Formal Security Proofs of Cryptographic Standards : A necessity achieved using EasyCrypt

Cécile Baritel-Ruet

▶ **To cite this version:**

Cécile Baritel-Ruet. Formal Security Proofs of Cryptographic Standards : A necessity achieved using EasyCrypt. Cryptography and Security [cs.CR]. Université Côte d'Azur, 2020. English. NNT : 2020COAZ4049 . tel-03150443v2

HAL Id: tel-03150443

https://theses.hal.science/tel-03150443v2

Submitted on 23 Mar 2021

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

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

Distributed under a Creative Commons Attribution 4.0 International License

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Preuves Formelles de la Sécurité de Standards Cryptographiques

Un objectif nécessaire, possible grâce à EasyCrypt

Cécile BARITEL-RUET

INRIA Sophia Antipolis

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur.

Dirigée par : Yves BERTOT

Co-encadrée par :
Benjamin GRÉGOIRE

Soutenue le : 02/10/2020

Devant le jury, composé de :
Manuel Barbosa,
Yves Bertot,
Sandrine Blazy,
François Dupressoir,
Steve Kremer,
Benjamin Grégoire,
Bruno Martin

Preuves Formelles de la Sécurité de Standards Cryptographiques

Un objectif nécessaire, possible grâce à EasyCrypt

Jury :

Directeurs

Yves Bertot, Directeur de recherche, Inria Sophia Antipolis Méditerrannée
Benjamin Grégoire, Chargé de recherche, Inria Sophia Antipolis Méditerrannée

Rapporteurs

Sandrine Blazy, Professeur et Chercheur, Université de Rennes 1 et IRISA
Steve Kremer, Directeur de recherche, Inria Nancy

Examinateurs

Manuel Barbosa, Lecturer and Researcher, University of Porto and HASLab
François Dupressoir, Senior Lecturer, University of Bristol
Bruno Martin, Professeur et Chercheur, Université Côte d'Azur et I3S

Formal Security Proofs of Cryptographic Standards

A necessity achieved using EasyCrypt

Jury :

Advisors

Yves Bertot, Research director, Inria Sophia Antipolis Méditerrannée
Benjamin Grégoire, Researcher, Inria Sophia Antipolis Méditerrannée

Rapporteurs

Sandrine Blazy, Lecturer and Researcher, Université de Rennes 1 and IRISA
Steve Kremer, Research director, Inria Nancy

Examiners

Manuel Barbosa, Lecturer and Researcher, University of Porto and HASLab
François Dupressoir, Senior Lecturer, University of Bristol
Bruno Martin, Lecturer and Researcher, Université Côte d'Azur et I3S

Titre : Preuves Formelles de la Sécurité de Standards Cryptographiques

Résumé :

En cryptographie, Shannon a montré que le secret parfait n'existe pas. Ainsi, la cryptographie moderne considère des propriétés de sécurité dans lesquelles un attaquant peut briser l'algorithme cryptographique mais seulement avec une faible probabilité. Dans ce contexte, les algorithmes cryptographiques et les propriétés/hypothèses de sécurité sont exprimés sous forme de programmes probabilistes. Les preuves de sécurité consistent à borner la probabilité d'un événement dans de tels programmes. Ces preuves sont difficiles à prouver et malgré le système de relecture académique des erreurs continuent d'être publiées. Nous proposons l'utilisation des preuves formelles pour assurer une fiabilité suffisante des standards cryptographiques.

Ma thèse fournit les preuves formelles de sécurité de trois standards dans l'assistant de preuve EasyCrypt. Ces schémas sont CMAC (qui fournit l'authentification et l'intégrité des messages), SHA-3 (une fonction de hachage cryptographique), et ChaCha20-Poly1305 (un schéma de chiffrement authentifié avec données associées). L'objectif de la thèse n'est pas seulement de formaliser la preuve de sécurité de ces standards, mais aussi de développer des techniques génériques et des bibliothèques qui peuvent être réutilisées. Toutefois, les preuves formelles de sécurité n'assurent que la sécurité des algorithmes et non de leurs implémentations. Pour contourner cette lacune, avec mes collaborateurs, nous lions formellement nos implémentations sûres et efficaces avec la preuve de sécurité, ceci conduit à la première preuve de sécurité cryptographique d'implémentations.

Mots clés : Cryptographie, Standards, Sécurité prouvable, EasyCrypt, Preuve formelle, CMAC, SHA-3, ChaCha20-Poly1305.

Title: Formal Security Proofs of Cryptographic Standards

Abstract:

In cryptography, Shannon showed that perfect secrecy does not exist. Thus, modern cryptography considers security properties in which attackers may break the cryptographic algorithm only with a small (negligible) probability. In this context, cryptographic algorithms and security properties/assumptions are expressed as probabilistic programs. Security proofs consist of bounding the probability of an event in such programs. Such proofs have been peer-reviewed for some decades, but since they are difficult to prove and to verify, fallacies keep emerging. We propose to use formal proofs to provide enough trustworthiness for crypto-systems such as cryptographic standards.

My thesis provides the formal security proofs of three standards that are formally verified using the proof assistant EasyCrypt. The cryptographic standards I have worked on are CMAC (that provides message authentication and integrity), SHA-3 (a cryptographic hash function), and ChaCha20-Poly1305 (an authenticated encryption scheme with associated data). The goal of the thesis is not only to provide formal proof of those standards, but also to develop generic techniques and libraries that can be reused. However, the formal security proofs only ensure the security of the algorithms and not its implementation. To circumvent this gap, with my collaborators, we have developed fast and secure implementations of the last two schemes that are also side-channel resistant. Furthermore, we formally link the implementation with the security proof, leading to the first formal security proof of an implementated standard.

Keywords: Cryptography, Standards, Provable security, EasyCrypt, Formal proof, CMAC, SHA-3, ChaCha20-Poly1305.

Remerciements — Acknowledgements

Je souhaite remercier en tout premier lieu mes deux directeurs, Yves Bertot et Benjamin Grégoire, pour leur temps, leur soutien et leurs conseils que vous avez su m'offrir pendant ces quatre années de travail ensemble.

Merci à toi Benjamin pour le soutien indéfectible et la bienveillance que tu as su témoigner durant notre travail ensemble. Merci de m'avoir posé toutes ces questions auxquelles je ne savais pas répondre immédiatement, m'enseignant beaucoup et en particulier cette compétence fort utile qu'est la prise de recul. Je te suis reconnaissante pour tes efforts de relecture, que j'espère savoir apprécier à leur juste valeur.

Merci à toi Yves pour ton humanité, tes conseils et ton écoute pendant toute la durée de ma thèse. Malgré la différence dans nos domaines d'expertise respectifs, nos échanges ont toujours été non seulement enrichissant, mais aussi ont contribué à la prise de recul sur mon domaine de recherche et le domaine de la recherche de manière générale. Plus particulièrement, merci pour ton soutien pendant ce confinement de mars à mai 2020.

I want to humbly thank my jury for accepting to review my work.

In particular, I am wholeheartedly grateful to both Sandrine Blazy and Steve Kremer, who dared to be rapporteurs, even though the health crisis and the summer break would leave them little time to read this manuscript and write their report.

Thank you, François Dupressoir, Manuel Barbossa, and Bruno Martin for showing enough interest in my work to participate in this jury.

Je tiens à remercier tous les membres de l'équipe Marelle — maintenant l'équipe Stamp — qu'ils soient permanents ou non. L'ambiance que vous avez su maintenir est à la hauteur du temps de la région : chaleureuse, agréable et humaine. Merci Laurence, pour ta gentillesse, tes conseils, ta bienveillance et ton énergie. Merci à Laurent pour ses conseils sur l'Agreg, même si je ne souhaite plus m'y diriger. Merci à Reynald pour toutes les inspirations que tu m'as données à propos de la constitution de ces remerciements. Merci à Sophie et Damien pour ces discussions entre doctorants toujours très intéressantes et enrichissantes, notamment sur l'éducation. Thank you, Anders, for the pleasant co-habitation at the office. Merci à Mohammad et Adam pour la bonne humeur qu'ils ont instaurée dans le bureau. Merci à Pierre pour ces discussions autour d'un (un ?) café. Merci à Cyril pour son point de vue rafraichissant et bienveillant. Merci à Maxime et Enrico pour avoir animé les repas du midi. Merci à Nathalie pour son soutien, son humanité, sa bonne humeur et sa bienveillance. Son travail nous soulage d'un poids administratif et je pense ne pas savoir mesurer son importance tant elle fait bien son travail pour qu'on ne le remarque pas. Merci pour sa trilogie, que j'attends avec impatience en version électronique. Je souhaite rendre hommage à José qui nous a quitté en 2019, on pense à lui.

Merci à tous ceux qui ont bien voulu m'écouter parler de ma recherche. Je tiens notamment à remercier les deux membres de mon comité de suivi doctoral, François Dupressoir et Cinzia Di Guisto, pour vos retours, vos encouragements et vos éclairements.

Merci aux enseignants Cinzia Di Guisto, Sid Touati et Étienne Lozes, pour m'avoir permis de travailler avec vous pour mon service en tant que chargé de TP/TD. Je tiens aussi à remercier mes élèves avec qui le contact humain lors de l'échange pédagogique a été source de beaucoup d'enthousiasme et de motivation pour moi.

Merci à Madtree pour cette discussion sur le cours de cryptographie sur Coursera, et dont la présente thèse en est le produit fortuit. Merci à Xavier Urbain pour son aide et son soutien pour poursuivre le parcours académique qui m'intéressait. Merci à Serge Haddad pour m'avoir présenté le second concours d'entrée à l'ENS de Cachan. Merci à Stéphanie Delaune pour son soutien et de m'avoir fait découvrir qui allait être mon encadrant de thèse. Merci à Pierre-Alain Fouque pour son co-encadrement de mon stage de M2 et que j'ai eu le plaisir de retrouver à Crypto 2019. Merci à Cristina Onete pour son soutien pendant ce stage, et pour m'avoir fait découvrir le workshop CrossFyre. Je tiens à remercier François Dupressoir qui m'a accueilli pour une visite juste après l'édition 2018 de CrossFyre. Merci à Pierre-Yves Strub pour son soutien pendant la demande de bourse de thèse et pour le travail commun pendant la thèse. Merci à Gilles Barthe pour sa ligne de recherche qui a été indirectement déterminante pour mon travail de recherche.

Enfin, je remercie tous ceux qui m'ont soutenu et encouragé. Merci en particulier à mes amis et ma famille, que j'ai peu vus ces dernières années à cause de la distance, mais qui comptent beaucoup pour moi et m'accueillent comme si je n'étais jamais partie.

Merci à Nicole, Pascal, Audrey, Nathanaël, le petit Maël (qui est arrivé le 17 mars 2020), Marthe, Joëlle, Pauline, Luca, Hervé, Catherine, Philippe, Rémy, Thomas, Ophélie, Djamal, Thomas, Élodie, Fabien, Louis, Charly, Pierre, Tristan, Amélie, Nicolas, Olivier, Karina, Stéphane, Daniel et aux autres que j'oublie. Merci à Ahri, ma chienne, qui a été d'une aide précieuse, en particulier pour continuer à sortir pendant le confinement.

Contents

Introduction

Whenever I speak about my PhD in a non-expert context, I start talking about *computer security* and I get simple but meaningful questions that resemble to: "Are you actually breaking into software to exploit vulnerabilities ?", "Do you know about this threatening vulnerability X ?", or "Which anti-virus should I install to protect my data ?". This kind of questions belongs to one particular field of *computer security* where the focus is on *breaking* security, while I work in the field where the focus is on *ensuring* security.

In computer networks, data flows from places to places. Some of the data is sensitive and attackers like hackers or spies want to find an access through vulnerabilities of the data flow. The objective of computer security is to analyze the flow of data, to find vulnerabilities and place barriers to prevent intruders from accessing it. Some of those barriers come from *cryptography* and my work is to analyze the "robustness" of some particular barriers.

Cryptography prior to the modern age was actually synonymous with *encryption*, the conversion of information from a readable meaningful state to apparent nonsense. The originator of an encrypted message shares the decoding technique (the *encryption* and *decryption algorithms* and the *secret key*) only with intended recipients to preclude access from *adversaries*. Because the adversary cannot understand and know the content of the encrypted message, this is one way of how the barriers are implemented.

There are two ways to ensure security, either the secrecy depends on hiding both the algorithm and the secret key, or the algorithm is known to the adversary and only the secret key needs to be hidden. History has shown that the first one is not practical, and leads to greater security flows, as the amount of secret data to keep the secrecy is much larger if the algorithm is included. This was first explicitly stated as a fundamental principle in 1883 by Auguste Kerckhoffs [Kerckhoffs, 1883] and is generally called Kerckhoffs's Principle; alternatively and more bluntly, it was restated by Claude Shannon, the inventor of *information theory* and the fundamentals of theoretical cryptography in [Shannon, 1949]:

> The enemy knows the system.

Since the development of rotor cipher machines in World War I and the advent of computers in World War II, the methods used to carry out cryptology have become increasingly complex and its application more widespread. In recent decades, the field has expanded beyond confidentiality concerns to include techniques for message integrity checking, sender/receiver identity authentication, digital signatures, interactive zero-knowledge proofs and secure computation, among others, as shown in Figure 1.

Modern cryptography is heavily based on mathematical theory and computer science practice. Cryptographic algorithms, or more generally *cryptosystems*, are based on *mathematical problems* that are easy to state but have been found difficult to solve. If one can

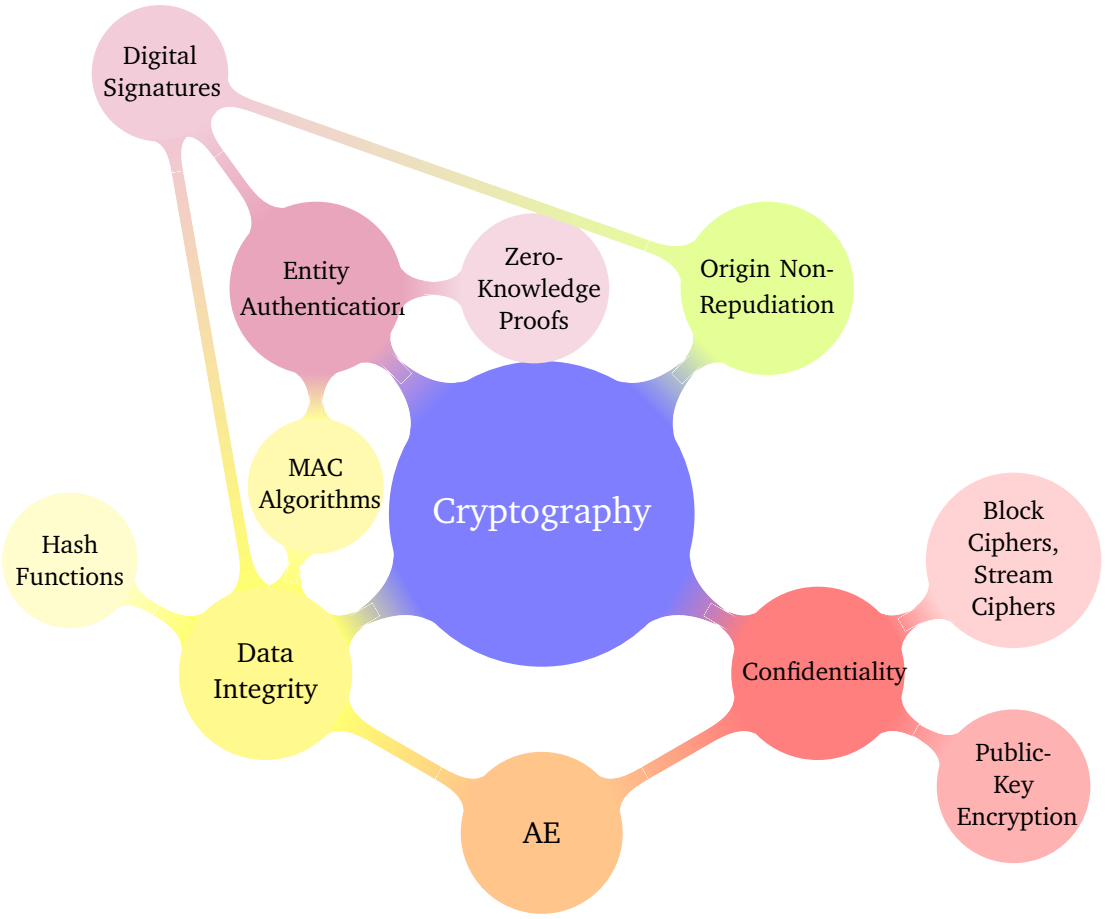


Figure 1 – Cryptography: a field composed of various security goals and techniques

reduce any *adversary* that efficiently breaks a cryptosystem into an efficient solution to a hard mathematical problem, this means that breaking this cryptosystem is harder than finding an efficient solution to this hard mathematical problem. This reasoning pattern is called a *security reduction proof* (or *reduction*, or *security proof*) and is instrumental in the modern approach to cryptography: *provable security* [Katz and Lindell, 2014].

We use a mathematical method for the basis of our work that includes the reduction of the number of concepts, for instance by reducing the number of actors, the form of attacks, and using probabilities. This approach makes explicit a simple model and is well accepted in the community. For instance, the standardization institute NIST recommends: « The review of technical merit includes a precise, formal statement of security claims, based on minimal security assumptions and supported as far as possible by documented cryptanalysis and security reduction proofs. » (*NIST Cryptographic Standards and Guidelines Development Process* [Regenscheid, 2016])

Provable Security

Traditionally, provable security is asymptotic: it classifies the hardness of computational problems using *polynomial-time reducibility*. Secure schemes are defined to be those in which the *advantage* of any computationally bounded adversary to break the cryptosystem is *negligible*. While such a theoretical guarantee is important, in practice one needs to know exactly how efficient a reduction is because of the need to instantiate the *security parameter* (e.g. key length). It is not enough to know that "sufficiently large" security parameters will

do. An inefficient reduction results either in the success probability for the adversary or the resource requirement of the scheme being greater than desired.

Concrete security is a practice-oriented approach [Bellare, 1997] that needs a more precise estimate of the computational complexity of adversarial tasks than polynomial equivalence would allow. Concrete security aims at quantifying an *upper bound* on the probability of any adversary to break the system studied. More precisely, a proof of security involves an upper bound on the advantage of the adversary to break the system as a function of adversarial resources and of the problem size. The resource available to the adversary are commonly *running time* and *memory*, and other resources specific to the system in question, such as the number of plaintexts it can obtain or the number of *queries* it can make to any available oracle. The problem size usually is the size of the key, but it can also include the size of blocks when blocks are defined in the system.

There are several lines of research in provable security. One is to establish the "correct" *definition of security* for a given, intuitively understood task. Another is to suggest possible solutions: cryptographic constructions and their security proofs. The cryptography community is confronted to the difficulty to produce correct proofs. Several researchers found mathematical fallacies in proofs; proofs that had been used to make claims about the security of important protocols. This is illustrated in the list below where the name of a cryptosystem is followed by the initial paper when it was claimed to be proven secure then by the reference in which the first fallacy was reported. For some of those constructions, successive corrections were also shown to be defective.

— OAEP [Bellare and Rogaway, 1994] [Shoup, 2002];

— HMQV [Krawczyk, 2005] [Menezes, 2007];

— CBC-MAC and EMAC [Bellare et al., 2005] [Jha and Nandi, 2016];

— Boneh-Franklin IBE [Boneh and Franklin, 2003] [Galindo, 2005];

— GCM [McGrew and Viega, 2004b] [Iwata et al., 2012];

— XLS [Ristenpart and Rogaway, 2007] [Nandi, 2014];

— RSA screening [Bellare et al., 1998b] [Coron and Naccache, 1999];

— XCB [McGrew and Fluhrer, 2007] [Chakraborty et al., 2015];

— Cascade encryption [Bellare and Rogaway, 2006] [Gaži and Maurer, 2009];

— RSA-FDH [Coron, 2002] [Kakvi and Kiltz, 2012];

— OCB2 [Rogaway, 2004a] [Inoue et al., 2019].

The scientific community thinks that for any claim the claimer should be the one to produce enough non-fallacious material to convince of the correctness of its logical arguments. The referring process is here to ensure that no wrong proof is accepted. However, the list above shows that this is not enough. Delegating the verification of a proof's correctness to a computer is a way to answer this issue. This is my second field of expertise which is part of the larger field of *formal verification*.

Formal Verification

The verification of a mathematical proof is a tedious work, and as a lot of tedious works, this can be eased by the use of machines. This is well explained by Thomas C. Hales in his advocacy for formal proofs [Hales, 2008]:

> Traditional mathematical proofs are written in a way to make them easily understood by mathematicians. Routine logical steps are omitted. An enormous amount of context is assumed on the part of the reader. Proofs, especially in topology and geometry, rely on intuitive arguments in situations where a trained mathematician would be capable of translating those intuitive arguments into a more rigorous argument.
>
> A formal proof is a proof in which every logical inference has been checked all the way back to the fundamental axioms of mathematics. All the intermediate logical steps are supplied, without exception. No appeal is made to intuition, even if the translation from intuition to logic is routine. Thus, a formal proof is less intuitive, and yet less susceptible to logical errors.

Formal verification involves the use of logical and computational methods to establish claims that are expressed in precise mathematical terms. These can include ordinary mathematical theorems, as well as claims that pieces of hardware or software, network protocols, and mechanical and hybrid systems meet their *specifications* (expected behavior defined by a mathematical property). Formal verification requires describing hardware and software systems in mathematical terms, at which point establishing claims about their correctness becomes a form of theorem proving.

Interactive theorem proving focuses on the verification aspect of theorem proving requiring that every claim is supported by a proof in a suitable axiomatic foundation. This sets a very high standard: every rule of inference and every step of a calculation has to be justified by appealing to prior definitions and theorems, all the way down to basic axioms and rules. HOL [Gordon, 1988], Isabelle [Paulson, 1994], PVS [Owre et al., 1992], Coq [Coq, 1984] and Lean [Lea, 2012] are some of the most well-known interactive theorem provers, also known as *proof assistants*.

Without minimizing nor dismissing their respective strengths, they primary work on mathematical proofs and aim to reduce towards a minimum its axiomatic basis. My work involves the game-based technique of provable security, a feature that is included in EasyCrypt, a proof assistant introduced as a proof of concept to address the problem of the security of OAEP [Barthe et al., 2011b]. It implements a model in which the game-based technique and distributions (probabilities) set up an interactive theorem prover for the formalization of concrete security bounds. I will describe EasyCrypt along with the methodology used for security proofs in detail in the next part.

Cryptographers have the intuition of why a system is secure, but translating this intuition into formal logical steps is a long and tedious work and requires expertise in both cryptography and formal verification. Deductive verification has the disadvantage that it requires the user to understand in detail how the system works exactly, and to convey this information to the verification system. There exist a lot of cryptographic constructions and not so many cryptographers are also versed into formal verification.

Contributions

My work aims at extending the number of formally proven secure schemes with two objectives in mind:

1. for the sake of verifying concrete security bounds of cryptographic standards,
2. to extend the proof of concept into a proof of usability, so that formal verification with EasyCrypt may be included in the process of standardization.

This extension includes the formalization of the security proof of three different cryptographic standards, each of them answering a different security requirement. Because of the downsides of formalization efforts, we have intentionally prioritized cryptographic standards. Those standards are from either NIST, or the IETF (Internet Engineering Task Force).

My first contribution is the study of the formal security bound of a standard MAC (message authentication code) scheme named CMAC [Baritel-Ruet et al., 2018].

My second contribution is part of a joint work to produce a fast and secure implementation of a standard hash function named SHA-3 [Almeida et al., 2019b]. My personal contribution on this is situated in the formal security bound.

The third contribution presented in my manuscript is the study of the formal security bound of an authenticated symmetric encryption scheme named ChaCha20-Poly1305. This last work has not been published yet.

Organization of this thesis

Part I explains the *scientific methodology* and is composed of two chapters.

— Chapter 1 explains in more details practice-oriented provable security, and the limits and controversies of provable security.

— Chapter 2 exposes EasyCrypt.

Part II explores the notion of *message authentication* with the study of the security of one *standardized* MAC scheme and is composed of two chapters.

— Chapter 3 explains what message authentication actually means and state the formal security definition of a MAC scheme.

— Chapter 4 explains the security proof of the standardized MAC scheme named CMAC.

Part III explores the notion of *cryptographic hash function* with the study of the security of one *standardized* hash function and is composed of two chapters.

— Chapter 5 states the formal security definition of a cryptographic hash function.

— Chapter 6 explains the security proof of the standardized hash function named SHA-3.

Part IV explores the notion of *authenticated encryption* with the study of the security of an authenticated encryption and is composed of two chapters.

— Chapter 7 states the formal security definition of an authenticated encryption.

— Chapter 8 explains the security proof of ChaCha20-Poly1305, an authenticated encryption scheme.

Part I

Methodology

Chapter 1

Practice-Oriented Provable Security

The security of a service is its ability to be resilient against potential risks. In providing a service, the goal is to maximize its usefulness while minimizing the risks associated with its use. This balance between risk and efficiency is essential in IT security. However, this is not up to cryptographers to provide such a balance in the use of a cryptosystem, but rather the "architect" of the service. Instead, the responsibility of the cryptographers is to provide sufficient analytical material to the "architect" so that the choice on which cryptosystems and their parameters should be based on an *informed one*.

The choice of such a cryptosystem can be guided by the fact that the longer the cryptosystem remains unattacked, the more secure it is believed to be. Before modern cryptography, the common practice was to find a particular attack on a cryptographic scheme, then update the scheme, and do it again (and again). This practice is tantamount to playing the cat and mouse game and reveals an empirical approach that is not sufficiently relevant for computer risk management. On the other hand, the scientific approach of *provable security* refers to mathematically proven security properties that quantify on *all* attacking strategies.

Remark

This paradigm shift in cryptography — from empirical to mathematically proven — was first introduced for public-key signature and encryption. The security of the famous public-key encryption RSA [Rivest et al., 1983] relies on the fact that any efficient adversary that can break the security of RSA can be turned into an efficient algorithm for factoring the product of two large random numbers, which is assumed to be a *hard* mathematical problem.

In his 1998 survey article, *Why chosen ciphertext security matters* [Shoup, 1998], Shoup explained the rationale for attaching great importance to reductionist security arguments:

> This is the preferred approach of modern, mathematical cryptography. Here, one shows with mathematical rigor that any attacker that can break the cryptosystem can be transformed into an efficient program to solve the underlying well-studied problem (e.g., factoring large numbers) that is widely believed to be very hard. Turning this logic around: if the hardness assumption is correct as presumed, the cryptosystem is secure. This approach is about the best we can do. If we can prove security in this way, then we essentially rule out all possible shortcuts, even ones we have not yet even imagined. The only way to attack the cryptosystem is a full-frontal attack on the underlying hard problem. Period.

Classical provable safety is mainly aimed at studying the *asymptotic* relationship between objects. Secure schemes are then defined as those in which any algorithm capable of breaking the cryptographic system can be asymptotically reduced to an algorithm that can solve a difficult mathematical problem. While such a theoretical guarantee is important, in practice it is necessary to know exactly how effective a reduction is because of the practical need to instantiate the security parameter. It is not enough for an informed choice to know that "sufficiently large" security parameters will suffice. Therefore, a reduction that is in polynomial time but in an ineffective way is reflected either in the probability of success for the adversary or in the resource requirements of the regime being greater than desired. As Rogaway stated in [Rogaway, 2009]:

> The provable-security approach begins by identifying the cryptographic problem of interest, an often ignored but crucial first step. Next, one gives a precise *definition* for the problem at hand. This entails identifying *what* the adversary can do and *when* it is deemed successful. A protocol is then given for problem, that protocol depending on some other, hopefully-simpler protocol for solving some other, hopefully-more-basic problem that has likewise been defined. Evidence of security for the high-level protocol takes the form of a *reduction*, the reduction showing how to transform an adversary attacking the high-level goal into one for attacking the low-level goal.
>
> [...] The association of an adversary to a real number is the *definition of security*.

The next sections are organized as follows.

— Section 1.1 introduces different models used in provable security.

— Section 1.2 discusses about about limitations of provable security and its models, and then discusses related work.

1.1 Provable Security Models

A model for provable security defines "*what* the adversary can do" and the collection of assumptions and proof methods in which the reduction can be expressed.

Traditionally, cryptographic protocols are analyzed with the *Dolev-Yao* symbolic model [Dolev and Yao, 1983]. This model assumes perfect cryptography. Messages are assumed to be elements of some abstract algebra, and cryptographic primitives, such as encryption, are abstract operations on this algebra. In this model, the adversary can hear, intercept, and synthesize any message and is limited only by the constraints of the cryptographic method used. In other words, "the adversary carries the message." These messages consist of formal terms that may reveal part of their internal structure, but some parts remain opaque to the adversary. Modelled as a specific non-deterministic state machine, the only way for the adversary to extract this internal structure is to perform some of the operations on messages it already "knows".

This model has an extremely nice feature: simplicity, but has a disadvantage: Dolev-Yao's adversary is actually quite weak. Because the calculation model is symbolic and complete knowledge of the adversary can be computed, it is possible to explicitly represent all possible behaviors of the adversary in a compact way. Although he can choose among the allowed operations in a non-deterministic way, the set of allowed operations is fixed and quite small.

Therefore, the models are very well suited to be automatically analyzed to show that a protocol is broken, rather than secure, under the assumptions about the adversary's capabilities, e.g. [Dalal et al., 2010, Blanchet, 2013].

