in EasyCrypt to perform a case distinction on the statement $\mathsf{ischain}(T, \mathsf{IV}, c, z)$: If $c = \mathsf{nil}$, then it holds that $z = \mathsf{IV}$; if c is not empty, then there is an element in the domain of T (namely, the last element of c) that maps to the chaining value z.

Corollary 6.22. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, and z be a chaining value. Assume that $\mathsf{ischain}(T, \mathsf{IV}, c, z)$ holds. Then it holds true that $z = \mathsf{IV} \vee \exists xy.\ xy \in \mathsf{dom}(T) \wedge T[xy] = z$.*

Proof. If $c = \mathsf{nil}$, then we immediately get $z = \mathsf{IV}$ from Definition 6.12. If c has the form $al \parallel [a]$, then Lemma 6.21 applies and we conclude $a \in \mathsf{dom}(T)$ and $\mathsf{ischain}(T, T[a], \mathsf{nil}, z)$ which implies $T[a] = z$. □

The second corollary, useful later, states that any element that belongs to a chain in a map T is in the domain of T.

Corollary 6.23. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, xy be a pair of a block and a chaining value and z be a chaining value. Assume the following:*

1. $\mathsf{ischain}(T, \mathsf{IV}, c, z)$
2. $xy \in c$

Then it holds that $xy \in \mathsf{dom}(T)$.

Proof. Since $xy \in c$ then c has the form $c_\ell \parallel xy{::}c_r$, for appropriate c_ℓ and c_r. Hence Lemma 6.21 applies and we find $xy \in \mathsf{dom}(T)$. □

In the transformations presented in Chapter 5, to show that some invariant of a game relating to maps maintained in that game is valid, it is often necessary to state under which conditions a map may be updated without afflicting the validity of the statement expressed by this invariant. In turn, the validity of these conditions is usually guaranteed by the failure events. The invariants described in Section 5.2 involve predicates such as set_bad3 or Claim5, which in turn involve the predicates valid_chain or the operator findseq. This are built upon the predicate ischain. Hence, when breaking down the necessary lemmas that ensure that some assertion in a game holds after a map update when it held before the map update, it often comes to show that under sensible conditions, a chain in a map T remains a chain when T gets updated. This is what the next two lemmas are used for. The first states that when a map T contains a chain and gets updated with a value not part of this chain, then this chain is still contained in the updated map.

Lemma 6.24. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, xy be a pair of a block and a chaining value and z and z' be chaining values. Assume the following:*

1. $\mathsf{ischain}(T, \mathsf{IV}, c, z)$
2. $xy \notin c$

Then it holds that $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), \mathsf{IV}, c, z)$.

Proof. We first *generalize* the constant IV as an arbitrary chaining value y; that is, we show the claim for arbitrary values y where the claim focuses only on the instance where $y = \mathsf{IV}$. This is necessary so as to obtain a strong enough induction hypothesis. We prove the resulting generalized statement by a structural induction over the list c.

Base case: $c = \mathsf{nil}$**.** We know $\mathsf{ischain}(T, y, \mathsf{nil}, z)$ which implies $y = z$. This implies $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, \mathsf{nil}, z)$ (see Definition 6.12).

Induction step: $c = a{::}al$**.**

Induction hypothesis:

$$\forall y.\ \mathsf{ischain}(T, y, al, z) \Rightarrow xy \notin al \Rightarrow \mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, al, z)$$

We must show that $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, a{::}al, z)$. By Lemma 6.21, this is equivalent to the following three conditions:

(i) $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, \mathsf{nil}, \mathsf{snd}(a))$: By Definition 6.12 we have to show that $y = \mathsf{snd}(a)$. This is immediately implied by the premise $\mathsf{ischain}(T, y, a{::}al, z)$ (Lemma 6.20).

(ii) $a \in \mathsf{dom}(\mathsf{upd}(T, xy, z'))$: By Lemma 6.21, $\mathsf{ischain}(T, y, a{::}al, z)$ implies that $a \in \mathsf{dom}(T)$, and therefore it also holds that $a \in \mathsf{dom}(\mathsf{upd}(T, xy, z'))$.

(iii) $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), \mathsf{upd}(T, xy, z')[a], al, z)$: Here we need the induction hypothesis. First note that $xy \neq a$ since we know that $xy \notin a{::}al$. Therefore it holds true that $\mathsf{upd}(T, xy, z')[a] = T[a]$. Hence, it suffices to show $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), T[a], al, z)$. This is implied by taking $y := T[a]$ in the induction hypothesis, and we are only left to show the two premises of the induction hypothesis: First, we know $\mathsf{ischain}(T, y, a{::}al, z)$, and thus it holds that $\mathsf{ischain}(T, T[a], al, z)$ by Lemma 6.21. Second, we know $xy \notin a{::}al$, which implies $xy \notin al$.

□

Next, we prove that a chain in a map T' remains a chain in an updated map T as long as it holds that $T \supseteq T'$. That is, a map T' containing a chain can be updated with arbitrarily many values outside of its domain and still contain the same chain. The proof is mostly analogous to the proof of the previous lemma, except in the case where the induction hypothesis needs to be applied.

Lemma 6.25. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be two maps,* $c : (\{0,1\}^k \times \{0,1\}^n)$ list *be a list, and* y *and* z *be chaining values. Assume the following:*

1. $\mathsf{ischain}(T', y, c, z)$
2. $\mathsf{Inclusion}(T', T)$

Then it holds that $\mathsf{ischain}(T, y, c, z)$*.*

Proof. As in Lemma 6.24, we perform a structural induction over the list c.

Base case: $c = \mathsf{nil}$. As in the proof of Lemma 6.24.

Induction step: $c = a{::}al$.

Induction hypothesis:

$$\forall y.\ \mathsf{ischain}(T', y, al, z) \Rightarrow \mathsf{ischain}(T, y, al, z)$$

As in the previous lemma, we show the three necessary and sufficient conditions according to Lemma 6.21:

(i) $\mathsf{ischain}(T, y, \mathsf{nil}, \mathsf{snd}(a))$: We know $\mathsf{ischain}(T', y, a{::}al, z)$, and therefore it holds that $y = \mathsf{snd}(a)$, which yields the conclusion by Definition 6.12.

(ii) $a \in \mathsf{dom}(T)$: We know that $\mathsf{ischain}(T', y, a{::}al, z)$, which implies $a \in \mathsf{dom}(T')$ by Lemma 6.21. Since $\mathsf{Inclusion}(T', T)$ this further implies $a \in \mathsf{dom}(T)$.

(iii) $\mathsf{ischain}(T, T[a], al, z)$: This is the case where we apply the induction hypothesis. We instantiate the induction hypothesis taking $y := T'[a]$. From the premise $\mathsf{ischain}(T', y, a{::}al, z)$, we obtain using Lemma 6.21 that $a \in \mathsf{dom}(T')$ and that $\mathsf{ischain}(T', T'[a], al, z)$; the latter fulfills the premise of the induction hypothesis. Therefore we obtain that $\mathsf{ischain}(T, T'[a], al, z)$, which is almost our goal. It only remains to show that $T'[a] = T[a]$. But this is clear since we have $a \in \mathsf{dom}(T')$ and $\mathsf{Inclusion}(T', T)$.

□

The previous lemma also yields an immediate corollary that allows a single update to a map T with a value outside of the domain of T.

Corollary 6.26. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, xy be a pair of a block and a chaining value, and y, z, z' be chaining values. Assume the following:*

1. $\mathsf{ischain}(T, y, c, z)$
2. $xy \notin \mathsf{dom}(T)$

Then it holds that $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, c, z)$.

Proof. This follows immediately from Lemma 6.25 since $xy \notin \mathsf{dom}(T)$ clearly implies that $\mathsf{Inclusion}(T, \mathsf{upd}(T, xy, z'))$. □

A dual of Lemma 6.24 is the following statement. If a list c is *not* a chain in some map T, but becomes a chain after a single update of this map, then the element which was used to update the map must be part of the chain.

Lemma 6.27. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, xy be a pair of a block and a chaining value, and z and z' be chaining values. Assume:*

1. $\neg\mathsf{ischain}(T, \mathsf{IV}, c, z)$.
2. $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), \mathsf{IV}, c, z)$.

Then it holds that $xy \in c$.

Proof. We generalize the constant IV as an arbitrary variable y, and perform a structural induction over the list c.

Base case: $c = \mathsf{nil}$**.** In this case we get a contradiction since the first premise implies $y \neq z$, but the second premise implies $y = z$ (see Definition 6.12).

Induction step: $c = a{::}al$**.**

Induction hypothesis:

$$\forall y.\ \neg\mathsf{ischain}(T, y, al, z) \Rightarrow \mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, al, z) \Rightarrow xy \in al$$

We must show that $xy \in a{::}al$. That is, we have to show that $xy = a \vee xy \in al$. To accomplish this, assume that $xy \neq a$; under this hypothesis we will prove that $xy \in al$. We perform a case distinction on the structure of al.

Case $al = \mathsf{nil}$**.** In this case, we obtain a contradiction. On the one hand, we know $\neg\mathsf{ischain}(T, y, [a], z)$. On the other hand, we know $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, [a], z)$ and $xy \neq a$. We show below that this implies $\mathsf{ischain}(T, y, [a], z)$, yielding the contradiction. In fact, the latter is easily seen, since it suffices to unfold the definition of ischain and show the following:

(i) $a \in \mathsf{dom}(T)$: This follows from $a \in \mathsf{dom}(\mathsf{upd}(T), xy, z')$ and $xy \neq a$.

(ii) $y = \mathsf{snd}(a)$: This immediately follows from $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, [a], z)$.

(iii) $T[a] = z$: This follows from $\mathsf{upd}(T, xy, z')[a] = z$ and $xy \neq a$.

Case $al = a'{::}al'$**.** We have to show $xy \in a'{::}al'$. We instantiate the induction hypothesis taking $y := \mathsf{upd}(T, xy, z')[a]$, which yields the goal. It remains to show the two premises of the induction hypothesis. The second premise of the induction hypothesis, namely $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), \mathsf{upd}(T, xy, z')[a], a'{::}al', z)$, follows quickly since it holds that $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, a{::}a'{::}al', z)$, and we apply Lemma 6.21. We are only left to show the first premise of the induction hypothesis, namely $\neg\mathsf{ischain}(T, \mathsf{upd}(T, xy, z')[a], a'{::}al', z)$. Recall that $\neg\mathsf{ischain}(T, y, a{::}a'{::}al', z)$, that is

$$\neg\big(a \in \mathsf{dom}(T) \wedge y = \mathsf{snd}(a) \wedge T[a] = \mathsf{snd}(a') \wedge \mathsf{ischain}(T, \mathsf{snd}(a'), a'{::}al', z)\big).$$

Next, recall that $\mathsf{ischain}(\mathsf{upd}(T, xy, z'), y, a{::}a'{::}al', z)$. Hence, by Definition 6.12 we know (i) $a \in \mathsf{dom}(\mathsf{upd}(T), xy, z')$ and hence $a \in \mathsf{dom}(T)$ since $xy \neq a$; (ii) $y = \mathsf{snd}(a)$, and (iii) $\mathsf{upd}(T, xy, z')[a] = \mathsf{snd}(a')$, implying $T[a] = \mathsf{snd}(a')$ since $xy \neq a$. Hence, it must hold that $\neg\mathsf{ischain}(T, \mathsf{snd}(a'), a'{::}al', z)$. Lastly, assume for contradiction that $\mathsf{ischain}(T, \mathsf{upd}(T, xy, z')[a], a'{::}al', z)$ were true. By Lemma 6.20 this would imply $\mathsf{ischain}(T, \mathsf{snd}(a'), a'{::}al', z)$, yielding a contradiction. Therefore, we obtain $\neg\mathsf{ischain}(T, \mathsf{upd}(T, xy, z')[a], a'{::}al', z)$, which finishes the proof.

□

Next, we show that any chain in a map T and the value that it hashes to are uniquely defined by the padding associated to it.

Lemma 6.28. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list be lists, and let z_1 and z_2 be chaining values. Assume the following:*
1. $\mathsf{ischain}(T, y, c_1, z_1)$
2. $\mathsf{ischain}(T, y, c_2, z_2)$
3. $\mathsf{mapfst}(c_1) = \mathsf{mapfst}(c_2)$

Then it holds that $c_1 = c_2$ and $z_1 = z_2$.

Proof. We perform a structural induction over c_1.

Base case: $c_1 = \mathsf{nil}$. If $c_2 \neq \mathsf{nil}$, we obtain a contradiction to $\mathsf{mapfst}(c_1) = \mathsf{mapfst}(c_2)$. If $c_2 = \mathsf{nil}$, it only remains to show that $z_1 = z_2$. From the first and second premises, we obtain $y = z_1$ and $y = z_2$ (unfolding Definition 6.12), which finishes this case.

Induction step: $c_1 = a{::}al$.