Dolev-Yao's symbolic model is often compared [Herzog, 2005] to the *computational model* [Goldwasser and Micali, 1984, Goldwasser et al., 1988, Yao, 1982], because the latter is much more realistic, but complicates the proofs. Instead of defining adversarial *capabilities* in the symbolic model, adversarial restrictions extend security claims to a larger set of adversaries. In the computational model, a cryptographic construction exchanges messages that are bitstrings from some distribution, an adversary is an arbitrary probabilitic algorithm with *restrictions* and primitives are modeled as (tuples of) algorithms that satisfy security and constructive assumptions (e.g. decryption and encryption are inverse to each other).

The computational model in which the adversary is limited only by time and computing power is called the *standard model*. Schemes whose safety can be proven using only complexity assumptions are said to be *secure in the standard model*. Proofs of security are known to be difficult to achieve in the standard model. In many proofs, cryptographic primitives are replaced by idealized versions.

In the next two subsections, I detail two other computational models used in the formal security proofs I have worked on, namely the *random oracle model* and the *ideal cipher model*.

1.1.1 Random Oracle Model

A *random oracle* is an *oracle* — a theorical black-box — that answers to every fresh query with a truly random response chosen from the uniform distribution on its output domain. Random oracles are typically used as an ideal replacement for cryptographic hash functions (see Chapter 5) where strong randomness on its output domain is needed. In the *random oracle model*, a random oracle is assumed to exist and be accessible to the adversary. In [Bellare and Rogaway, 1995], Bellare and Rogaway were the first to advocate their use in cryptographic constructions. In their definition, the random oracle produces a bit-string of infinite length which can be truncated to the desired length. However, the formalization of such a random oracle would raise problems such as how to sample a value uniformly from the output set $\{0,1\}^{\infty}$. Therefore, a random oracle is formalized to either set the output type to be finite or the random oralce waits for a second input: the desired length [1].

Such a model has its own limitations. In [Impagliazzo and Rudich, 1989], Impagliazzo and Rudich showed that the existence of a random oracle alone is not sufficient for a secure exchange of secret keys. Moreover, if a cryptographic construct proves to be secure in the standard model, attacks against this construct must either be outside the behaviour of the adversary considered in the model or break one of the assumptions. For example, if the proof is based on the hardness of the factorization of large integers, in order to break it, a fast integer factorization algorithm must be discovered. Instead, to break the random oracle hypothesis, one must only discover an unknown and undesirable property of the current hash function. For a good hash function when such properties are deemed unlikely or difficult to discover by computation, the intended cryptographic construct remains secure.

Furthermore, consider a cryptographic construction built using a hash function that has a security proof in the random oracle model. Even though some hash function is found to be broken, e.g. SHA-1 [Stevens et al., 2017], the cryptographic construction remains secure in the sense that one only has to change the hash function to restore security. In

1. see Chapter 6 for more details

other words, the difficulty to find a secure hash function does not affect the overall security of the cryptograhpic construction that has a proof in the random oracle model.

1.1.2 Ideal Cipher Model

An *ideal cipher* is a random permutation oracle that is used to model an idealized block cipher (see Chapter 3). A random permutation decrypts each ciphertext block into one and only one plaintext block and vice versa, so there is a one-to-one correspondence. Some cryptographic proofs give to the adversary also access to the reverse permutation.

Remark

Recent works showed that an ideal cipher can be constructed from a finite random oracle using Feistel networks, with 10 rounds [Dachman-Soled et al., 2016] or 8 rounds [Dai and Steinberger, 2016a].

1.2 Limitations and Related Work

When cryptographers publish their security analysis, often as a security proof, a non-specialist is very unlikely to read (or even think about) the proof, hoping to find some instructions about how to implement the cryptosystem. This involves a certain trust in security proofs from non-specialist communities about their credibility. Since security proofs are left aside, the responsibility of cryptographers is to strive for clarity in security statements, including the theoretical model, meaning of all parameters, and all assumptions (security assumptions, independance assumptions, access granted or banned to oracles, etc...).

The promise of quantifiable proof of safety on each adversarial strategy is attractive and tight security statements are great tools to enable practitioners to make informed choices, when they are sufficiently clear. For this reason, one of the emphasis of this manuscript is about clear security statements and proof models. However, clarity does not guarantee the correctness of the proof of security, as shown in the next two examples.

Example

In [McGrew and Viega, 2004b], the AES-GCM authenticated encryption scheme was proven secure. Eight years after [Iwata et al., 2012], the proof was found to have some fallacies and the authors gave a procedure to break the first security bound, without contradicting the overall security of AES-GCM. Indeed, they also proposed a repaired security proof with a worse security bound than the incorrect one.

Example

In [Inoue et al., 2019], they found a fallacy in the security proof of OCB2, leading to a very dangerous and powerful attack on this authenticated encryption still in use. They propose a fix for OCB2, and argue that OCB1 and OCB3 do not share this flaw.

As Stern, Pointcheval, Malone-Lee, and Smart [Stern et al., 2002] reported:

> Methods from provable security, developed over the last twenty years, have been recently extensively used to support emerging standards. However, the fact that proofs also need time to be validated through public discussion was somehow overlooked. [...] the use of provable security is more subtle than it appears, and flaws in security proofs themselves might have a devastating effect on the trustworthiness of cryptography.

This is not to claim that the correctness of a proof is a problem that stems from the limits of provable safety. However, the impact on real systems that rely on the correctness of security proofs can be extremely detrimental, as in the case of OCB2 [Inoue et al., 2019]. Instead, my thesis tackles this problem using *formal proofs* which gives a stronger guarantee that the proof is correct, increasing the confidence non-specialists may have in cryptosystems, when the clarity of the security statement is emphasized.

The application of formal proofs on reductionist arguments also has its limitations in terms of the cost of such an analysis. They involve the time necessary to analyse a cryptosystem and expertise in two different fields of computer science. This is a huge cost on time and expertise, therefore an emphasis should be made when this effort is reasonably expectable. A potential candidate would be the process of standardization (e.g. SHA-3 [Kayser, 2007], see Chapter 6), or in the process of a competition that aims to determine the best cryptosystem, e.g. the CEASAR competition [CAE, 2013].

I also argue that for each security property about each cryptosystem, only a single formal proof is needed. Indeed, once a result has been formally proven using a proof assistant, this result can be trusted — at least until a flaw is found in the proof assistant. For this reason, one could argue that a proof assistant is as trustworthy as a cryptographer, if not less. However, pen-and-paper proofs are much more sensible to intuition shortcuts that may lead to a fallacy. The constant requirement to formalize all of the logical steps of every intuition is still a fastidious effort, but on the other side it gives the opportunity to deeply understand the reductionist argument and all of the assumptions made about the cryptosystem.

Example

When a proof assistant is implemented so that composition of cryptosystems is possible, assumptions on independance of components are checked along the way, which in practice are even hardly looked at. For instance, in a hybrid system that uses a cryptosystem — also used elsewhere — to generate the key of another cryptosystem, no guarantee about the security of the combinantion is provided, even if each primitive is provably secure and their parameters are set so that security requirements are met. A security proof of the combination needs to be provided (see Chapter 8).

This discussion remains in the field of the study of the security of cryptographic algorithms. The final purpose of provable security and security proofs is to guarantee the real security of a cryptosystem's implementation, not just of its algorithm.

Even the computational model is just a model that ignores many important aspects of reality. In particular, it ignores physical attacks against devices: side-channel attacks that may exploit for instance power consumption, timing, noise and fault attacks — an attack that introduces faults in the system in order to break its security from provoking unexpected behaviour. As cryptosystems are better studied and verified formally, physical attacks become

increasingly important and are an area of active research. Some of my collaborators have worked in this direction, producing the Jasmin programming language whose « compiler is designed to achieve predictability and efficiency of the output code (currently limited to x64 platforms), and is formally verified in the Coq proof assistant. » ([Almeida et al., 2017])

In [Almeida et al., 2019b], we propose both a formal security proof and an efficient implementation of the SHA-3 hash function family:

> Our implementation is written in the Jasmin programming language, and is formally verified for functional correctness, provable security and timing attack resistance in the EasyCrypt proof assistant. [...] Concretely, our mechanized proofs show that: 1) the SHA-3 hash function is indifferentiable from a random oracle, and thus is resistant against collision, first and second preimage attacks; 2) the SHA-3 hash function is correctly implemented by a vectorized x86 implementation. Furthermore, the implementation is provably protected against timing attacks in an idealized model of timing leaks.

In [Almeida et al., 2019a], my collaborators propose a fast and side-channel secure implementation of the authenticated encryption ChaCha20&Poly1305. In Chapter 8, I propose a formal security proof of ChaCha20&Poly1305, that is functionally equivalent to their implementation in Jasmin.

Chapter 2

EasyCrypt and General Formalization Techniques

The interactive theorem prover EasyCrypt was first presented in [Barthe et al., 2011a] as an implementation of a *code-based game-based* approach to practice-oriented provable security in the computational model. It diverged from CertiCrypt [Barthe et al., 2009], a Coq framework used to prove the formal semantic security of OAEP.

This chapter gives a general presentation of EasyCrypt, states clearly *what an adversary can do* (i.e. adversarial strategies included in the security statements), and regroups formalization techniques that are used in the different proofs.

2.1 Foundations

EasyCrypt is a proof assistant that relies on the *goal directed proof* approach. Also described in [Bertot, 2006] for Coq, this approach usually has the following type of scenario:

1. the user enters a statement that he wants to prove (using the command lemma) along with a name for later reference,
2. the EasyCrypt system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this particular proof,
3. the user enters a command to decompose the goal into simpler ones, ideally upto axioms or formula that evaluate to true,
4. the EasyCrypt system displays a list of formulas that still need to be proven,
5. back to step 3.

The commands used at step 3 are called *tactics*. Some of these tactics actually decrease the number of goals. When there are no more goals, the proof is complete. When the user enters the command qed, the lemma is saved with the name given at step 1.

Before stating a lemma to prove, the user may define the structures he will work on, some axioms he assumes, or import libraries that contain such definitions. For that, EasyCrypt has a typed expression language based on the polymorphic typed lambda calculus. In EasyCrypt, *types* are non-empty sets of values and *operators* are typed functions on these sets. The internal kernel of EasyCrypt provides basic built-in types such as bool, int, real, and unit (the type inhabited by the single element tt or ()). The standard libraries includes formalization of lists, arrays, sets, finite sets, maps, finite maps, distributions, etc....

Remark

In EasyCrypt, an integer n can be recast to the real type using the notation n%r.

EasyCrypt

The polymorphic option type encapsulates an optional value. A value of type α option is either Some x where x is of type α, or None.

```
type α option = [ None | Some of α ].
```

The operator oget extracts the optional value by associating to the value None a default value named witness in EasyCrypt (as mentioned above, all types are inhabited).

```
op oget [α] (o : α option) : α =
  with o = None ⇒ witness
  with o = Some a ⇒ a.
```

The system of *theory* in EasyCrypt allows the user to regroup related types, predicates, operators, modules, axioms, and lemmas. EasyCrypt allows users to declare and specify their data-types and operators, including inductive data-types and operators defined by pattern matching. Types and operators without definitions are said *abstract* and can be seen as *parameters* to the rest of the context. Theory parameters (types, operators, and predicates) left abstract when proving its lemmas may be instantiated via a *cloning* process. For instance, theory cloning allows the development of generic proofs that can later be instantiated with concrete parameters.

EasyCrypt

Axioms can restrict parameters and are discharged during particular instanciation.

```
op σ : int.
axiom σ_gt0 : 0 < σ. (* we assume that σ is positive *)
```

2.1.1 Probability Distributions

To every type t is associated the type t distr of *real discrete sub-distribution*. In some cases, we may use the notation $\mathscr{D}(\mathsf{t})$ to express the set of all sub-distributions on a type t. A discrete sub-distribution over a type t is fully defined by its *mass function*. A mass function f is a non-negative function from t to $\mathbb{R}$ and is defined over a discrete support while verifying:

$$\sum_{x:\mathsf{t}} f(x) \leq 1$$

EasyCrypt

When the sum is equal to 1, the sub-distribution is called a *distribution* and the keyword for this assumption is is_lossless.

```
type t.
op dt : t distr.
axiom dt_ll : is_lossless dt.
```

When the mass function of the sub-distribution is positive over all of the elements of type t, the sub-distribution is said to be *full*.

```
axiom dt_full : is_full dt.
```

When the mass function of the sub-distribution associates every value of the support of the sub-distribution to the same mass, the sub-distribution is said to be *uniform*.

```
axiom dt_uniform : is_uniform dt.
```

The keyword for a sub-distribution that is full and uniform is is_funiform.

```
axiom dt_funiform : is_funiform dt.
```

The mass function of a sub-distribution is represented by the operator mu1, while mu dt P represents the mass of the set of all values that satisfy the proposition P. In fact, internally, mu1 is defined using the operator mu.

```
op mu : t distr → t prop → real.
op mu1 (dt : t distr) (x : t) : real = mu dt (fun y ⇒ x = y).
```

2.1.2 Modules in EasyCrypt

Programs are formalized in EasyCrypt as *modules*: stateful "objects" consisting of *global variables* and *procedures*. Global variables are visible outside the modules and define the internal state of the module at any given time. A procedure is composed of the declarations of local variables and a sequence of *instructions*. An instruction can be an assignment, a random sampling (denoted by $\xleftarrow{\$}$), a call to another procedure (possibly obtaining the return value), a conditional branch, a while loop, or a return statement. This simple imperative probabilistic programming language is named **pWhile**.

EasyCrypt

This exemple defines a module with one global variable Multiply.factor, and defines one procedure Multiply.mul that outputs the product of its input to its global variable.

```
module Multiply = {
  var factor : int (* global variable *)
  proc mul (n : int) : int = {
    (* factor refers to Multiply.factor *)
    return n * factor;
}}.
```

The module system in EasyCrypt manages higher-order modules. This feature allows a cryptographer to define a general game by taking an adversary and a cryptosystem as parameters. Before declaring a game that takes some modules as parameters, a *module type* needs to be defined for each different type of parameter.

EasyCrypt

In this example, the module type Oracle declares the interface of the cryptosystem, Adversary declares the interface of the adversary, and the module Game defines the cryptographic game when given an adversary and a cryptosystem.

```
module type Oracle = {
  proc init () : unit
  proc f (_: int) : bool
}.
```

```
module type Adversary (O : Oracle) = {
  proc guess () : bool
}.
```

```
module Game (A : Adversary) (O : Oracle) = {
  proc main () : bool = {
    var b : bool;
    O.init();
    b ← A(O).guess();
    return b;
}}.
```

Probability of some event in a game

Formally, an instruction operates on program *memories*, which map local and global variables to values. Let $\mathscr{M}$ be the set of memories.

EasyCrypt

A memory is handled differently from other types. For instance, anonymous functions (fun (x : t) => ...) and operators cannot take memories as input, while quantification on memories is allowed. The quantification ∀ &m ranges over all memories with domain equal to the set of all variables declared as global in currently declared modules. For instance, with those declarations:

```
module X = { var a : int }.
module Y = { var b : int }.
```

the following formula is well-formed and evaluates to true:

```
∀ &m, X.a{m} < Y.b{m} ⇒ X.a{m} + 1 ≤ Y.b{m}
```

An *event* is a boolean expression over non-free logical variables and program variables that are tagged with a memory. If &m is a memory and x is a program variable in the domain of &m, then $x\langle m\rangle$ [1] is the expression for the value of x in &m, and $\psi\langle m\rangle$ is the expression where all the program variables occurring in ψ are replaced with their value in &m.

The semantics of a sequence c of instructions is a function $[\![c]\!] : \mathscr{M} \to \mathscr{D}(\mathscr{M})$ from program memories to sub-distributions on program memories. If one possible execution of c does not terminate, c generates a sub-distribution with total probability less than 1. The *probability* of some *event* ψ in a program c when starting with initial memory &m is defined as the sum of the masses of all memories in the distribution $[\![c]\!]\langle m\rangle$ that satisfy ψ:

$$\Pr[c, m : \psi]$$

The EasyCrypt notation Pr[c@&m : ψ] is equivalent to Pr[c, m : ϕ], the latter being more readable and the former more accurate to what EasyCrypt is expecting.

Adversary as a module, Quantification over all adversaries

Adversaries are modeled in EasyCrypt as *abstract* modules of a defined module type. The module type defines the type of procedures, but the code of an abstract module is unknown.

1. This notation has been chosen for text readability, even if x{m} is the notation used in EasyCrypt

EasyCrypt

An abstract module is declared locally using quantification over a module type: ∀ (A <: Adversary). Another way to declare an abstract module is using inside a section the declare module feature of EasyCrypt. The system of *sections* has the primary goal to hide intermediate details of a proof. Inside a section, which may be named, an abstract module can be declared so that all lemmas that appear in the section will appear with a supplementary quantification. In the following, some notations have not been described so far, but they are not relevant for the EasyCrypt notion of sections, so we explain them in the following sections.

Imagine the following example of a lemma that is declared in a section in a context with the module Game declared as follows and with a defined (but not presented) module O : Oracle.

```
module Game (A : Adversary) (O : Oracle) = {
  var i : bool
  proc main () = {
    A(O).guess();
    i ←$ {0,1};
}}.
```

```
section Proof.
  declare module A : Adversary.
  axiom A_ll : (∀ (O1 <: Oracle{A}),
    islossless O1.f ⇒ islossless A(O1).guess.
  lemma toto &m : Pr [ Game(A,O).main() @ &m : Game.i = true ] = 1%r / 2%r.
end section Proof.
```

When used outside of the section, the lemma toto looks like:

```
lemma toto : ∀ (A <: Adversary),
  (∀ (O1 <: Oracle{A}), islossless O1.f ⇒ islossless A(O1).guess) ⇒
  ∀ &m, Pr [ Game(A,O).main() @ &m : Game.i = true ] = 1%r / 2%r.
```

Remark

In the previous example, the procedure A(O).guess may call all procedures of O, but EasyCrypt features restrictions on which procedures can be queried. This point is described in more details in Section 2.2.2.

Remark

Consider a module A declared with module type Adversary. Its procedures may be referred to (in lemmas, judgments, or code) only when an oracle O is provided. The procedure A(O).guess is correct, whereas A.guess is not well-formed.

EasyCrypt allows the user to reason about three different types of facts about probabilistic programs. They are called *judgments* and each type specifies what it actually means for the programs:

— HL, *Hoare Logic*, with probabilistic programs,

— pHL, *probabilistic Hoare Logic*, that allows one to carry out proofs about the probability of a procedure's execution resulting in a post-condition holding, and

— pRHL, *probabilistic Relational Hoare Logic*, that relates a pair of procedures.

Hoare Logic

A *Hoare judgment* is a triplet $[\mathsf{c} : \phi \Longrightarrow \psi]$ composed of a (probabilistic) program c and two predicates ϕ and ψ. It states that for all memory &m satisfying the *precondition* ϕ, then all memories in the support of the sub-distribution $[\![\mathsf{c}]\!]\langle \mathsf{m}\rangle$ satisfy the *post-condition* ψ. When the program c is deterministic, $[\![\mathsf{c}]\!]\langle \mathsf{m}\rangle$ has a support of at most one element.

EasyCrypt

The main tactic for an abstract procedure call is the tactic named proc. Consider a Hoare judgment $[\mathsf{A(O).g} : \phi \Longrightarrow \psi]$ in the context of an abstract procedure named A(O).g of the abstract module A that takes a module O as parameter. The tactic proc takes one input: a formula I that represents an invariant.

One constraint needs to be checked for the tactic proc to work: A *should have no write access to any global variable that appears in* I. The intuition is that the rule for the adversary is analogous to the rule with the loops. If each iteration keeps the validity of the invariant, and the invariant is valid before the loop, then the full loop respects the invariant. Unspecified code with oracle access to some procedures is analogous to a "loop" that "iterates" on oracle calls, with some code between calls.

The rule checks that the unspecified code of the procedure A(O).g cannot modify the invariant I, and asks to prove that each oracle call which A(O).g can query respects the invariant. The procedures accessible by A(O).g are from the module type of A.

$$\frac{\forall \mathsf{f} \text{ accessible procedure by } \mathsf{A(O).g}, [\mathsf{O.f} : \mathsf{I} \Longrightarrow \mathsf{I}]}{[\mathsf{A(O).g} : \mathsf{I} \Longrightarrow \mathsf{I}]}\ [\mathsf{proc\ I}]$$

Probabilistic Hoare Logic

A *probabilistic Hoare judgment* is a quintuplet $[\mathsf{c} : \phi \Longrightarrow \psi] \diamond p$ composed of a program c, two predicates ϕ and ψ, a relation on reals $\diamond$ taken from the set $\{\leq, =, \geq\}$, and a real p. It states that $\forall \&\mathsf{m}, \phi\,\langle \mathsf{m}\rangle \Rightarrow \Pr[\mathsf{c}, \mathsf{m} : \psi] \diamond p$.

EasyCrypt

A very useful tactic in the context of a pHL statement about a sequence of instructions $[\mathsf{c}_1; \mathsf{c}_2 : \phi \Longrightarrow \psi] \diamond \mathsf{p}$ is the tactic named seq i : b p_1 p_2 p_3 p_4 I. The arguments mean:

- i: the line number that specify the beginning of c_2,
- b: a boolean formula that may be evaluated to true or false,
- p_1: the probability that b holds at the end of c_1, i.e. $[\mathsf{c}_1 : \phi \Longrightarrow \mathsf{b}] \diamond \mathsf{p}_1$,
- p_2: the probability that ψ holds after c_2 if b holds, i.e. $[\mathsf{c}_2 : \mathsf{I} \wedge \mathsf{b} \Longrightarrow \psi] \diamond \mathsf{p}_2$,
- p_3: the probability that b does not hold at the end of c_1, i.e. $[\mathsf{c}_1 : \phi \Longrightarrow \neg\mathsf{b}] \diamond \mathsf{p}_3$,
- p_4: the probability that ψ holds after c_2 if b does not hold, i.e. $[\mathsf{c}_2 : \mathsf{I} \wedge \neg\mathsf{b} \Longrightarrow \psi] \diamond \mathsf{p}_4$,
- I: a property always satisfied at the end of c_1, regardless of the evaluation of b, i.e. $[\mathsf{c}_1 : \phi \Longrightarrow \mathsf{I}]$.

When applied, the tactic also asks the user to prove the goal: $\mathsf{p}_1\mathsf{p}_2 + \mathsf{p}_3\mathsf{p}_4 \diamond \mathsf{p}$.

Probabilistic Relational Hoare Logic

A *probabilistic relational Hoare judgment* [2] is a quadruplet $[c_1 \sim c_2 : \phi \Longrightarrow \psi]$ composed of two programs c_1 and c_2, and two relations ϕ and ψ. More specifically, the precondition ϕ and post-condition ψ are first-order formulate built from *relational expressions* that are then interpreted as a relation on program memories.

Whereas for HL and pHL judgments only one memory is considered, a pRHL judgment distinguishes between the two memories for each side. Therefore, relational expressions are arbitrary boolean expressions over program variables tagged with $\langle 1 \rangle$ or $\langle 2 \rangle$ (to specify from which of the programs they are from: $\langle 1 \rangle$ for c_1, $\langle 2 \rangle$ for c_2) and logical variables that only appear when quantified. By abuse of notation, $e\,\langle i \rangle$ stands for the expression e in which all program variables have been tagged with $\langle i \rangle$.

EasyCrypt

The notation for the equality of a program variable X.x in both memories is ={X.x}, and the notation ={X.x, Y.y} stands for ={X.x}∧ ={Y.y}. For any module M, the set of all the variables a module accesses is denoted glob M. The precondition ={glob A} is particularly useful for an abstract module A when one wants to state that it will have the same behavior in both memories.

A pRHL judgment $[c_1 \sim c_2 : \phi \Longrightarrow \psi]$ states that for any pair of initial memories m_1, m_2 satisfying the precondition ϕ, denoted $\langle m_1, m_2 \rangle \vdash \phi$, the distributions $[\![c_1]\!]\langle m_1 \rangle$ and $[\![c_2]\!]\langle m_2 \rangle$ satisfy the lifting $\mathscr{L}$ of post-condition ψ, denoted $\langle ([\![c_1]\!]\langle m_1 \rangle), ([\![c_2]\!]\langle m_2 \rangle) \rangle \vdash \mathscr{L}(\psi)$. The lifting of a relation on memories to a relation on memory distributions is defined as a max-cut min-flow problem, in the style of [Jonsson et al., 2001]. Formally, let $\mu_1 \in \mathscr{D}(A)$ be a probability distribution on a set A and $\mu_2 \in \mathscr{D}(B)$ a probability distribution on a set B. The lifting $\langle \mu_1, \mu_2 \rangle \vdash \mathscr{L}(R)$ of a relation $R \subseteq A \times B$ to μ_1 and μ_2 is defined as follows:

$$\exists \mu : \mathscr{D}(A \times B), \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \forall (a, b) \in A \times B, \mu(a, b) > 0 \Rightarrow \langle a, b \rangle \vdash R$$

where the projections $\pi_1(\mu)$ and $\pi_2(\mu)$ are defined as

$$\pi_1(\mu)(a) \stackrel{\text{def}}{=} \sum_{b \in B} \mu(a, b) \qquad \pi_2(\mu)(b) \stackrel{\text{def}}{=} \sum_{a \in A} \mu(a, b)$$

Claims about probabilities can be derived from valid pRHL judgments by means of the following rules:

$$\frac{\langle m_1, m_2 \rangle \vdash \phi \quad [c_1 \sim c_2 : \phi \Longrightarrow \psi] \quad \forall \&m_1', \&m_2', \langle m_1', m_2' \rangle \vdash \psi \Rightarrow \left(E_1 \langle m_1' \rangle \Leftrightarrow E_2 \langle m_2' \rangle\right)}{\Pr[c_1, m_1 : E_1] = \Pr[c_2, m_2 : E_2]} \text{ [PrEq]}$$

$$\frac{\langle m_1, m_2 \rangle \vdash \phi \quad [c_1 \sim c_2 : \phi \Longrightarrow \psi] \quad \forall \&m_1', \&m_2', \langle m_1', m_2' \rangle \vdash \psi \Rightarrow \left(E_1 \langle m_1' \rangle \Rightarrow E_2 \langle m_2' \rangle\right)}{\Pr[c_1, m_1 : E_1] \leq \Pr[c_2, m_2 : E_2]} \text{ [PrLe]}$$

2. This description is from [Barthe et al., 2011a], in which more details can be found.

EasyCrypt

The tactic proc also exists in pRHL. It is used to relate two instances of the same abstract procedure A(·).g that may use different implementations of its oracles. The tactic takes one input: a relation I that represents an invariant. *The invariant should not include any global variable that may be overwritten by the abstract module* A (same as for HL).

This tactic formalizes the intuition that the abstract procedure will have the same behavior with both implementations and preserve the invariant I if the abstract module starts with the same global state and, for every procedure, both implementations stay in relation by the invariant I while answering the same output for the same query.

$$\frac{\forall \mathsf{f} \text{ accessible procedure by A}, [\mathsf{O1.f} \sim \mathsf{O2.f} : \mathrm{I} \wedge =\{\mathsf{arg}\} \Longrightarrow \mathrm{I} \wedge =\{\mathsf{res}\}]}{[\mathsf{A(O1).g} \sim \mathsf{A(O2).g} : \mathrm{I} \wedge =\{\mathsf{arg}, \mathsf{glob\ A}\} \Longrightarrow \mathrm{I} \wedge =\{\mathsf{res}, \mathsf{glob\ A}\}]}\ [\mathsf{proc\ I}]$$

Remark

When an equivalence of two events in each program is lifted to relate those two programs, then, from the definition of the projections, both must have the same termination probability. This is particularly important for the theorem about failure events that is explained in the next paragraph.

EasyCrypt

The EasyCrypt keyword islossless is a predicate on a procedure that states that for all input, the procedure terminates with probability 1, i.e.

$$\mathsf{islossless\ O.f} := \text{“}[\mathsf{O.f} : \mathsf{true} \Longrightarrow \mathsf{true}] = 1\text{”}$$

Failure Event

A relation between games defines a transition of equivalent games in a cryptographic security proof. However, games are not always equivalent, and a lossy transformation may be needed. One common technique to justify such a lossy transformation is to annotate both games with a fresh boolean flag (often named bad) that is initialized to the false value and set to true whenever the code of games differ. Such transitions are justified using the Fundamental Lemma, first used and introduced in [Shoup, 2004, Bellare and Rogaway, 2006].

Lemma 2.1.1 (Fundamental Lemma [Barthe et al., 2010])**.** *For any pair of games* G_1, G_2 *and events* E_1, E_2*, and* B*, such that* E_1 *is well-defined in* G_1*,* E_2 *in* G_2*, and* B *in both games:*

$$\Pr[G_1, m : E_1 \wedge \neg B] = \Pr[G_2, m : E_2 \wedge \neg B]$$
$$\Rightarrow |\Pr[G_1, m : E_1] - \Pr[G_2, m : E_2]| \leq \max(\Pr[G_1, m : B], \Pr[G_2, m : B])$$

When an adversary is involved in both games, then one more assumption is needed: the adversarial computation should terminate with probability 1 in at least one game. Assuming that G_1 terminates with probability 1, then $\Pr[G_2, m : B] \leq \Pr[G_1, m : B]$. In this case, the upper bound on the loss of the transition is quantified by $\Pr[G_1, m : B]$.

EasyCrypt

The assumption on the termination of adversarial code is generalized in EasyCrypt; it requires the adversary to end with probability 1 with any terminating oracles. Such an hypothesis should be manually added whenever it is required.

The following axiom is an example using the module types of Section 2.1.2:

```
declare module A : Adversary.
axiom A_ll : ∀ (O <: Oracle{A}),
  islossless O.init ⇒ islossless O.f ⇒
  islossless A(O).guess.
```

EasyCrypt

The tactic proc can also be applied for an upto-bad reasoning that relates the same abstract procedure with different implementations of its oracles. The tactic proc B I J takes three inputs — a failure event B (defined in the right implementation), the invariant I that holds as long as B does not hold, and the invariant J that holds when B holds. The tactic proc B I is a shortcut for proc B I true.

To prove the conclusion C, EasyCrypt creates the sub-goals H_1, H_2, H_3, H_4:

H_1 The adversary always terminates if its oracles terminate:

$$\forall (O <: \mathsf{Oracle}\{A\}), (* \ \forall f \ *) \ \mathsf{islossless}\ O.f \Rightarrow \mathsf{islossless}\ A(O).g$$

H_2 Assuming the failure event B does not hold before a query, the relation at the end of the query depends on the occurrence of the failure event:

$$\forall f \text{ procedure}, [O1.f \sim O2.f : \neg B\langle 2\rangle \wedge =\{arg\} \wedge I \Longrightarrow \text{if } B\langle 2\rangle \text{ then } J \text{ else } =\{res\} \wedge I]$$

H_3 If the failure B event has already occurred, then, in the left implementation, the relation J remains true with probability 1 even when the right memory is fixed:

$$\forall f \text{ procedure}, \forall \&m_2, B\langle m_2\rangle \Rightarrow [O1.f : J\langle \cdot, m_2\rangle \Longrightarrow J\langle \cdot, m_2\rangle] = 1$$

where the event $J\langle \cdot, m_2\rangle$ is the predicate: $\&m \to \langle m, m_2\rangle \vdash J$.

H_4 The failure event B in the right implementation remains true with probability 1:

$$\forall f \text{ procedure}, [O2.f : B \Longrightarrow B] = 1$$

C If all those conditions are provable, this use case for the tactic proc makes it possible to prove the following goal:

$$[A(O1).g \sim A(O2).g : =\{arg, glob\ A \wedge I\} \Longrightarrow \text{if } B\langle 2\rangle \text{ then } J \text{ else } =\{res, glob\ A\} \wedge I]$$

2.2 Adversarial Model

All security statements in my manuscript rely on the same definition of adversaries based on the EasyCrypt adversarial model. Therefore, any strategy that cannot be captured in this

model is not captured by the security statements. Some restrictions on adversaries are local to the security statements, but are expressed using the same terminology.

Cryptographic security statements are expressed by quantifying on all adversaries. In EasyCrypt, this quantification is done using abstract modules and module types. Therefore, all algorithms that may be expressed as an EasyCrypt module that satisfy all the restrictions of the security statement are captured by the quantification.

2.2.1 Restrictions on Oracle Procedure Calls

In the implementation of an oracle, some of its procedures are used to initialize its internal state. This kind of procedure is meant to be accessible to the game from the security definition, but not to the adversary. This kind of restriction can be achieved either using different module types between the adversary's oracles and the game's oracles or by restricting the oracle procedures allowed to query directly in the module type of an adversary.

EasyCrypt

In the following example, any module A : Adversary should have a procedure named guess that may only call the procedures f and g of its oracle O. Therefore, a module with a procedure guess that calls its oracle's procedure init is not of module type Adversary.

```
type f_input, g_input, f_output, g_output.
module type Oracle = {
  proc init () : unit
  proc f (x : f_input) : f_output
  proc g (x : g_input) : g_output
}.
module type Adversary (O : Oracle) = {
  proc guess () : bool { O.f O.g }
}.
```

2.2.2 Adversarial Restrictions on Program Variables

In EasyCrypt, *program variables* may be *local* to a procedure or *global* to the full scope. Local program variables are inaccessible to any other procedure other than the one in which they are defined. In a security game, the adversary should not be able to access some global variables, e.g. the secret key of an encryption scheme. This is expressed in EasyCrypt when an abstract module is declared either in a quantification or in a section using declare module.

Some tactics require the declared abstract module to have no access on global variables that appear in the invariant, e.g. the proc tactic and all its declinations. This kind of restriction is possible when the abstract module is declared.

EasyCrypt

In the following example, the adversary A on which the property is quantifying over should never use the global variable Enc.key, either reading or overwriting its content.

```
type key.
module Enc {
  var key : key
  proc (* ··· *)
}.
lemma security : ∀ (A <: Adversary { Enc }),
    (* some property *).
```

In the case of a module declared inside a section, new modules may be implemented after this declaration. The local keyword binds the implementation of a module inside a section and states that any abstract module previously declared may have no access to its global variables. For instance, the two following scripts are equivalent in meaning, except that in practice the module O is hidden in the left script when applying any lemma from the section.

```
section.
  declare module A : Adversary.
  local module O : Oracle = { var c : int }.
end section.
```

```
module O : Oracle = { var c : int }.
section.
  declare module A : Adversary { O }.
end section.
```

Remark

This feature restricts all accesses, both for reading and writing. If one may want an abstract module to have reading access to a global variable without being able to alter it, direct access to the global variable should remain restricted in the declaration of the abstract module. Then, provide to the module type another oracle which outputs the value of the desired global variable, this acts like a "getter". This concept can be extended to the inclusion of a "setter".

2.2.3 Computation Time Restrictions

Usually, in cryptography, adversaries are *probabilistic polynomial-time* (PPT) algorithms [3], i.e. their running time is upper bounded by a polynomial expression in the size of the *input* of the algorithm, also named *security parameter* in cryptography.

Security statements are usually of the form:

$$\forall \mathscr{A} \text{ PPT}, \exists \mathscr{B} \text{ PPT}, \mathsf{Adv}^{\mathrm{xxx}}_{\mathrm{X}}(\mathscr{A}) \leq \alpha + \mathsf{Adv}^{\mathrm{yyy}}_{\mathrm{Y}}(\mathscr{B})$$

while *security assumptions* are usually:

$$\forall \mathscr{B} \text{ PPT}, \mathsf{Adv}^{\mathrm{yyy}}_{\mathrm{Y}}(\mathscr{B}) \text{ is "negligible" in the security parameter.}$$

3. Complexity is also bounded in terms of computation space, but we do not discuss it here.

Remark

The PPT property for adversarial algorithms is important for the "negligibility" of the bound. Often in cryptography, a negligible bound is about (or less than) 2^{-80}. Of course, this is dependent on the risks and security goals of particular applications.

EasyCrypt

Computation time, for example number of processor clock cycles, is not embedded in the logics of EasyCrypt but can be modeled using, for example, ghost variables that simulates computation times. Since adversaries can be any module of the specified module type, security statements are expressed in EasyCrypt without PPT restriction:

$$\forall \mathscr{A}, \mathsf{Adv}_{\mathrm{X}}^{\mathrm{xxx}}(\mathscr{A}) \leq \alpha + \mathsf{Adv}_{\mathrm{Y}}^{\mathrm{yyy}}(\mathscr{B}(\mathscr{A}))$$

with a concrete implementation of $\mathscr{B}(\cdot)$ from $\mathscr{A}$ that needs to be checked so that:

$$\forall \mathscr{A}, \mathscr{A} \text{ is PPT} \Rightarrow \mathscr{B}(\mathscr{A}) \text{ is PPT.}$$

Example

In certain games, there may exist a non-PPT adversary C that has a non-negligible advantage $\mathsf{Adv}_{\mathrm{Y}}^{\mathrm{yyy}}(\mathscr{C})$. For instance, consider the case of any scheme that relies on the difficulty in finding the decomposition in prime factors of a large number, e.g. RSA, or the discrete logarithm, e.g. ElGamal. There exists a non-PPT adversary decomposing a large number into prime factors: the one that tests every prime inferior to the input number for divisibility. This adversary has the maximum advantage because it always succeeds. However, this cannot be considered to be a successful attack, since it would take too much time and effort because of the (sub-)exponential complexity.

When analyzing the security of a cryptosystem, adversarial computation time can be restricted indirectly. For instance, a common restriction is to bound the number of queries an adversary can make, where oracle queries formalize the computation of cryptographic primitives. In those cases, the bound on the number of query calls often appears in the security statement and the security bound. Counting queries may be more complex and such an example is shown in Chapter 6.

EasyCrypt

To summarize *what* an adversary can do:

— It is a probabilistic program, expressible in EasyCrypt as a module from the given module type.

— It has access to some oracles (accessible procedures).

— It is restricted in the internal states it can access and interfere with.

— Its computation space is *unbounded*.

— Its computation time is *unbounded*. If the proof involves an upto-bad argument, it should terminate with probability 1 for any implementation of its oracles.

2.3 Formalization Techniques

In the following chapters of my manuscript, I describe security proofs of three cryptographic standards. In those proofs, a few fomalization techniques are often used, and I choose to gather them in this autonomous section so that each proof can be followed on its own, without pointing to another security proof.

2.3.1 PRP-PRF switching lemma

In game-based proofs, it is often useful to switch between a random permutation and a random function (see Chapter 4 and Chapter 6). This section starts with the definition of a random permutation and a random function. Then I describe their formalization in EasyCrypt and state the PRP-PRF switching and the strong PRP-PRF switching lemmas.

Remark

This formalization has been done time and time again (in [Affeldt et al., 2007, Barthe et al., 2010], to only cite a few formalizations), and is not part of my personal contribution. However, this is a good introduction to the techniques of EasyCrypt.

Let S be a finite set, Perm(S) the set of all permutations over S, and Fun(S) be the set of all functions over S. In this context, the sampling $p \xleftarrow{\$} \mathsf{Perm}(S)$ means sampling a random permutation p *uniformly* from the set of all permutations. Respectively $f \xleftarrow{\$} \mathsf{Fun}(S)$ means sampling a random function f uniformly from the set of all functions.

EasyCrypt

The formalization in EasyCrypt of a random permutation and a random function uses modules that are defined over an abstract type t that represents the set S and an abstract distribution operator dt that represents the uniform distribution over the finite abstract type t.

```
type t.
op dt : t distr.
axiom dt_ll : is_lossless dt.
axiom dt_funi : is_funiform dt.
```

Listing 2.1 – Abstract parameters.

The module Map declares the finite map of the oracle that is visible by the adversary. The module type Oracle defines the expected oracle interface.

```
module Map = { var m : (t, t) fmap }.
module type Oracle = {
  proc init () : unit
  proc f (_: t) : t
}.
```

The predicate rng Map.m x is true if the element x is in the range of the finite map Map.m. The distribution dt \ rng Map.m is the uniform distribution over the set of elements in S that do not validate the predicate rng Map.m.

```
module RF : Oracle = {
  proc init () : unit = { Map.m ← empty; }
  proc f (x : t) : t = {
    if (x \notin Map.m) {
      Map.m[x] ←$ dt;
    }
    return oget Map.m[x];
}}.
```

Listing 2.2 – Random function

```
module RP : Oracle = {
  proc init () : unit = { Map.m ← empty; }
  proc f (x : t) : t = {
    if (x \notin Map.m) {
      Map.m[x] ←$ dt \ rng Map.m;
    }
    return oget Map.m[x];
}}.
```

Listing 2.3 – Random permutation

The indistinguishability advantage is bounded by a formula that involves the maximum number of oracle calls it can make. Indeed, if the adversary can query all values of the domain of its oracle, it has a high probability to distinguish between RP/RF. This restriction is formalized in the following module named Count.

```
op c : int.
```

```
module Count (E : Oracle) : Oracle = {
  var counter : int
  proc init () : unit = {
    counter ← 0;
    E.init();
  }
  proc f (x : t) : t = {
    var y : t ← witness;
    if (counter < c) {
      y ← E.f(x);
      counter ← counter + 1;
    }
    return y;
}}.
```

An adversary may not (re-)initialize its oracle. This is either done by restricting procedure accesses in the declaration of the procedures, or by declaring another module type for oracles. The second option requires module type management and is possible, but may be tedious.

```
module type Adversary (O : Oracle) = {
  proc guess () : bool { O.f }
}.
```

The distinguishing game Dist, when given an adversary and an oracle, initializes the oracle and the counter, gives to the adversary the restricted oracle, and outputs the answer of the adversary.

```
module Dist (A : Adversary) (O : Oracle) = {
  proc main () : bool = {
    var b;
    Count(O).init();
    b ← A(Count(O)).guess();
    return b;
}}.
```

Lemma 2.3.1 (PRP–PRF switching). *For any natural number c, any lossless adversary $\mathscr{A}$ distinguishes a random permutation from a random function with probability:*

$$\left|\Pr\left[\pi \xleftarrow{\$} \mathrm{Perm}(S); b \leftarrow \mathscr{A}^{\pi} : b = 1\right] - \Pr\left[f \xleftarrow{\$} \mathrm{Fun}(S); b \leftarrow \mathscr{A}^{f} : b = 1\right]\right| \leq \frac{c \cdot (c-1)}{2 \cdot |S|}$$

when the oracle starts to always output a default value witness *after c queries.*

```
lemma prp_prf_switching &m :
(* ∀  adversary A that cannot access internal states of Map and Count *)
∀  (A <: Adversary { Map, Count }),
  `| Pr[Dist(A,RP).main() @ &m : res] − Pr[Dist(A,RF).main() @ &m : res] | ≤
```

```
  (c*(c-1)/2) * mu dt witness.
(* 'mu dt witness' represents the probability a value sampled from 'dt' is equal to 'witness' *)
```

Proof. Let &m be the initial memory, and Event.bad be the failure event.

```
module Event = { var bad : bool }.
```

This proof has a simple sequence of games:

1. $\Pr[\text{Dist}(A, \text{RP}).\text{main}(), m : \text{res}] = \Pr[\text{Dist}(A, \text{RPbad}).\text{main}(), m : \text{res}]$
2. $\Pr[\text{Dist}(A, \text{RF}).\text{main}(), m : \text{res}] = \Pr[\text{Dist}(A, \text{RFbad}).\text{main}(), m : \text{res}]$
3. $|\Pr[\text{Dist}(A, \text{RFbad}).\text{main}(), m : \text{res}] - \Pr[\text{Dist}(A, \text{RPbad}).\text{main}(), m : \text{res}]| \leq \Pr[\text{Dist}(A, \text{RFbad}).\text{main}(), m : \text{Event.bad}]$
4. $\Pr[\text{Dist}(A, \text{RFbad}).\text{main}(), m : \text{Event.bad}] \leq c \cdot (c-1)/(2 \cdot |S|)$

where the oracles RFbad and RPbad are defined in the following way and are respectively equivalent to RF and RP while capturing the failure event directly in their code.

```
module RFbad : Oracle = {
  proc init () : unit = {
    Map.m ← empty;
    Event.bad ← false;
  }
  proc f (x : t) : t = {
    var y : t ← witness;
    if (x \notin Map.m) {
      y $← dt;
      if (rng Map.m y) {
        Event.bad ← true;
        (* y $← dt \ rng Map.m; *)
      }
      Map.m[x] ← y;
    }
    return oget Map.m[x];
}}.
```

```
module RPbad : Oracle = {
  proc init () : unit = {
    Map.m ← empty;
    Event.bad ← false;
  }
  proc f (x : t) : t = {
    var y : t ← witness;
    if (x \notin Map.m) {
      y $← dt;
      if (rng Map.m y) {
        Event.bad ← true;
        y $← dt \ rng Map.m;
      }
      Map.m[x] ← y;
    }
    return oget Map.m[x];
}}.
```

1. The functional equivalence between RF and RFbad is easy to show in EasyCrypt.
2. The functional equivalence between RP and RPbad relies on the fact that sampling from either of the following distributions yields the same probability:
   — sample directly from 'dt \ rng Map.m' and
   — sample from 'dt', then if the value belongs to the range of 'Map.m', re-sample another value from 'dt \ rng Map.m'.
3. The upto-bad step is a direct application of the Fundamental Lemma.
4. The computation of the probability of setting Event.bad ← true is done using the advanced tactic named `fel` for which a documentation can be found online at https://www.easycrypt.info/documentation/refman.pdf. This tactic captures the intuition that if the adversary has a limited number of queries it can

make, if the probability of the event for one query is bounded by some expression, then the probability the adversary triggers the event is bounded by the sum of all those expressions.

□

Stronger version

The stronger version of this theorem, used in Chapter 6, gives to the adversary an access to the *inverse*. Because the inverse of a random function is not defined on elements (but mathematically on sets), it is replaced it by a random *transformation*. It is formalized as a module with an internal state that keeps track of all previous queries to both the direct oracle and its inverse, and for any fresh query to either samples an output and updates its internal state. This last step may overwrite a previous query to the other oracle, therefore, a random transformation does not ensure the full consistency of the output of previous queries.

Because of this instability of behavior, the set of all transformations is not well defined. Therefore, by abuse of notation, *in the next lemma* the sampling $f \xleftarrow{\$} \mathsf{Fun(S)}$ represents the initialization of a random *transformation*, given to the adversary along with its "inverse" f^{-1}, and whose internal state is updated as the adversary asks its queries.

EasyCrypt

The formalization of the random transformation involves two finite maps that keep track of all the queries, one for the direct oracle Map.m and one for its inverse Map.mi.

Both maps are updated for each individual query, and in the case of the random transformation, no check is made to ensure the consistency of the output of previous queries.

```
module Map = {
  var m : (t, t) fmap
  var mi : (t, t) fmap
}.
module type Oracle = {
  proc init () : unit
  proc f (_: t) : t
  proc fi (_: t) : t
}.
```

```
module Count (O : Oracle) = {
  var counter : int
  proc init () : unit = {
    counter ← 0;
    O.init();
  }
  proc f (x : t) : t = {
    var y : t ← witness;
    if (counter < c) {
      y ← O.f(x);
    }
    counter ← counter + 1;
    return y;
  }
  proc fi (x : t) : t = {
    var y : t ← witness;
    if (counter < c) {
      y ← O.fi(x);
    }
    counter ← counter + 1;
    return y;
}}.
```

```
module RT : Oracle = {
  proc init () : unit = {
    Map.m ← empty;
    Map.mi ← empty;
  }
  proc f (x : t) : t = {
    var y : t;
    if (x \notin Map.m) { y <$- dt; }
    Map.m[x] ← y;
    Map.mi[y] ← x;
    return y;
  }
  proc fi (x : t) : t = {
    var y : t;
    if (x \notin Map.mi) { y <$- dt; }
    Map.m[y] ← x;
    Map.mi[x] ← y;
    return y;
}}.
```

Listing 2.4 – Random transformation

```
module RP : Oracle = {
  proc init () : unit = {
    Map.m ← empty;
    Map.mi ← empty;
  }
  proc f (x : t) : t = {
    var y : t;
    if (x \notin Map.m) { y <$- dt \ rng Map.m; }
    Map.m[x] ← y;
    Map.mi[y] ← x;
    return y;
  }
  proc fi (x : t) : t = {
    var y : t;
    if (x \notin Map.mi) { y <$- dt \ rng Map.mi; }
    Map.m[y] ← x;
    Map.mi[x] ← y;
    return y;
}}.
```

Listing 2.5 – Random permutation

The definition of the adversarial module type and the game are close, but they differ in the set of procedures the adversary can call from its input module. In this example, the adversary can call both O.f and O.fi.

```
module type Adversary (O : Oracle) = {
  proc guess () : bool { O.f O.fi }
}.
```

```
module Dist (A : Adversary) (O : Oracle) = {
  proc main () : bool = {
    var b;
    Count(O).init();
    b ← A(Count(O)).guess();
    return b;
}}.
```

Lemma 2.3.2 (Strong PRP–PRF switching)**.** *For any natural number c, any lossless adversary $\mathscr{A}$ can distinguish a random permutation from a random transformation with probability:*

$$\left|\Pr\left[\pi \xleftarrow{\$} \mathrm{Perm}(S); b \leftarrow \mathscr{A}^{\pi,\pi^{-1}} : b = 1\right] - \Pr\left[f \xleftarrow{\$} \mathrm{Fun}(S); b \leftarrow \mathscr{A}^{f,f^{-1}} : b = 1\right]\right| \leq \frac{c \cdot (c-1)}{2 \cdot |S|}$$

when its oracle starts to always answer a default value witness *after c queries.*

```
lemma strong_prp_prf_switching &m :
∀ (A <: Adversary { Map, Count }),
  `| Pr[Dist(A,RP).main() @ &m : res] − Pr[Dist(A,RT).main() @ &m : res] | ≤
   (c∗(c−1)/2) ∗ mu dt witness.
```

The proof of the stronger version of the PRP-PRF switching lemma is very similar and has the same structure as previously.

2.3.2 Eager Sampling [Almeida et al., 2019b]

Eager sampling is a standard argument when working with random oracles in cryptographic proofs, which consists in switching between two different views of a random oracle:

i. an *eager* view, in which the random oracle is sampled at random in a distribution over the space of functions (of the appropriate type) when the game is initialized; and

ii. a *lazy* view, in which the random oracle's responses are individually sampled upon fresh requests.

Although it is clear that no adversary that can only query the random oracle can distinguish between these two views, this restriction is often much too strict for the argument to be applicable directly.

Example

In Section 6.3.2, the step in the proof that states that padding and truncation preserve indifferentiability, boils down to proving that the oracles E and L shown on the outside of Figure 2.1 cannot be distinguished — when instantiated with a random oracle RF with domain $\mathrm{D} \times \mathbb{N}$ and range $\{0,1\}$ — by any algorithm (even unbounded) with oracle access to one of them. Note here that it is impossible to simply eagerly sample the random oracle RF, since its domain is countably infinite, preventing us from defining a uniform distribution over the function space.

$\mathrm{E}(x \in \mathrm{D}, n \in \mathbb{N})$

$z \leftarrow \varepsilon;$
for $(i = 0; i < n; i{+}{+})$
  $z \leftarrow z \,\|\, \mathrm{RF}.\mathtt{get}(x, i);$
for $(i = n; i < \lceil n/r \rceil \cdot r; i{+}{+})$
  $\mathrm{RF}.\mathtt{get}(x, i);$
return $z;$

$\mathrm{H}^{\mathrm{O}}(x \in \mathrm{D}, n \in \mathbb{N})$

$z \leftarrow \varepsilon;$
for $(i = 0; i < n; i{+}{+})$
  $z \leftarrow z \,\|\, \mathrm{O}.\mathtt{get}(x, i);$
for $(i = n; i < \lceil n/r \rceil \cdot r; i{+}{+})$
  $\mathrm{O}.\mathtt{sample}(x, i);$
return $z;$

$\mathrm{L}(x \in \mathrm{D}, n \in \mathbb{N})$

$z \leftarrow \varepsilon;$
for $(i = 0; i < n; i{+}{+})$
  $z \leftarrow z \,\|\, \mathrm{RF}.\mathtt{get}(x, i);$
return $z;$

Figure 2.1 – Eager-lazy random sampling example.

Rather than forcing the EasyCrypt user to use low-level tactics to reason about this equivalence, we define a new abstraction for random oracles that supports eager arguments, not only in situations like the one of Figure 2.1 (where definitional difficulties arise from the use of infinite domains), but also in the presence of programming queries, such as scenarios in which answers to some queries are deterministically `set` or `removed` by the context.

Our new abstraction models a random oracle with input in set D, output in C, and output distribution d_{C} as 4 oracles that share state, and whose canonical eager and lazy implementations are shown in Figure 2.2. They differ only in their `sample` oracle: an eager implementation (shown on a grey background), which samples values even though they are not returned; and a lazy implementation, which does nothing.

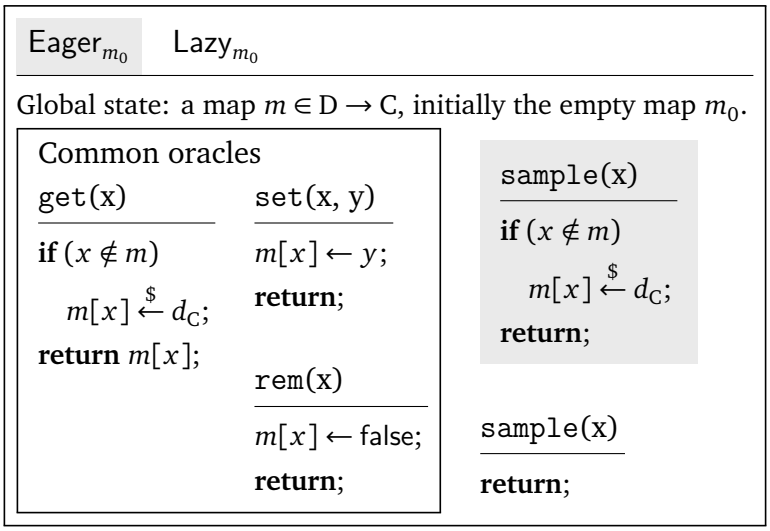


Figure 2.2 – The eager and lazy programmable random oracle

Even given this much extended interface (over the "traditional" interface which only exposes `get`), we can prove the following lemma, which states that the eager and lazy implementations are strictly equivalent.

Lemma 2.3.3 (Eager sampling for programmable random oracles)**.** *For any map $m_0 \in D \to C$, and for any unbounded adversary $\mathscr{D}$ with unbounded oracle access to `get`, `set`, `rem` and `sample` oracles as specified above, we have*

$$\mathscr{D}^{\mathsf{Eager}_{m_0}} \sim \mathscr{D}^{\mathsf{Lazy}_{m_0}}$$

Proving this lemma makes heavy use of EasyCrypt's advanced `eager` tactic which can be found online at https://www.easycrypt.info/documentation/refman.pdf. This tactic formalizes and confirms the standard intuition that, even when oracles can be externally programmed, sampling operations whose results do not influence the adversary's view can safely be delayed, either until the point where they do influence the adversary's view, or until the end of the game's execution.

Example

Continuing our example, the extended interface can be used as shown in Figure 2.1 to define a hybrid oracle H^O that uses O.`sample` as well as O.`get` for $O \in \{\mathsf{Eager}, \mathsf{Lazy}\}$. Then E and H^{Eager} are equivalent, since their second loops do nothing with the values they sample: both oracles sample $\lceil n/r \rceil \cdot r$ bits, returning the first n of them. Also H^{Lazy} is equivalent to L, since the second loop in H^{Lazy} does nothing. Finally, if $\mathscr{D}$ is a distinguisher for E/L, then by Lemma 2.3.3, we have that $\mathscr{D}^{E} \sim \mathscr{D}^{H^{\mathsf{Eager}}} \sim \mathscr{D}^{H^{\mathsf{Lazy}}} \sim \mathscr{D}^{L}$, considering the distinguisher $\mathscr{D}^{H^{\cdot}}$ when applying Lemma 2.3.3.

Remark

In the previous Section, the sampling $p \xleftarrow{\$} \mathsf{Perm}(S)$ represents the full-eager sampling whereas the EasyCrypt lemma involves the lazy representations of a random per-

mutation and a random function. From this section, those representations are equivalent and one can switch between the two as much as one's will.

2.3.3 Split Random Oracle

A random oracle is a useful tool that cryptographers manipulate in the random oracle model. Common random oracle manipulations involve splitting the domain of the random oracle separated by a predicate, or splitting the co-domain when the output type is isomorphic to a pair. We have developed this simple yet useful library for Chapter 8.

Split Domain

In the situation of all calls to a random oracle that satisfy a predicate test may be delayed and the rest may stay as is, the best way to delay such queries is to split the domain according to the predicate test and then use the previous library to delay the wanted ones.

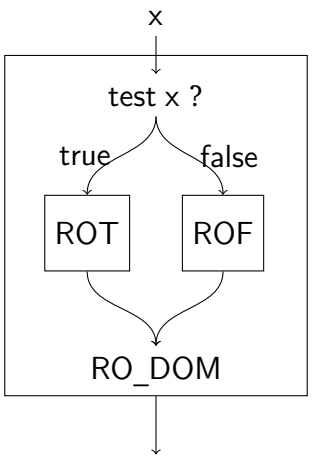


Figure 2.3 – Split domain

EasyCrypt

The abstract theory named SplitDom implements this idea. It uses the same interface (module types) as the lazy-eager theory named PROM (and described in Section 2.3.2). It provides two random oracles (named ROT and ROF) and the module RO_DOM.

```
abstract theory SplitDom.
type from, to, input, output.
op sampleto : from → to distr.
clone import PROM.Ideal as IdealAll with
  type from ← from,
  type to   ← to,
  type input ← input,
  type output ← output,
  op sampleto ← sampleto.
module RO = PROM.RO.
clone IdealAll.MkRO as ROT.
clone IdealAll.MkRO as ROF.
```

Represented in Figure 2.3, the module RO_DOM builds a full domain random oracle from two random oracles separated by predicate named test. Specifically it redirects queries: RO_DOM[ROT, ROF].get(x) will either output ROT.get(x) if x satisfies test, or output ROF.get(x) otherwise.

```
op test : from → bool.
module RO_DOM(ROT:RO, ROF:RO): RO = {
  proc init () = { ROT.init(); ROF.init(); }
  proc get(x : from) = {
    var r;
    if (test x) r ← ROT.get(x);
    else r ← ROF.get(x);
    return r;
  }
  proc set(x : from, y : to) = {
    if (test x) ROT.set(x, y);
    else ROF.set(x, y);
  }
  proc rem(x : from) = {
    if (test x) ROT.rem(x);
    else ROF.rem(x);
  }
  proc sample(x : from) = {
    if (test x) ROT.sample(x);
    else ROF.sample(x);
  }
}.
```

The more general lemma states that any game using RO_DOM[ROT, ROF] is equivalent to the same game using a direct random oracle RO. The following lemma is a pRHL equivalence of procedures:

```
section PROOFS.
  declare module D: IdealAll.RO_Distinguisher { RO, ROT.RO, ROF.RO }.
  lemma RO_split:
    equiv [ IdealAll.MainD(D,RO).distinguish
          ~ IdealAll.MainD(D,RO_DOM(ROT.RO,ROF.RO)).distinguish
          : ={glob D, x}
          ⇒ ={res, glob D} ∧ RO.m{1} = union_map ROT.RO.m{2} ROF.RO.m{2} ∧
            (∀ x, x ∈ ROT.RO.m{2} ⇒ test x) ∧
            (∀ x, x ∈ ROF.RO.m{2} ⇒ ¬test x) ].
  proof. (* ··· *) qed.
```

The corrolary pr_RO_split is in the form of equality of probabilities, where some of the information about the relations between the finite maps is lost.

```
  lemma pr_RO_split (p: glob D → output → bool) &m x0:
    Pr[ MainD(D,RO).distinguish(x0) @ &m : p (glob D) res] =
    Pr[ MainD(D,RO_DOM(ROT.RO,ROF.RO)).distinguish(x0) @ &m : p (glob D) res].
  proof. by byequiv RO_split. qed.
end section PROOFS.
end SplitDom.
```

Split co-domain

When the output type to of a random oracle is isomorphic to a pair of two different types, to1 and to2, one may want to split the random oracle into the pairing of two uncorrelated random oracles with output type to1 and to2, respectively. This co-domain split allows cryptographers to delay (or erase) one of them, making the proof of an upper bound on some event easier, as in Chapter 8.