Induction hypothesis:

$$\forall c_2.\ \mathsf{mapfst}(al) = \mathsf{mapfst}(c_2) \Rightarrow \forall y.\ \mathsf{ischain}(T, y, al, z_1) \Rightarrow \mathsf{ischain}(T, y, c_2, z_2) \Rightarrow al = c_2 \wedge z_1 = z_2$$

If $c_2 = \mathsf{nil}$, we obtain a contradiction to $\mathsf{mapfst}(c_1) = \mathsf{mapfst}(c_2)$. Therefore assume that c_2 has the form $b{::}bl$. We instantiate the induction hypothesis with the list bl. From $\mathsf{mapfst}(a{::}al) = \mathsf{mapfst}(b{::}bl)$, we obtain that $\mathsf{fst}(a) = \mathsf{fst}(b)$ and that $\mathsf{mapfst}(al) = \mathsf{mapfst}(bl)$. The former hypothesis will yet be important later; the latter can be used to fulfill the first premise of the induction hypothesis. We obtain a new induction hypothesis:

$$\forall y.\ \mathsf{ischain}(T, y, al, z_1) \Rightarrow \mathsf{ischain}(T, y, bl, z_2) \Rightarrow al = bl \wedge z_1 = z_2$$

Next, we show that $\mathsf{snd}(a) = \mathsf{snd}(b)$. Indeed, from the first and second premises we obtain (using Lemma 6.21) that $\mathsf{ischain}(T, y, \mathsf{nil}, \mathsf{snd}(a))$ and $\mathsf{ischain}(T, y, \mathsf{nil}, \mathsf{snd}(b))$. By unfolding Definition 6.12, we obtain $y = \mathsf{snd}(a)$ and $y = \mathsf{snd}(b)$. Since we now know $\mathsf{snd}(a) = \mathsf{snd}(b)$ and we already knew $\mathsf{fst}(a) = \mathsf{fst}(b)$, we conclude $a = b$. Further, we conclude from the first and second premises (again using Lemma 6.21) that $\mathsf{ischain}(T, T[a], al, z_1)$ and $\mathsf{ischain}(T, T[b], bl, z_2)$. As we have $a = b$, we rewrite the latter hypothesis as $\mathsf{ischain}(T, T[a], bl, z_2)$. We can now instantiate the induction hypothesis taking $y := T[a]$, and we have already proven the two resulting premises of the induction hypothesis. Thus we conclude $al = bl$ and $z_1 = z_2$. Since $a = b$, we immediately get $c_1 = a{::}al = b{::}bl = c_2$.

□

Now, we want to show a central property of chains arising from the well-formedness conditions of maps (see Section 6.1.1). Namely, we will show that any chain in a well-formed map T is uniquely defined by the value that it hashes to. Formally, two chains in a well-formed map T that hash to the same value are, in fact, equal. This lemma also has many important implications that we will see later. However, the induction needed for the proof of this statement is somewhat tricky. To make it go through easier, we first show two closely related claims, considering the cases where the two chains have the same length or a different length, respectively.

Lemma 6.29. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list be lists, and let y_1, y_2 and z be chaining values. Assume the following:*

1. $|c_1| = |c_2|$
2. $\mathsf{Injective}(T)$
3. $\mathsf{ischain}(T, y_1, c_1, z)$
4. $\mathsf{ischain}(T, y_2, c_2, z)$

Then it holds that $c_1 = c_2$.

Proof. Fix c_1. Given the premise $\mathsf{Injective}(T)$, we show that

$$\forall c_2, y_1, y_2.\ |c_1| = |c_2| \Rightarrow \mathsf{ischain}(T, y_1, c_1, z) \Rightarrow \mathsf{ischain}(T, y_2, c_2, z) \Rightarrow c_1 = c_2,$$

using a structural induction over c_1.

Base case: $c_1 = \mathsf{nil}$.

If $c_2 = \mathsf{nil}$, the claim is trivial. If c_2 has the form $b{::}bl$, we get a contradiction since we assume $|c_1| = |c_2|$, i.e. $|\mathsf{nil}| = |b{::}bl|$.

Induction step: $c_1 = a{::}al$.

Induction hypothesis:

$$\forall c_2, y_1, y_2.\ |al| = |c_2| \Rightarrow \mathsf{ischain}(T, y_1, al, z) \Rightarrow \mathsf{ischain}(T, y_2, c_2, z) \Rightarrow al = c_2$$

We must show $a{::}al = c_2$. If $c_2 = \mathsf{nil}$, we get a contradiction to the premise $|c_1| = |c_2|$. Therefore assume that c_2 has the form $b{::}bl$. We can now sensibly instantiate the induction hypothesis as follows:

$$|al| = |bl| \Rightarrow \mathsf{ischain}(T, T[a], al, z) \Rightarrow \mathsf{ischain}(T, T[b], bl, z) \Rightarrow al = bl.$$

The first premise of this hypothesis follows easily since we know that $|a{::}al| = |b{::}bl|$. Further, we know that $\mathsf{ischain}(T, y_1, a{::}al, z)$ and $\mathsf{ischain}(T, y_2, b{::}bl, z)$. Hence, the second and third premises follow by using Lemma 6.21. We conclude that $al = bl$. Thus it only remains to show that $a = b$. Note that we now have $\mathsf{ischain}(T, y_1, a{::}al, z)$ and $\mathsf{ischain}(T, y_2, b{::}al, z)$. Using Lemma 6.21 we obtain that $a \in \mathsf{dom}(T)$ and $b \in \mathsf{dom}(T)$. Further, we also know that $T[a] = T[b]$: Indeed, in the case where $al \neq \mathsf{nil}$, we have $T[a] = \mathsf{hd}(al)$ and $T[b] = \mathsf{hd}(al)$ (from Lemma 6.20), and in the case where $al = \mathsf{nil}$, we have $T[a] = z$ and $T[b] = z$ (from Definition 6.12). Thus, we can use the injectivity of T to conclude that $a = b$, which finishes the proof.

□

Lemma 6.30. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list be lists, and let y_1, y_2 and z be chaining values. Assume the following:*

1. $|c_1| > |c_2|$
2. $\mathsf{Injective}(T)$
3. $\mathsf{ischain}(T, y_1, c_1, z)$
4. $\mathsf{ischain}(T, y_2, c_2, z)$

Then there exists a list pl and a chaining value p such that $c_1 = pl \parallel p{::}c_2$.

Proof. Fix c_2. Given the premise $\mathsf{Injective}(T)$, we show that

$$\forall c_1, y_1, y_2.\ |c_1| > |c_2| \Rightarrow \mathsf{ischain}(T, y_1, c_1, z) \Rightarrow \mathsf{ischain}(T, y_2, c_2, z) \Rightarrow \\ \exists p', pl'.\ c_1 = pl' \parallel p'{::}c_2$$

using a structural induction over c_2.

Base case: $c_2 = \mathsf{nil}$.

If $c_1 = \mathsf{nil}$, we get a contradiction to $|c_1| > |c_2|$. Therefore assume that c_1 has the form $a{::}al$. We must show that there exist pl, p such that $a{::}al = pl \parallel [p]$, which is easily seen.

Induction step: $c_2 = b{::}bl$.

Induction hypothesis:

$$\forall c_1, y_1, y_2.\ |c_1| > |bl| \Rightarrow \mathsf{ischain}(T, y_1, c_1, z) \Rightarrow \mathsf{ischain}(T, y_2, bl, z) \Rightarrow \\ \exists p', pl'.\ c_1 = pl' \parallel p'{::}bl$$

We must show that there exist p, pl such that $c_1 = pl \parallel p{::}b{::}bl$. If $c_1 = \mathsf{nil}$, we get a contradiction to the premise $|c_1| > |c_2|$. Therefore assume that c_1 has the form $a{::}al$. We can now sensibly instantiate the induction hypothesis as follows:

$$|al| > |bl| \Rightarrow \mathsf{ischain}(T, T[a], al, z) \Rightarrow \mathsf{ischain}(T, T[b], bl, z) \Rightarrow \exists p', pl'.\ al = pl' \parallel p'{::}bl$$

The first premise of this hypothesis follows easily since we know that $|a{::}al| > |b{::}bl|$. Further, we know that $\mathsf{ischain}(T, y_1, a{::}al, z)$ and $\mathsf{ischain}(T, y_2, b{::}bl, z)$. Hence, the second and third premises follow by using Lemma 6.21. We conclude that that there exist p', pl' such that $al = pl' \parallel p'{::}bl$, and we fix p' and pl'. We still have to show that there exist p, pl such that $a{::}al = pl \parallel p{::}b{::}bl$. Since we have $al = pl' \parallel p'{::}bl$, this means that we have to show that there exist p, pl such that $a{::}pl' \parallel p'{::}bl = pl \parallel p{::}b{::}bl$. It is easily seen that there exist p, pl such that $a{::}pl' = pl \parallel [p]$. Thus it only remains to show that $p' = b$. Note that we have $\mathsf{ischain}(T, y_1, a{::}al, z)$, which can be rewritten as $\mathsf{ischain}(T, y_1, a{::}pl' \parallel p'{::}bl, z)$. Using Lemma 6.21, this implies $\mathsf{ischain}(T, T[p'], bl, z)$ and $p' \in \mathsf{dom}(T)$. On the other hand, we also know that $\mathsf{ischain}(T, y_2, b{::}bl, z)$, which similarly implies $\mathsf{ischain}(T, T[b], bl, z)$ and $b \in \mathsf{dom}(T)$. Hence we obtain $T[p'] = T[b]$ (using Definition 6.12 if $bl = \mathsf{nil}$, and Lemma 6.20 if $bl \neq \mathsf{nil}$). Finally, using the injectivity of T, we conclude that $p' = b$, which finishes the proof.

□

The previous two lemmas only included one of two well-formedness conditions, namely that T is injective. Basically, we saw that if an injective map T contains two chains c_1 and c_2 that hash to the same value, then by the injectivity of T this means that the shorter chain is a suffix of the longer chain. At this point we can exploit the other well-formedness condition, namely that $\mathsf{IV} \notin \mathsf{ran}(T)$. Indeed, if two chains using IV as a first chaining value hash to the same value, and one of the two chains is longer than the other, then the longer chain includes an element that maps to the chaining value IV at the boundary of its suffix that is equal to the shorter chain. However, as IV is not in the range of a well-formed map, there cannot be two chains (starting with the chaining value IV) of different length in a well-formed map. We formally derive the contradiction in the following corollary.

Corollary 6.31. *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list *be lists, and let* z *be a chaining value. Assume the following:*

1. $|c_1| > |c_2|$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $\mathsf{ischain}(T, \mathsf{IV}, c_1, z)$
5. $\mathsf{ischain}(T, \mathsf{IV}, c_2, z)$

Then we obtain a contradiction, i.e. the above premises cannot hold all at once.

Proof. Using all but the third premise, we obtain using Lemma 6.30 a chaining value p and a list pl such that $c_1 = pl \parallel p{::}c_2$. Thus, the fourth premise can be rewritten as $\mathsf{ischain}(T, \mathsf{IV}, pl \parallel p{::}c_2, z)$. We now use a case distinction on the structure of c_2.

Case $c_2 = \mathsf{nil}$**.** By applying Lemma 6.21 we obtain $p \in \mathsf{dom}(T)$ and $\mathsf{ischain}(T, T[p], \mathsf{nil}, z)$. By Definition 6.12, the latter implies $T[p] = z$. Therefore $z \in \mathsf{ran}(T)$. By the fifth premise we know $\mathsf{ischain}(T, \mathsf{IV}, \mathsf{nil}, z)$ which implies $z = \mathsf{IV}$, and thus $\mathsf{IV} \in \mathsf{ran}(T)$. Contradiction to the third premise.

Case $c_2 = b{::}bl$**.** This case is analogous, but the value $T[p]$ is equal to $\mathsf{snd}(b)$ instead of z, and we use that $\mathsf{snd}(b)$ is equal to IV, yielding a contradiction.

□

Finally, using both Lemma 6.29 and Corollary 6.31, it is straightforward to conclude the uniqueness lemma mentioned above: We show that any chain in a well-formed map is uniquely defined by its final hash.

Lemma 6.32 (Uniqueness of chains)**.** *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list *be lists, and let* z *be a chaining value. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{ischain}(T, \mathsf{IV}, c_1, z)$
4. $\mathsf{ischain}(T, \mathsf{IV}, c_2, z)$

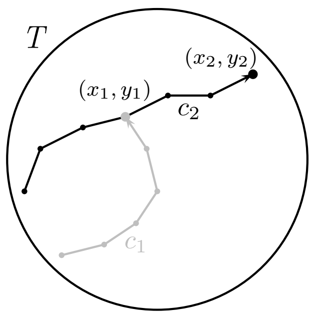

Figure 6.1: Colliding chains, an impossibility because of *Claim 4*

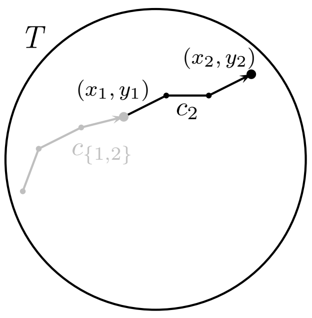

Figure 6.2: Length extension, an impossibility because of prefix-freeness

Then it holds that $c_1 = c_2$.

Proof. It cannot hold that either $|c_1| > |c_2|$ or $|c_2| > |c_1|$, since in both cases we obtain a contradiction using Corollary 6.31. Therefore it must hold that $|c_1| = |c_2|$. Thus Lemma 6.29 applies and we obtain $c_1 = c_2$. □

The last lemma of this section is another important one, with many consequences seen in later sections. Essentially, it states that there can be no colliding chains in a well-formed map T, as illustrated in Figure 6.1. It is named *Claim 4* in reminiscence of the same claim appearing in the proof by Coron [39], which our proof of indifferentiability is based on. It is a consequence of the uniqueness of chains described above, and the assumption that the padding function is prefix-free. Figures 6.1 and 6.2 illustrate the idea. They depict respectively two complete chains c_1 and c_2, ending respectively in (x_1, y_2) and (x_2, y_2), in a well-formed map T. By the uniqueness of chains, the chain c_1 is a prefix of chain c_2, but this contradicts our assumption of prefix-freeness.

Lemma 6.33 (Claim 4)**.** *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* $c_1, c_2 : (\{0,1\}^k \times \{0,1\}^n)$ list *be lists, let* x_1, x_2 *be blocks, and let* y_1, y_2 *be chaining values. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{ischain}(T, \mathsf{IV}, c_1, y_1)$
4. $\mathsf{unpad}(\mathsf{mapfst}(c_1) \parallel [x_1]) \neq \mathsf{None}$
5. $\mathsf{ischain}(T, \mathsf{IV}, c_2, y_2)$
6. $\mathsf{unpad}(\mathsf{mapfst}(c_2) \parallel [x_2]) \neq \mathsf{None}$

Then it holds that $(x_1, y_1) \notin c_2$.

Proof. We prove the claim by contradiction. Assume $(x_1, y_1) \in c_2$. Then, clearly c_2 has the form $c_2^\ell \parallel (x_1, y_1)::c_2^r$, for appropriate c_2^ℓ and c_2^r. Thus, we can rewrite the fifth premise as $\mathsf{ischain}(T, \mathsf{IV}, c_2^\ell \parallel (x_1, y_1)::c_2^r, y_2)$. By Lemma 6.21 this implies $\mathsf{ischain}(T, \mathsf{IV}, c_2^\ell, y_1)$. Now, since we also know $\mathsf{ischain}(T, \mathsf{IV}, c_1, y_1)$ and additionally the well-formedness conditions on map T hold, Lemma 6.32 applies and we conclude that $c_2^\ell = c_1$. Hence, we can rewrite the sixth premise as $\mathsf{unpad}(\mathsf{mapfst}(c_1 \parallel (x_1, y_1){::}c_2^r) \parallel [x_2]) \neq \mathsf{None}$. From this we will obtain a contradiction to the prefix-freeness of pad. By the distributivity of mapfst, the latter statement can be rewritten as $\mathsf{unpad}(\mathsf{mapfst}(c_1) \parallel [x_1] \parallel \mathsf{mapfst}(c_2^r) \parallel [x_2]) \neq \mathsf{None}$. Now, taking $p_1 := \mathsf{mapfst}(c_1) \parallel [x_1]$ and $p_2 := \mathsf{mapfst}(c_2^r) \parallel [x_2]$ in Lemma 6.18, we get $\mathsf{unpad}(\mathsf{mapfst}(c_1) \parallel [x_1]) = \mathsf{None}$, yielding a contradiction to the fourth premise. □

6.5 Lemmas on findseq

We will now discuss several lemmas concerning the function findseq. Many of these are immediately relevant to enable the automated tools used in the EasyCrypt proof to solve arising side conditions, since the function findseq is directly used in the games presented in Chapter 5 – by contrast, the lemmas discussed in the previous section are more often relevant as a basis to prove the lemmas in this section, or in later sections.

The first statement presented here is a more powerful variant of the completeness of the function findseq (see Axiom 6.14). Indeed, the definition of findseq's completeness postulates only that whenever there is a chain in a well-formed map T ending in (x, y), then $\mathsf{findseq}(x, y, T)$ will find *something*. Now, the uniqueness of chains presented in the previous section gives us that there can be but a single chain ending in (x, y) in a well-formed map T. Therefore, it can be derived from both the soundness and completeness of findseq that the chain returned by $\mathsf{findseq}(x, y, T)$ is indeed the very chain ending in (x, y).

Lemma 6.34 (Completeness and uniqueness of findseq). *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* x *a block,* y *a chaining value, and* $c : (\{0,1\}^k \times \{0,1\}^n)$ list *a list. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{ischain}(T, \mathsf{IV}, c, y)$
4. $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, T) = \mathsf{Some}(c)$.

Proof. By the completeness of findseq, we obtain that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$. By the soundness of findseq, this implies $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T)}, y)$. Now, Lemma 6.32 applies and we obtain $\pi_{\mathsf{findseq}(x,y,T)} = c$, which implies $\mathsf{findseq}(x, y, T) = \mathsf{Some}(c)$. □

Dually, we observe that if there is a chain ending in (x, y) in a well-formed map T which does *not* correspond to a valid padding, then $\mathsf{findseq}(x, y, T)$ will return nothing at all.

Lemma 6.35. *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* x *a block,* y *a chaining value, and* $c : (\{0,1\}^k \times \{0,1\}^n)$ list *a list. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{ischain}(T, \mathsf{IV}, c, y)$
4. $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) = \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, T) = \mathsf{None}$.

Proof. Assume for contradiction that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$. Then, by the soundness of $\mathsf{findseq}$, we get $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T)}, y)$ and $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T)}) \parallel [x]) \neq \mathsf{None}$. Furthermore, Lemma 6.32 implies that $\pi_{\mathsf{findseq}(x,y,T)} = c$. Therefore, we obtain that $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) \neq \mathsf{None}$, which yields the contradiction. □

We also show that the padding associated to a chain returned by findseq uniquely determines this chain (similarly as in Lemma 6.28), and further uniquely determines the arguments of findseq that must have been used to return this chain. Therefore, any complete chain can only be retrieved using uniquely defined arguments to the function findseq.

Lemma 6.36. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, x_1, x_2 be blocks, and y_1, y_2 be chaining values. Assume the following:*

1. $\mathsf{findseq}(x_1, y_1, T) \neq \mathsf{None}$
2. $\mathsf{findseq}(x_2, y_2, T) \neq \mathsf{None}$
3. $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_1,y_1,T)}) \parallel [x_1]) = \mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_2,y_2,T)}) \parallel [x_2])$

Then it holds that $(x_1, y_1) = (x_2, y_2)$.

Proof. First, since we have that the function unpad is injective (see Axiom 6.9) and that $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_1,y_1,T)}) \parallel [x_1]) \neq \mathsf{None}$ (by the soundness of findseq), we obtain that $\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_1,y_1,T)}) \parallel [x_1] = \mathsf{mapfst}(\pi_{\mathsf{findseq}(x_2,y_2,T)}) \parallel [x_2]$. Therefore, it holds that $\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_1,y_1,T)}) = \mathsf{mapfst}(\pi_{\mathsf{findseq}(x_2,y_2,T)})$ and $x_1 = x_2$. Additionally, the soundness of findseq implies that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x_1,y_1,T)}, y_1)$ and $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x_2,y_2,T)}, y_2)$. Hence, Lemma 6.28 applies and we obtain $\pi_{\mathsf{findseq}(x_1,y_1,T)} = \pi_{\mathsf{findseq}(x_2,y_2,T)}$ and $y_1 = y_2$, which finishes the proof. □

The following lemma makes explicit a relation between the function findseq and the predicate ischain. Namely, as already discussed in Section 6.1.4, the function findseq searches in a map T for a specific complete chain, but a complete chain does not require its last element to be in T (see Definition 5.2); by contrast, the predicate ischain requires all the elements of its list argument to be in the domain of T. Thus, for the predicate $\mathsf{ischain}(T, \mathsf{IV}, c, z)$ to hold for a complete chain c in T, we need that the last element of c is also in the domain of T. We state this in a slightly generalized fashion.

Lemma 6.37. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x be a block, and y be a chaining value. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$
3. $(x, y) \in \mathsf{dom}(T)$

Then it holds that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')} \parallel [(x, y)], T[(x, y)])$.

Proof. To show the conclusion of the statement, we show the three necessary and sufficient conditions required by Lemma 6.21:

(i) $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')}, y)$: The soundness of the function findseq immediately implies that $\mathsf{ischain}(T', \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')}, y)$. Therefore, we obtain this subgoal by applying Lemma 6.25.

(ii) $(x, y) \in \mathsf{dom}(T)$: We require this as a premise in the main statement.

(iii) $\mathsf{ischain}(T, T[(x, y)], \mathsf{nil}, T[(x, y)])$: This is obvious since this statement is equivalent to $T[(x, y)] = T[(x, y)]$ by Definition 6.12.

□

Similarly, if $\mathsf{findseq}(x, y, T')$ is successful in a well-formed map $T' \subseteq T$, and there is an element in T which maps to y, then this element must be the penultimate element of the chain found by findseq, and therefore it is also in the domain of T'.

Lemma 6.38. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x, x' be blocks, and y, y' be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x', y') \in \mathsf{dom}(T)$
5. $T[(x', y')] = y$
6. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

Then it holds that $(x', y') \in \mathsf{dom}(T')$.

Proof. First, by the soundness of findseq we know that $\mathsf{ischain}(T', \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')}, y)$. Next, we will show that $\pi_{\mathsf{findseq}(x,y,T')}$ has the form $c \parallel [(x', y')]$, for an appropriate list c. To do this, we perform a case distinction on the structure of the list $\pi_{\mathsf{findseq}(x,y,T')}$.

Case $\pi_{\mathsf{findseq}(x,y,T')} = \mathsf{nil}$**.** Here we obtain a contradiction. Indeed, in this case it holds $\mathsf{ischain}(T', \mathsf{IV}, \mathsf{nil}, y)$ and thus $y = \mathsf{IV}$. Recall that we have $(x', y') \in \mathsf{dom}(T)$ and $T[(x', y')] = y = \mathsf{IV}$, contradicting the assumption $\mathsf{IV} \notin \mathsf{ran}(T)$.

Case $\pi_{\mathsf{findseq}(x,y,T')} = al \parallel [a]$**.** It suffices to show that $a = (x', y')$. From the assumption $\mathsf{ischain}(T', \mathsf{IV}, al \parallel [a], y)$ we conclude (using Lemma 6.21) that $a \in \mathsf{dom}(T')$ and $\mathsf{ischain}(T', T'[a], \mathsf{nil}, y)$, implying $T'[a] = y$ by Definition 6.12. Therefore, it also holds $a \in \mathsf{dom}(T)$ and $T[a] = y$. So by the injectivity of T, we conclude $(x', y') = a$.

Now we know that $\pi_{\mathsf{findseq}(x,y,T')}$ has the form $c \parallel [(x', y')]$. We fix c, and finally obtain $\mathsf{ischain}(T', \mathsf{IV}, c \parallel [(x', y')], y)$. By Lemma 6.21, we therefore obtain $(x', y') \in \mathsf{dom}(T')$. □

In the next few lemmas, we will compare the behavior of the calls $\mathsf{findseq}(x, y, T')$ and $\mathsf{findseq}(x, y, T)$ where $T' \subseteq T$. More precisely, we are interested to see under which conditions we have $\mathsf{findseq}(x, y, T') = \mathsf{findseq}(x, y, T)$, or if not, what conclusions we can draw from this. We first introduce the notion of a *completing* update.

Definition 6.39 (Completing update)**.** Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, x, x' be blocks, and y, y' and z' be chaining values. Assume $\mathsf{findseq}(x, y, T) = \mathsf{None}$ and $\mathsf{findseq}(x, y, \mathsf{upd}(T, (x', y'), z')) \neq \mathsf{None}$. Then the update of T with the pair $((x', y'), z')$ is called *completing* for the chain ending in (x, y).

We show below that when an update is completing for some chain in a well-formed map, then the element which the map was updated with is part of this chain.

Lemma 6.40. *Let* $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a map,* x, x' *be blocks, and* y, y' *and* z' *be chaining values. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{findseq}(x, y, T) = \mathsf{None}$
4. $\mathsf{findseq}(x, y, \mathsf{upd}(T, (x', y'), z')) \neq \mathsf{None}$

Then it holds that $(x', y') \in \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}$.

Proof. First, note that the fourth premise implies by the soundness of the function $\mathsf{findseq}$ that $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}) \parallel [x]) \neq \mathsf{None}$. Therefore, it must hold that $\neg\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}, y)$: Assume for contradiction that it held true that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}, y)$, then the completeness of $\mathsf{findseq}$ would imply that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$, contradicting the third premise. Thus, it must indeed hold that $\neg\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}, y)$. On the other hand, the fourth premise implies that $\mathsf{ischain}(\mathsf{upd}(T, (x', y'), z'), \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}, y)$ by the soundness of $\mathsf{findseq}$. Hence we can apply Lemma 6.27 to conclude $(x', y') \in \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x',y'),z'))}$. □

As mentioned above, we will now see under which conditions we can indeed say that $\mathsf{findseq}(x, y, T') = \mathsf{findseq}(x, y, T)$. Indeed, it is often relevant in the proof outlined in Section 5.2 to know that whenever $\mathsf{findseq}$ finds a complete chain for some arguments x, y in a well-formed map T', then it will still find the same chain after benign updates to T'. Similarly, if it does not find such a complete chain in T', then it should still not find any chain after benign updates that are not completing. We will now see several lemmas that describe different conditions under which we expect the behavior of $\mathsf{findseq}$ to be the same before and after updates to a map.

We begin with the following helper lemma: Whenever the function call $\mathsf{findseq}(x, y, T')$ finds a chain in a well-formed map T' for a block x and chaining value y, then it will still find the same chain in a well-formed map $T \supseteq T'$. This lemma is also interesting on its own. For instance, in the transformation described in Section 5.2, this lemma intuitively implies that any complete chain in the map T' maintained by the simulator is also a complete chain in the bigger map T maintained by the FIL-RO f (resp. $f_{\mathbf{bad}}$) in the game $\mathsf{G}_{\mathsf{real}'}$.

Lemma 6.41. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x *be a block, and* y *be a chaining value. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, T) = \mathsf{findseq}(x, y, T')$.

Proof. By the soundness of function $\mathsf{findseq}$, we find $\mathsf{ischain}(T', \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')}, y)$ and $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T')}) \parallel [x]) \neq \mathsf{None}$. By Lemma 6.25, the former moreover implies that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')}, y)$. Hence, by the completeness and uniqueness of findseq (see Lemma 6.34), we conclude $\mathsf{findseq}(x, y, T) = \mathsf{Some}(\pi_{\mathsf{findseq}(x,y,T')})$, which implies the main conclusion. □

Using the above lemma, we can now easily derive a corollary which explicitly imposes a single benign update, but allows arbitrarily many of them (as we speak about a well-formed map $T \supseteq T'$, which may be obtained from T' by a number of appropriate updates).