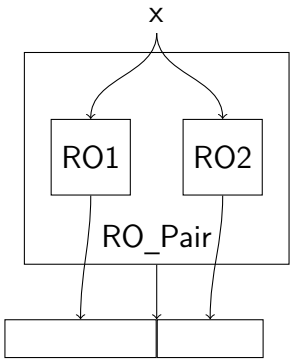


Figure 2.4 – Split co-domain

EasyCrypt

The abstract theory SplitCodom implements this idea. It uses the same interface for module types as the theory PROM and provides two sub theories cloned from PROM, named I1 and I2.

```
abstract theory SplitCodom.
type to1, to2.
op topair : to → to1 * to2.
op ofpair : to1 * to2 → to.
axiom topairK: cancel topair ofpair.
axiom ofpairK: cancel ofpair topair.
op sampleto1 : from → to1 distr.
op sampleto2 : from → to2 distr.
axiom sample_spec (f:from) :
  sampleto f =
  dmap (sampleto1 f `*` sampleto2 f) ofpair.
```

```
clone PROM.Ideal as I1 with
  type from ← from,
  type to ← to1,
  type input ← input,
  type output ← output,
  op sampleto ← sampleto1.
```

```
clone PROM.Ideal as I2 with
  type from ← from,
  type to ← to2,
  type input ← input,
  type output ← output,
  op sampleto ← sampleto2.
```

```
module RO_Pair(RO1:I1.RO, RO2:I2.RO): RO = {
  proc init () = { RO1.init(); RO2.init(); }
  proc get(x : from) = {
    var r1, r2;
    r1 ← RO1.get(x);
    r2 ← RO2.get(x);
    return ofpair(r1,r2);
  }
  proc set(x : from, y : to) = {
    var y1, y2;
    (y1,y2) ← to pair y;
    RO1.set(x,y1); RO2.set(x,y2);
  }
  proc rem(x : from) = {
    RO1.rem(x); RO2.rem(x);
  }
  proc sample(x : from) = {
    RO1.sample(x); RO2.sample(x);
}}.
```

The more general lemma states that any game using RO_pair[I1.RO, I2.RO] is equivalent to the same game using a direct random oracle RO. The following lemma is a pRHL equivalence of procedures:

```
section PROOFS.
  declare module D : RO_Distinguisher { RO, I1.RO, I2.RO }.
  lemma RO_split:
    equiv [ MainD(D,RO).distinguish
          ~ MainD(D,RO_Pair(I1.RO,I2.RO)).distinguish
          : ={glob D, x}
          ⇒ ={res, glob D} ∧
            RO.m{1} = map (fun _ ⇒ ofpair) (pair_map I1.RO.m{2} I2.RO.m{2}) ∧
            ∀ x, x ∈ RO.m{1} = x ∈ I1.RO.m{2} ∧ x ∈ RO.m{1} = x ∈ I2.RO.m{2}.
  proof. (* ··· *) qed.
```

The corrolary pr_RO_split is in the form of equality of probabilities, where some of the information about the relations between the finite maps is lost.

```
  lemma pr_RO_split (p: glob D → output → bool) &m x0:
    Pr[ MainD(D,RO).distinguish(x0) @ &m : p (glob D) res] =
    Pr[ MainD(D,RO_Pair(I1.RO,I2.RO)).distinguish(x0) @ &m : p (glob D) res].
  proof. by byequiv RO_split. qed.
end section PROOFS.
end SplitCodom.
```

Part II

Message Authentication

Chapter 3

Security of a MAC Scheme

Passive attacks such as eavesdropping, can be prevented using confidentiality preserving techniques. A different requirement is to protect against active attack, such as falsification of data and transactions. Protection against such attacks is known as *message authentication*.

Example

If you're using Windows, all the Windows operating files on disk are the same for each Windows operating system, and are frequently used. Even if they are public and known to the world, it is rather important to ensure that they are not modified by a malware or some other unauthorized entity, without having to decrypt them anytime you access a different fonctionality of your operating system.

In the case of an ads' banner, the ads' provider does not care about the ads being copied and shown to other people (in fact, it may even benefit them). There is no confidentiality issue at all. What they do care about is modifying the ads in an unauthorized way. An example would be to change the name and the logo of the ads' provider.

Remark

In both cases, the usability of the functionality should not be more difficut than accessing the plaintext. This issue corresponds to the balance between security and usability of systems. The importance of such a balance according to the risks in the system's environment cannot be emphasized enough, even if I do not discuss further this issue in this manuscript.

Message authentication

Message authentication is a procedure that allows communicating parties to verify that the source is *authentic* — the actual source can be verified to be the expected one — and *message integrity* — the message content has not been altered.

The overall intent of any message integrity technique is the same: upon retrieval, ensure the message is exactly the same as it was originally transmitted over a possibly malicious network. In other words, message integrity aims to prevent unintentional and unauthorized changes to transmitted information while the message itself is not encrypted and can be read by anyone receiving the message or listening to the transmission.

One authentication technique involves the use of a secret key to generate a small block of data, known as *message authentication code* (MAC) or a *tag*, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key k_{AB}. When A has a message m to send to B, whatever the message length is, it computes the message authentication code as a complex function of the message and the key: $tag_m = MAC_{k_{AB}}(m)$. The message plus tag are transmitted to the intended recipient. The recipient performs the same computation on the received message, using the same secret key, to generate a new tag. The received tag is compared to the computed tag. If they are equal, the message is authenticated, otherwise the couple message–tag can be assumed to be falsified.

3.1 Security Definition : Forgery Resistance

Assuming the secret key is only possessed by intended parties, if any entity that does not know the secret key is able to produce a valid tag for a message, the authentication procedure described before does not provide authentication nor message integrity at all. Therefore, the security of such a MAC algorithm resides in the *difficulty* to *forge* a valid tag to a message without knowing the secret key, versus the *efficiency* to compute a valid tag for any message using the secret key. A forgery is the production of (m, t) where t is a valid tag for m when the entity providing it does not own the secret key.

A stronger definition of a *forgery* is to allow the entity, called *adversary*, to query the MAC algorithm so that it can obtain valid tags for any message of the adversary's choosing before the attempt of the forgery. The adversary's forgery attempt is successful if the tag is valid for the message, and the message was not previously queried. In other words, even if the adversary can obtain valid tags for any message, it should still be very difficult to produce a valid tag that does not come from the parties knowing the secret key.

Definition 3.1.1. The *forgery advantage* $\mathsf{Adv}^{\mathsf{forge}}_{M}(\mathscr{A})$ of an adversary $\mathscr{A}$ against a MAC scheme M is the probability of $\mathscr{A}$ succeeding in producing a valid forgery :

$$\mathsf{Adv}^{\mathsf{forge}}_{M}(\mathscr{A}) := \Pr\left[k \xleftarrow{\$} \mathbb{K}; (m,t) \leftarrow \mathscr{A}^{M_k(\cdot)}; t^* \leftarrow M_k(m) : t = t^* \wedge m \notin \mathscr{Q}_{\mathscr{A}}^{M_k(\cdot)}\right]$$

where $\mathscr{Q}_{\mathscr{A}}^{M_k(\cdot)}$ is the set of all queries $\mathscr{A}$ made to its oracle during its computation.

Forgery Resistance of a Random Function

EasyCrypt

The forgery game is formalized in EasyCrypt as a *module* which takes two modules as parameters, the advsersary A and the MAC scheme M.

The Trace module is the formalization of the restriction given to the adversary that depends on the cryptographic game. In the forgery game, the adversary cannot win if the message in the forge attempt has already been queried to its oracle. The module Trace logs all the queries asked by the adversary in its *global variable* Trace.queries and the game tests at the end if the forgery attempt was not previously asked. The init procedure of Trace sets its variable to be empty at the beginning of the forgery game. It then calls the init procedure of the MAC algorithm M. Very often, it stands for the sampling of the MAC key that is stored in a global variable of the module M. This key

is then used for all queries to the procedure mac.

```
type message, mac.
```

```
module type MAC = {
  proc init () : unit
  proc mac (m : message) : mac
}.
```

```
module type Adversary (O : MAC) = {
  proc forge () : message * mac { O.mac }
}.
```

```
module Trace (M : MAC) : MAC = {
  var queries : (message, mac) fmap
  proc init () : unit = {
    queries ← empty;
    M.init();
  }
  proc mac (m : message) : mac = {
    if (¬ m ∈ Trace.queries) {
      Trace.queries[m] ← M.mac(m);
    }
    return oget Trace.queries[m];
}}.
```

```
module Forge (A : Adversary, M : MAC) = {
  proc game () : bool = {
    var m, t, t';
    Trace(M).init();
    (m,t) ← A(Trace(M)).forge();
    t' ← M.mac(m);
    return t = t' ∧ ¬ m ∈ Trace.queries;
}}.
```

In EasyCrypt, to state the security of a MAC scheme, the forgery advantage is proven to be bounded *for all adversaries* by an *explicit* bound. The adversary module can appear in the bound, and, in this case, it exhibits the security assumption on which the MAC algorithm is secure. This is the general approach to provide a proof of any provable secure scheme in EasyCrypt.

In this setting, the adversary with access to either Trace.queries or the secret MAC key M.k has a trivial forgery attack. Therefore, in the quantification for all adversaries, the adversaries that have access to those global variables should not be considered. A lemma about forgery resistance will appear in the proof script as :

```
module M : MAC = {
  var k : key
  proc init () = { (* ··· *) }
  proc mac (m : message) = {
    (* code of the MAC scheme *)
}}.
section Proof.
  declare module A : Adversary{M, Trace}.
  (* intermediate lemmas *)
  lemma forgery : ∀ &m,
    Pr [ Forge(A, M).game() @ &m : res ]
    ≤ (* explicit bound *).
  proof.
  (* formal proof script *)
  qed.
end section Proof.
```

3.2 Indistinguishability from a Random Function implies Forgery Resistance

In this section, I state the forgery advantage of such a random oracle in order to relate it to the forgery advantage of any MAC scheme and the advantage to distinguish this MAC scheme from a random oracle.

3.2.1 Forgery Resistance of a Random Function

EasyCrypt

A random oracle (or random function) named RF is modelized as follows.

```
type message, mac.
op dmac : mac distr.
axiom dmac_funi : is_funiform dmac.
axiom dmac_ll : is_lossless dmac.
```

The type mac models the output set F of the MAC algorithm.

```
module RF : MAC = {
  var map : (message, mac) fmap
  proc init () : unit = { map ← empty; }
  proc mac (x : message) : mac = {
    if (¬ x ∈ map) { map[x] ←$ dmac; }
  return oget map[x];
}}.
```

Theorem 3.2.1 (Forgery Advantage against a Random Function)**.** *For all adversary* A *that have no access to* Trace *nor* RF*, the forgery advantage of* A *against* RF *is bounded by the inverse of the size of the output space* F*.*

$$\mathsf{Adv}^{\mathrm{forge}}_{\mathsf{RF}}(\mathsf{A}) \leq \frac{1}{|\mathrm{F}|} = \mathsf{mu1\ dmac\ witness}$$

Proof. The forgery advantage of A against RF is:

— equal to 0 if m ∈ Trace.queries, since Trace.queries = RF.map is an invariant;

— equal to $\frac{1}{|\mathrm{F}|}$ otherwise, since we assume that dmac is the uniform distribution on F.

□

EasyCrypt

```
lemma random_function_is_forgery_resistant :
  ∀ &m (A <: Adversary{Trace, RF}),
  Pr [ Forge(A,RF).game() @ &m : res ] ≤ mu1 dmac witness.
```

Remark

Consider the adversary that samples uniformly at random its forgery attempt with no query. This adversary has a forgery advantage of $\frac{1}{|\mathrm{F}|}$ against every MAC scheme that has the output space F. Therefore, when bounding the forgery advantage for all adversaries, it can never be less than $\frac{1}{|\mathrm{F}|}$. As a consequence, a random function acheives the *optimal bound* on the forgery advantage when quantified *for all* adversaries.

3.2.2 Indistinguishability

In the real world, random function does not exist. Since they achieve the best security in terms of forgery resistance, one could want to prove that even if a MAC scheme is not a random function, its behaviour is as close to a random function as possible so that the forgery advantage of any adversary is close to the forgery advantage of a random function.

In cryptography, two distributions are *computationally*[1] *indistinguishable* if no efficient adversary can tell the difference between them except with small probability. This probability is called the *distinguishing advantage* of an adversary. It is called *prf-distinguishing advantage*, for pseudo-random function, when one of them is a random function.

Definition 3.2.1 (Distinsguishing Advantage)**.** Let D and E be two distributions of functions, for any adversary $\mathscr{A}$ that has access to one oracle sampled from one of the distributions and outputs one bit guessing which distribution it was given. The *distinguishing advantage* of $\mathscr{A}$ about D and E is defined as:

$$\mathsf{Adv}^{\mathsf{dist}}_{\mathsf{D},\mathsf{E}}(\mathscr{A}) := \left|\Pr\left[d \xleftarrow{\$} \mathsf{D}; b \leftarrow \mathscr{A}^{d(\cdot)} : b = 1\right] - \Pr\left[e \xleftarrow{\$} \mathsf{E}; b \leftarrow \mathscr{A}^{e(\cdot)} : b = 1\right]\right|$$

EasyCrypt

```
type t_in, t_out.
module type Function = {
  proc init () : unit
  proc f (x : t_in) : t_out
}.
module type Distinguisher (F : Function) = {
  proc guess () : bool { F.f }
}.
module Dist (D : Distinguisher) (F : Function) = {
  proc game () : bool = {
    var b : bool;
    F.init();
    b ← D(F).guess();
    return b;
  }
}.
module F : Function = (* some function *).
module G : Function = (* some function *).
lemma indistinguishable :
  ∀ &m (D <: Distinguisher{F, G}),
  `| Pr [ Dist(D,F).game() @ &m : res ] − Pr [Dist(D,G).game() @ &m : res] |
    ≤ (* some concrete bound *).
proof. (* some proof script *) qed.
```

Forgery Resistance from Indistinguishability with a Random Function

Theorem 3.2.2. *Let the program $\mathscr{B}(\cdot)$ build a distinguishing adversary given a forgery adversary $\mathscr{A}$ that aims to distinguish a MAC scheme* M *from a random function pictured as \$, i.e.* $\mathscr{B}(\mathscr{A})^{\mathsf{O}(\cdot)} := \mathsf{Forge}[\mathscr{A}, \mathsf{O}].\mathsf{main}$. *For any adversary $\mathscr{A}$, its forgery advantage against* M

1. The specification "computationally" refers to the fact that only *efficient* adversaries are taken into account in this definition of indistinguishability. In EasyCrypt, there is no analysis of complexity in time but analysis of probabilistic terminaison. Therefore all adversaries that are considered in the proofs are not bounded in time. Instead they are often assumed to be *lossless*. This is the assumption that adversaries will terminate with probability 1. This is a stronger result on indistinguishability than computational indistinguishability.

is bounded by the advantage of the distinguishing adversary $\mathscr{B}(\mathscr{A})$ to distinguish M *from a random function plus the inverse of the size of the tag space* $\mathbb{B}$.

$$\mathsf{Adv}^{\text{forge}}_{\mathsf{M}}(\mathscr{A}) \leq \mathsf{Adv}^{\text{dist}}_{\mathsf{M},\$}(\mathscr{B}(\mathscr{A})) + \frac{1}{|\mathbb{B}|}$$

Proof. By unfolding the definition of $\mathscr{B}$ and the distinguishing advantage:

$$\mathsf{Adv}^{\text{dist}}_{\mathsf{M},\mathsf{LazyRF}}(\mathscr{B}(\mathscr{A})) = \left|\mathsf{Adv}^{\text{forge}}_{\mathsf{M}}(\mathscr{A}) - \mathsf{Adv}^{\text{forge}}_{\mathsf{LazyRF}}(\mathscr{A})\right| \geq \mathsf{Adv}^{\text{forge}}_{\mathsf{M}}(\mathscr{A}) - \frac{1}{|\mathbb{B}|}$$

□

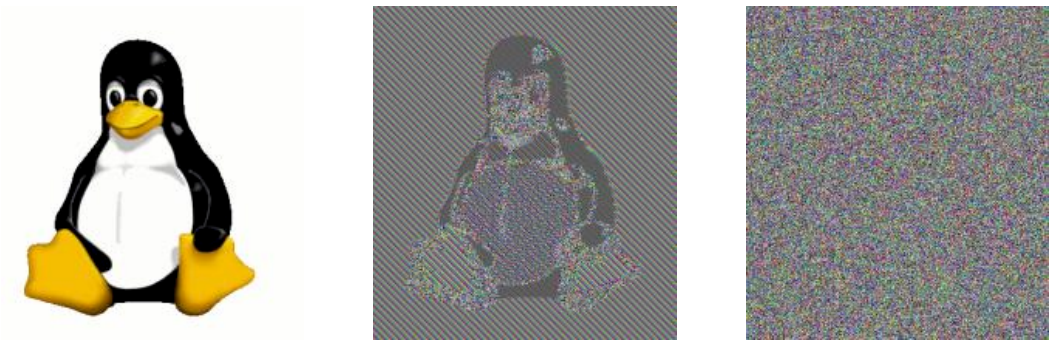
3.3 Block Cipher Mode of Operation

A *block cipher* is a deterministic algorithm operating on fixed-length group of bits, called *block*, with an unvarying transformation (encryption or decryption) that is specified by a *symmetric key*. The most used block cipher is the Advanced Encryption Standard (AES [Daemen and Rijmen, 1999]), even if older block ciphers are still used: Data Encryption Standard (DES [Morris et al., 1977]), or triple DES.

A block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one block. A *block cipher mode of operation* is an algorithm that uses a block cipher to provide information security such as confidentiality or authenticity. However, the security of a mode of operation does not entirely depends on the security of the underlying block cipher. When a block cipher is used in a given mode of operation, the resulting algorithm should ideally be about as secure as the underlying block cipher itself.

Example

Consider the algorithm that encrypts an arbitrary long message by encrypting each block separately, with the same symmetric key. It is named ECB, for Electronic Codebook. Even if the block cipher correctly hides information of each block, the pattern of a longer message may be revealed as shown in Figure 3.1.



Plaintext. Encrypted using ECB. Securely encrypted.

Figure 3.1 – ECB reveals patterns, even when using a secure block cipher.

3.3.1 Security Definition of a Block Cipher

A block cipher consists of two paired algorithms, one for encryption denoted E, and one for decryption denoted D. Both algorithms expect two inputs: a block in $\mathbb{B}$ and a private key $\mathsf{k} \in \mathbb{K}$; and both algorithms output a block in $\mathbb{B}$. The decryption is defined to be the inverse function of encryption, when they both use the same private key.

More formally, a block cipher is specified by an encryption function:

$$\mathrm{E}_{\mathsf{k}}(p) := \mathrm{E}(\mathsf{k}, p) : \mathbb{K} \times \mathbb{B} \to \mathbb{B}$$

which takes as input a key $\mathsf{k} \in \mathbb{K}$, and the *plaintext*, a block $p \in \mathbb{B}$ and returns the *ciphertext*, a string $c \in \mathbb{B}$. For each key $\mathsf{k} \in \mathbb{K}$, the function $\mathrm{E}_{\mathsf{k}}(\cdot)$ is required to be an invertible mapping on $\mathbb{B}$, where the inverse D_{k} is defined as:

$$\mathrm{D}_{\mathsf{k}}(c) := \mathrm{D}(\mathsf{k}, c) : \mathbb{K} \times \mathbb{B} \to \mathbb{B}$$

We denote $\mathsf{Perm}(\mathbb{B})$ the family of all permutations on $\mathbb{B}$ that is equipped with the uniform distribution. The *prp-distinguishing advantage* (for pseudo-random permutation) against a block cipher is the advantage to distinguish it, when the key is randomly sampled, from a random permutation sampled from $\mathsf{Perm}(\mathbb{B})$.

Definition 3.3.1 (prp-distinguishing advantage)**.** The prp-distinguishing advantage of any adversary $\mathscr{A}$ to tell apart a block cipher (E, D) equipped with a random key from a random permutation over the same block space $\mathbb{B}$ is defined as follows:

$$\mathsf{Adv}^{\mathrm{prp}}_{(\mathrm{E},\mathrm{D})}(\mathscr{A}) := \left|\Pr\left[\mathsf{k} \xleftarrow{\$} \mathbb{K}; \mathsf{b} \leftarrow \mathscr{A}^{\mathrm{E}_k(\cdot),\mathrm{D}_k(\cdot)} : b = 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}); \mathsf{b} \leftarrow \mathscr{A}^{\pi(\cdot),\pi^{-1}(\cdot)} : b = 1\right]\right|$$

3.3.2 Security Definition of a Block Cipher Mode of Operation

Some MAC algorithms are constructed using a *mode of operation*. A mode of operation describes how to repeatedly apply a block cipher operation to securely transform amounts of data larger than a block. The security of such a mode often relies on the security assumption that the underlying block cipher is secure, so that in the analysis of the structure of the mode, one can replace calls to a block cipher with different keys by independent random permutations.

EasyCrypt

In EasyCrypt, such a formalization looks like the following.

```
type key.
type block.
op dblock : block distr.
axiom dblock_funi : is_funiform dblock.
axiom dblock_ll : is_lossless dblock.
```

```
module type Permutation = {
  proc init () : unit
  proc enc (p : block) : block
  proc dec (p : block) : block
}.
```

```
module type Distinguisher
    (O : Permutation) = {
  proc distinguish () : bool { O.enc O.dec }
}.
```

```
module Dist (D : Distinguisher)
    (P : Permutation) = {
  proc game () : bool = {
    var b : bool;
    P.init();
    b ← D(P).distinguish();
    return b;
}}.
```

```
module type OperationMode
    (P : Permutation) = MAC.
```

```
module RP : Permutation = {
  var map  : (block, block) fmap
  var mapi : (block, block) fmap
  proc init () : unit = {
    map  ← empty;
    mapi ← empty;
  }
  proc enc (x : block) : block = {
    if (¬ x ∈ map) {
      map[x] ←$ dblock \ (rng map);
      mapi[oget map[x]] ← x;
    }
    return oget map[x];
  }
  proc dec (x : block) : block = {
    if (¬ x ∈ mapi) {
      mapi[x] ←$ dblock \ (rng mapi);
      map[oget mapi[x]] ← x;
    }
    return oget mapi[x];
}}.
```

```
lemma mode_of_operation_secure :
  ∀ &m
  (OM <: OperationMode{Trace, RP})
  (P <: Permutation{Trace, RP, OM})
  (A <: Adversary{Trace, RP, OM, P}),
  Pr [ Forge(A,OM(P)).game() @ &m : res ]
  ≤ Pr [ Forge(A,OM(RP)).game() @ &m : res] (* concrete upper bound *)
    + `|Pr [ Dist(B(A),P).game() @ &m : res] − Pr [ Dist(B(A),RP).game() @ &m : res ]|.
      (* security assumption *)
```

Chapter 4

CMAC's Formal Security Proof [Baritel-Ruet et al., 2018]

The Cipher-based Message Authentication Code, shortened as CMAC [Song et al., 2006], is a block cipher-based message authentication code algorithm. The structure of CMAC's algorithm is sequential and its computation does not require to know the length of the message to authenticate.

4.1 Historical presentation

The CMAC scheme was first introduced by Iwata and Kurosawa as OMAC1, for One-key MAC, in [Iwata and Kurosawa, 2003]. It is now a standardized MAC [Dworkin, 2016]. The scheme is based on the CBC (*chained block cipher*) mode of operation [Ehrsam et al., 1978], and more precisely on the MAC scheme derived from it. We denote CBC-MAC$_{E_k}$ to represent the CBC-MAC scheme using the underlying block cipher E_k.

In CBC-MAC, the message to authenticate is split into a list of blocks. The first block is encrypted by E_k, and each next block is first xored using the last computed cipher then encrypted by E_k. This constructs a chain linked by the output of the block cipher. This dependence ensures that a change to any bit of a message will change the rest of the MAC in an unpredictable way.

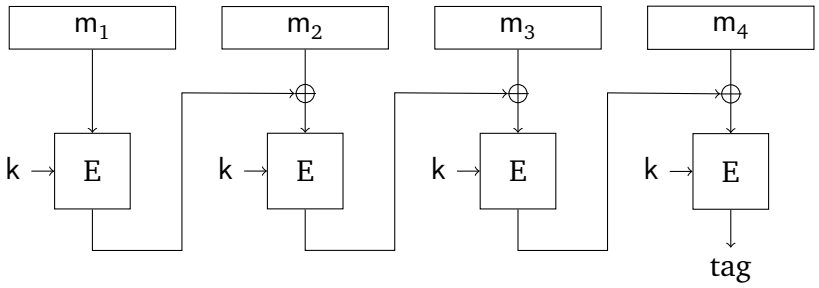


Figure 4.1 – Representation of CBC-MAC$_{E_k}(m_1||m_2||m_3||m_4)$

EasyCrypt

```
module type MACN = {
  proc init () : unit
  proc mac (m : block list) : block
}.
module CBCMAC (F : Permutation) : MACN = {
  proc init = F.init
  proc mac (m : block list) : block = {
    var i : int ← 0;
    var c : block ← zeros;
    while (i < size m) {
      c ← F.enc(m[i] ^+ c); (* ^+ denotes the xor operation *)
      i ← i + 1;
    }
    return c;
}}.
```

Bellare, Kilian and Rogaway [Bellare et al., 1994] prove the security of the CBC-MAC construction, by proving, as is the practice in cryptography, that any algorithm that finds CBC-MAC forgeries with non-negligible probability can be used to break the underlying block cipher with the same time complexity. However, their proof — and indeed the security of CBC-MAC— is limited to the case where the length m of the message (in blocks) is fixed in advance. Indeed, it is easy to produce a chosen message forgery as follows when the tagging and verification algorithm accepts arbitrary length messages:

1. Through chosen-message queries, obtain $t = CBC(m)$ and $t' = CBC(m')$ for some messages m and m' of respective lengths $m > 0$ and $m' > 0$.
2. Then t' is also a valid tag for the *fresh* message: $m_1 || \ldots || m_m || [t \oplus m'_1] || \ldots || m'_{m'}$.

Thus CBC-MAC suffers from two major issues: i. the size of messages should be a multiple of n, the block size, and ii. all messages must contain the same, fixed, number of blocks. To overcome these restrictions, Black and Rogaway [Black and Rogaway, 2005] propose three extensions of CBC-MAC: [1]i. ECBC, ii. FCBC and iii. XCBC. CMAC is a variant of XCBC that was first proposed by Iwata and Kurosawa [Iwata and Kurosawa, 2003] as OMAC1.

Description of ECBC

The first issue of CBC-MAC, that is being limited to computing tags for messages whose length is a multiple of the block length, is often dealt with using some injective padding scheme. However, such schemes require flexibility in the number of blocks that can be processed, as their injectivity may require them to add a full block of padding onto a message. The MAC scheme ECBC combines two ideas to: i. allow padding messages to a multiple of the block length without overhead; and ii. securely support computing MAC tags for messages of different lengths.

1. A simpler fix is to be prepend the length of the input message to the message itself before processing it: if this does indeed yield a secure MAC algorithm, it is not always possible to know the length of the payload when the processing begins. For example, the TLS protocol computes a MAC over the concatenation of all messages exchanged during its handshake protocol–the length of which is only known after it is over. These engineering considerations are often kept separate from security considerations when defining the syntax and security of cryptographic schemes. This provides abstraction but brings its own set of problems.

First, in order to support messages with different numbers of blocks, it is possible to apply additional treatment to the final tag before releasing it. This is to ensure that it cannot predictably be used as a known intermediate value in the computation of a tag for a chosen message. In ECBC, this is done, using an idea by Vaudenay [Vaudenay, 2000], which was first proposed in the book [Bosselaers and Preneel, 1995], by computing a CBC-MAC tag using a block cipher key k_1 and encrypting this tag with a different key k_2 before releasing it.

This first idea then brings an opportunity to do padding only when needed. Indeed, rather than ensuring the padding is injective, it is possible to apply padding to the message only when its length is not a multiple of n, but then compute the final encryption using a different key k_3 when padding was applied.

Given some injective padding function $\text{pad} : \Sigma^* \to \mathbb{B}^+$ that pads arbitrary strings over some alphabet Σ as non-empty sequence of blocks, and combining the two ideas above, the ECBC scheme (also shown in Figure 4.2) can be defined as the following three-key construction.

$$\text{ECBC}_{E_{k_1},E_{k_2},E_{k_3}}(m) = \begin{cases} E_{k_2}(\text{CBC}_{k_1}(m)) & \text{if } m \in \mathbb{B}^+ \\ E_{k_3}(\text{CBC}_{k_1}(\text{pad}(m)) & \text{otherwise} \end{cases}$$

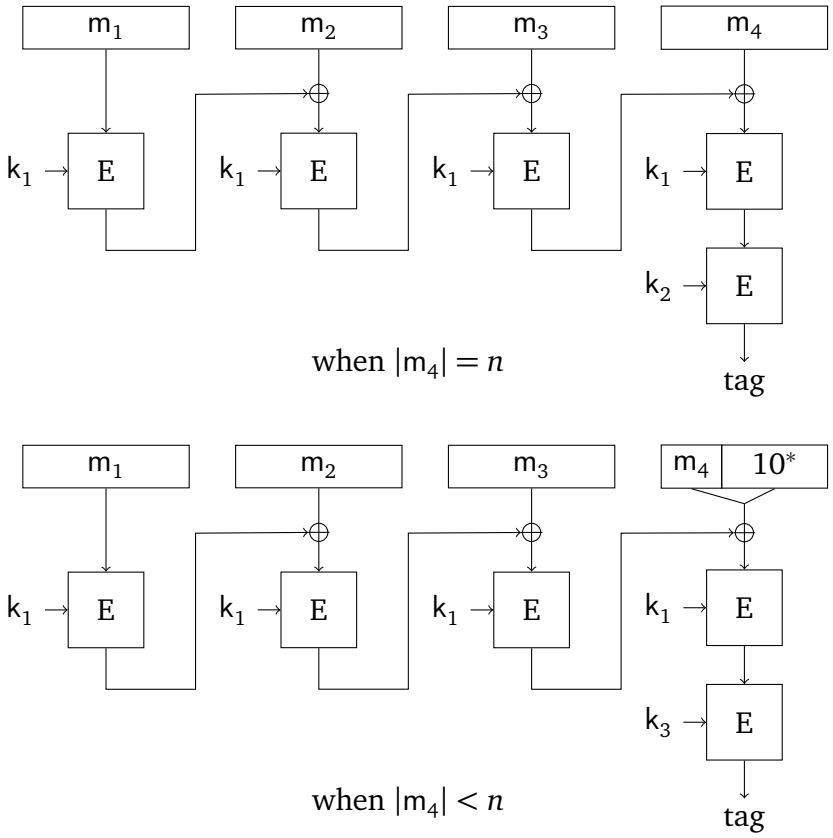


Figure 4.2 – Representation of $\text{ECBC}_{E_{k_1},E_{k_2},E_{k_3}}(m_1||m_2||m_3||m_4)$

EasyCrypt

```
op pad (s:bitstring) : bitstring =
  if `|s| mod n = 0 then s else s || ones 1 || zeros (n - 1 - (`|s| mod n)).
op towords (s:bitstring) : block list = let indexes = iota_ 0 (`|pad s| %/ n) in
  let l = map (fun (i:int) => from_bits (sub (pad s) (i*n) n)) indexes in
  mkarray l.
op ofwords (t : block list) : bitstring = let l = ofarray t in
  foldr (fun a b => to_bits a || b) (ABitstring.zeros 0) l.
module type MAC = {
  proc init () : unit
  proc mac (_: bistring) : block
}.
module EPAD (H : MACN) (F2 F3 : BlockCipher) : MAC = {
  proc init() : unit = {
    H.init(); F2.init(); F3.init();
  }
  proc mac (m : bitstring) : block = {
    var t : block;
    if ((size m) mod n = 0) {
      t ← H.mac(towords m);
      t ← F2.f(t);
    } else {
      t ← H.mac(towords m);
      t ← F3.f(t);
    }
    return t;
}}.
module ECBC (F1 F2 F3 : BlockCipher) : MAC = EPAD(CBCMAC(F1),F2,F3).
```

Description of FCBC

In ECBC, as shown in Figure 4.2, processing the last block of message involves two consecutive calls to the block cipher with independent keys. This brings an opportunity for optimization. Indeed, since block ciphers are meant to be indistinguishable from random permutations, a single block cipher invocation is in fact sufficient for security. The FCBC construction, shown in Figure 4.3, implements this idea.

EasyCrypt

```
module PartialCBCMAC (F : BlockCipher) : MACN = {
  proc init = F.init
  proc mac (m : word list) : word = {
    var i : int ← 0;
    var c : word ← zeros;
    while (i < size m - 1) {
      c ← F.f(m[i] ^+ c); (* ^+ denotes the xor operation *)
      i ← i + 1;
    }
    return c +^ m[i];
}}.
```

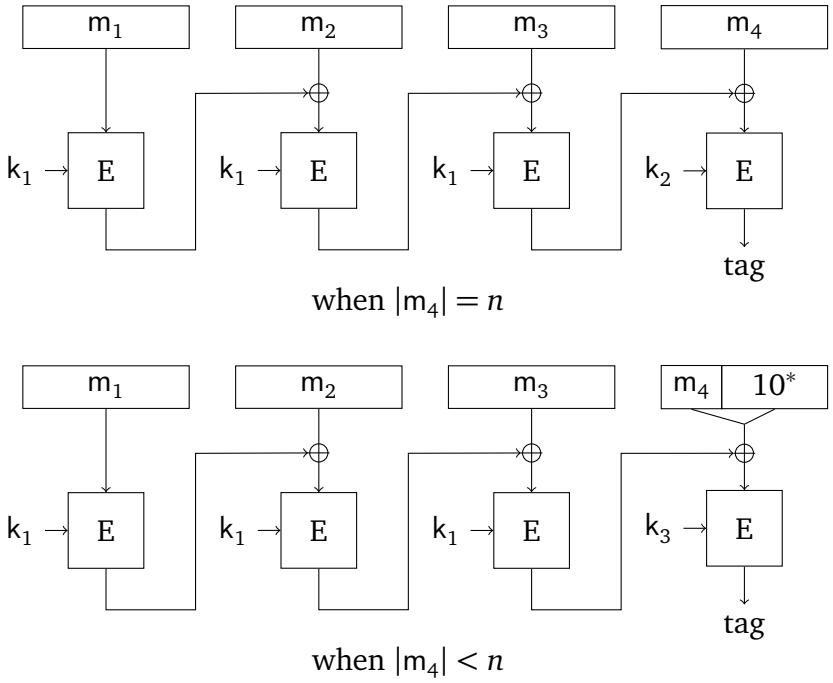


Figure 4.3 – Representation of $\mathrm{FCBC}_{E_{k_1},E_{k_2},E_{k_3}}(m_1||m_2||m_3||m_4)$.

```
module FCBC (F1 F2 F3 : BlockCipher) : MAC = EPAD(PartialCBCMAC(F1),F2,F3).
```

Description of XCBC

Both of the above constructions do solve CBC-MAC's main issues, with FCBC giving a small performance advantage. However, their use of block ciphers keyed with independent keys makes them expensive to compute using block ciphers like the AES [Daemen and Rijmen, 2013]. Indeed, AES involves a costly key expansion phase that can only be amortized if the same key is used many times. To better suit this engineering constraint, the XCBC construction (Figure 4.4) uses the same key k for all block cipher invocations, and uses the other two keys (k_2, k_3) to mask the final block before processing it. This is done without loss of security.

EasyCrypt

```
module XOR2 (E : BlockCipher) = {
  proc init(q : int) : unit = {}
  proc f (x : word) : word = {
    var y : word;
    y ← E.f(x ^+ XOR.k2);
    return y;
}}.
module XOR3 (E : BlockCipher) = {
  proc init(q : int) : unit = {}
  proc f (x : word) : word = {
    var y : word;
    y ← E.f(x ^+ XOR.k3);
    return y;
```

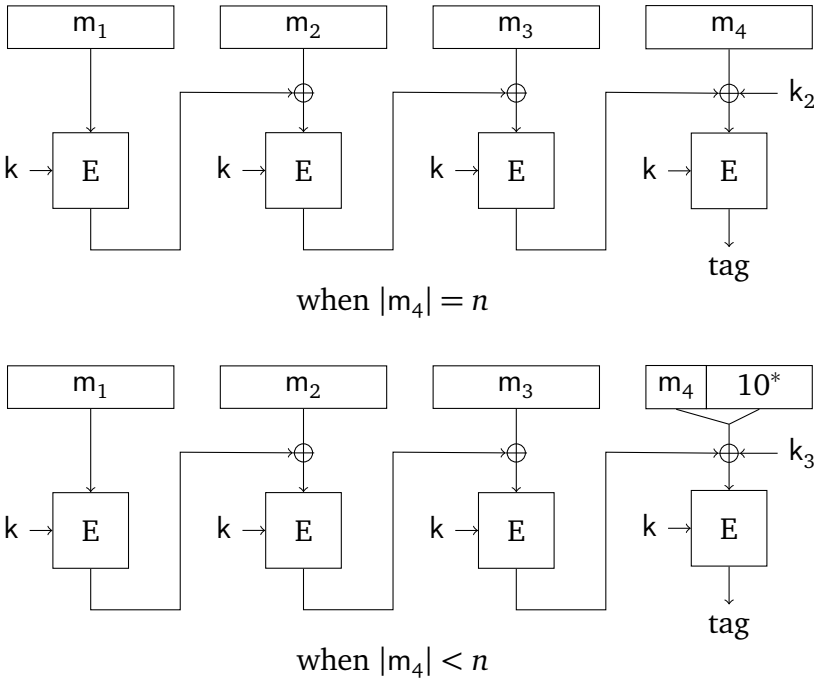


Figure 4.4 – Representation of $\mathrm{XCBC}_{E_k,k_2,k_3}(m_1||m_2||m_3||m_4)$

```
}}.
module FCBC (F1 F2 F3 : BlockCipher) : MAC =
  EPAD(PartialCBCMAC(F1),XOR2(F1),XOR3(F3)).
```

Description of CMAC

One final performance constraint needs dealt with: the key size for XCBC is indeed $2n$ plus the key size for CBC-MAC. CMAC (also known as OMAC1) proposes to derive k_2 and k_3 from k using the block cipher. We note that this makes CMAC easier to express as a refinement of FCBC—rather than XCBC. In particular, the security of XCBC cannot be easily used to prove the security of CMAC, as it requires that the three keys be independent.

At this stage, it is necessary to concretely instantiate the block space $\mathbb{B} = \mathrm{GF}(2^n)$, that is represented as the quotient of the polynomial ring GF(2)[x] by a fixed irreductible polynomial of degree n. With this choice of block space, CMAC derives k_2 and k_3 as:

— $k_2 := 2 \times \mathrm{E_k}(0^n)$, when the constant 2 can be seen as the hexadecimal number `0x02`, or the bitstring $0\ldots010$, or the polynomial $x \in \mathrm{GF}(2^n)$, and

— $k_3 := 4 \times \mathrm{E_k}(0^n)$, when the constant 4 can be seen as the hexadecimal number `0x04`, or the bitstring $0\ldots0100$, or the polynomial $x^2 \in \mathrm{GF}(2^n)$.

EasyCrypt

```
module Tweak1 (E : BlockCipher) = {
  proc init(q : int) : unit = {}
  proc f (x : word) : word = {
    var y : word;
```

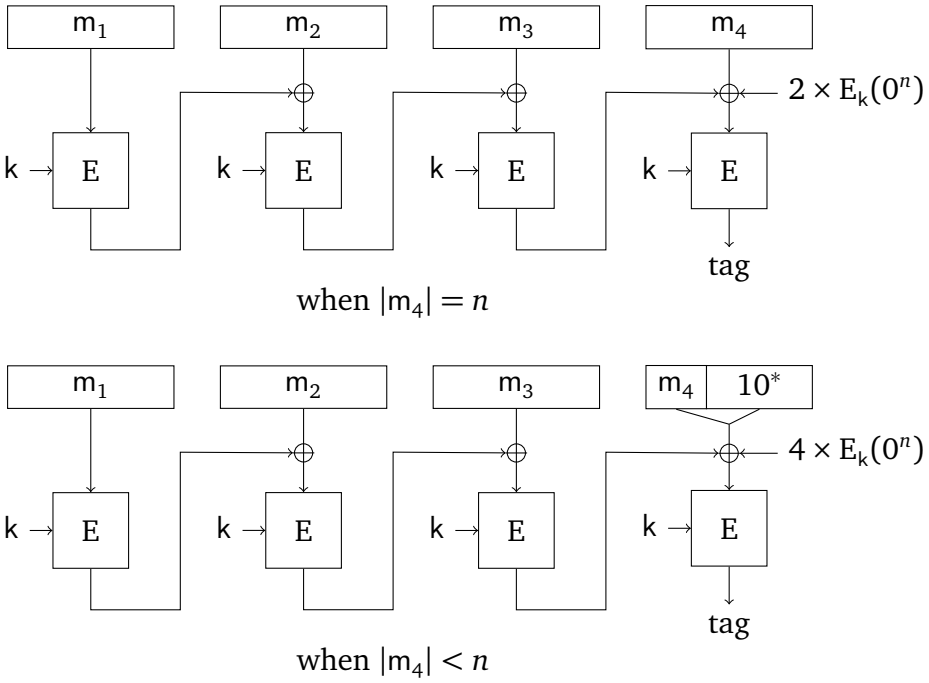


Figure 4.5 – Representation of $\mathrm{CMAC}_{\mathrm{E_k}}(m_1||m_2||m_3||m_4)$

```
    y ← E.f(x ^+ tweak1 XOR.k);
    return y;
  }
}.
module Tweak2 (E : BlockCipher) = {
  proc init(q : int) : unit = {}
  proc f (x : word) : word = {
    var y : word;
    y ← E.f(x ^+ tweak2 XOR.k);
    return y;
  }
}.
module CMAC (E1 : BlockCipher) = {
  proc init (σ : int, q : int) : unit = {
    E1.init(σ+1);  (* this is an artefact from the security proof about counters *)
    XOR.k ← E1.f(AWord.zeros);
  }
  proc f = EPAD(CBCpartial(E1),Tweak1(E1),Tweak2(E1)).f
}.
```

4.2 MAC Security Proofs

This section describes a formalization verifying a concrete bound of the security of CMAC. The proof is articulated as shown in Figure 4.6. The security of both CMAC and XCBC relies on the security proof of FCBC, which is equivalent to the security of ECBC. Finally, the security of ECBC relies on the unlikeliness that two different messages chosen before the block cipher key k is sampled will have the same $\text{CBC-MAC}_{\mathrm{E_k}}$ value.

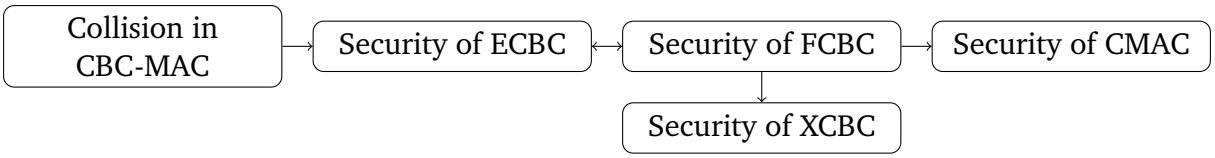


Figure 4.6 – Layers of the CMAC's security proof.

Our security proofs for ECBC, FCBC, XCBC and CMAC are in fact proofs that they cannot be distinguished from a random function assuming that *the underlying block cipher is secure*. Therefore, the block ciphers used with independent random keys can be replaced by independent random permutations, using the lemma in section 3.3.2. Only in the last theorem statements, the advantage of an adversary distinguishing between a block cipher instance and a random permutation will appear. From now on, the implicit function parameters of a parameterized MAC scheme are independent random permutations.

We provide a verified security bound on the advantage of any adversary to:
a) distinguish ECBC and a random function;
b) distinguish FCBC and ECBC (the bound is actually equal to 0);
c) distinguish XCBC and FCBC; and
d) distinguish CMAC and FCBC.
The first three statements follow those by Black and Rogaway [Black and Rogaway, 2005].

4.2.1 Indistinguishability of ECBC from a Random Function

For any independent random permutations π_1, π_2, π_3 :

$$\begin{aligned} \mathrm{ECBC}[\pi_1,\pi_2,\pi_3](m) &= \mathrm{EPAD}[\mathrm{CBC\text{-}MAC}[\pi_1],\pi_2,\pi_3](m) \\ &= \begin{cases} \pi_2(\mathrm{CBC\text{-}MAC}[\pi_1](m)) & \text{if m's size is a multiple of } n \\ \pi_3(\mathrm{CBC\text{-}MAC}[\pi_1](\mathrm{pad}(m)) & \text{otherwise} \end{cases} \end{aligned}$$

Idea of the proof. Since the output of ECBC is either the output of π_2 or π_3, which are both random permutations, it should look random if their input is fresh. This input is the output of CBC-MAC, therefore the probability of distinguishing ECBC from a random function is closely related to the probability to *find a collision* of outputs in CBC-MAC for two different messages without knowing the values CBC-MAC produced. We call this distinguishing event *collision finding*. It is unrelated to the primitive CBC-MAC, so we choose to generalize it.

Generalization. The security of ECBC is formalized by generalizing the first parameter of EPAD. In EasyCrypt, we quantify over all modules `(H <: MACN)`.

Definition of collision-finding probability. This probability is defined as a game-based notion, through the following *collision-finding game*. The adversary $\mathscr{A}$ is asked to provide a list of messages of its choice, without access to the hash values and *before* the hash function represented by the module (H <: MACN) is initialized. The success of $\mathscr{A}$ is defined as the event that the list of messages contains two distinct messages that have the same image by H. This yields the following definition of an adversary $\mathscr{A}$'s *collision-finding advantage* against a probabilistic program represented by the module H.

$$\Pr\left[\mathsf{I} \leftarrow \mathscr{A}; \mathsf{H.init}() : \exists\, m, m' \in \mathsf{I}.\ m \neq m' \wedge \mathsf{H}(m) = \mathsf{H}(m')\right]$$

Statement. If H is collision-finding resistant, then EPAD[H] is a pseudo-random function.

Proof. First, the independent random permutations π_2, π_3 are replaced by independent random functions f_2, f_3 using twice the PRP–PRF switching lemma (Lemma 2.3.1). This adds a term in the bound of indistinguishability of EPAD[H] from a random function.

We apply the PRP–PRF switching lemma on both π_2 and π_3. Following our oracle query counting, the adversary could cause either of these to be queried at most q times, yielding the following probability bound :

$$\left|\Pr\left[\mathsf{h} \xleftarrow{\$} \mathrm{H}; \pi_2 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}); \pi_3 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}); \mathsf{b} \leftarrow \mathscr{A}_{\sigma,q}^{\mathsf{EPAD}[\mathsf{h},\pi_2,\pi_3]} : \mathsf{b} = 1\right]\right.$$
$$\left. - \Pr\left[\mathsf{h} \xleftarrow{\$} \mathrm{H}; \mathsf{f}_2 \xleftarrow{\$} \mathsf{Rand}(\mathbb{B}); \mathsf{f}_3 \xleftarrow{\$} \mathsf{Rand}(\mathbb{B}); \mathsf{b} \leftarrow \mathscr{A}_{\sigma,q}^{\mathsf{EPAD}[\mathsf{h},\mathsf{f}_2,\mathsf{f}_3]} : \mathsf{b} = 1\right]\right| \leq \frac{q(q-1)}{|\mathbb{B}|}$$

From now on, we refer using EPAD[h] to the function composed of EPAD[h, $\mathsf{f}_2, \mathsf{f}_3$] when $\mathsf{f}_2, \mathsf{f}_3$ are two independent random functions and $\mathsf{h} \xleftarrow{\$} \mathrm{H}$.

In EPAD[h], all inputs of both f_2 and f_3 are the output of h. As $\mathsf{f}_2, \mathsf{f}_3$ are two independent random functions, the output of EPAD[h] on fresh messages will be sampled at random unless h maps it to an already produced output value. This corresponds to the definition of a collision in h. Formalizing this argument, we prove that EPAD[h] is indistinguishable from a random function unless a collision, as formally defined below, occurs in h.

$$\begin{aligned} \mathsf{Coll}_h(\mathrm{M}, \mathrm{M}') &:= \mathrm{M} \neq \mathrm{M}' \wedge h(\mathrm{M}) = h(\mathrm{M}') \\ \mathsf{Coll}_h(\mathrm{S}) &:= \exists (\mathrm{M}, \mathrm{M}') \in \mathrm{S}^2, \mathsf{Coll}_h(\mathrm{M}, \mathrm{M}') \end{aligned}$$

This proof is a standard cryptographic reduction, whereby, given a prf–distinguishing adversary $\mathscr{A}$, we construct a collision-finding adversary $\mathscr{B}(\mathscr{A})$ against h that operates with similar time complexity. $\mathscr{B}$ is constructed as follows. $\mathscr{B}$ samples a random function $f \xleftarrow{\$} \mathsf{Rand}(*, \mathbb{B})$ and runs $\mathscr{A}^f$. While doing so, $\mathscr{B}$ stores all the queries of $\mathscr{A}$ in a set [2], pads all the queries whose length is not a multiple of n, then outputs the resulting set. It can then be shown that the following inequality holds, and it remains to bound the right-hand side, which is an instance of the collision-finding game.

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{EPAD}^{\mathsf{f}}_{\mathrm{H}}}(\mathscr{A}) \leq \Pr\left[\mathrm{S} \leftarrow \mathscr{B}(\mathscr{A})_{\sigma,q}; h \xleftarrow{\$} \mathrm{H} : \mathsf{Coll}_h(\mathrm{S})\right]$$

CBC-MAC is collision-finding Resistant

The mode of operation CBC-MAC can be seen as a function family indexed by the block cipher key space $\mathbb{K}$. Sampling a key uniformly at random in $\mathbb{K}$ induces a distribution over $\mathbb{B}^+ \to \mathbb{B}$ for the hash function family $\mathrm{H} = \text{CBC-MAC}_{\mathrm{E}_{\mathbb{K}}}$.

2. We note that the choice of data structure for formalizing collisions does not keep track of query order or multiplicity (in particular, when padding maps an unpadded message to a padded message that is separately queried to h). This relaxation is done without loss of precision since CBC_f is deterministic and two consecutive calls to CBC_f can be swapped without effect on the random function – even if it is sampled lazily.

Difference from Black and Rogaway's proof [Black and Rogaway, 2005]. From here on, our proof for ECBC differs from that by Black and Rogaway [Black and Rogaway, 2005], due to the impossibility of precisely formalizing their arguments in EasyCrypt.

Using their notations, in the proof of their Lemma 3, they compute probabilities of events of the form $\Pr\left[Y_{i-1} \oplus M_i = Y_{j-1} \oplus M_j\right]$, relying on their ability to compute the probability of sampling a particular value for Y_{i-1}. However, this sampling occurs in a previous iteration of the loop, and may in fact be overwritten, losing its "randomness" for the next iteration where the events are tested.

One may argue that every value it may be overwritten with in fact follows the same distribution. However, EasyCrypt's logics–and indeed entire proof methodology relies on reasoning about values rather than distributions, and its logics cannot express the fact that some intermediate value follows a particular distribution. On the other hand, a standard way of dealing with similar issues would be to delay the random sampling until the value is used, allowing a precise probability computation. However, in this case, the value could in fact be overwritten between the point where it is initially sampled and the point where it is used. This introduces dependencies between random values and the adversary's view of the system that make it impossible to delay sampling operations as desired.

Our method. If we cannot formalize precisely their argument, we can formalize a simpler and less precise bound that does not discount internal collisions when they are caused by a common prefix. In particular, instead of computing the collision probability for the full set S of messages, we guess which of the couples of messages may produce a collision and use standard arguments to bound the probability that any of them collide. We sample two of the messages from S and only compute their collision probability. Let $\mathscr{C}(\mathscr{B})$ be the adversary that calls the adversary $\mathscr{B}$ which outputs a set $S \subset \mathbb{B}^+$, then samples two distinct messages $M, M' \xleftarrow{\$} S$ and outputs (M, M'). There are at most $\frac{|S|(|S|-1)}{2}$ different pairs.

Lemma 4.2.1. *For any function family $H \in \{\mathbb{B}^+ \to \mathbb{B}\}$, any adversary $\mathscr{B}$ that outputs a set of size at most c (with $2 \leq c$) has a comparable collision probability to the adversary that tries to guess where the collision may happen and only test this collision.*

$$\Pr\left[S \leftarrow \mathscr{B}_c; h \xleftarrow{\$} H : \mathsf{Coll}_h(S)\right] \leq$$
$$\frac{c(c-1)}{2} \cdot \Pr\left[m, m' \leftarrow \mathscr{C}(\mathscr{B})_c; h \xleftarrow{\$} H : \mathsf{Coll}_h(m, m')\right]$$

In the context of ECBC, the hash function family is concretely CBC-MAC, parametrized by a random permutation π. However, the PRP–PRF switching lemma applied on π allows to consider the collision-finding probability on CBC-MAC that is parametrized by a random function instead of a random permutation. This makes the collision-finding probability to be more easily bounded in CBC-MAC.

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{prf}}_{\mathrm{ECBC}}(\mathscr{A}) \\
&\leq \frac{q(q-1)}{2^n} + \Pr\left[S \leftarrow \mathscr{B}[\mathscr{A}]_\sigma; \pi \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}) : \mathsf{Coll}_{\mathrm{CBC\text{-}MAC}_\pi}(S)\right] \\
&\leq \frac{q(q-1)}{2^n} + \frac{\sigma(\sigma-1)}{2^{n+1}} + \frac{q(q-1)}{2} \cdot \Pr\left[m, m' \leftarrow \mathscr{C}[\mathscr{B}[\mathscr{A}]]_q; f \xleftarrow{\$} \mathsf{Rand}(\mathbb{B}) : \mathsf{Coll}_{\mathrm{CBC\text{-}MAC}_f}(m, m')\right]
\end{aligned}$$

It therefore remains to bound the probability of collisions of two different messages in $\mathrm{CBC\text{-}MAC}_f$ when f is a random function.

Theorem 4.2.2 (CBC-MAC collision probability of two messages)**.** *For any natural number c, an adversary $\mathscr{A}_c$ that outputs two bit-strings $(\mathsf{m}, \mathsf{m}')$ whose lengths are at most cn, with $0 < c$, has a low probability of producing a collision with CBC-MAC when parameterized by a random function.*

$$\Pr\Big[\mathsf{m}, \mathsf{m}' \leftarrow \mathscr{A}_c; \mathsf{f} \xleftarrow{\$} \mathsf{Rand}(\mathbb{B}) : \mathsf{Coll}_{\mathit{CBC\text{-}MAC}_\mathsf{f}}(\mathsf{m}, \mathsf{m}')\Big] \leq \frac{2c(2c-1)}{2^n} + \frac{1}{2^n}$$

Proof. Let $(\mathsf{m}, \mathsf{m}')$ be two distinct messages output by the adversary $\mathscr{A}_c$, and f the random function sampled by the experiment. If there is no collision in any of the inputs of f during the CBC chaining, $\mathrm{CBC}_\mathsf{f}(\mathsf{m})$ will collide with $\mathrm{CBC}_\mathsf{f}(\mathsf{m}')$ only if the final call to f yields a collision. This occurs with probability at most $\frac{1}{2^n}$.

It remains to bound the probability that a collision occurs somewhere along the chain of inputs to f. The chaining for the longest common prefix of m and m' is computed only once, and collisions are only considered afterwards. We then bound the probability of a collision occurring in one of the inputs to f by the probability of a collision occurring when those $2c$ inputs are sampled. □

Combining all results above allows us to conclude with a security bound for ECBC.

Theorem 4.2.3 (prf-advantage of ECBC)**.** *For any natural numbers q, l, σ, n, an adversary $\mathscr{A}$ making at most q queries, each query of maximum size ln and of total size in the number of blocks at most σ, has a low probability to distinguish ECBC from a random function, when parameterized by independent random permutations.*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathit{ECBC}}(\mathscr{A}) \leq \frac{0.5\sigma^2 + 2q^2l^2 + q^2}{2^n} \tag{4.1}$$

4.2.2 Indistinguishability of FCBC from ECBC

We now relate the structure of FCBC with that of ECBC. Indeed, we note that ECBC is a particular instance of FCBC with non-independent permutations. Given three independent permutations π_1, π_2, π_3, we have

$$\mathrm{ECBC}_{\pi_1,\pi_2,\pi_3} = \mathrm{FCBC}_{\pi_1,\pi_2\circ\pi_1,\pi_3\circ\pi_1}$$

The security proof for FCBC then simply relies on the fact that the composition of two independent random permutations remains independent from the first one. In other words, given two independent random permutations π_1, π_2, the distributions of $(\pi_1, \pi_2 \circ \pi_1)$ and (π_1, π_2) are the same.

Lemma 4.2.4. *An adversary $\mathscr{A}$ making an unbounded number of oracle queries cannot distinguish the composition of two independent random permutations.*

$$\Pr\Big[\pi_1, \pi_2 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B})^2 : \mathscr{A}^{\pi_1,\pi_2} = 1\Big] = \Pr\Big[\pi_1, \pi_2 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B})^2 : \mathscr{A}^{\pi_1,\pi_2\circ\pi_1} = 1\Big]$$

Proof of Lemma 4.2.4. It is possible to pre-sample outputs for all inputs of both π_1 and π_2. This makes it clear that answers to $\pi_2 \circ \pi_1$ queries are independent from past queries to π_1 and that, with π_1 fixed, the distributions of $\pi_2 \circ \pi_1$ and π_2 are equal. □

Theorem 4.2.5 (Security of FCBC)**.** *An adversary $\mathscr{A}$ cannot distinguish FCBC from ECBC, when parameterized by independent random permutations.*

$$\mathsf{Adv}^{\mathsf{prf}}_{FCBC}(\mathscr{A}) = \mathsf{Adv}^{\mathsf{prf}}_{ECBC}(\mathscr{A}) \tag{4.2}$$

Proof. This is a direct application of Lemma 4.2.4 (applied twice). □

4.2.3 Indistinguishability of XCBC from FCBC

The proof that XCBC is indistinguishable from FCBC is based on a more general lemma from [Black and Rogaway, 2005], which states that an adversary with oracle access to two independent random permutations $\pi_1(\cdot)$ and $\pi_2(\cdot)$ cannot distinguish them from oracles $\pi(\cdot)$ and $\pi(k \oplus \cdot)$, when π is a random permutation and k is chosen uniformly at random and independently from π (and remains hidden from the adversary).