Corollary 6.42. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x, x' be blocks, and y, y' and z' be a chaining values. Assume the following:*
1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x', y') \notin \mathsf{dom}(T)$
5. $z' \notin \mathsf{ran}(T)$
6. $z' \neq \mathsf{IV}$
7. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, \mathsf{upd}(T, (x', y'), z')) = \mathsf{findseq}(x, y, T')$.

Proof. The corollary quickly follows from Lemma 6.41. To be able to apply the latter lemma, we need to fulfill its premises by showing that (i) $\mathsf{Inclusion}(T', \mathsf{upd}(T, (x', y'), z'))$, (ii) $\mathsf{Injective}(\mathsf{upd}(T, (x', y'), z'))$ and (iii) $\mathsf{IV} \notin \mathsf{ran}(\mathsf{upd}(T, (x', y'), z'))$. The first condition is easily fulfilled by the first and fourth premises. The second condition is fulfilled by the the second and fifth premises. Finally, the last condition is fulfilled by the third and sixth premises. □

In the above corollary, we spoke about the *same* parameters x and y for the call of findseq in the premises and the conclusion. We will now state two lemmas stating sensible conditions under which findseq will yield the same return values for *arbitrary* parameters x', y' in the conclusion. The first considers the case where a map T' is updated with a value that corresponds to the last element of an already complete chain in T'. It makes use of Lemma 6.33: this lemma essentially stated that the end of any complete chain in a map T' may not be part of any other complete chain in T'. Therefore, if we know that $\mathsf{findseq}(x, y, T')$ is successful, and therefore (x, y) is the end of a complete chain in T', then extending the domain of T' with the element (x, y) will not influence the return value of findseq for arbitrary parameters x', y': Neither in the case $(x, y) = (x', y')$, since the end of a complete chain is not required to be in T' for findseq to be successful; nor in the case $(x, y) \neq (x', y')$, since (x, y) cannot be part of any other chain in T'.

Lemma 6.43. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x, x' be blocks, and y, y' be chaining values. Assume the following:*
1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

5. $(x, y) \in \mathsf{dom}(T)$

Then it holds that $\mathsf{findseq}(x', y', \mathsf{upd}(T', (x, y), T[(x, y)])) = \mathsf{findseq}(x', y', T')$.

Proof. First, we perform a case distinction on the return value of $\mathsf{findseq}(x', y', T')$.

Case $\mathsf{findseq}(x', y', T') \neq \mathsf{None}$**.** In this case, the conclusion follows from Lemma 6.41, since it is easily seen that the update with the pair $((x, y), T[(x, y)])$ is benign, i.e. it holds $\mathsf{Inclusion}(T', \mathsf{upd}(T', (x, y), T[(x, y)]))$, and the map $\mathsf{upd}(T', (x, y), T[(x, y)])$ is well-formed.

Case $\mathsf{findseq}(x', y', T') = \mathsf{None}$**.** In this case, we have to show that the update of the map T' with the pair $((x, y), T[(x, y)])$ is not completing for any chain in T', i.e. it also holds that $\mathsf{findseq}(x', y', \mathsf{upd}(T', (x, y), T[(x, y)])) = \mathsf{None}$. For the sake of readability in the remainder of the proof, let us define $T'' := \mathsf{upd}(T', (x, y), T[(x, y)])$. Note that $T' \subseteq T'' \subseteq T$, and that all these maps are well-formed. Now, assume for contradiction that $\mathsf{findseq}(x', y', T'') \neq \mathsf{None}$. Then, on the one hand, the update of T' was completing for the chain ending in (x', y'): We can use Lemma 6.40 to conclude that $(x, y) \in \pi_{\mathsf{findseq}(x', y', T'')}$. On the other hand, we also know (fourth premise) that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$. Therefore, by Lemma 6.41 it also holds that $\mathsf{findseq}(x, y, T'') \neq \mathsf{None}$. Hence, by the soundness of $\mathsf{findseq}$ it holds that $\mathsf{ischain}(T'', \mathsf{IV}, \pi_{\mathsf{findseq}(x, y, T'')}, y)$ and that $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x, y, T'')}) \mathbin{\|} [x]) \neq \mathsf{None}$. Furthermore, since we assume that $\mathsf{findseq}(x', y', T'') \neq \mathsf{None}$, it similarly holds that $\mathsf{ischain}(T'', \mathsf{IV}, \pi_{\mathsf{findseq}(x', y', T'')}, y')$ and that $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x', y', T'')}) \mathbin{\|} [x']) \neq \mathsf{None}$. Hence, Lemma 6.33 applies and we obtain $(x, y) \notin \pi_{\mathsf{findseq}(x', y', T'')}$, which yields the contradiction. Thus, we must indeed have $\mathsf{findseq}(x', y', T'') = \mathsf{None}$, which finishes the proof.

□

In a corollary, we state the restriction that the update has to be benign in a more intuitive fashion, similarly as in Corollary 6.42.

Corollary 6.44. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x, x' *be blocks, and* y, y' *and* z *be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x, y) \notin \mathsf{dom}(T)$
5. $z \notin \mathsf{ran}(T)$
6. $z \neq \mathsf{IV}$
7. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x', y', \mathsf{upd}(T', (x, y), z)) = \mathsf{findseq}(x', y', T')$.

Proof. By instantiating Lemma 6.43 with the two maps T' and $\mathsf{upd}(T', (x, y), z)$, we obtain the conclusion $\mathsf{findseq}(x', y', \mathsf{upd}(T', (x, y), \mathsf{upd}(T', (x, y), z)[(x, y)])) = \mathsf{findseq}(x', y', T')$, which is equivalent to the conclusion of this corollary, as clearly $\mathsf{upd}(T', (x, y), z)[(x, y)] = z$. We are left to fulfill the premises of Lemma 6.43. This is not difficult:

(i) $\mathsf{Inclusion}(T', \mathsf{upd}(T', (x, y), z))$: By the premises *1* and *4*.
(ii) $\mathsf{Injective}(\mathsf{upd}(T', (x, y), z))$: By the premises *1*, *2* and *5*.
(iii) $\mathsf{IV} \notin \mathsf{ran}(\mathsf{upd}(T', (x, y), z))$: By the premises *1*, *3* and *6*.
(iv) $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$: By premise *7*.
(v) $(x, y) \in \mathsf{dom}(\mathsf{upd}(T', (x, y), z))$: Trivial.

□

Lemma 6.43 considered the situation where an update to a map T with a pair $((x, y), z)$ did not complete a chain in T by asserting that (x, y) is already the end of some complete chain in T. The next lemma also ensures that the update is not completing, however it does not require (x, y) to be the end of some complete chain; instead, it requires that (x, y) does not fill a gap in an incomplete chain ending in (x', y'). More precisely, it requires the update to be benign along with the usual well-formedness conditions for T. Additionally, it also requires that $((x, y), z)$ is not a fixed-point (in the transformation described in Section 5.2, this is guaranteed by the failure event $\mathbf{bad}_2$), and that (x, y) is not the predecessor of another element of a chain in T (this is guaranteed by the failure events $\mathbf{bad}_2$ and $\mathbf{bad}_3$). Now, observe that if (x, y) is the penultimate element of an incomplete chain ending in (x', y'), and (x', y') is not in the domain of $\mathsf{upd}(T, (x, y), z)$, then the update with the pair (x, y) may indeed be completing for the chain ending in (x', y') without violating any of the other conditions (and therefore the results of the calls to findseq in the conclusion of the following lemma would differ); the last assumption of the lemma allows us to exclude this case.

Lemma 6.45. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x, x' *be blocks, and* y, y' *and* z *be chaining values. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $(x, y) \notin \mathsf{dom}(T)$
4. $z \notin \mathsf{ran}(T)$
5. $z \neq \mathsf{IV}$
6. $z \neq y$
7. $\nexists x_0.\ (x_0, z) \in \mathsf{dom}(T)$
8. $(x', y') \in \mathsf{dom}(\mathsf{upd}(T, (x, y), z))$

Then it holds that $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) = \mathsf{findseq}(x', y', T)$.

Proof. In the case where $\mathsf{findseq}(x', y', T) \neq \mathsf{None}$, the statement immediately follows from Corollary 6.42. Therefore, let us concentrate on the case where $\mathsf{findseq}(x', y', T) = \mathsf{None}$. Then, we need to show that also $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) = \mathsf{None}$; that is, the update of T is not completing for any chain. We prove the claim by contradiction: Assume that $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) \neq \mathsf{None}$. We expect (and show below) that this yields a contradiction to the premises of the statement. Since we assume $\mathsf{findseq}(x', y', T) = \mathsf{None}$ and $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) \neq \mathsf{None}$, Lemma 6.40 applies and we conclude $(x, y) \in \pi_{\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z))}$. Furthermore, by the soundness of findseq we obtain that $\mathsf{ischain}(\mathsf{upd}(T, (x, y), z), \mathsf{IV}, \pi_{\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z))}, y')$. For the remainder of the proof, let us define $c := \pi_{\mathsf{findseq}(x',y',\mathsf{upd}(T,(x,y),z))}$. Now, using only the premises $(x,y) \in c$ and $\mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\mathsf{IV},c,y')$ and the premises *6* through *8* of the main statement, we derive a contradiction using an induction over c, generalizing the constant IV as an arbitrary value $\hat{y}$. That is, under the premises *6* through *8* we prove the following statement:

$$\forall \hat{y}.\ \mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\hat{y},c,y') \Rightarrow (x,y) \in c \Rightarrow \bot$$

Base case: $c = \mathsf{nil}$**.** This yields a contradiction immediately since we assume $(x,y) \in \mathsf{nil}$.

Induction step: $c = a{::}al$**.**

Induction hypothesis:

$$\forall \hat{y}.\ \mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\hat{y},al,y') \Rightarrow (x,y) \in al \Rightarrow \bot$$

We treat the cases $a \neq (x,y)$ and $a = (x,y)$ separately.

Case $a \neq (x,y)$**.** This is the easier case, since in this case we can use the induction hypothesis to obtain the contradiction. Indeed, the two premises of the induction hypothesis are easily fulfilled: we have $\mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\hat{y},a{::}al,y')$, which immediately implies $\mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\mathsf{upd}(T,(x,y),z)[a],al,y')$ by Lemma 6.21. Furthermore, since we have $a \neq (x,y)$ and $(x,y) \in a{::}al$, we conclude $(x,y) \in al$.

Case $a = (x,y)$**.** This case is slightly more difficult, since we cannot use the induction hypothesis. However, since $a = (x,y)$, we now have the assumption $\mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\hat{y},(x,y){::}al,y')$. We will now show that this yields a contradiction to the premises *6* through *8*. To achieve this, we need another case distinction to distinguish whether there is another element in this chain after the element (x,y).

Case $al = \mathsf{nil}$**.** By Definition 6.12, we obtain that $\mathsf{upd}(T,(x,y),z)[(x,y)] = y'$, which immediately implies $z = y'$. Now, we quickly obtain a contradiction using a last case distinction.

Case $(x,y) = (x',y')$**.** Then $y' = y$, and thus $z = y$, contradiction to the sixth premise.

Case $(x,y) \neq (x',y')$**.** We have that $(x',y') \in \mathsf{dom}(\mathsf{upd}(T,(x,y),z))$ (eighth premise), and since $(x,y) \neq (x',y')$ we know $(x',y') \in \mathsf{dom}(T)$. Since $z = y'$, we obtain $(x',z) \in \mathsf{dom}(T)$, contradiction to the seventh premise.

Case $al = a'{::}al'$**.** By Definition 6.12, we get $\mathsf{upd}(T,(x,y),z)[(x,y)] = \mathsf{snd}(a')$, implying $z = \mathsf{snd}(a')$, and $\mathsf{ischain}(\mathsf{upd}(T,(x,y),z),\mathsf{snd}(a'),a'{::}al',y')$. Again, we perform a last case distinction to derive the contradiction.

Case $(x,y) = a'$**.** Then $\mathsf{snd}(a') = y$, and thus $z = y$, contradiction to the sixth premise.

Case $(x, y) \neq a'$. We have $\mathsf{ischain}(\mathsf{upd}(T, (x, y), z), \mathsf{snd}(a'), a'{::}al', y')$. By Lemma 6.21 this implies $a' \in \mathsf{dom}(\mathsf{upd}(T, (x, y), z))$. Since $(x, y) \neq a'$, we conclude $a' \in \mathsf{dom}(T)$. Because $z = \mathsf{snd}(a')$, we obtain $(\mathsf{fst}(a'), z) \in \mathsf{dom}(T)$, contradiction to the seventh premise.

Finally, our original assumption that $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) \neq \mathsf{None}$ led to a contradiction. Thus, it must indeed hold that $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) = \mathsf{None}$ and therefore $\mathsf{findseq}(x', y', \mathsf{upd}(T, (x, y), z)) = \mathsf{findseq}(x', y', T)$, which finishes the proof. □

6.6 Lemmas on valid_chain

The predicate valid_chain (see Definition 6.15) is closely related to the function findseq. Namely, $\mathsf{valid\_chain}(T, c \parallel [(x, y)])$ means that $c \parallel [(x, y)]$ is a complete chain in T and additionally $(x, y) \in \mathsf{dom}(T)$. By contrast, $\mathsf{findseq}(x, y, T) = \mathsf{Some}(c)$ only means that $c \parallel [(x, y)]$ is a complete chain in T, but the element (x, y) is not required to be in the domain of T. Therefore the former statement (involving valid_chain) is a slightly stronger statement than the latter statement (involving findseq); we use this predicate to speak about complete chains that are entirely contained in a map, i.e. in particular their last element is also in the domain of T.

We begin by observing two basic properties concerning the predicate valid_chain. First, we formally state the relation between findseq and valid_chain described above in a generalized fashion as follows.

Lemma 6.46. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x be a block, and y be a chaining value. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$
3. $(x, y) \in \mathsf{dom}(T)$

Then it holds that $\mathsf{valid\_chain}(T, \pi_{\mathsf{findseq}(x,y,T')} \parallel [(x, y)])$.