We view XCBC as a particular instance of FCBC. Indeed, given a permutation π and two blocks k_2, k_3, we have

$$\mathrm{XCBC}_{\pi,k_2,k_3} = \mathrm{FCBC}_{\pi,\pi(k_2\oplus\cdot),\pi(k_3\oplus\cdot)}$$

We thus prove that the security of XCBC is implied by that of FCBC. The proof crucially relies on the following lemma, which states that one can always (computationally) simulate two independent random permutations using a unique random permutation and a random constant.

Lemma 4.2.6. *For any natural number c, an adversary $\mathscr{A}$ making at most c oracle queries, with $0 < c \leq \frac{2^n}{2}$, has a low probability of distinguishing between two independent permutations (π_1, π_2) and the pair of permutations $(\pi, \pi(k \oplus \cdot))$.*

$$\left|\Pr\left[\pi_1, \pi_2 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B})^2; b \leftarrow \mathscr{A}_c^{\pi_1,\pi_2} : b = 1\right] - \Pr\left[\pi, k \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}) \times \mathbb{B}; b \leftarrow \mathscr{A}_c^{\pi,\pi(k\oplus\cdot)} : b = 1\right]\right| \leq \frac{1.25c^2}{2^n}$$

Proof of Lemma 4.2.6. The proof closely follows that of [Black and Rogaway, 2005]. Let $\mathscr{A}$ be an adversary that expects two permutations (as in the given game). The adversary's goal here is to find which distribution the oracles it was given come from. On one hand, there are two independent random permutations, and on the other hand there is a random permutation that simulates two permutations with a random constant.

We prove that the distributions are equal *unless* the adversary makes two queries x and y to $[\mathsf{O}]_1$ and $[\mathsf{O}]_2$ such that $[\mathsf{O}]_1(x) = [\mathsf{O}]_2(y)$. When two independent permutations are used, this may happen with low probability for any pair (x, y). On the other hand, when the second permutation is a masked version of the first, this can only occur when $x \oplus y = k$. Once such a pair of queries has been found, the adversary can decide with very high probability which oracles she is interacting with by simply checking, for some $z \neq x$ whether $[\mathsf{O}]_1(z) = [\mathsf{O}]_2(z \oplus x \oplus y)$, which is very unlikely if $[\mathsf{O}]_1$ and $[\mathsf{O}]_2$ are independent, but will hold with probability 1 otherwise.

We further prove that an adversary that makes such a pair of queries must have either: i. queried x to $[\mathsf{O}]_1$ and $x \oplus k$ to $[\mathsf{O}]_2$ without knowing the value of k; or ii. queried x to $[\mathsf{O}]_1$ and $y \neq x \oplus k$ to $[\mathsf{O}]_2$ and obtained $\mathsf{O}_1(x) = \mathsf{O}_2(y)$. We can then bound the probability of the first event by $\frac{0.25c^2}{2^n}$, and the probability of the second event by $\frac{c^2}{2^n}$. □

Lemma 4.2.7 (Security of XCBC)**.** *For any natural numbers q, σ, n, an adversary $\mathscr{A}$ making at most q queries, of total size in the number of blocks at most σ, has a low probability of distinguishing XCBC from FCBC, when XCBC is parameterized by a random permutation and two independent random constants, and FCBC is parameterized by three independent random permutations.*

$$\mathsf{Adv}^{\mathsf{prf}}_{XCBC}(\mathscr{A}) \leq \mathsf{Adv}^{\mathsf{prf}}_{FCBC}(\mathscr{A}) + \frac{2.5\sigma^2}{2^n} \tag{4.3}$$

Proof. When an adversary succeeds in distinguishing XCBC from a random function, either it has distinguished XCBC from FCBC, or FCBC from a random function. To bound the distinguishing advantage of any adversary between FCBC and XCBC, let π_1, π_2, π_3 be three independent random permutations and k_1, k_2 two independent random constants. Lemma 4.2.6 is instantiated twice with the bound σ, making $\mathrm{FCBC}_{\pi_1,\pi_2,\pi_3}$ indistinguishable from $\mathrm{FCBC}_{\pi_1,\pi_1(k_1\oplus\cdot),\pi_3}$, and indistinguishable from $\mathrm{FCBC}_{\pi_1,\pi_1(k_1\oplus\cdot),\pi_1(k_2\oplus\cdot)}$, i.e. $\mathrm{XCBC}_{\pi_1,k_1,k_2}$. □

A small flaw Note that our bound on the security of XCBC is different from Black and Rogaway's [Black and Rogaway, 2005]. They instantiate their lemma called *Two Permutations From One* (Lemma 4.2.6 in the present paper) with the wrong bound on the total number of oracle queries. Indeed, the bound c on the number of queries in Lemma 4.2.6 is a bound on the *total* number of oracle queries. On the other hand, in both instantiations of Lemma 4.2.6, Black and Rogaway only count the number of MAC queries that do not need padding (accounting only for queries to the first oracle in Lemma 4.2.6). The flaw is subtle, and has no effect on the security bound (which we improve below), but is present nonetheless, adding to the body of evidence that cryptographic proofs are difficult both to write and to evaluate.

An improvement Furthermore, to tighten the bound back, we extend Lemma 4.2.6 into the following Lemma 4.2.8, which states that a random permutation can simulate three independent random permutations using two independent random constants. Proving this lemma directly, rather than relying on Lemma 4.2.6 twice allows us to improve the bound slightly, by allowing us to count queries to the first permutation once only.

Lemma 4.2.8. *For any natural number c, an adversary $\mathscr{A}$ making at most c total oracle queries, with $0 < c \leq 2^{n-1}$, has a low probability to distinguish between three independent permutations (π_1, π_2, π_3) and the tuple of permutations $(\pi, \pi(k_1 \oplus \cdot), \pi(k_2 \oplus \cdot))$.*

$$\left|\Pr\left[\pi_1, \pi_2, \pi_3 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B})^3; b \leftarrow \mathscr{A}_c^{\pi_1,\pi_2,\pi_3} : b = 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}); (k_1, k_2) \xleftarrow{\$} \mathbb{B}^2; b \leftarrow \mathscr{A}_c^{\pi,\pi(k_1\oplus\cdot),\pi(k_2\oplus\cdot)} : b = 1\right]\right| \leq \frac{1.75c^2}{2^n}$$

Proof. This is very similar to Lemma 4.2.6 and Lemma 4.2.9. The proof needs to bound the events $\mathsf{coll}_{\mathsf{rng}}$ (defined for Lemma 4.2.9) and another one. The bound of $\mathsf{coll}_{\mathsf{rng}}$ has been proved as a generic result and then instanciated in Lemmas 4.2.8 and 4.2.9. The probability of the other one is bounded by $\frac{0.75\sigma^2}{2^n}$. □

4.2.4 Indistinguishability of CMAC from FCBC

The security of CMAC cannot be easily deduced from that of XCBC, since the core lemma of the security of XCBC imposes that the masks be independent from the permutation. Since this is not the case for CMAC, we use ideas from [Iwata and Kurosawa, 2003] to relate the security of CMAC to that of FCBC. Their key idea is to prevent the adversary from directly accessing the block cipher oracle by adding an independent random variable.

Our security proof generalises CMAC slightly in that we use abstract public functions $f_1, f_2 : \mathbb{B} \to \mathbb{B}$ to capture the derivation of k_2 and k_3 from $\pi(0^n)$. The security of CMAC follows by instantiating $f_1 = x \mapsto 2 \times x$ and $f_2 = x \mapsto 4 \times x$.

The security of this generalised CMAC (or indeed of concrete CMAC) does not immediately follow from that of XCBC, since its constants $k_2 = f_1(\pi(0^n))$ and $k_3 = f_2(\pi(0^n))$ are not independent between themselves, and furthermore not independent from π. However, the security of CMAC is easily implied by that of FCBC, of which it is an instance. Indeed, given π,

$$\mathrm{CMAC}_\pi = \mathrm{FCBC}_{\pi,\pi(f_1(\pi(0^n))\oplus\cdot),\pi(f_2(\pi(0^n))\oplus\cdot)}$$

Here again, our security proof is close to that of [Iwata and Kurosawa, 2003], which relies on a more general construction, MOMAC, that generalises both CMAC and FCBC. We provide a game-based proof and go into more details.

MOMAC

The MOMAC construction [Iwata and Kurosawa, 2003] is parameterised by 6 oracles $\mathsf{O}_{1\le i\le 6}$, which are used as follows.

$$\mathsf{MOMAC}_{\mathsf{O}_1,\mathsf{O}_2,\mathsf{O}_3,\mathsf{O}_4,\mathsf{O}_5,\mathsf{O}_6}(\mathsf{m}) \quad := \quad \begin{cases} \mathsf{O}_5(\mathsf{pad}(\mathsf{m})) & \text{if } 0 \le |\mathsf{m}| < n \\ \mathsf{O}_3(\mathsf{m}) & \text{if } |\mathsf{m}| = n \\ \mathrm{FCBC}_{\mathsf{O}_1,\mathsf{O}_4,\mathsf{O}_6}(\mathsf{m}) & \text{if } n < |\mathsf{m}| \le 2n \\ \mathrm{FCBC}_{\mathsf{O}_2,\mathsf{O}_4,\mathsf{O}_6}(\mathsf{m}_{\mathsf{O}_1}) & \text{if } 2n < |\mathsf{m}| \wedge \lceil |\mathsf{m}|/n \rceil = m \end{cases}$$

when $\mathsf{m}_{\mathsf{O}_1} = (\mathsf{O}_1(\mathsf{m}_1) \oplus \mathsf{m}_2) \,||\, \ldots \,||\, \mathsf{m}_m$.

The resulting construction is also illustrated in Figure 4.7.

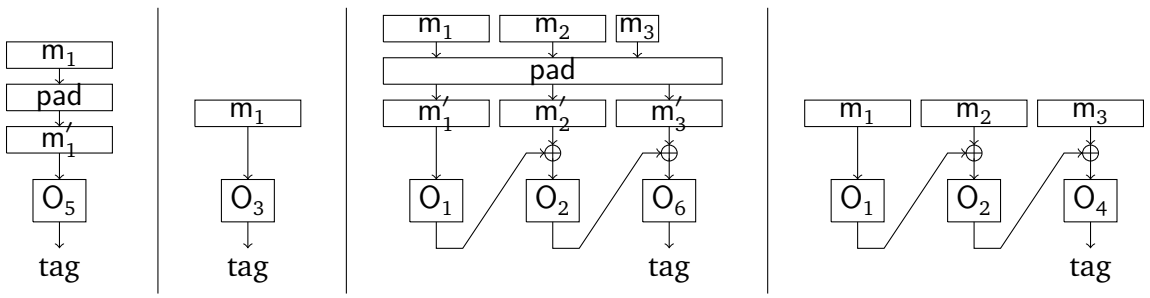


Figure 4.7 – Illustration of MOMAC, by increasing size order: $|\mathsf{m}| < n$, $|\mathsf{m}| = n$, $2n < |\mathsf{m}| < 3n$, $|\mathsf{m}| = 3n$,

CMAC and FCBC as instances of MOMAC

Given a random permutation π and a random n-bit string r, the following six oracles $\mathrm{Q}_{1\le i\le 6}$, where $\mathrm{L}^{\pi} = \pi(0^n)$, can be used with MOMAC to build CMAC.

$$\begin{aligned} \mathrm{Q}_1(x) &:= \pi(x) \oplus \mathsf{r} & \mathrm{Q}_2(x) &:= \pi(x \oplus \mathsf{r}) \oplus \mathsf{r} \\ \mathrm{Q}_3(x) &:= \pi(x \oplus f_1(\mathrm{L}^{\pi})) & \mathrm{Q}_4(x) &:= \pi(x \oplus \mathsf{r} \oplus f_1(\mathrm{L}^{\pi})) \\ \mathrm{Q}_5(x) &:= \pi(x \oplus f_2(\mathrm{L}^{\pi})) & \mathrm{Q}_6(x) &:= \pi(x \oplus \mathsf{r} \oplus f_2(\mathrm{L}^{\pi})) \end{aligned}$$

Given three independent random permutations π_1, π_2, π_3 and a random n-bit string r, the following six oracles $\mathrm{R}_{1\le i\le 6}$ can be used with MOMAC to build FCBC.

$$\begin{aligned} \mathrm{R}_1(x) &:= \pi_1(x) \oplus \mathsf{r} & \mathrm{R}_2(x) &:= \pi_1(x \oplus \mathsf{r}) \oplus \mathsf{r} \\ \mathrm{R}_3(x) &:= \pi_2(x) & \mathrm{R}_4(x) &:= \pi_2(x \oplus \mathsf{r}) \\ \mathrm{R}_5(x) &:= \pi_3(x) & \mathrm{R}_6(x) &:= \pi_3(x \oplus \mathsf{r}) \end{aligned}$$

Security proof

For any value of r, the following functional equivalences follow from these definitions.

$$\begin{aligned} \mathrm{CMAC}_{\pi} &\sim \mathrm{MOMAC}_{\mathrm{Q}_1,\mathrm{Q}_2,\mathrm{Q}_3,\mathrm{Q}_4,\mathrm{Q}_5,\mathrm{Q}_6} \\ \mathrm{FCBC}_{\pi_1,\pi_2,\pi_3} &\sim \mathrm{MOMAC}_{\mathrm{R}_1,\mathrm{R}_2,\mathrm{R}_3,\mathrm{R}_4,\mathrm{R}_5,\mathrm{R}_6} \end{aligned}$$

Therefore, distinguishing CMAC from FCBC can be reduced to distinguishing the Q_i from the R_i. In the following, for any adversary $\mathscr{A}$, expecting six oracles and making at most q oracle queries to $\mathsf{O}_3, \mathsf{O}_4, \mathsf{O}_5, \mathsf{O}_6$ and at most σ total oracle queries before returning a boolean, we name $\mathrm{Q}(\mathscr{A})$ (resp. $\mathrm{R}(\mathscr{A})$) the game that initializes the oracles $\mathrm{Q}_{1\le i\le 6}$ (resp. $\mathrm{R}_{1\le i\le 6}$), then calls the adversary $\mathscr{A}^{\mathrm{Q}_i}$ (resp. $\mathscr{A}^{\mathrm{R}_i}$).

$$\begin{aligned} \mathrm{Q}(\mathscr{A}) &:= \left[\pi \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}), \mathsf{r} \xleftarrow{\$} \mathbb{B}; b \leftarrow \mathscr{A}^{\mathrm{Q}_{1\le i\le 6}}_{\sigma,q} \right] \\ \mathrm{R}(\mathscr{A}) &:= \left[\pi_1, \pi_2, \pi_3 \xleftarrow{\$} \mathsf{Perm}(\mathbb{B}), \mathsf{r} \xleftarrow{\$} \mathbb{B}; b \leftarrow \mathscr{A}^{\mathrm{R}_{1\le i\le 6}}_{\sigma,q} \right] \end{aligned}$$

Lemma 4.2.9. *For any bijective f_1 and f_2 such that $x \mapsto x \oplus f_1(x)$, $x \mapsto x \oplus f_2(x)$, $x \mapsto f_1(x) \oplus f_2(x)$, $x \mapsto x \oplus f_1(x) \oplus f_2(x)$ are also bijective, an adversary $\mathscr{A}$, making at most q oracle queries to $\mathsf{O}_3, \mathsf{O}_4, \mathsf{O}_5, \mathsf{O}_6$ and at most σ total oracle queries, has a low probability of distinguishing the game with the R_i from the game with the Q_i.*

$$|\Pr[\mathrm{Q}(\mathscr{A}) : b = 1] - \Pr[\mathrm{R}(\mathscr{A}) : b = 1]| \le \frac{\sigma(\sigma+1)}{2^n} + \frac{\frac{1}{4}(\sigma+1)^2 + \frac{1}{2}q^2}{2^n} + \frac{\frac{3}{4}(\sigma+1)^2}{2^n}$$

We first give an overview of the proof, using names that are defined later. The rest of this section details individual steps.

We formally prove that, for any adversary $\mathscr{A}$, the game $\mathrm{Q}(\mathscr{A})$ is equivalent to $\mathrm{R}(\mathscr{A})$ upto the event $\mathsf{find}_{\mathrm{L}} \vee \mathsf{coll}_{\mathsf{rng}}$. It is easy to bound the probability of $\mathsf{coll}_{\mathsf{rng}}$ occurring, but the probability of $\mathsf{find}_{\mathrm{L}}$ occurring in $\mathrm{R}(\mathscr{A})$ requires additional work. We introduce two new games $\mathrm{S}(\mathscr{A})$ and $\mathrm{T}(\mathscr{A})$, represented in Figure 4.8. Game $\mathrm{S}(\mathscr{A})$ is perfectly equivalent to $\mathrm{R}(\mathscr{A})$, thus the probability of $\mathsf{find}_{\mathrm{L}}$ occurring is equal in both games. We then prove that $\mathrm{T}(\mathscr{A})$ is equivalent to $\mathrm{S}(\mathscr{A})$ upto a third event $\mathsf{find}_{\mathsf{r}}$.

Game	O_1	O_2	O_3	O_4	O_5	O_6
$Q(\mathscr{A})$	$\pi(\cdot)\oplus r$	$\pi(\cdot\oplus r)\oplus r$	$\pi(\cdot\oplus f_1(\pi(0^n)))$	$\pi(\cdot\oplus r\oplus f_1(\pi(0^n)))$	$\pi(\cdot\oplus f_2(\pi(0^n)))$	$\pi(\cdot\oplus r\oplus f_2(\pi(0^n)))$
$R(\mathscr{A})$	$\pi_1(\cdot)\oplus r$	$\pi_1(\cdot\oplus r)\oplus r$	$\pi_2(\cdot)$	$\pi_2(\cdot\oplus r)$	$\pi_3(\cdot)$	$\pi_3(\cdot\oplus r)$
$S(\mathscr{A})$	$\pi_1(\cdot)$	$\pi_1(\cdot\oplus r)$	$\pi_2(\cdot)$	$\pi_2(\cdot\oplus r)$	$\pi_3(\cdot)$	$\pi_3(\cdot\oplus r)$
$T(\mathscr{A})$	$\pi_1(\cdot)$	$\pi'_1(\cdot)$	$\pi_2(\cdot)$	$\pi'_2(\cdot)$	$\pi_3(\cdot)$	$\pi'_3(\cdot)$

Figure 4.8 – Sequence of games.

The following sequence of inequalities shows an outline of our detailed proof, with the justification for each non-trivial step given in the paragraph whose heading is listed beside the step.

$$
\begin{aligned}
&|\Pr[Q(\mathscr{A}) : b = 1] - \Pr[R(\mathscr{A}) : b = 1]| && \text{(using } Q(\mathscr{A}) \sim R(\mathscr{A}) \text{ upto } \mathsf{coll}_{\mathsf{rng}} \vee \mathsf{find}_{\mathrm{L}}\text{)}\\
&\quad \le \Pr\left[R(\mathscr{A}) : \mathsf{coll}_{\mathsf{rng}}\right] + \Pr[R(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] && \text{(using } R(\mathscr{A}) \sim S(\mathscr{A})\text{)}\\
&\quad = \Pr\left[R(\mathscr{A}) : \mathsf{coll}_{\mathsf{rng}}\right] + \Pr[S(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}]\\
&\quad \le \Pr\left[R(\mathscr{A}) : \mathsf{coll}_{\mathsf{rng}}\right] + \Pr[T(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] + |\Pr[S(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] - \Pr[T(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}]|\\
&\qquad\qquad && \text{(using } S(\mathscr{A}) \sim T(\mathscr{A}) \text{ upto } \mathsf{find}_{\mathrm{r}}\text{)}\\
&\quad \le \Pr\left[R(\mathscr{A}) : \mathsf{coll}_{\mathsf{rng}}\right] + \Pr[T(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] + \Pr[T(\mathscr{A}) : \mathsf{find}_{\mathrm{r}}]
\end{aligned}
$$

The conclusion then only needs to bound the probability of $\mathsf{coll}_{\mathsf{rng}}$ and $\mathsf{find}_{\mathrm{r}}$ in $T(\mathscr{A})$. We now discuss individual steps in the proof, including definitions for the events. In Figure 4.8 and below, L^π denotes the value $\pi(0^n)$ and L_i^π denotes the value $f_i(L^\pi)$ for $i \in \{1,2\}$.

$Q(\mathscr{A}) \sim R(\mathscr{A})$ **upto** $\mathsf{coll}_{\mathsf{rng}} \vee \mathsf{find}_{\mathrm{L}}$

In a proof reminiscent of that of Lemma 4.2.6, we note that games $Q(\mathscr{A})$ and $R(\mathscr{A})$ are equivalent unless $\mathscr{A}$ queries some x to O_i and some y to O_j, with $(i,j) \in \{(1,3),(1,5),(3,5)\}$ and such that $O_i(x) = O_j(y)$. If this event (which we simply call bad below) occurs, then $\mathscr{A}$ can distinguish the two sets of oracles with high probability by testing, for some $z \neq x$ whether $O_i(z) = O_j(z \oplus x \oplus y)$. We now prove that event bad as defined above completely captures all cases in which the two games can be distinguished. To simplify bounding the probability of bad occurring, we consider instead a disjunction of two disjoint events $\mathsf{find}_{\mathrm{L}}$ and $\mathsf{coll}_{\mathsf{rng}}$, depending on the value of $x \oplus y$. Event $\mathsf{find}_{\mathrm{L}}$ occurs if $\mathscr{A}$ queries, for some x:

— $O_1(x)$ and $O_3(x \oplus f_1(L^\pi))$; or
— $O_1(x)$ and $O_5(x \oplus f_2(L^\pi))$; or
— $O_3(x)$ and $O_5(x \oplus f_1(L^\pi) \oplus f_2(L^\pi))$.

Event $\mathsf{coll}_{\mathsf{rng}}$ captures the remainder of the cases in bad, and thus occurs when $\mathscr{A}$ queries, for some x and y:

— $O_1(x)$ and $O_3(y)$ such that $O_1(x) = O_3(y)$ and $y \neq x \oplus f_1(L^\pi)$; or
— $O_1(x)$ and $O_5(y)$ such that $O_1(x) = O_5(y)$ and $y \neq x \oplus f_2(L^\pi)$; or
— $O_3(x)$ and $O_5(y)$ such that $O_3(x) = O_5(y)$ and $y \neq x \oplus f_1(L^\pi) \oplus f_2(L^\pi)$.

We consider two games $X_1(\mathscr{A})$ and $X_2(\mathscr{A})$, which are respectively equivalent to $Q(\mathscr{A})$ and $R(\mathscr{A})$. $X_1(\mathscr{A})$ and $X_2(\mathscr{A})$ are shown in Figure 4.9, where code inside the dotted boxes appears only in $X_1(\mathscr{A})$. Figure 4.9 only shows definitions for O_1, O_3 and O_5, which are sufficient to define the entire games. Indeed, we note that, in both $Q(\mathscr{A})$ and $R(\mathscr{A})$, we have $O_2(x) = O_1(x \oplus L_1^\pi)$ and that O_4 and O_3, and O_6 and O_5 obey the same relation.

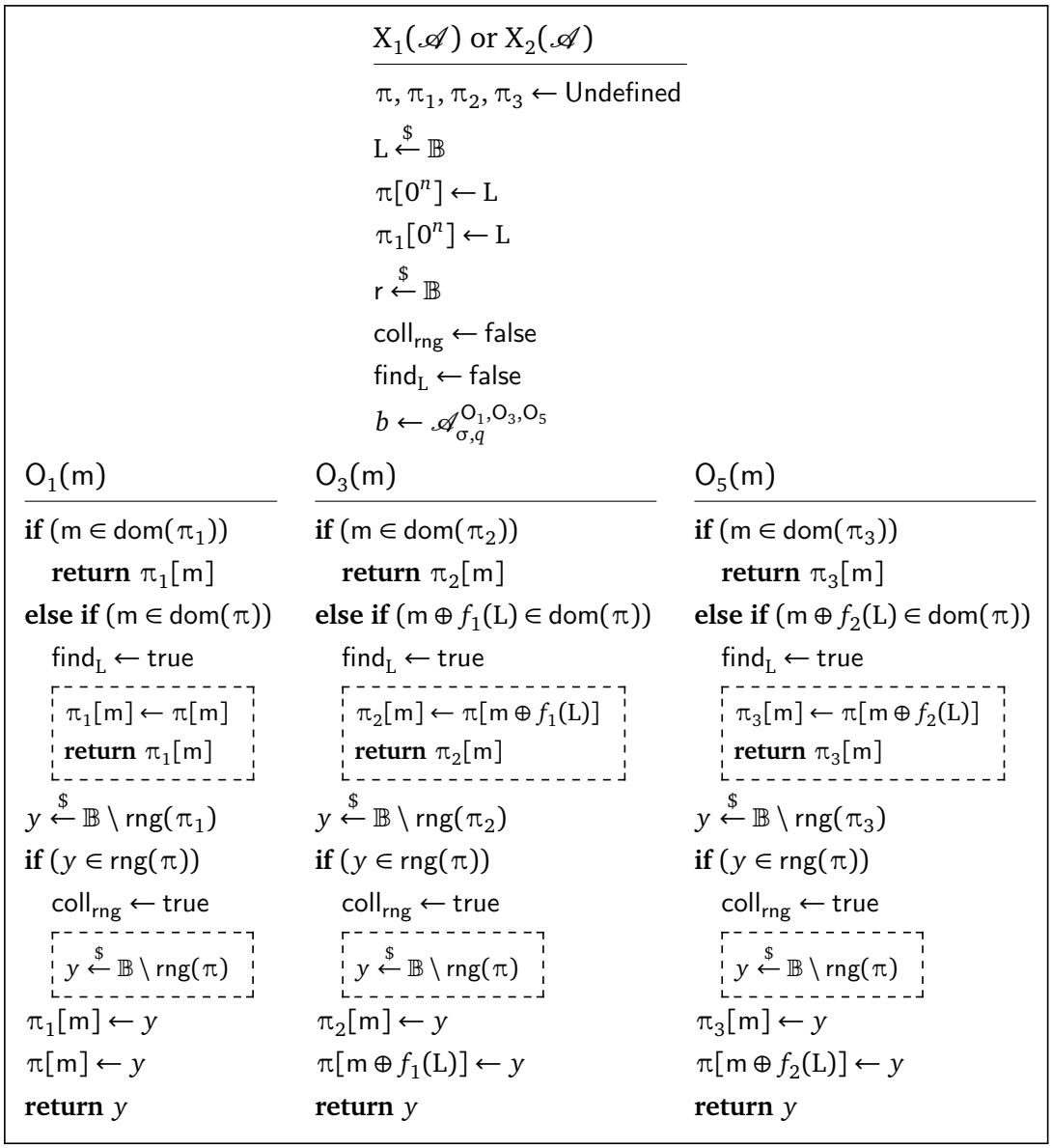


Figure 4.9 – Games $X_1(\mathscr{A})$ (including dotted boxes) and $X_2(\mathscr{A})$ (excluding dotted boxes).

Figure 4.9 shows very clearly that $Q(\mathscr{A})$ (or X_1) and $R(\mathscr{A})$ (or X_2) cease to be equivalent only when one of find_L or coll_{rng} becomes true. Thus, we have:

$$\begin{aligned}
&|\Pr[Q(\mathscr{A}) : b = 1] - \Pr[R(\mathscr{A}) : b = 1]| \\
&= |\Pr[X_1(\mathscr{A}) : b = 1] - \Pr[X_2(\mathscr{A}) : b = 1]| \\
&\leq \Pr\left[X_2(\mathscr{A}) : \mathsf{coll}_{rng} \vee \mathsf{find}_L\right] \\
&\leq \Pr\left[X_2(\mathscr{A}) : \mathsf{coll}_{rng}\right] + \Pr[X_2(\mathscr{A}) : \mathsf{find}_L]
\end{aligned}$$

Bounding $\Pr\left[X_2(\mathscr{A}) : \mathsf{coll}_{rng}\right]$

We note that coll_{rng} is essentially the probability of a freshly sampled variable already appearing in some set. This can easily be bound, knowing that the total number of oracle queries is at most $\sigma \leq 2^{n-1}$.