Proof. By Lemma 6.37, we obtain $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T')} \parallel [(x, y)], T[(x, y)])$. Furthermore, by the soundness of findseq we know $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T')}) \parallel [x]) \neq \mathsf{None}$. Thus it holds $\mathsf{valid\_chain}(T, \pi_{\mathsf{findseq}(x,y,T')} \parallel [(x, y)])$ by the definition of valid_chain. □

We also observe that the predicate cannot hold for a non-empty list in the empty map (clearly, it never holds for the empty list because of Axiom 6.6). This is needed in the game transition outlined in Section 5.2 to conclude that there are no valid chains in the initial games (i.e., before the simulator has made any calls).

Lemma 6.47. *Let $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list and xy be a pair of a block and a chaining value. Then it holds that* $\neg\mathsf{valid\_chain}(\emptyset, c \parallel [xy])$.

Proof. Assume $\mathsf{valid\_chain}(\emptyset, c \parallel [xy])$, then we know $\mathsf{ischain}(\emptyset, \mathsf{IV}, c \parallel [xy], z)$ for some z. By Lemma 6.21 we obtain $xy \in \mathsf{dom}(\emptyset)$, yielding a contradiction. □

We also state an alternative variant of the completeness and uniqueness of the function $\mathsf{findseq}$ (see Lemma 6.34), using the predicate $\mathsf{valid\_chain}$.

Lemma 6.48 (Completeness and uniqueness of $\mathsf{findseq}$ under $\mathsf{valid\_chain}$)**.** *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, x a block, y a chaining value, and $c : (\{0,1\}^k \times \{0,1\}^n)$ list a list. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $\mathsf{valid\_chain}(T, c \parallel [(x,y)])$

Then it holds that $\mathsf{findseq}(x, y, T) = \mathsf{Some}(c)$.

Proof. We obtain from the premise $\mathsf{valid\_chain}(T, c \parallel [(x,y)])$ that for some z it holds that $\mathsf{ischain}(T, \mathsf{IV}, c \parallel [(x,y)], z)$, and that $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x]) \neq \mathsf{None}$. The former implies $\mathsf{ischain}(T, \mathsf{IV}, c, y)$ by Lemma 6.21. Hence, the conclusion follows by the completeness and uniqueness of $\mathsf{findseq}$ (Lemma 6.34). □

Finally, we consider the behavior of predicate $\mathsf{valid\_chain}$ when the map that it relates to gets updated. The first lemma simply states that we can extend a map T with an element not present in its domain without afflicting the validity of the predicate.

Lemma 6.49. *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, xy be a pair of a block and a chaining value, and z be a chaining value. Assume:*

1. $\mathsf{valid\_chain}(T, c)$
2. $xy \notin \mathsf{dom}(T)$

Then it holds that $\mathsf{valid\_chain}(\mathsf{upd}(T, xy, z), c)$.

Proof. We have to show $\exists z'.\ \mathsf{ischain}(\mathsf{upd}(T, xy, z), \mathsf{IV}, c, z') \land \mathsf{unpad}(\mathsf{mapfst}(c)) \neq \mathsf{None}$. From $\mathsf{valid\_chain}(T, c)$ we obtain a z' such that $\mathsf{ischain}(T, \mathsf{IV}, c, z')$ and $\mathsf{unpad}(\mathsf{mapfst}(c)) \neq \mathsf{None}$. It remains only to show $\mathsf{ischain}(\mathsf{upd}(T, xy, z), \mathsf{IV}, c, z')$. This follows quickly from Lemma 6.24; to apply it, we only need to show $xy \notin c$. But this is clear: If we had $xy \in c$, then by Corollary 6.23 it would hold $xy \in \mathsf{dom}(T)$, yielding a contradiction. □

The last two lemmas of this section are variants of Lemma 6.45. Intuitively, we would like to say that when a well-formed map T gets updated with a pair $((x,y), z)$, and this update is benign and not completing, then for any list c, we have $\mathsf{valid\_chain}(T, c)$ if and only if $\mathsf{valid\_chain}(\mathsf{upd}(T, (x,y), z), c)$. However, we need an additional condition for this to hold: Indeed, it could happen that (x,y) is the *last* element of c. In this case, the latter statement may hold, but the former may not, even though the update was not completing. This is the main difference to Lemma 6.45; there, an additional assumption was needed to ensure that the element (x,y) was not the *penultimate* element of the chain returned by $\mathsf{findseq}$. Both of the following two lemmas are needed at different points in the EasyCrypt proof outlined in Section 5.2. The first uses the same premises as Lemma 6.45, and additionally assumes $\mathsf{findseq}(x, y, T) = \mathsf{None}$, thereby restricting the statement to the case where (x,y) is not the last element of a complete chain in T.

Lemma 6.50. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x, x' be blocks, and y, y' and z be chaining values. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $(x,y) \notin \mathsf{dom}(T)$
4. $z \notin \mathsf{ran}(T)$
5. $z \neq \mathsf{IV}$
6. $z \neq y$
7. $\nexists x_0.\ (x_0, z) \in \mathsf{dom}(T)$
8. $(x', y') \in \mathsf{dom}(\mathsf{upd}(T, (x,y), z))$
9. $\mathsf{findseq}(x, y, T) = \mathsf{None}$

Then it holds that $\mathsf{valid\_chain}(\mathsf{upd}(T,(x,y),z), c \parallel [(x',y')]) \Leftrightarrow \mathsf{valid\_chain}(T, c \parallel [(x',y')])$.

Proof. The direction "$\Leftarrow$" of the conclusion follows immediately from Lemma 6.49. The interesting case is the direction "$\Rightarrow$": Assume that $\mathsf{valid\_chain}(\mathsf{upd}(T,(x,y),z), c \parallel [(x',y')])$. Then, we show that $\mathsf{valid\_chain}(T, c \parallel [(x',y')])$. First, we apply the completeness and uniqueness of findseq under valid_chain to conclude that $\mathsf{findseq}(x', y', \mathsf{upd}(T,(x,y),z)) \neq \mathsf{None}$ and $\pi_{\mathsf{findseq}(x',y',\mathsf{upd}(T,(x,y),z))} = c$ (note that $\mathsf{upd}(T,(x,y),z)$ is still well-formed since the update is benign). Now, since Lemma 6.45 implies $\mathsf{findseq}(x', y', \mathsf{upd}(T,(x,y),z)) = \mathsf{findseq}(x', y', T)$, we also know that $\mathsf{findseq}(x', y', T) \neq \mathsf{None}$, and what we need to show is $\mathsf{valid\_chain}(T, \pi_{\mathsf{findseq}(x',y',T)} \parallel [(x',y')])$. We perform a case distinction.

Case $(x,y) = (x',y')$**.** This is the case where the new element used for the update is the last element of a complete chain. It yields a contradiction to the ninth premise, since we have $\mathsf{findseq}(x', y', T) \neq \mathsf{None}$.

Case $(x,y) \neq (x',y')$**.** In this case, the premise $(x',y') \in \mathsf{dom}(\mathsf{upd}(T,(x,y),z))$ implies $(x',y') \in \mathsf{dom}(T)$. Then, the claim follows by Lemma 6.46.

□

The next lemma uses the same premises as Lemma 6.45, except that the premise $(x',y') \in \mathsf{dom}(\mathsf{upd}(T,(x,y),z))$ is strengthened as $(x',y') \in \mathsf{dom}(T)$: Thereby, as in the previous lemma we exclude the case where the new element is the last element of a complete chain.

Lemma 6.51. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x, x' be blocks, and y, y' and z be chaining values. Assume the following:*

1. $\mathsf{Injective}(T)$
2. $\mathsf{IV} \notin \mathsf{ran}(T)$
3. $(x,y) \notin \mathsf{dom}(T)$
4. $z \notin \mathsf{ran}(T)$
5. $z \neq \mathsf{IV}$
6. $z \neq y$
7. $\nexists x_0.\ (x_0, z) \in \mathsf{dom}(T)$
8. $(x', y') \in \mathsf{dom}(T)$

Then it holds that $\mathsf{valid\_chain}(\mathsf{upd}(T, (x, y), z), c \parallel [(x', y')]) \Leftrightarrow \mathsf{valid\_chain}(T, c \parallel [(x', y')])$.

Proof. Note that $(x', y') \in \mathsf{dom}(T)$ implies $(x', y') \in \mathsf{dom}(\mathsf{upd}(T, (x, y), z))$. Now, the proof is analogous to the proof of Lemma 6.50, except in the case where $(x, y) = (x', y')$: In this case, here we obtain a contradiction between the premises $(x, y) \notin \mathsf{dom}(T)$ and $(x', y') \in \mathsf{dom}(T)$. $\square$

6.7 Lemmas on set_bad3

In this section, we discuss lemmas concerning the predicate set_bad3, described in Section 6.1.5. More precisely, we will see under which conditions the maps T and T' in game $\mathsf{G}_{\mathsf{real}'}$ may be updated with the same pair $((x, y), z)$ (this may happen upon calls to the simulator f_q) such that $\mathsf{set\_bad3}(z_0, T', T) = \mathsf{set\_bad3}(z_0, \mathsf{upd}(T', (x, y), z), \mathsf{upd}(T, (x, y), z))$. Essentially, as for the Lemmas 6.43 and 6.45 discussed at the end of Section 6.5, we need to ensure that the update with $((x, y), z)$ is not completing for any chain in T. Firstly, we need that T is well-formed and that the update is benign. Then, we again consider two cases: either the pair (x, y) is the last element of a complete chain in T', but then (x, y) cannot be part of any other chain in T' (this is the first lemma); or the pair (x, y) does not correspond to the last element of any complete chain, but in this case we require that it does not fill a gap by requiring that the pair $((x, y), z)$ is not a fixed-point and that z was not already used as a chaining value in a query to the simulator (this is the second lemma); in both cases, the update cannot complete a chain.

In the first lemma, to ensure that the newly added value is the end of a complete chain, we require that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$, similarly as was the case in Corollary 6.44.

Lemma 6.52. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x *be a block, and* y, z *and* z_0 *be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x, y) \notin \mathsf{dom}(T)$
5. $z \notin \mathsf{ran}(T)$
6. $z \neq \mathsf{IV}$
7. $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$

Then it holds that $\mathsf{set\_bad3}(z_0, \mathsf{upd}(T', (x, y), z), \mathsf{upd}(T, (x, y), z)) \Leftrightarrow \mathsf{set\_bad3}(z_0, T', T)$.

Proof. For the sake of readability of the proof, let us define $T'_{xyz} := \mathsf{upd}(T', (x, y), z)$ and $T_{xyz} := \mathsf{upd}(T, (x, y), z)$.

We begin with the direction "$\Rightarrow$", which is the easier one. By unfolding Definition 6.16, we obtain the assumptions $z_0 \in \mathsf{ran}(T_{xyz})$, $T^{-1}_{xyz}[z_0] \notin \mathsf{dom}(T'_{xyz})$ and $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}_{xyz}[z_0]], T_{xyz})$. Furthermore, observe that $z_0 \neq z$: otherwise, we would have $T^{-1}_{xyz}[z_0] = (x, y)$, and the assumption $(x, y) \notin \mathsf{dom}(T'_{xyz})$ in itself is contradictory. Therefore, we respectively obtain $z_0 \in \mathsf{ran}(T)$, $T^{-1}[z_0] \notin \mathsf{dom}(T')$, and $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T_{xyz})$.

From the latter, we obtain $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T)$ using Lemma 6.49. Hence, we conclude $\mathsf{set\_bad3}(z_0, T', T)$.

For the direction "$\Leftarrow$", we have the assumptions $z_0 \in \mathsf{ran}(T)$, $T^{-1}[z_0] \notin \mathsf{dom}(T')$ and $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T)$. Again, observe that $z_0 \neq z$, since otherwise we obtain a contradiction to the premise $z \notin \mathsf{ran}(T)$. Similarly, note that $T^{-1}[z_0] \neq (x, y)$, since we have $z_0 \in \mathsf{ran}(T)$, but $(x, y) \notin \mathsf{dom}(T)$. We have to show the following:

(i) $z_0 \in \mathsf{ran}(T_{xyz})$: This is clear, since we have $z_0 \in \mathsf{ran}(T)$ and $z_0 \neq z$.

(ii) $T^{-1}_{xyz}[z_0] \notin \mathsf{dom}(T'_{xyz})$: This is also clear, since we have $T^{-1}[z_0] \notin \mathsf{dom}(T')$, $z_0 \neq z$ and $T^{-1}[z_0] \neq (x, y)$.

(iii) $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}_{xyz}[z_0]], T_{xyz})$: Note that $T^{-1}_{xyz}[z_0] = T^{-1}[z_0]$ since $z_0 \neq z$. To show the goal, assume for contradiction that $\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T_{xyz})$ for some list c. This implies $\mathsf{ischain}(T_{xyz}, \mathsf{IV}, c \parallel [T^{-1}[z_0]], z_0)$ and $\mathsf{unpad}(\mathsf{mapfst}(c \parallel [T^{-1}[z_0]])) \neq \mathsf{None}$. Now, on the one hand, we have $\neg\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T)$, and therefore it must hold that $\neg\mathsf{ischain}(T, \mathsf{IV}, c \parallel [T^{-1}[z_0]], z_0)$. Thus we can apply Lemma 6.27 to conclude $(x, y) \in c \parallel [T^{-1}[z_0]]$, and since $T^{-1}[z_0] \neq (x, y)$, we get $(x, y) \in c$. On the other hand, we have $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$, which implies $\mathsf{findseq}(x, y, T'_{xyz}) \neq \mathsf{None}$ by Corollary 6.42. Hence, we know from the soundness of findseq that $\mathsf{ischain}(T_{xyz}, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T_{xyz})}, y)$ and $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T_{xyz})}) \parallel [x]) \neq \mathsf{None}$. From $\mathsf{ischain}(T_{xyz}, \mathsf{IV}, c \parallel [T^{-1}[z_0]], z_0)$, we obtain by Lemma 6.21 that $\mathsf{ischain}(T_{xyz}, \mathsf{IV}, c, \mathsf{snd}(T^{-1}[z_0]))$. Recall that $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel \mathsf{fst}([T^{-1}[z_0]])) \neq \mathsf{None}$. Finally, Lemma 6.33 applies and we obtain $(x, y) \notin c$, which yields the contradiction.

□

In the second lemma, as mentioned above we ensure that the update of T with the pair $((x, y), z)$ is not completing by requiring that the pair $((x, y), z)$ is not a fixed-point and that z was not previously used as a chaining value, similarly as for Lemma 6.45.

Lemma 6.53. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x *be a block, and* y, z *and* z_0 *be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x, y) \notin \mathsf{dom}(T)$
5. $z \notin \mathsf{ran}(T)$
6. $z \neq \mathsf{IV}$
7. $z \neq y$
8. $\nexists x_0.\ (x_0, z) \in \mathsf{dom}(T)$

Then it holds that $\mathsf{set\_bad3}(z_0, \mathsf{upd}(T', (x, y), z), \mathsf{upd}(T, (x, y), z)) \Leftrightarrow \mathsf{set\_bad3}(z_0, T', T)$.

Proof. As in the previous lemma, let us define $T'_{xyz} := \mathsf{upd}(T', (x,y), z)$ and $T_{xyz} := \mathsf{upd}(T, (x,y), z)$ for the sake of readability. The proof for the direction "⇒" is analogous to the previous proof, and the proof for direction "⇐" is analogous for the cases **(i)** $z_0 \in \mathsf{ran}(T_{xyz})$ and **(ii)** $T^{-1}_{xyz}[z_0] \notin \mathsf{dom}(T'_{xyz})$. For the case **(iii)** $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}_{xyz}[z_0]], T_{xyz})$, we note that $T^{-1}_{xyz}[z_0] = T^{-1}[z_0]$ since $z_0 \neq z$. Again, assume for contradiction $\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T_{xyz})$ for some list c. From this, as in the previous proof, we eventually obtain $(x,y) \in c$. Therefore, c has the form $c_\ell \parallel (x,y){::}c_r$ for appropriate c_ℓ and c_r. Hence we have $\mathsf{valid\_chain}(c_\ell \parallel (x,y){::}c_r \parallel [T^{-1}[z_0]], T_{xyz})$, which implies $\mathsf{ischain}(T_{xyz}, \mathsf{IV}, c_\ell \parallel (x,y){::}c_r \parallel [T^{-1}[z_0]], z_0)$. We will now show that the latter statement leads to a contradiction to the premises *7* and *8*. We begin with a case distinction on the structure of the list c_r.

Case $c_r = \mathsf{nil}$**.** Using Lemma 6.21, we obtain $\mathsf{ischain}(T_{xyz}, T_{xyz}[(x,y)], [T^{-1}[z_0]], z_0)$, and therefore $\mathsf{ischain}(T_{xyz}, z, [T^{-1}[z_0]], z_0)$. By Definition 6.12, this implies $T^{-1}[z_0] \in \mathsf{dom}(T_{xyz})$ and $z = \mathsf{snd}(T^{-1}[z_0])$. We use another case distinction.

Case $T^{-1}[z_0] = (x,y)$**.** In this case we obtain $\mathsf{snd}(T^{-1}[z_0]) = y$ and therefore $z = y$, contradiction to the seventh premise.

Case $T^{-1}[z_0] \neq (x,y)$**.** In this case we obtain $T^{-1}[z_0] \in \mathsf{dom}(T)$, and therefore $(\mathsf{fst}(T^{-1}[z_0]), z) \in \mathsf{dom}(T)$, contradiction to the eighth premise.

Case $c_r = a{::}al$**.** By Lemma 6.21, it holds $\mathsf{ischain}(T_{xyz}, T_{xyz}[(x,y)], a{::}al \parallel [T^{-1}[z_0]], z_0)$, and therefore $\mathsf{ischain}(T_{xyz}, z, a{::}al \parallel [T^{-1}[z_0]], z_0)$. We perform another case distinction on the structure of al.

Case $al = \mathsf{nil}$**.** Then, we have $\mathsf{ischain}(T_{xyz}, z, a{::}T^{-1}[z_0]{::}\mathsf{nil}, z_0)$. By Definition 6.12, this implies $a \in \mathsf{dom}(T_{xyz})$ and $z = \mathsf{snd}(a)$. We use another case distinction.

Case $a = (x,y)$**.** Then it holds that $\mathsf{snd}(a) = y$, and therefore $z = y$, contradiction to the seventh premise.

Case $a \neq (x,y)$**.** Then it holds that $a \in \mathsf{dom}(T)$, and therefore $(\mathsf{fst}(a), z) \in \mathsf{dom}(T)$, contradiction to the eighth premise.

Case $al = a'{::}al'$**.** In this case we have $\mathsf{ischain}(T_{xyz}, z, a{::}a'{::}al' \parallel T^{-1}[z_0]{::}\mathsf{nil}, z_0)$, and the rest is analogous to the case where $al = \mathsf{nil}$.

Hence, the assumption $\mathsf{valid\_chain}(c \parallel [T^{-1}[z_0]], T_{xyz})$ leads to a contradiction and thus, since $T^{-1}_{xyz}[z_0] = T^{-1}[z_0]$ we must indeed have $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}_{xyz}[z_0]], T_{xyz})$, which finishes the last case, and the proof. □

6.8 Lemmas on Claim5

In the last section of this chapter, we will discuss a few lemmas concerning the predicate Claim5, described in Section 6.1.6. Recall that $\mathsf{Claim5}(T', T)$, for well-formed maps $T' \subseteq T$, means that for any element (x,y) that belongs to a chain in T and that is also in the domain of T', all elements of this chain preceding the element (x, y) are also in the domain of T', provided that there is no complete chain in T whose last element maps to the value y.

In a first lemma, we make this relation explicit: Assume that there is a complete chain of the form $c \parallel [(x', y'), (x, y)]$ in T, and (x', y') is also in the domain of T'. Then $c \parallel [(x', y')]$ is a chain in T', since we can reason inductively that all elements of c are in the domain of T'.

Lemma 6.54. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, $c : (\{0,1\}^k \times \{0,1\}^n)$ list be a list, x, x' be blocks, and y, y' be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $\mathsf{ischain}(T, \mathsf{IV}, c \parallel [(x', y')], y)$
5. $\mathsf{unpad}(\mathsf{mapfst}(c \parallel [(x', y')]) \parallel [x]) \neq \mathsf{None}$
6. $(x', y') \in \mathsf{dom}(T')$
7. $\mathsf{Claim5}(T', T)$

Then it holds that $\mathsf{ischain}(T', \mathsf{IV}, c \parallel [(x', y')], y)$.

Proof. We will prove the claim by a structural induction over the list c. However, the premise $\mathsf{unpad}(\mathsf{mapfst}(c \parallel [(x', y')]) \parallel [x]) \neq \mathsf{None}$ is not suitable for such an induction, since in general the fact that some chain is in the domain of unpad does not imply that the same holds for a suffix of this chain. Therefore, we replace this premise with a premise more suitable for induction; namely,

$$\forall x_0, y_0.\ (x_0, y_0) \in c \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}.$$

We eventually obtain the validity of this premise using Lemma 6.33: Fix an arbitrary element $(x_0, y_0) \in c$, and assume for contradiction that $\mathsf{findseq}(x_0, y_0, T) \neq \mathsf{None}$. By the soundness of function $\mathsf{findseq}$, this implies that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x_0, y_0, T)}, y_0)$ and that $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x_0, y_0, T)}) \parallel [x_0]) \neq \mathsf{None}$. Hence, Lemma 6.33 applies and we obtain $(x_0, y_0) \notin c \parallel [(x', y')]$, which yields the contradiction since we have $(x_0, y_0) \in c$.

We will now show the claim using an induction over c, generalizing the constant IV as an arbitrary value $\hat{y}$. That is, under the premises *1* through *3*, *6* and *7* we show:

$$\forall \hat{y}.\ \mathsf{ischain}(T, \hat{y}, c \parallel [(x', y')], y) \Rightarrow (\forall x_0, y_0.\ (x_0, y_0) \in c \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}) \Rightarrow \\ \mathsf{ischain}(T', \hat{y}, c \parallel [(x', y')], y)$$

Base case: $c = \mathsf{nil}$. Since we have $\mathsf{ischain}(T, \hat{y}, [(x', y')], y)$ we know by Definition 6.12 that $\hat{y} = y'$ and $T[(x', y')] = y$, and we have to show $(x', y') \in \mathsf{dom}(T')$, $\hat{y} = y'$ and $T'[(x', y')] = y$. This is trivial using premises *1* and *6*.

Induction step: $c = a{::}al$.

Induction hypothesis:

$$\begin{aligned}&\forall \hat{y}.\ \mathsf{ischain}(T, \hat{y}, al \parallel [(x', y')], y) \Rightarrow \\ &\qquad (\forall x_0, y_0.\ (x_0, y_0) \in al \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}) \Rightarrow \\ &\qquad\qquad \mathsf{ischain}(T', \hat{y}, al \parallel [(x', y')], y)\end{aligned}$$

We use a case distinction on the structure of the list al, and concentrate first on the case where $al = a'{::}al'$.

Case $al = a'{::}al'$**.** First, from the assumption $\mathsf{ischain}(T, \hat{y}, a{::}a'{::}al' \parallel [(x', y')], y)$ we immediately obtain by Definition 6.12 that $a \in \mathsf{dom}(T)$, $\hat{y} = \mathsf{snd}(a)$, $T[a] = \mathsf{snd}(a')$ and $\mathsf{ischain}(T, \mathsf{snd}(a'), a'{::}al' \parallel [(x', y')], y)$. The latter statement fulfills the first premise of the induction hypothesis (instantiated with $\hat{y} := \mathsf{snd}(a')$). The second premise of the induction hypothesis is also easily fulfilled: since we know that $\forall x_0, y_0.\ (x_0, y_0) \in a{::}al \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}$, clearly it holds that $\forall x_0, y_0.\ (x_0, y_0) \in al \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}$. Hence, we obtain from the induction hypothesis that $\mathsf{ischain}(T', \mathsf{snd}(a'), a'{::}al' \parallel [(x', y')], y)$. Since we also have $\hat{y} = \mathsf{snd}(a)$, to prove the goal $\mathsf{ischain}(T', \hat{y}, a{::}a'{::}al' \parallel [(x', y')], y)$ it only remains to show that $a \in \mathsf{dom}(T')$ and $T'[a] = \mathsf{snd}(a')$ (see Definition 6.12). Note that we have $\mathsf{Inclusion}(T', T)$, $a \in \mathsf{dom}(T)$ and $T[a] = \mathsf{snd}(a')$, so if $a \in \mathsf{dom}(T')$ then $T'[a] = \mathsf{snd}(a')$ immediately follows. Therefore we are only left to prove $a \in \mathsf{dom}(T')$. This is where we need the premise $\mathsf{Claim5}(T', T)$, which yields the conclusion since it implies that

$$\begin{aligned} a \in \mathsf{dom}(T) &\Rightarrow \\ T[a] = \mathsf{snd}(a') &\Rightarrow \\ a' \in \mathsf{dom}(T') &\Rightarrow \\ \mathsf{findseq}(\mathsf{fst}(a), \mathsf{snd}(a), T) = \mathsf{None} &\Rightarrow a \in \mathsf{dom}(T').\end{aligned}$$

We have already fulfilled the first and second premises. Moreover, since we have $\mathsf{ischain}(T', \mathsf{snd}(a'), a'{::}al' \parallel [(x', y')], y)$ then $a' \in \mathsf{dom}(T')$ follows immediately by Lemma 6.21. Finally, the last premise also holds since we have that $\forall x_0, y_0.\ (x_0, y_0) \in a{::}al \Rightarrow \mathsf{findseq}(x_0, y_0, T) = \mathsf{None}$, and we take $x_0 = \mathsf{fst}(a)$ and $y_0 = \mathsf{snd}(a)$. Thus, we obtain $a \in \mathsf{dom}(T')$, which finishes this case.

Case $al = \mathsf{nil}$**.** This case is analogous to the case $al = a'{::}al'$: Simply replace in the above proof the list $a'{::}al'$ with nil, respectively the variable a' (when on its own) with the pair (x', y').

□

In a sense, both the predicates set_bad3 and Claim5 ensure that queries of the distinguisher can only be made in order. The difference is that set_bad3 does this locally, i.e. for a given chaining value y used in a query from the distinguisher, while Claim5 guarantees a correct ordering of all queries globally, i.e. for all elements in map T'. Therefore we expect, and state formally below, the following relation between these two predicates. If Claim5(T', T) holds for given well-formed maps $T' \subseteq T$, then both maps can be updated with a pair $((x, y), z)$ such that Claim5(upd(T', $(x, y), z$), upd(T, $(x, y), z$)) still holds, provided that the update is benign, and event $\mathbf{bad}_3$ is not triggered for the chaining value y (i.e., it holds that ¬set_bad3(y, T', T)). This lemma constitutes an important element to derive that as long as none of the failure events is triggered, the statement Claim5(T', T) is an invariant of game $\mathsf{G}_{\mathsf{real'}}$, meaning that the distinguisher can only make queries in order up to the failure events – this is an essential property needed in the transformation described in Section 5.2.