For any adversary $\mathscr{A}$ that makes at most σ oracle queries, we have,

$$\Pr\left[\mathrm{X}_2(\mathscr{A}) : \mathsf{coll}_{\mathsf{rng}}\right] \leq \frac{\sigma(\sigma+1)}{2} \cdot \frac{1}{2^n - (\sigma+1)} \leq \frac{\sigma(\sigma+1)}{2^n}$$

$\mathrm{R}(\mathscr{A}) \sim \mathrm{S}(\mathscr{A})$

Event $\mathsf{find}_{\mathrm{L}}$ in game $\mathrm{X}_2(\mathscr{A})$ can also be expressed as follows.

$$\mathsf{find}_{\mathrm{L}} \Leftrightarrow \bigvee \left\{ \begin{array}{ll} \mathrm{L}_1^{\pi_1} & \in \mathsf{dom}(\pi_1) \oplus \mathsf{dom}(\pi_2), \\ \mathrm{L}_2^{\pi_1} & \in \mathsf{dom}(\pi_1) \oplus \mathsf{dom}(\pi_3), \\ \mathrm{L}_1^{\pi_1} \oplus \mathrm{L}_2^{\pi_1} & \in \mathsf{dom}(\pi_2) \oplus \mathsf{dom}(\pi_3) \end{array} \right\}$$

Its probability cannot be computed directly in $\mathrm{X}_2(\mathscr{A})$, since the collision involves L^{π}, which is not independent from the adversary's view. We therefore need to modify $\mathrm{X}_2(\mathscr{A})$ somewhat into a game $\mathrm{S}(\mathscr{A})$ shown in Figure 4.10 to bound this probability. The objective of game $\mathrm{S}(\mathscr{A})$ is to extend the $\mathsf{find}_{\mathrm{L}}$ event to include fresh and independent randomness r in its definition. This will then enable us to make use of it to bound the probability of $\mathsf{find}_{\mathrm{L}}$, since it is independent from the oracles' outputs. It is easy to see that for any adversary $\mathscr{A}$, $\mathrm{S}(\mathscr{A})$ is perfectly equivalent to $\mathrm{X}_2(\mathscr{A})$. We formally prove

$$\Pr[\mathrm{X}_2(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] = \Pr[\mathrm{S}(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}]$$

Event $\mathsf{find}_{\mathrm{L}}$ in game $\mathrm{S}(\mathscr{A})$

In game $\mathrm{X}_2(\mathscr{A})$, the value of L^{π_1} as it appears in event $\mathsf{find}_{\mathrm{L}}$ is equal to $\pi_1(0^n)$. In game $\mathrm{S}(\mathscr{A})$, its value is related to r is equal to $\mathsf{r} \oplus \pi_1(0^n)$. Event $\mathsf{find}_{\mathrm{L}}$ in game $\mathrm{S}(\mathscr{A})$ can now be expressed as follows.

$$\mathsf{find}_{\mathrm{L}} \Leftrightarrow \bigvee \left\{ \begin{array}{ll} f_1(\mathrm{L}^{\pi_1} \oplus \mathsf{r}) & \in \mathsf{dom}(\pi_1) \oplus \mathsf{dom}(\pi_2), \\ f_2(\mathrm{L}^{\pi_1} \oplus \mathsf{r}) & \in \mathsf{dom}(\pi_1) \oplus \mathsf{dom}(\pi_3), \\ f_1(\mathrm{L}^{\pi_1} \oplus \mathsf{r}) \oplus f_2(\mathrm{L}^{\pi_1} \oplus \mathsf{r}) & \in \mathsf{dom}(\pi_2) \oplus \mathsf{dom}(\pi_3) \end{array} \right\}$$

$\mathrm{S}(\mathscr{A}) \sim \mathrm{T}(\mathscr{A})$ **upto** $\mathsf{find}_{\mathsf{r}}$

As it stands in game $\mathrm{S}(\mathscr{A})$, the use of r in $\mathsf{find}_{\mathrm{L}}$ is not sufficient to allow us to bound its probability of occurring. To do so, we wish to show that it is possible to delay sampling r until the end of the game. However, r is correlated to the output of oracles S_2, S_4 and S_6. Again, we observe that r is a random value, unknown to the adversary, and independent from the permutations. Rather than using this fact as in Lemma 4.2.6 to replace the three permutations and their re-randomised version with six truly independent permutations, we simply use it to replace the three permutations with six permutations that are simply independent from the value of r.

In our formal proof, we use three pairs of incomplete functions (π_i, π'_i) that are guaranteed to never output the same result for any fresh input. We implement this by sampling each new fresh output from the uniform distribution over $\mathbb{B} \setminus (\mathsf{rng}(\pi_i) \cup \mathsf{rng}(\pi'_i))$. The games we use to formally prove that (π_1, π'_1) are equivalent to $(\pi_1, \pi_1(\cdot \oplus \mathsf{r}))$ upto the event $\mathsf{find}_{\mathsf{r}}$ are illustrated in Figure 4.10. We extend this definition to (π_2, π'_2) and (π_3, π'_3) and name $\mathrm{T}(\mathscr{A})$ the game that uses the corresponding definitions for $\mathrm{O}_{1 \leq i \leq 6}$ (which can be seen in Figure 4.8).

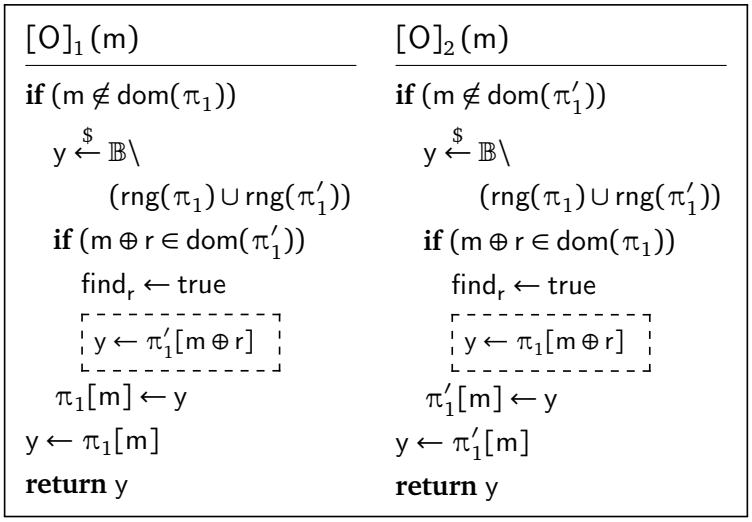


Figure 4.10 – Games S($\mathscr{A}$) (including dotted boxes) and T($\mathscr{A}$) (excluding dotted boxes).

Thanks to this more flexible goal we need only consider a smaller event, find_r, whose occurrence allows the adversary to distinguish between the S_i and the T_i. In essence, find_r corresponds in this setting to the $\mathsf{coll}_{\mathsf{rng}}$ event from the first step. With this in mind, it is easy to prove that the behaviour of S($\mathscr{A}$) and T($\mathscr{A}$) can only diverge if find_r occurs. This allows us to prove the following inequality.

$$|\Pr[\mathsf{S}(\mathscr{A}) : \mathsf{find}_\mathsf{L}] - \Pr[\mathsf{T}(\mathscr{A}) : \mathsf{find}_\mathsf{L}]| \leq \Pr[\mathsf{T}(\mathscr{A}) : \mathsf{find}_\mathsf{r}]$$

Bounding $\Pr[\mathsf{T}(\mathscr{A}) : \mathsf{find}_\mathsf{r}]$

As when bounding find_L earlier, we note that find_r is the probability of a freshly sampled value already appearing in a set. For all adversary $\mathscr{A}$ that makes at most q to O_3, O_4, O_5 and O_6, and at most σ total oracle queries, we have

$$\Pr[\mathsf{T}(\mathscr{A}) : \mathsf{find}_\mathsf{r}] \leq \frac{1}{4}\frac{(\sigma+1)^2}{2^n} + \frac{1}{2}\frac{q^2}{2^n}$$

The proof relies on the fact that, $\forall x, y, z$, if $0 \leq x + y \leq z$ then $xy \leq \frac{z^2}{4}$, which allows us to clean complex bounds.

Bounding $\Pr[\mathsf{T}(\mathscr{A}) : \mathsf{find}_\mathsf{L}]$

It now remains to bound the probability that find_L occurs in T($\mathscr{A}$). As shown in Figure 4.8, game T($\mathscr{A}$) no longer uses the value of r, and that variable can therefore be leveraged to bound the probability of find_L.

Therefore, we can use the randomness of r to bound the probability of find_L in T($\mathscr{A}$). The transformation from S($\mathscr{A}$) to T($\mathscr{A}$) have modified find_L, in the sense that every occurence of $\mathsf{dom}(\pi_i)$ is now replaced by $\mathsf{dom}(\pi_i) \cup \mathsf{dom}(\pi'_i)$, for $i \in \{1, 2, 3\}$. To simplify all this, find_L is equivalent to a disjunction of six sub-events. Denoting $\mathsf{D}_i := \mathsf{dom}(\pi_i)$ and $\mathsf{D}'_i := \mathsf{dom}(\pi'_i)$, find_L can be expressed, as it appears in T($\mathscr{A}$) as follows.

$$\mathsf{find}_{\mathrm{L}} \Leftrightarrow \bigvee \left\{ \begin{array}{ll} f_1(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_1 \oplus \mathrm{D}_2) \cup (\mathrm{D}'_1 \oplus \mathrm{D}'_2), \\ \mathrm{r} \oplus f_1(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_1 \oplus \mathrm{D}'_2) \cup (\mathrm{D}'_1 \oplus \mathrm{D}_2), \\ f_2(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_1 \oplus \mathrm{D}_3) \cup (\mathrm{D}'_1 \oplus \mathrm{D}'_3), \\ \mathrm{r} \oplus f_2(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_1 \oplus \mathrm{D}'_3) \cup (\mathrm{D}'_1 \oplus \mathrm{D}_3), \\ f_1(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) \oplus f_2(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_2 \oplus \mathrm{D}_3) \cup (\mathrm{D}'_2 \oplus \mathrm{D}'_3), \\ \mathrm{r} \oplus f_1(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) \oplus f_2(\mathrm{L}^{\pi_1} \oplus \mathrm{r}) & \in (\mathrm{D}_2 \oplus \mathrm{D}'_3) \cup (\mathrm{D}'_2 \oplus \mathrm{D}_3) \end{array} \right\}$$

Recall that f_1, f_2, $x \mapsto x \oplus f_1(x)$, $x \mapsto x \oplus f_2(x)$, $x \mapsto f_1(x) \oplus f_2(x)$ and $x \mapsto x \oplus f_1(x) \oplus f_2(x)$ are bijective. Thus, for any adversary $\mathscr{A}$ that makes at most σ total queries to its oracles, we can prove that

$$\Pr[\mathrm{T}(\mathscr{A}) : \mathsf{find}_{\mathrm{L}}] \leq \frac{3}{4} \frac{(\sigma+1)^2}{2^n}$$

This concludes the proof of Lemma 4.2.9, and we can now seek to apply it to the security of CMAC.

Lemma 4.2.10 (Indistinguishability of CMAC and FCBC)**.** *For any natural numbers q, σ, n, an adversary $\mathscr{A}$ making at most q queries, of total size in the number of blocks of at most σ, has a low probability of distinguishing CMAC from FCBC, when CMAC is parameterized by a random permutation and FCBC is parameterized by three independent random permutations.*

$$\mathsf{Adv}^{\mathsf{prf}}_{CMAC}(\mathscr{A}) \leq \mathsf{Adv}^{\mathsf{prf}}_{FCBC}(\mathscr{A}) + \frac{2(\sigma+1)^2}{2^n} + \frac{0.5q^2}{2^n} \tag{4.4}$$

Proof. This is a direct application of the definition of MOMAC and Lemma 4.2.9. In CMAC, the assumptions involving bijections are reduced in $\mathrm{GF}(2^n)$ to the fact that, for every $p \in \{\mathrm{x}, \mathrm{x} \oplus 1, \mathrm{x}^2, \mathrm{x}^2 \oplus 1, \mathrm{x}^2 \oplus \mathrm{x}, \mathrm{x}^2 \oplus \mathrm{x} \oplus 1\}$, the function $q \mapsto p \times q$ is bijective. □

Theorem 4.2.11 (Security of CMAC)**.** *For any natural numbers q, l, σ, n, an adversary $\mathscr{A}$ making at most q queries, each query of maximum size ln and of total size in the number of blocks of at most σ, has a low probability of distinguishing CMAC from a random function, when parameterized by a random permutation.*

$$\mathsf{Adv}^{\mathsf{prf}}_{CMAC}(\mathscr{A}) \leq \frac{2.5(\sigma+1)^2 + 1.5q^2 + 2q^2l^2}{2^n} \tag{4.5}$$

Proof. This is a direct application of Theorem 4.2.3, Theorem 4.2.5 and Lemma 4.2.10. □

Part III

Cryptographic Hash Function

Chapter 5

Security of a Hash Function

A *hash function* is a mathematical algorithm that maps data of *arbitrary size* (called the message) to a *fixed-size* bitstring (called a *hash value*, *hash*, or *message digest*). A *cryptographic hash function* — also called a *one-way hash function* — is an efficient deterministic hash function that is almost infeasible to invert, i.e. given a hash h, finding a message whose hash value is h would be so expensive that it would take millions of years to compute.

Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs and hope to find a match. One can use rainbow tables [1] [Oechslin, 2003] to improve brute-force techniques, but for a secure hash function, it should still be too expensive in time and memory.

Example

In a database that stores user passwords for authenticity, one can store them in plaintext. However any leak may reveal all the users' password, compromising later authentication. As a solution (simple and not sufficient), one can store the password hashes so that it should be difficult – nearly impossible – to anyone stealing the database to learn user passwords.

5.1 Security Definitions and Formalization

The ideal cryptographic hash function has the following main properties.

Determinism: The same message always results in the same hash.

Efficiency: The computation of a hash value for any given message is efficient.

Preimage resistance: It is infeasible to generate a message that yields a given hash value.

Second preimage resistance: It is infeasible to find another message that shares the same hash value as a given one.

Collision resistance: It is infeasible to find two different messages sharing the same hash.

Pseudo-random: A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.

I give the formalization of the three resistances in the next paragraphs. Then, using the definition of a random function from Section 2.3.2, I give the advantage of any adversary for each resistance against a random function, concluding that a random function is secure.

1. A rainbow table is a table that stores all computed hashes.

EasyCrypt

In the formalization of the signature of a hash function, I use an option type to express the validity of a query, as it answers None when the query is prohibited.

```
type message, hash.
module type HashFunction = {
  proc init () : unit
  proc get (m : message) : hash option
}.
```

In the formalization of the security statement in the next chapter, the number of queries an adversary can make is bounded by a fixed value, here named bound.

```
op bound : int.
axiom bound_gt0 : 0 < bound.
```

However, only the number of queries may not be enough to express more complex costs of queries, since some queries may be more costly than others in different models. Therefore, we generalize the function increase_counter that increases the counter instead of only increment it. This generalization is implemented using abstract operators in the general theory, and then, when the theory is used in a more defined model, one should clone this theory and instantiate the query cost function.

```
op increase_counter : int → message → int.
```

The module Bounder encapsulates the validity of queries in terms of the accumulated cost of queries allowed. For each query, it tests if the increased counter stays below the bound. In this case it actually increases the counter and returns the hash value, otherwise it returns the default value None.

```
module Bounder (F : HashFunction) : HashFunction = {
  var bounder : int
  proc init () : unit = { bounder ← 0; F.init(); }
  proc get(x : message) : hash option = {
    var y : hash option ← None;
    if (increase_counter bounder x ≤ bound) {
      bounder ← increase_counter bounder x;
      y ← F.get(x);
    }
    return y;
}}.
```

Preimage Advantage

EasyCrypt

```
module Preimage (A : AdvPreimage, F : HashFunction) = {
  module O = Bounder(F)
  proc main (t : hash) : bool = {
    var m, t';
    O.init();
    m ← A(O).guess(t);
    t' ← O.get(m);
```

```
    return t' = Some t;
}}.
```

The *preimage advantage* of any adversary $\mathscr{A}$ making at most bound queries against a hash function h is defined for every hash value t as:

$$\mathsf{Adv}_h^{\mathrm{p1}}(\mathscr{A}, \mathsf{t}) := \Pr[\mathsf{Preimage}(\mathscr{A}, h).\mathsf{main}(\mathsf{t}) : \mathsf{res}]$$

Second Preimage Advantage

EasyCrypt

```
module SecondPreimage (A : AdvSecondPreimage, F : HashFunction) = {
  module O = Bounder(F)
  proc main (m1 : message) : bool = {
    var m2, t1, t2;
    O.init();
    m2 ← A(O).guess(m1);
    t1 ← O.get(m1);
    t2 ← O.get(m2);
    return m1 ≠ m2 ∧ ∃ h, t1 = Some h ∧ t2 = Some h;
}}.
```

The *second preimage advantage* of any adversary $\mathscr{A}$ making at most bound queries against a hash function h is defined for all message m_1 as:

$$\mathsf{Adv}_h^{\mathrm{p2}}(\mathscr{A}, \mathsf{m}_1) := \Pr[\mathsf{SecondPreimage}(\mathscr{A}, h).\mathsf{main}(\mathsf{m}_1) : \mathsf{res}]$$

Collision Advantage

EasyCrypt

```
module Collision (A : AdvCollision, F : HashFunction) = {
  module O = Bounder(F)
  proc main () : bool = {
    var m1, m2, t1, t2;
    O.init();
    (m1,m2) ← A(O).guess();
    t1 ← O.get(m1);
    t2 ← O.get(m2);
    return m1 ≠ m2 ∧ ∃ h, t1 = Some h ∧ t2 = Some h;
}}.
```

The *collision advantage* of any adversary $\mathscr{A}$ making at most bound queries against a hash function h is defined as:

$$\mathsf{Adv}_h^{\mathrm{coll}}(\mathscr{A}) := \Pr[\mathsf{Collision}(\mathscr{A}, h).\mathsf{main}() : \mathsf{res}]$$

5.2 Random Function Security

The ideal version of a hash function is a random function, already defined in Section 2.3.2. In this section, I only give the upper bound of each advantage against a random function and do not describe the EasyCrypt formal proofs as the techniques are already described in the previous chapters.

EasyCrypt

In this section, the abstract function increase_counter is instanciated by the function that increases by 1 its first input, regardless of its second input.

```
clone Hash as MyHash with
  op increase_counter ← fun (c : int) (_ : message) ⇒ c + 1.
```

Against a random function, an adversary has a low probability to find a preimage.

Theorem 5.2.1 (Preimage resistance of a random function)**.** *For any hash value* $t \in S$*, the preimage advantage of any adversary* $\mathscr{A}$ *that queries its oracle at most* σ *times against a random function is bounded by:*

$$\mathrm{Adv}_{F}^{p1}(\mathscr{A}, t) \leq \frac{\sigma + 1}{|S|}$$

Against a random function, an adversary has a low probability to find a second preimage.

Theorem 5.2.2 (Second preimage resistance of a random function)**.** *For any message* m_1*, the second preimage advantage of any adversary* $\mathscr{A}$ *that queries its oracle at most* σ *times against a random function is bounded by:*

$$\mathrm{Adv}_{F}^{p2}(\mathscr{A}, m_1) \leq \frac{\sigma + 1}{|S|}$$

Against a random function, an adversary has a low probability to find a collision.

Theorem 5.2.3 (Collision resistance of a random function)**.** *The collision advantage of any adversary* $\mathscr{A}$ *that queries its oracle at most* σ *times against a random function is bounded by:*

$$\mathrm{Adv}_{F}^{coll}(\mathscr{A}) \leq \frac{\sigma^2 - \sigma + 2}{2\,|S|}$$

5.3 Indifferentiability, not indistinguishability

A hash function is by definition publicly known, as opposed to a MAC scheme that includes a secret key. Therefore, any hash function that is built upon a permutation should provide security (preimage, second preimage, and collision resistances) when the underlying permutation is publicly known. This is translated by giving to the adversary direct access to the permutation. Introduced by Maurer, Renner, and Holenstein in [Maurer et al., 2004], the notion of *indifferentiability* generalizes over the standard notion of indistinguishability by considering settings where the adversary has oracle access to both the construction and its underlying primitive.

It has been used as a way of reducing concerns in the design of block ciphers (with proofs for Feistel networks [Dachman-Soled et al., 2016, Dai and Steinberger, 2016b] and substitution-permutation networks [Cogliati et al., 2018]) and hash functions (with proofs for the Merkle-Damgård construction [Coron et al., 2005] and the SPONGE construction [Bertoni et al., 2008]). In each case, it formally captures the intuition that the construction does not introduce any structural vulnerabilities when the underlying primitive is seen as an ideal black-box permutation.

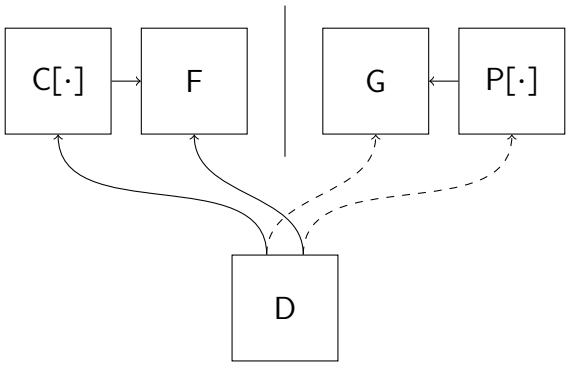


Figure 5.1 – Indifferentiability

Definition 5.3.1 (Indifferentiability [Maurer et al., 2004])**.** A construction C with oracle access to an ideal primitive F is said to be (q_D, q_S, ϵ)-indifferentiable from an ideal functionality G if there exists a simulator P with oracle access to G such that for any distinguisher D that makes queries of total cost at most q_D, and P[G] makes at most q_S queries to G, when:

$$\left|\Pr\left[b \leftarrow D^{C[F],F} : b = 1\right] - \Pr\left[b \leftarrow D^{G,P[G]} : b = 1\right]\right| < \epsilon$$

EasyCrypt

```
type state, message, thash.
module type Primitive = {
  proc init () : unit
  proc f  (x : state) : state
  proc fi (x : state) : state
}.
module type Functionality = {
  proc init () : unit
  proc f (x : message) : thash list
}.
module type Construction (F : Primitive) = Functionality.
module type Simulator (P : Functionality) = Primitive.
module type Adversary (Fun : Functionality) (Prim : Primitive) = {
  proc guess () : bool { Fun.f Prim.f Prim.fi } (* accessible oracles *)
}.
module Real (D : Adversary) (C : Construction) (F : Primitive) = {
  proc game = A(C(F),F).guess
}
module Ideal (A : Adversary) (G : Functionality) (P : Simulator) = {
  proc game = A(G,P(G)).guess
```

```
}.
```

5.4 Security from Indifferentiability

When the adversary has access to the underlying permutation, it changes the security bounds of collision resistance and first- and second-preimage resistance in the sense that the query cost is not the number of queries the adversary calls the hash function. Instead the query cost is defined as the number of primitive calls incurred by the combined queries to the construction and to the primitive itself.

EasyCrypt

In this case, the abstract function increase_counter is instanciated as follows, using an abstract operator pad, for which an instanciation is defined in Section 6.1.

```
type block, message.
op pad : message → block list.
clone Hash as MyHash with
  op increase_counter ← fun (c : int) (l : message) ⇒ c + size (pad l).
```

Theorem 5.4.1 (Preimage resistance from indifferentiability)**.** *The advantage of any adversary $\mathscr{A}$ to succeed in producing a preimage for a construction* $\mathsf{C}(\mathsf{P})$ *(over the primitive* P*) that is* $(\sigma, \sigma, \epsilon)$*-indifferentiable from a random oracle is bounded by:*

$$\mathsf{Adv}^{\mathrm{p1}}_{\mathsf{C}(\mathsf{P})}(\mathscr{A}) \leq \frac{\sigma + 1}{|S|} + \epsilon$$

Theorem 5.4.2 (Second-preimage resistance from indifferentiability)**.** *The advantage of any adversary $\mathscr{A}$ to succeed in producing a second-preimage for a construction* $\mathsf{C}(\mathsf{P})$ *(over the primitive* P*) that is* $(\sigma, \sigma, \epsilon)$*-indifferentiable from a random oracle is bounded by:*

$$\mathsf{Adv}^{\mathrm{p2}}_{\mathsf{C}(\mathsf{P})}(\mathscr{A}) \leq \frac{\sigma + 1}{|S|} + \epsilon$$

Theorem 5.4.3 (Collision resistance from Indifferentiability)**.** *The advantage of any adversary $\mathscr{A}$ to succeed in producing a collison for a construction* $\mathsf{C}(\mathsf{P})$ *(over the primitive* P*) that is* $(\sigma, \sigma, \epsilon)$*-indifferentiable from a random oracle is bounded by:*

$$\mathsf{Adv}^{\mathrm{coll}}_{\mathsf{C}(\mathsf{P})}(\mathscr{A}) \leq \frac{\sigma^2 - \sigma + 2}{2\,|S|} + \epsilon$$

Remark

The results in this chapter are part of my personal contributions and are proven generally (for all constructions and primitives).

Chapter 6

SHA3's Formal Security Proof [Almeida et al., 2019b]

Remark

This chapter's content has been published at CCS 2019 in [Almeida et al., 2019b]. I am using the text of our joint work with their agreement. My contributions will be made clear when the description matches the boundaries of my personal contributions.

A stated goal of recent competitions for cryptographic standards is to gain trust from the broad cryptographic community through open and transparent processes. These processes generally involve open-source reference and optimized implementations for performance evaluation, rigorous security analysis for provable security evaluation, and, often, informal evaluation of security against side-channel attacks.

These artifacts contribute to building trust in candidates, and ultimately in the new standard. However, the disconnect between implementation and security analyses is a major cause of concern. Our paper [Almeida et al., 2019b] explores how formal approaches could eliminate this disconnect and bring together implementation (most importantly, efficient implementations) and software artifacts, in particular machine-checked proofs, supporting security analyses. We put forward four desirable properties for formal approaches:

— *functional correctness*: efficient implementations should be proved equivalent to reference implementations and to algorithmic specifications of the standardized cryptographic construction that are both human-readable and interpreted by machines. Such specifications and implementations should be *proved* to have the same input/output behavior (or interactive behavior in protocols);

— *provable security*: rigorous security proofs should be provided both for algorithms and for implementations. For the highest level of assurance, security proofs should be machine-checked and establish guarantees for the (machine-readable) algorithmic specifications. Security for both efficient and reference implementations will follow from the functional correctness proofs, using the baseline adversarial models from provable security;

— *side-channel resistance*: implementations should be provably secure against side-channel attacks, in relevant ideal models. For instance, it is commonly required that implementations are secure in an abstract model of timing, where implementations leak secrets if they contain secret dependent memory-accesses or control-flow instructions, a notion known as "cryptographic constant-time". Combined with provable security,

it entails security in a stronger adversarial model where side-channel leakage is available to the adversary;

— *efficiency*: formal proofs should remain fully compatible with efficiency considerations. They should neither constrain in any way the code of the implementations (although constrained intermediate implementations could be used as proof artifacts) nor impact its performance.

We demonstrate through a relevant use case of the feasibility of formal approaches concerning the stated goals of functional correctness, provable security, side-channel resistance, and efficiency. Our use case is the SHA3 standard [Dworkin, 2015], the last member of the *Secure Hash Algorithm* family released by NIST in 2015. Our choice is guided by two main considerations. Firstly, the SHA3 standard will likely be used to protect real-world applications for many years to come. Secondly, its security proof is intricate and involves techniques that are not routinely addressed in machine-checked security proofs.

Concretely, our implementation is written in Jasmin (described in [Almeida et al., 2017], with another use case in [Almeida et al., 2019a]), a framework that targets high-assurance and high-speed implementations using "assembly in the head" (a mixture of high-level and low-level, platform-specific programming) and a formally verified and predictable compiler which empowers programmers to write highly efficient fine-tuned code. The generated (verified) `x86-64` assembly code matches in performance the best available implementations for this primitive, including for example a current OpenSSL version. Machine-checked proofs of equivalence and provable security are developed in EasyCrypt, using the embedding developed in [Almeida et al., 2019a]. More precisely, as illustrated in Figure 6.1, we establish:

— *functional correctness*: the highly efficient implementations are proved functionally equivalent to a readable Jasmin reference implementation of the SHA3 standard;

— *provable security*: we prove that the SPONGE [1] construction is indifferentiable from a random oracle when the underlying permutation is modeled as a random object. From this result we derive concrete bounds for the standard notions of collision-resistance and resistance against first- and second-preimage attacks in the random permutation model;

— *side-channel resistance*: we prove that the implementation only leaks the length of public data, in the abstract model of timing used to reason about "cryptographic constant-time". This property is useful when the hash function is integrated into higher-level primitives, say key-derivation functions, when hashed inputs are secret.

Our results are established at different levels. Our provable security analysis is based on an EasyCrypt model of the SPONGE construction, which matches the (bit-oriented) specification in the SHA3 standard. At this level, we adopt the standard approach for cryptographic proofs of indifferentiability and treat the underlying permutation as an ideal object. In contrast, constant-time security is therefore established as close to the computational platform as possible. Our analysis of potential timing side-channels is carried out over highly optimized (byte- and word-oriented) Jasmin implementations of SHA3.

We then use automatic extraction and equivalence proofs in EasyCrypt to bridge these two levels of results. First, our optimized Jasmin implementations are proved equivalent to a readable reference implementation of the standard, which includes the SHA3 permutation. Finally, we also prove that the model of the SPONGE that proved theoretically secure is

1. The construction is described in more details in Section 6.1.

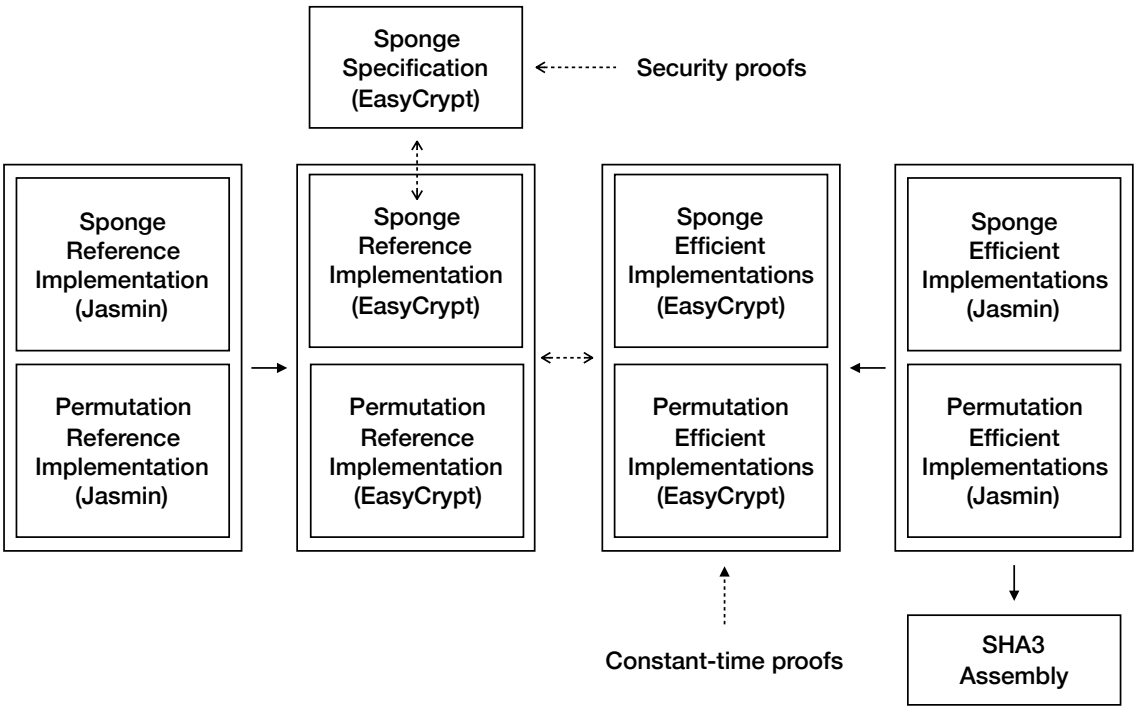


Figure 6.1 – Our results. Full lines represent extraction to EasyCrypt and compilation to assembly by the Jasmin compiler. Dashed lines represent equivalence and security proofs, formalized in EasyCrypt.

functionally equivalent to the Jasmin reference implementation of this construction when instantiated with the same permutation. This establishes a link between theoretical security and implementation security.