Lemma 6.55. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x *be a block, and* y *and* z *be chaining values. Assume the following:*

1. Inclusion(T', T)
2. Injective(T)
3. IV $\notin$ ran(T)
4. $(x, y) \notin$ dom(T)
5. $z \notin$ ran(T)
6. $z \neq$ IV
7. ¬set_bad3(y, T', T)
8. Claim5(T', T)

Then it holds that Claim5(upd(T', $(x, y), z$), upd(T, $(x, y), z$)).

Proof. We want to show the following:

$$\begin{aligned}\forall (x_0, y_0), (x_0', y_0').\ & (x_0', y_0') \in \mathsf{dom}(\mathsf{upd}(T, (x, y), z)) \wedge \\ & (x_0, y_0) \in \mathsf{dom}(\mathsf{upd}(T', (x, y), z)) \wedge \\ & \mathsf{upd}(T, (x, y), z)[(x_0', y_0')] = y_0 \wedge \\ & \mathsf{findseq}(x_0', y_0', \mathsf{upd}(T, (x, y), z)) = \mathsf{None} \Rightarrow \\ & \qquad (x_0', y_0') \in \mathsf{dom}(\mathsf{upd}(T', (x, y), z)).\end{aligned}$$

Fix x_0, y_0, x_0' and y_0'. Assume that the four premises of the above statement hold. Then, we show that $(x_0', y_0') \in$ dom(upd(T', $(x, y), z$)). If $(x_0', y_0') = (x, y)$, this is trivial. Therefore assume that $(x_0', y_0') \neq (x, y)$. Now, since we have $(x_0, y_0) \in$ dom(upd(T', $(x, y), z$)), it either holds that $(x_0, y_0) = (x, y)$, or $(x_0, y_0) \in$ dom(T'). The basic idea of the proof is that in the former case, we obtain the conclusion because we know ¬set_bad3(y, T', T), and in the latter case, we obtain the conclusion because we know Claim5(T', T). We perform a case distinction.

Case $(x_0, y_0) = (x, y)$**.** Then, we obtain upd(T, $(x, y), z$)$[(x_0', y_0')] = y$. Since $(x_0', y_0') \neq (x, y)$, we furthermore conclude $(x_0', y_0') \in$ dom(T) and $T[(x_0', y_0')] = y$. Moreover, we also know findseq(x_0', y_0', T) = None: If we had findseq(x_0', y_0', T) $\neq$ None, we would obtain by Corollary 6.42 that $\mathsf{findseq}(x_0', y_0', \mathsf{upd}(T, (x, y), z)) \neq \mathsf{None}$, yielding a contradiction. We will now use the premise $\neg\mathsf{set\_bad3}(y, T', T)$ to conclude $(x_0', y_0') \in \mathsf{dom}(T')$. Unfolding the definition of set_bad3, this premise states:

$$\neg \big(y \in \mathsf{ran}(T) \wedge T^{-1}[y] \notin \mathsf{dom}(T') \wedge \forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[y]], T)\big)$$

First, note that since we know $(x_0', y_0') \in \mathsf{dom}(T)$ and $T[(x_0', y_0')] = y$, we obtain $T^{-1}[y] = (x_0', y_0')$. Therefore we can rewrite the above statement as

$$y \notin \mathsf{ran}(T) \vee (x_0', y_0') \in \mathsf{dom}(T') \vee \exists c.\ \mathsf{valid\_chain}(c \parallel [(x_0', y_0')], T)$$

However, we know that $y \in \mathsf{ran}(T)$ since $(x_0', y_0') \in \mathsf{dom}(T)$ and $T[(x_0', y_0')] = y$. Furthermore, if there existed a c such that $\mathsf{valid\_chain}(c \parallel [(x_0', y_0')], T)$ were true, then by the definition of valid_chain (Definition 6.15) and the completeness of findseq (Axiom 6.14) we would obtain $\mathsf{findseq}(x_0', y_0', T) \neq \mathsf{None}$, contradicting the fact that $\mathsf{findseq}(x_0', y_0', T) = \mathsf{None}$. Hence, it must hold $(x_0', y_0') \in \mathsf{dom}(T')$. Thus it also holds $(x_0', y_0') \in \mathsf{dom}(\mathsf{upd}(T', (x, y), z))$, which finishes this case.

Case $(x_0, y_0) \in \mathsf{dom}(T')$**.** Since $(x_0', y_0') \neq (x, y)$, we obtain that $(x_0', y_0') \in \mathsf{dom}(T)$ and $T[(x_0', y_0')] = y_0$. Furthermore, we also know that $\mathsf{findseq}(x_0', y_0', T) = \mathsf{None}$ by an analogous argument as in the previous case. Therefore, all premises of $\mathsf{Claim5}(T', T)$ are fulfilled, implying that $(x_0', y_0') \in \mathsf{dom}(T')$. Thus, it clearly also holds that $(x_0', y_0') \in \mathsf{dom}(\mathsf{upd}(T', (x, y), z))$, which finishes the proof.

□

The final two statements described in this section concern the behavior of the function findseq in the maps T' and T, where $T' \subseteq T$. Firstly, assume that $\mathsf{findseq}(x, y, T)$ is successful in the map T. Then, if $\mathsf{Claim5}(T', T)$ holds and event $\mathbf{bad}_3$ is not triggered by the simulator for a query using y as a chaining value, all elements of the chain returned by findseq must already be in the domain of T', and therefore $\mathsf{findseq}(x, y, T')$ will also be successful.

Lemma 6.56. *Let $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be maps, x be a block, and y be a chaining value. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $\neg\mathsf{set\_bad3}(y, T', T)$
5. $\mathsf{Claim5}(T', T)$
6. $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$.

Proof. First, since $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$, by the soundness of function findseq, it holds that $\mathsf{ischain}(T, \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,T)}, y)$ and $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,T)}) \parallel [x]) \neq \mathsf{None}$. First we will consider the case where $\pi_{\mathsf{findseq}(x,y,T)} = \mathsf{nil}$, which is easy. Next, we will prove the claim in the case where $\pi_{\mathsf{findseq}(x,y,T)}$ has the form $c \parallel [(x', y')]$.

Case $\pi_{\mathsf{findseq}(x,y,T)} = \mathsf{nil}$**.** From $\mathsf{ischain}(T, \mathsf{IV}, \mathsf{nil}, y)$, we get that $\mathsf{IV} = y$ by Definition 6.12, and therefore also $\mathsf{ischain}(T', \mathsf{IV}, \mathsf{nil}, y)$. Then, since furthermore $\mathsf{unpad}(\mathsf{mapfst}([x])) \neq \mathsf{None}$, and $T' \subseteq T$ is well-formed, the completeness of function findseq implies that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$.

Case $\pi_{\mathsf{findseq}(x,y,T)} = c \parallel [(x', y')]$**.** In this case, we perform a proof by contradiction: Assume that $\mathsf{findseq}(x, y, T') = \mathsf{None}$. Then, we show that event $\mathbf{bad}_3$ would have been triggered for the value y, i.e. we show that $\mathsf{set\_bad3}(y, T', T)$ holds true, contradicting the fourth assumption. Unfolding the definition of set_bad3, this means that we have to prove the following:

$$y \in \mathsf{ran}(T) \wedge T^{-1}[y] \notin \mathsf{dom}(T') \wedge \forall c.\ \neg\mathsf{valid\_chain}(c \parallel [T^{-1}[y]], T)$$

We prove each of the operands in the above conjunction below; the assumption $\mathsf{Claim5}(T', T)$ will only be needed for the proof of the second operand. In each of the three instances, we will need the following. Since $\pi_{\mathsf{findseq}(x,y,T)}$ has the form $c \parallel [(x', y')]$, we have $\mathsf{ischain}(T, \mathsf{IV}, c \parallel [(x', y')], y)$ and $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x', x]) \neq \mathsf{None}$. By Lemma 6.21, the former implies that $(x', y') \in \mathsf{dom}(T)$ and $T[(x', y')] = y$. Thus, we get $T^{-1}[y] = (x', y')$. We will henceforth speak only about (x', y').

(i) $y \in \mathsf{ran}(T)$: Since $(x', y') \in \mathsf{dom}(T)$ and $T[(x', y')] = y$, we immediately conclude $y \in \mathsf{ran}(T)$.

(ii) $(x', y') \notin \mathsf{dom}(T')$: Assume for contradiction that $(x', y') \in \mathsf{dom}(T')$. Then, we can make use of the assumption that $\mathsf{Claim5}(T', T)$: by Lemma 6.54, we obtain $\mathsf{ischain}(T', \mathsf{IV}, c \parallel [(x', y')], y)$. Since furthermore $\mathsf{unpad}(\mathsf{mapfst}(c) \parallel [x', x]) \neq \mathsf{None}$ and the map $T' \subseteq T$ is well-formed, by the completeness of findseq we obtain that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$, contradicting our assumption. Therefore, $(x', y') \notin \mathsf{dom}(T')$.

(iii) $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [(x', y')], T)$: Assume for contradiction that there exists a chain c' such that $\mathsf{valid\_chain}(c' \parallel [(x', y')], T)$ holds. That is, for some z' we have $\mathsf{ischain}(T', \mathsf{IV}, c' \parallel [(x', y')], z')$ and $\mathsf{unpad}(\mathsf{mapfst}(c') \parallel [x']) \neq \mathsf{None}$ (see Definition 6.15). Then, by Lemma 6.21 it also holds that $\mathsf{ischain}(T', \mathsf{IV}, c', y')$. Now, on the other hand, we also have that $\mathsf{ischain}(T, \mathsf{IV}, c \parallel [(x', y')], y)$ and $\mathsf{unpad}(\mathsf{mapfst}(c \parallel [(x', y')]) \parallel [x]) \neq \mathsf{None}$. This yields a contradiction since now Lemma 6.33 implies that $(x', y') \notin c \parallel [(x', y')]$, which is absurd. Hence, we must indeed have $\forall c.\ \neg\mathsf{valid\_chain}(c \parallel [(x', y')], T)$, and this finishes the proof.

□

A corollary of the above lemma is that, if $\mathsf{findseq}(x, y, \mathsf{upd}(T, (x, y), z))$ is successful, and the update with $((x, y), z)$ is benign, then $\mathsf{findseq}(x, y, T)$ must also have been successful (since the last element of a complete chain is not required to be in the domain of the chain's map), and thus, as above, $\mathsf{findseq}(x, y, T')$ must also be successful.

Corollary 6.57. *Let* $T, T' : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be maps,* x *be a block, and* y *and* z *be chaining values. Assume the following:*

1. $\mathsf{Inclusion}(T', T)$
2. $\mathsf{Injective}(T)$
3. $\mathsf{IV} \notin \mathsf{ran}(T)$
4. $(x, y) \notin \mathsf{dom}(T)$
5. $z \notin \mathsf{ran}(T)$
6. $z \neq \mathsf{IV}$
7. $\neg\mathsf{set\_bad3}(y, T', T)$
8. $\mathsf{Claim5}(T', T)$
9. $\mathsf{findseq}(x, y, \mathsf{upd}(T, (x, y), z)) \neq \mathsf{None}$

Then it holds that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$.

Proof. We will first show that it holds that $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$. For this, assume for contradiction that $\mathsf{findseq}(x, y, T) = \mathsf{None}$. But then, by Lemma 6.40 we obtain that $(x, y) \in \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x,y),z))}$, i.e. the last element of the complete chain found by $\mathsf{findseq}$ also appears at another point in this chain. This cannot hold true. To see this, note that from the soundness of $\mathsf{findseq}$ we get $\mathsf{ischain}(\mathsf{upd}(T, (x, y), z), \mathsf{IV}, \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x,y),z))}, y)$ and $\mathsf{unpad}(\mathsf{mapfst}(\pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x,y),z))}) \parallel [x]) \neq \mathsf{None}$. By instantiating Lemma 6.33 twice with the same chain, we now find out that $(x, y) \notin \pi_{\mathsf{findseq}(x,y,\mathsf{upd}(T,(x,y),z))}$, which yields the contradiction. Therefore, it must indeed hold $\mathsf{findseq}(x, y, T) \neq \mathsf{None}$. Now, we can apply Lemma 6.56 to immediately conclude that $\mathsf{findseq}(x, y, T') \neq \mathsf{None}$, which finishes the proof. □

7 Conclusions

We formally showed that the plain Merkle-Damgård construction, when used in conjunction with a prefix-free padding function, is indifferentiable from a variable input length random oracle, under the assumption that the underlying compression function behaves like a fixed input length random oracle. We learned about EasyCrypt and how it comfortably enables the formalization and mechanical verification of game-based proofs. We followed the proof of indifferentiability previously published by Coron et al. [39], and implemented a similar sequence of game transformations in EasyCrypt. However, our game sequence turned out to be slightly more complex, to account for the technical difficulties encountered when formally verifying equivalence statements between games. For instance, the difference between eager and lazy sampling exhibited by the initial and final games, described in Section 5.4, was hardly tackled at all in [39]. Moreover, the latter paper also described the games only in natural language, and accordingly that proof was conducted in a rather semi-formal way. In our proof, all transformations were validated using the built-in probabilistic relational Hoare logic of EasyCrypt, and the logical side conditions that were needed to enable the supported automated tools to solve the necessary proof obligations were verified in the Coq proof assistant. This result strengthens our belief that EasyCrypt is a formidable tool to tackle the formal verification of real-world cryptographic constructions, and that EasyCrypt makes a tremendous step towards the realization of Halevi's vision [49]. Our chief aim is to formally verify the indifferentiability of the finalists of the SHA-3 competition; let us discuss how our present work applies to them.

7.1 Applicability to SHA-3 Finalists

First, we consider how closely our formalization of the Merkle-Damgård iteration matches the actual implementation of the five SHA-3 finalists. Second, we discuss if and how well the assumption that the compression function behaves like a fixed input length random oracle is applicable to the real-world compression functions of those algorithms.

As mentioned in Section 3.3, all finalists implement variants of the Merkle-Damgård construction (see Definition 3.1). All finalists, except for BLAKE, additionally chop a number of bits off the last chaining value, with the purpose to discard some information; this makes the compression function's computations even more difficult to invert. Furthermore, Grøstl also performs a final transformation before chopping. For a formal verification, those operations would have to be explicitly implemented in the initial and final games. In fact, such operations may actually be needed to perform an indifferentiability proof; for instance, Coron et al. also showed in [39] that instead of using a prefix-free padding function in the plain Merkle-Damgård construction, one may alternatively chop a non-trivial number of bits off the last chaining value, and this also yields an indifferentiable hash function.

In our formalization of Merkle-Damgård, the compression function takes only two arguments (a block and a chaining value). This is indeed the case for Grøstl, JH, and Keccak. However, BLAKE's compression function additionally takes (as specified by the HAIFA framework [27]) a counter and a salt. Skein's compression function is built on a *tweakable* block cipher, and takes as additional input for its compression function a tweak (which essentially encodes the number of bytes hashed so far, as well as some additional information [45]). The counters in BLAKE and the tweaks in Skein play a similar role, namely, each block is processed with a unique variant of its compression function. This is supposed to increase their security, since it makes it much harder to exploit e.g. fixed-points. We observe that it is possible to formalize the salt and counters (respectively the tweaks) as an integral part of the padding rule of these hash functions, i.e. the padding function can precompute the appropriate values and append them to the message blocks. Then, these compression functions can be seen as taking only the usual two arguments, since the additional arguments are simply encoded within the blocks. This makes it possible to also map the compression functions of BLAKE and Skein onto our formalization of the Merkle-Damgård iteration.

Lastly, while all of the finalists use suffix-free padding rules, only the padding rules of BLAKE and Skein are additionally prefix-free [5]. Factoring in this fact, we observe that the closest candidate where our proof may apply is BLAKE; as for Skein, as mentioned above the final chopping would have to be formalized. Unfortunately, the assumption that the compression function is ideal is too strong in any case; we discuss this next.

As mentioned in Section 3.3, all of the SHA-3 finalists have been proven (on paper) to be indifferentiable from a random oracle [2, 3, 20, 24, 26, 37]. Yet, as opposed to the proof of indifferentiability presented here, all of those proofs are based on the assumption that the underlying building blocks of the compression functions are ideal; these underlying building blocks are block ciphers or permutations in all cases. As those underlying primitives constitute lower building blocks than the compression function itself, assuming their ideality is a weaker assumption than the assumption that the whole compression function is ideal. In fact, assuming ideality of the whole compression function is not appropriate for most of the finalists:

- The compression functions of JH and Keccak are trivially insecure, as collisions and preimages can be found in only one query to the underlying permutation [5, 28];
- It is trivial to find fixed-points for the compression function of Grøstl [47];

- The compression function of BLAKE has recently been shown to exhibit non-random behavior: It is differentiable from a (fixed input length) random oracle even under the assumption that its underlying block cipher is ideal [2, 37].

For Skein, there are no immediate results invalidating the assumption that its compression function is indifferentiable from a random oracle. It should be noted however that non-randomness has been proven for reduced round versions of Threefish [52], the tweakable block cipher which Skein is based on.

Since our proof of indifferentiability of the prefix-free Merkle-Damgård construction relies on the assumption that the underlying compression function behaves like an ideal primitive, it cannot be applied to BLAKE, as this assumption has been invalidated by aforementioned result. As for Skein, although no such result is known, the assumption that its compression function is ideal is also seriously weakened by the attacks on Threefish mentioned above.

7.2 Future Work

Our next aim is to formally verify the indifferentiability of the candidates of the SHA-3 competition. To achieve this, our formalization and proof has to be adapted so as to match the actual implementations of those algorithms. Since only BLAKE and Skein employ a prefix-free padding function, they constitute the most appealing candidates to concentrate on first. For this, first the formalization of the respective iteration modes has to be adapted as outlined in the previous section. Second, as explained above the proof has to be based on weaker assumptions, namely that the underlying building blocks of the respective compression functions are ideal. More precisely, both BLAKE and Skein are based on a block cipher [7, 45], and their respective compression functions have to be formalized up to the point where the respective block cipher is used. Then, the proof has to be performed in the ideal cipher model, i.e. only under the assumption that the underlying block cipher is ideal. Since a block cipher provides not only an encryption algorithm (used in the compression function), but also a decryption algorithm, the distinguisher must additionally be given access to a decryption oracle, which may constitute the main challenge.

Several (pen-and-paper) indifferentiability proofs of hash designs under such weaker assumptions are known. In [39], Coron et al. proved the indifferentiability of the prefix-free Merkle-Damgård construction based on the Davies-Meyer compression function [73], defined as $f(x, y) := E_x(y) \oplus y$, in the ideal cipher model for E, and for similar constructions as well. For all the finalists, as mentioned above, direct indifferentiability proofs are available which only assume the ideality of the underlying block ciphers or permutations.

To conclude, our proof constitutes a non-trivial result about the prefix-free Merkle-Damgård construction, and a first, but significant step to formalize the indifferentiability proofs of the SHA-3 finalists in EasyCrypt as well. Indeed, these proofs essentially use the same techniques, albeit they are slightly more involved since they are based on weaker assumptions. Our result provides a good starting point for formalizing these proofs, and we will further this line of research in the future.

References

[1] Reynald Affeldt, Miki Tanaka, and Nicolas Marti. Formal proof of provable security by game-playing in a proof assistant. In *1st International Conference on Provable Security – ProvSec 2007*, volume 4784 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2007.

[2] Elena Andreeva, Atul Luykx, and Bart Mennink. Provable security of BLAKE with non-ideal compression function. Cryptology ePrint Archive, Report 2011/620, November 2011. `http://eprint.iacr.org/`.

[3] Elena Andreeva, Bart Mennink, and Bart Preneel. On the indifferentiability of the Grøstl hash function. In *7th International Conference on Security in Communication Networks – SCN 2010*, volume 6280 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2010.

[4] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions of the second round SHA-3 candidates. In *13th International Conference on Information Security – ISC 2010*, volume 6531 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2011.

[5] Elena Andreeva, Bart Mennink, Bart Preneel, and Marjan Škrobot. Security analysis and comparison of the SHA-3 finalists BLAKE, Grøstl, JH, Keccak, and Skein. In *Progress in Cryptology – AFRICACRYPT 2012*, volume 7374 of *Lecture Notes in Computer Science*, pages 287–305. Springer, 2012.

[6] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2007.

[7] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Online – `http://131002.net/blake/`, December 2010.

[8] Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified security of Merkle-Damgård. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium – CSF 2012*, pages 354–368. IEEE Computer Society, June 2012. Also in *Grande Region Security and Reliability Day – GRSRD 2012*.

[9] Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning – LPAR 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 353–376. Springer, 2008.

[10] Gilles Barthe, Marion Daubignard, Bruce Kapron, and Yassine Lakhnech. Computational indistinguishability logic. In *17th ACM Conference on Computer and Communications Security – CCS 2010*, pages 375–386. ACM, 2010.

[11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella Béguelin. Verified indifferentiable hashing into elliptic curves. In *1st Conference on Principles of Security and Trust – POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2012.

[12] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International Workshop on Formal Aspects in Security and Trust – FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2009.

[13] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[14] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[15] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL 2009*, pages 90–101. ACM, 2009.

[16] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Programming language techniques for cryptographic proofs. In *1st International Conference on Interactive Theorem Proving – ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2010.

[17] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of Sigma-protocols. In *23rd IEEE*

Computer Security Foundations Symposium – CSF 2010, pages 246–260. IEEE Computer Society, 2010.

[18] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic reasoning for differential privacy. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL 2012*, pages 97–110. ACM, 2012.

[19] Gilles Barthe, Federico Olmedo, and Santiago Zanella Béguelin. Verifiable security of Boneh-Franklin identity-based encryption. In *5th International Conference on Provable Security – ProvSec 2011*, volume 6980 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2011.

[20] Mihir Bellare, Tadayoshi Kohno, Stefan Lucks, Niels Ferguson, Bruce Schneier, Doug Whiting, Jon Callas, and Jesse Walker. Provable security support for the Skein hash family. Online – `http://skein-hash.info`, April 2009.

[21] Mihir Bellare, Ted Krovetz, and Phillip Rogaway. Luby-Rackoff backwards: Increasing security by making block ciphers non-invertible. In *Advances in Cryptology – EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 1998.

[22] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security – CCS 1993*, pages 62–73. ACM, 1993.

[23] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, November 2004. `http://eprint.iacr.org/`.

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[25] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference. Online – `http://keccak.noekeon.org`, January 2011.

[26] Rishiraj Bhattacharyya, Avradip Mandal, and Mridul Nandi. Security analysis of the mode of JH hash function. In *17th International Workshop on Fast Software Encryption – FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 168–191. Springer, 2010.

[27] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, July 2007. `http://eprint.iacr.org/`.

[28] John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. *Journal of Cryptology*, 22:311–329, 2009.

[29] John Black and Phillip Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. In *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2000.

[30] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397. Springer, 2002.

[31] Bruno Blanchet. Mechanizing game-based proofs of security protocols. In *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 1–25. IOS Press, 2012.

[32] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *15th ACM Conference on Computer and Communications Security – ASIACCS 2008*, pages 87–99. ACM, 2008.

[33] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.

[34] Dan Boneh and Matthew Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal on Computing*, 32(3):586–615, 2003.

[35] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010.

[36] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[37] Donghoon Chang, Mridul Nandi, and Moti Yung. Indifferentiability of the hash algorithm BLAKE. Cryptology ePrint Archive, Report 2011/623, November 2011. `http://eprint.iacr.org/`.

[38] Pierre Corbineau, Mathilde Duclos, and Yassine Lakhnech. Certified security proofs of cryptographic protocols in the computational model: An application to intrusion resilience. In *1st International Conference on Certified Programs and Proofs – CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2011.

[39] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[40] Judicaël Courant, Marion Daubignard, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *15th ACM Conference on Computer and Communications Security – CCS 2008*, pages 371–380. ACM, 2008.

[41] Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI and University of Amsterdam, 1996.

[42] Ivan Damgård. A design principle for hash functions. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[43] Marion Daubignard, Pierre-Alain Fouque, and Yassine Lakhnech. Generic indifferentiability proofs of hash designs. In *25th IEEE Computer Security Foundations Symposium – CSF 2012*, pages 340–353. IEEE Computer Society, June 2012.

[44] Jonathan Driedger. Formalization of game-transformations. Saarland University, January 2010. Bachelor's Thesis.

[45] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whithing, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family. Online – `http://skein-hash.info`, November 2008.

[46] Jean-Christophe Filliâtre. The WHY verification tool: Tutorial and Reference Manual Version 2.31. Online – `http://why.lri.fr`, July 2012.

[47] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Online – `http://www.groestl.info`, March 2011.

[48] Shai Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. In *5th International Conference on Cryptology in India – INDOCRYPT'04*, volume 3348 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2004.

[49] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, June 2005. `http://eprint.iacr.org/`.

[50] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer, 2003.

[51] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. In *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.

[52] Dmitry Khovratovich, Ivica Nikolić, and Christian Rechberger. Rotational rebound attacks on reduced Skein. In *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2010.

[53] Joe Kilian and Phillip Rogaway. How to protect DES against exhaustive key search. In *Advances in Cryptology – CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 252–267. Springer, 1996.

[54] Neal Koblitz. Another look at automated theorem-proving. *Journal of Mathematical Cryptology*, 1(4):385–403, 2008.

[55] Neal Koblitz. Another look at automated theorem-proving. II. *Journal of Mathematical Cryptology*, 5(3-4):205–224, 2012.

[56] Stefan Lucks. A failure-friendly design principle for hash functions. In *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[57] Yuri I. Manin. *A Course in Mathematical Logic*. Springer, 1977. Translated by Neal Koblitz.

[58] Stéphane Manuel. Classification and generation of disturbance vectors for collision attacks against SHA-1. *Designs, Codes and Cryptography*, 59:247–263, 2011.

[59] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *1st Theory of Cryptography Conference – TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

[60] Ralph Merkle. One way hash functions and DES. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[61] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[62] David Nowak. A framework for game-based security proofs. In *9th International Conference on Information and Communications Security – ICICS 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2007.

[63] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In *11th International Conference on Information Security and Cryptology – ICISC 2008*, volume 5461 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2009.

[64] David Nowak and Yu Zhang. A calculus for game-based security proofs. In *4th International Conference on Provable Security – ProvSec 2010*, volume 6402 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2010.

[65] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2011.

[66] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *8th ACM Conference on Computer and Communications Security – CCS 2001*, pages 196–205. ACM, 2001.

[67] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *11th International Workshop on Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

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

[69] Malte Skoruppa. Formal verification of ElGamal encryption using a probabilistic Lambda-calculus. Saarland University, January 2010. Bachelor's Thesis.

[70] The Coq development team. The Coq Proof Assistant Reference Manual Version 8.3. Online – `http://coq.inria.fr`, 2010.

[71] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[72] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 561–561. Springer, 2005.

[73] Robert S. Winternitz. A secure one-way hash function built from DES. In *5th IEEE Symposium on Security and Privacy – S&P 1984*, pages 88–90. IEEE Computer Society, 1984.

[74] Hongjun Wu. The hash function JH. Online – `http://www3.ntu.edu.sg/home/wuhj/research/jh/`, January 2011.

[75] Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs.* PhD thesis, Ecole Nationale Supérieure des Mines de Paris – Mines ParisTech, 2010.

[76] Santiago Zanella Béguelin, Benjamin Grégoire, Gilles Barthe, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy – S&P 2009*, pages 237–250. IEEE Computer Society, 2009.