Remark

All proof and implementation artifacts are available from `https://gitlab.com/easycrypt/sha3`. The `README.md` file contains therein further points to be relevant checking extraction and compilation tools and gives light instructions on how to use them to check the proofs and compile the code.

Remark

My personal contribution is situated in the provable security level. It involves the indifferentiability proof and the proof that indifferentiability implies collision-resistance and resistance against first- and second-preimage attacks.

This manuscript will stop describing the Jasmin implementation, but anyone interested in this part are welcomed to read the original paper [Almeida et al., 2019b].

In section 6.1, I describe the SPONGE construction and the SHA3 functions. Then, in section 6.2 I present the security statement for which I explain in section 6.3 the structure of the security proof by introducing a general approach to the decomposition of indifferentiability games. In section 6.3.2, section 6.3.3 and section 6.3.4, I detail the formalization of the first, second and third layers of the indifferentiability proof.

6.1 The SPONGE construction

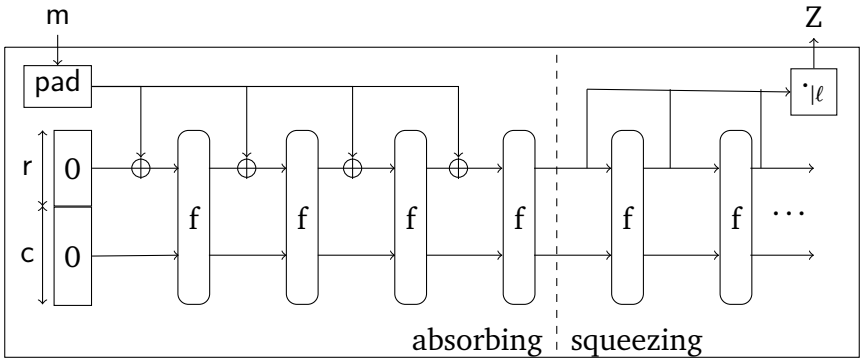


Figure 6.2 – Representation of $\text{SPONGE}_c[f, \text{pad}, r](\text{m}, \ell)$

$\text{SPONGE}_c[f, \text{pad}, r](\text{m}, \ell)$

```
1 :  m_0||...||m_{m-1} ← m||pad(r, |m|);
2 :     // absorption phase
3 :  s_a||s_c ← 0^r||0^c;
4 :  for i = 0...m−1 do
5 :     s_a||s_c ← f((s_a ⊕ m_i)||s_c);
6 :     // squeezing phase
7 :  Z ← ε; done ← false;
8 :  while ¬done
9 :     Z ← Z||s_a;
10 :    if |Z| < ℓ
11 :       s_a||s_c ← f(s_a||s_c);
12 :    else
13 :       done ← true;
14 :  return Z_{|ℓ};
```

Figure 6.3 – Pseudocode for the SPONGE construction [**?**]

The SHA3 standard defines a family of 4 hash functions and 2 extendable-output functions (XOFs). All functions rely on a generic construction, called the SPONGE, that is based on a fixed (unkeyed) permutation. The standard also defines modularly a permutation algorithm — KECCAK-$p[1600, 24]$ — which operates over a 1600–bit-wide state and is defined as an approved usable in other standards. In the following, we use the notation KECCAK-p as shorthand for this permutation.

Representation and pseudocode for the SPONGE construction are respectively shown in Figure 6.2 and 6.3. It is parametrised by: i. the permutation f, ii. the padding algorithm pad, and iii. the *rate* (or block size) r. I write c for the construction's *capacity*, defined as the permutation's bitwidth (1600 in the standard) minus r. The construction's internal state $\text{s}_a||\text{s}_c$ has two parts: s_a (r bits) and s_c (c bits). On input of a bitstring m, the SPONGE

construction pads it to a multiple of the block size and breaks it into blocks (Line 1). The padding scheme must be injective and length-regular (both properties are necessary in a padding scheme used in secure cryptographic hashing) and must also guarantee that no padded input ends with an all-zero block (which is a necessary condition for the security of the SPONGE). The padded input is then absorbed block-by-block into the SPONGE's internal state (initialized to all 0 bits on Line 3) by interleaving the addition of blocks into the state with applications of the permutation (Lines 4-5). Once all input blocks have been absorbed, the permutation is used, again, to extract the output by blocks of size r (Lines 7-13), truncating the final block to the requested ℓ bits (Line 14). [2]

In the description and formal treatment, I abstract the permutation's bitwidth (set to 1600 by the standard) to some positive integer b, refining it only in the final steps of the proof. Thus $r + c = b$. s_c is the part of the internal state that is not exposed to or controlled by the adversary. For a fixed state width, the capacity serves as the main security parameter for the SPONGE construction, and the rate as its main performance parameter. Therefore, as illustrated in Figure 6.3, I often use c and r to specify a particular SPONGE construction, rather than b and r.

The standard defines SHA3-224, SHA3-256, SHA3-384 and SHA3-512 — collectively referred to as SHA3-x in the following, as approved hash functions that accept any bitstring as input, and deterministically produce a fixed-length hash (of length x, for SHA3-x). For a fixed output length x, these functions instantiate the SPONGE construction with KECCAK-p, fix $r = 1600 - 2 \cdot x$ and use the *multi-rate padding* scheme pad10*1 defined as

$$\mathsf{pad10^*1}(r, \ell) := 1 \| 0^{(-\ell-2) \mod r} \| 1 \,.$$

The pad10*1 scheme simply appends a string composed of two 1 bits around as many 0 bits as necessary (from 0 to $r-1$) to the message. It is easy to see that it satisfies the properties required by the SPONGE construction. Formally, the SHA3-x functions are defined as:

$$\text{SHA3-}x(\mathsf{m}) = \text{SPONGE}_{2 \cdot x}[\text{KECCAK-}p, \mathsf{pad10^*1}, 1600 - 2 \cdot x](\mathsf{m} \| 01, x) \,,$$

where 01 denote two domain separation bits.

The SHA3 standard also defines two XOFs, SHAKE128 and SHAKE256 — collectively referred to as SHAKEx, that accept arbitrary bitstrings as input, and produce a caller-chosen-length prefix of an infinite bitstream deterministically defined by the input. On input a bitstring m and an output length d, the SHAKEx XOFs are defined as

$$\text{SHAKE}x(\mathsf{m}, d) = \text{SPONGE}_{2 \cdot x}[\text{KECCAK-}p, \mathsf{pad10^*1}, 1600 - 2 \cdot x](\mathsf{m} \| \mathtt{ss} \| 11, d) \,,$$

where ss denote two suffix bits and 11 denote two domain separation bits. The standard introduces suffix bits for future compatibility with coding schemes for tree hashing variants of the SHAKEx functions. They have no effect on security; in fact, our security proof applies for arbitrary application suffixes (of any length fixed in advance).

6.2 Security Statement

The SPONGE construction satisfies the strong security notion known as indifferentiability from a (extendable output) random oracle, defined in Section 5.3.

2. Figure 6.3 is as close to the standard as it could be made with a structured programming language. This specification differs slightly in that there is no squeezing at all when 0 bits of output are requested. Both specifications are proven functionnally equivalent.

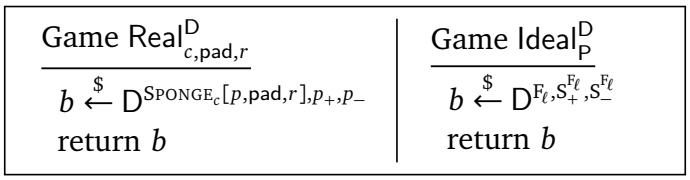


Figure 6.4 – Games defining the indifferentiability of the SPONGE construction from an (extendable output) RO.

For concreteness, we give the real experiment and ideal experiments in Figure 6.4 when the notion of indifferentiability is applied to the SPONGE construction, as used in the SHA3 standard and formalized in our proof. In the real game, p is a permutation sampled uniformly at random from the set of all permutations over bit strings of length 1600. The distinguisher is given access to oracles p_+ and p_- that allow it to query the permutation backwards as well as forwards. In the ideal game, the simulator $\mathsf{P} = (\mathsf{S}_+, \mathsf{S}_-)$ must fake the outputs of the p_+ and p_- oracles, while oblivious of the calls that the distinguisher places to the construction (which is replaced by a random object in the ideal world). We show the simulator as two different algorithms for clarity, but we allow them to share state. The ideal functionality is an extendable output random oracle. This is implemented as an *infinite random oracle* F that associates to each input an infinite (lazy) bitstring, each element of which is sampled uniformly at random. The distinguisher and simulator are restricted to queries to the ideal functionality of the form (m, ℓ), matching the syntax of the SPONGE interface; these queries return prefixes of size ℓ of the random oracle F outputs (denoted using F_ℓ notation in the security games). In our formalization we consider a random function $f \in \{0,1\}^* \times \mathbb{N} \to \{0,1\}$ and construct the observable prefix of length ℓ of the infinite random oracle F as follows:

$$\mathrm{F}(\mathsf{m}, \ell) = f(\mathsf{m}, 0) \| f(\mathsf{m}, 1) \| \ldots \| f(\mathsf{m}, \ell - 1)$$

We actually implement f lazily: representing it as a finite map from $\{0,1\}^* \times \mathbb{N}$ to $\{0,1\}$ to which we add new input/output pairs as needed.

Our machine-checked proof establishes the following security result for the EasyCrypt specification of the SPONGE, which corresponds to the pseudo code described in Figure 6.3.

Theorem 6.2.1 (Indifferentiability of SPONGE). *The* SPONGE *construction is* $(\sigma, \sigma, \frac{\sigma^2-\sigma}{2^{b+1}} + \frac{\sigma^2}{2^{b-r-2}})$*-indifferentiable from an extendable output random oracle for any* $\sigma < 2^{b-r}$*. Namely, the simulator* SIMULATOR *exhibited in Figure 6.13 makes at most* σ *queries when the adversary makes queries of total cost at most* σ*.*

$$\left| \Pr\left[\mathsf{Real}^{\mathsf{D}}_{b-r,\mathsf{pad},r} = 1\right] - \Pr\left[\mathsf{Ideal}^{\mathsf{D}}_{\mathsf{P}} = 1\right] \right| \leq \frac{\sigma^2}{2^{b-r-2}} + \frac{\sigma^2 - \sigma}{2^{b+1}}$$

6.3 Layered Indifferentiability

Our formalization of definitions and top-level statements is direct: we simply express the SPONGE, its ideal functionality, and the simulator, as the indifferentiability result in EasyCrypt is already stated in Theorem 6.2.1.

In order to carry out the proof, we layer it as shown in Figure 6.5 to account for individual aspects of the construction. Our layers separately deal with truncation and padding (SPONGE), and squeezing (BLOCKSPONGE) over a simplified CORESPONGE which outputs

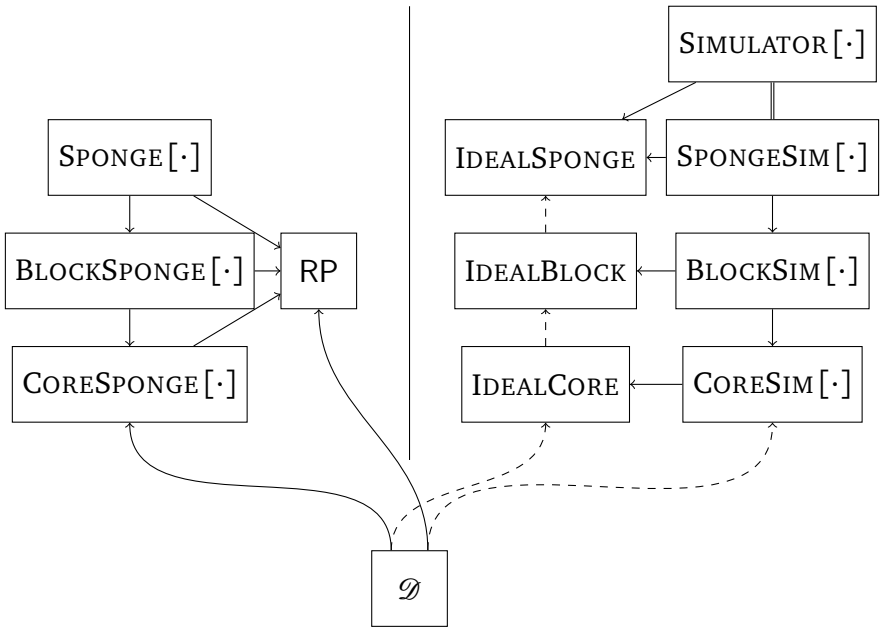


Figure 6.5 – A layered proof for the SPONGE construction.

only a single block. The simulator constructed by layers is not optimal in terms of query cost, and we then show its equivalence to the top-level simulator, allowing us to conclude with a tighter concrete security result.

Beyond the proof of indifferentiability for the bare SPONGE construction, we then show that the functions defined in the SHA3 standard [**?**] inherit specialized version of this indifferentiability, and formally establish concrete security bounds on an adversary's ability to produce collisions, preimages or second preimages on the SHA3 hash functions without first breaking the security assumption on the KECCAK-*p* permutation. At the highest level, our indifferentiability proof is instantiated to the hash functions SHA3-224, SHA3-256, SHA3-384 and SHA3-512, and extendable output functions SHAKE128 and SHAKE256 *as they are defined in the SHA3 standard* [**?**].

6.3.1 General Decomposition

The decomposition outlined in Figure 6.5 is made possible by the following general observation. Suppose we have a stateless "upper-level" construction $\mathsf{C}[\mathsf{RP}]$ that we want to prove to be indifferentiable from an upper-level ideal functionality J. Furthermore, let us assume that we already know that a stateless "lower-level" construction $\mathsf{E}[\mathsf{RP}]$ is indifferentiable from a lower-level ideal functionality I, where S is a lower-level simulator such that no adversary can effectively distinguish between $(\mathsf{E}[\mathsf{RP}], \mathsf{RP})$ and $(\mathsf{I}, \mathsf{S}[\mathsf{I}])$.

We construct a pair of stateless converters D and U that work as follows: D ("down") transforms an upper-level functionality into a lower-level one; and U ("up") transforms a lower-level functionality into a upper-level one. We define the upper-level simulator T such that $\mathsf{T}[\mathsf{J}] := \mathsf{S}[\mathsf{D}[\mathsf{J}]]$. And, for any upper-level adversary $\mathscr{A}$ that is asked to differentiate C from J, let the lower-level adversary $\mathscr{B}[\mathscr{A}]$ be defined as $\mathscr{B}[\mathscr{A}]^{\mathsf{X},\mathsf{Y}} := \mathscr{A}^{\mathsf{U}[\mathsf{X}],\mathsf{Y}}$.

Then, to prove that $\mathsf{C}[\mathsf{RP}]$ is indifferentiable from J, it will suffice to show the following

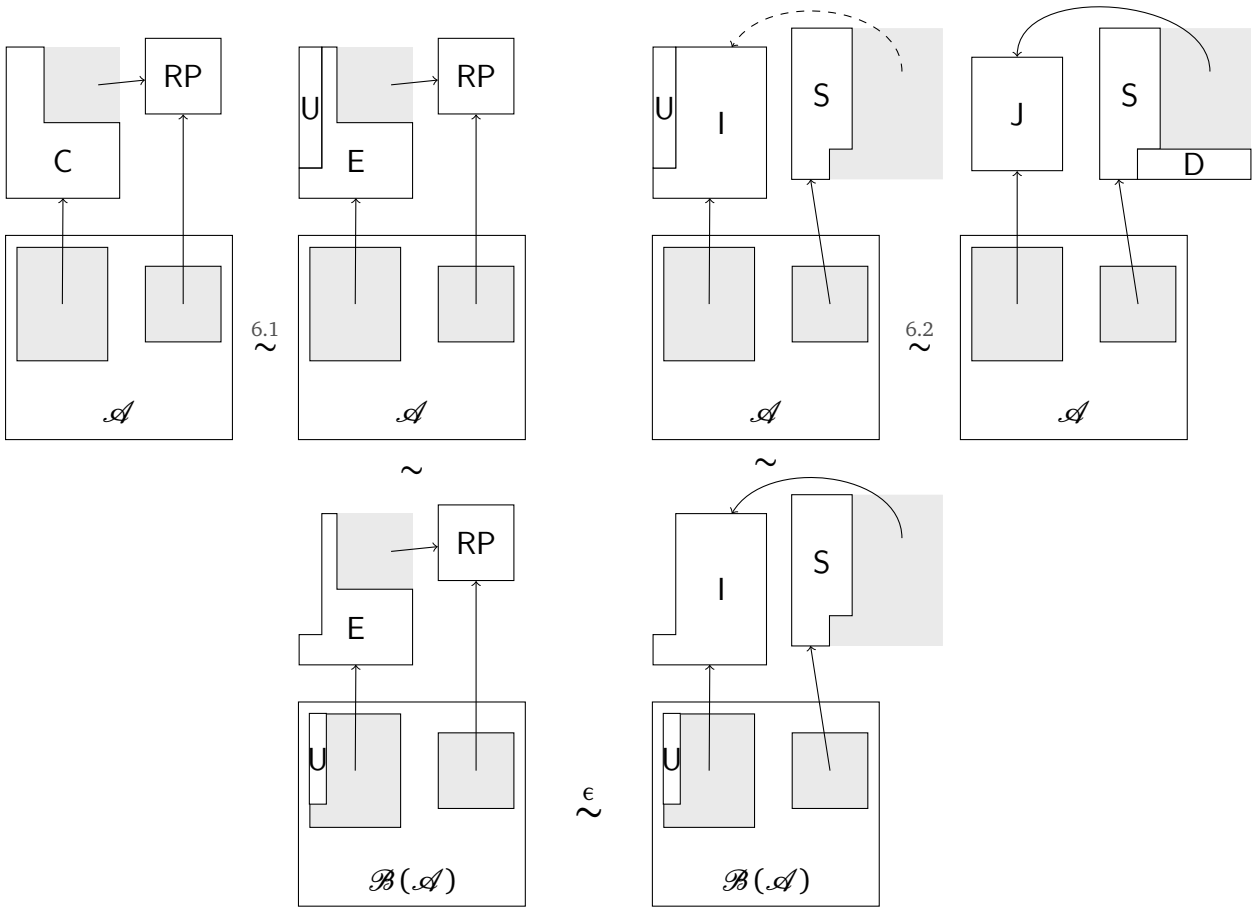


Figure 6.6 – General decomposition argument

two equivalences, as represented in Figure 6.6.

$$\mathscr{A}^{\mathsf{C}[\mathsf{RP}],\mathsf{RP}} \sim \mathscr{B}[\mathscr{A}]^{\mathsf{E}[\mathsf{RP}],\mathsf{RP}} \tag{6.1}$$

$$\mathscr{A}^{\mathsf{J},\mathsf{T}[\mathsf{J}]} \sim \mathscr{B}[\mathscr{A}]^{\mathsf{I},\mathsf{S}[\mathsf{I}]} \tag{6.2}$$

Equivalence (6.1) relates the "real" games, and simply reflects that the modular construction is correct. Equivalence (6.2), on the other hand, pertains to the "ideal" games. Since E is indifferentiable from I for all adversaries, this holds in particular for $\mathscr{B}[\mathscr{A}]$.

Because C and E (and U) are stateless, it is clear what $\mathsf{C}[\mathsf{RP}] \sim \mathsf{U}[\mathsf{E}[\mathsf{RP}]]$ should mean, and that it will be sufficient to imply that Equivalence (6.1) holds.

However the situation is more complex for the ideal equivalence (6.2), since our ideal functionalities have persistent local state (say, query maps) of *different* types. Consequently it is unclear what the statements $\mathsf{J} \sim \mathsf{U}[\mathsf{I}]$ and $\mathsf{I} \sim \mathsf{D}[\mathsf{J}]$ would even mean, and we must instead prove finer-grained equivalence statements where equivalence—over the whole game—also defines how the ideal functionalities' states are related.

6.3.2 First Layer : SPONGE (not part of my contributions)

The BLOCKSPONGE construction is similar to SPONGE, but works on blocks rather than bits, forgoing padding of inputs and truncation of outputs. The IDEALBLOCK ideal functionality is like the infinite random oracle IDEALSPONGE, except that it, too, works on blocks

rather than bits. Both the construction and ideal functionality should only be called with lists of blocks that can be successfully unpadded; when called with invalid arguments, they return the empty list.

We prove that, if BLOCKSPONGE is indifferentiable from IDEALBLOCK, then SPONGE is indifferentiable from IDEALSPONGE by instantiating the generic argument discussed in Section 6.3.1. In this instance, C is SPONGE, E is BLOCKSPONGE, J is IDEALSPONGE, I is IDEALBLOCK, and S is the block sponge simulator. We construct the transformers D and U as follows:

— D[J′] takes in a list of blocks and a requested length n. If given an input that is not correctly padded, it returns the empty list. Otherwise, it calls J′ with the unpadded input and $n*r$, and then chunks the resulting bitstring into n blocks. It is represented in Figure 6.7.

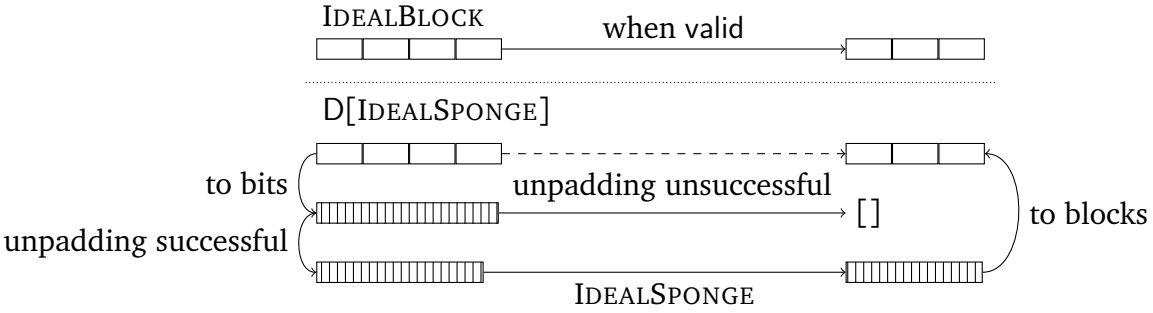


Figure 6.7 – Representation of the construction D with message input of size 4 blocks and asked for an ouput of size 3 blocks.

— U[I′] takes in a list of bits and a requested length n. It calls I′ on the padding of its input and $\lceil n/r \rceil$, and then truncates the result of turning the resulting blocks into a list of bits. It is represented in Figure 6.8.

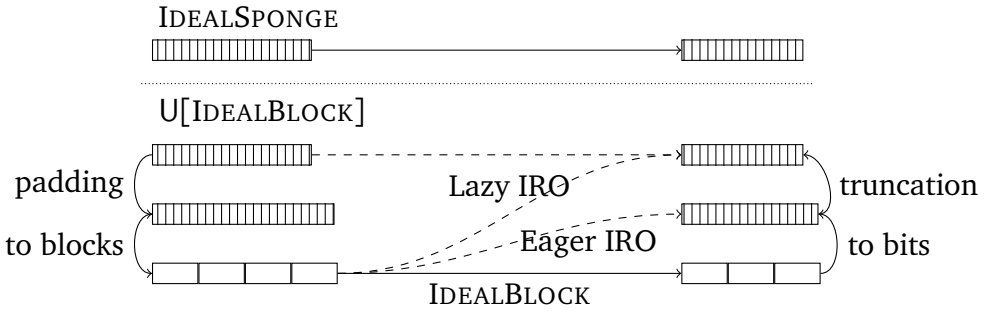


Figure 6.8 – Representation of the construction U with message input of size m and asked for an ouput of size ℓ such that $3r \leq m < 4r$ and $2r < \ell \leq 3r$.

For the real equivalence $\mathscr{A}^{\mathsf{C[RP]},\mathsf{RP}} \sim \mathscr{B}[\mathscr{A}]^{\mathsf{E[RP]},\mathsf{RP}}$, we simply prove that the construction and its modularly-constructed version are equivalent, as C[RP] ∼ U[E[RP]]. This involves inlining and code rewriting, noting that U simply truncates and pads exactly as the SPONGE does.

In contrast, and as discussed in Section 6.3.1, the proof of the ideal equivalence $\mathscr{A}^{\mathsf{J},\mathsf{T[J]}} \sim \mathscr{B}[\mathscr{A}]^{\mathsf{I},\mathsf{S[I]}}$ is more complex. We carry it out in three steps, involving *hybrid infinite random*

oracles (*hybrid IROs*), which are midway between J and I. An input to a hybrid IRO is a well-padded list of *blocks* and a desired number of output *bits*. Internally, they work with finite maps from $(\{0,1\}^r)^* \times \mathbb{N}$ to $\{0,1\}$. A hybrid IRO can be raised, for comparison with J, or lowered, for comparison with I. Two hybrid IROs are defined: a *lazy* one, and an *eager* one. The lazy one consults/updates just enough inputs of its finite map to provide the requested bits, whereas the eager one continues up to a block boundary, consulting/updating subsequent inputs of the finite map, as if it had been asked for a multiple of r bits.

The first step of the proof transitions from $\mathscr{A}^{\mathsf{J},\mathsf{T}[\mathsf{J}]}$ to a game involving the lazy IRO. This is done by employing a relational invariant between the maps of J and the lazy IRO.

The second step of the proof uses the eager sampling facility of Section 2.3.2 to transition from a game involving the lazy IRO to one involving the eager IRO. A graphical representation is displayed in Figure 6.8. The bridge between these games uses the eager sampling theory's `sample` oracle to sample the extra bits needed to take one up to a block boundary. The lazy version of `sample` then gives us the lazy IRO, whereas its eager version gives us the eager IRO.

The third step of the proof takes us from the game involving the eager IRO to $\mathscr{B}[\mathscr{A}]^{\mathsf{I},\mathsf{S}[\mathsf{I}]}$. This is done by employing an invariant between the maps of the eager IRO and I. The proof is rather involved, and makes use of: (a) EasyCrypt's library's support for showing the equivalence of randomly choosing a block versus forming a block out of r randomly chosen bits; and (b) a mathematical induction over a pRHL judgement.

6.3.3 Second Layer : BLOCKSPONGE

The next step in our proof is to show that squeezing, the operation through which the SPONGE's output is extended to any desired length, also preserves indifferentiability. Consider a functionality CORESPONGE that computes only the absorption stage of BLOCKSPONGE (lines 3-5 of Figure 6.3, taking as input a list of blocks, and outputting the final value of s_a).

We prove that, if CORESPONGE is indifferentiable from IDEALCORE, then BLOCKSPONGE is indifferentiable from IDEALBLOCK by instantiating the generic argument discussed in Section 6.3.1. In this instance, C is BLOCKSPONGE, E is CORESPONGE, J is IDEALBLOCK, I is IDEALCORE, and S is CORESIM. We construct the transformers D and U as follows:

— D is renamed as SQUEEZE. SQUEEZE[J′] takes a list of blocks m (a padded bitstring) and a desired output length ℓ (in blocks). It iteratively calls J′ ℓ times, with inputs $(\mathsf{m} \,\|\, 0^{r \cdot j})_{0 \le j < \ell}$, each call producing a single block. It outputs the list of all these blocks in order, as shown in Figure 6.9.

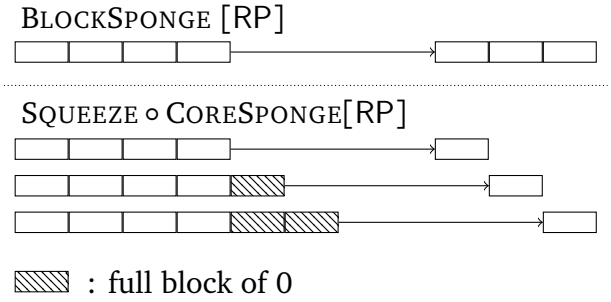


Figure 6.9 – BLOCKSPONGE [RP] and SQUEEZE ∘ CORESPONGE[RP] on inputs that are a list of 4 blocks and output size 3.

— U is renamed as LAST. LAST[I′] takes in a list of blocks and outputs a single block. It parses the list of blocks into a pair of a valid list of blocks — that terminates with a non full-0 block — and an output size. It then calls I′ on this pair, obtaining a list of blocks from which it outputs only the last block, as shown in Figure 6.10.

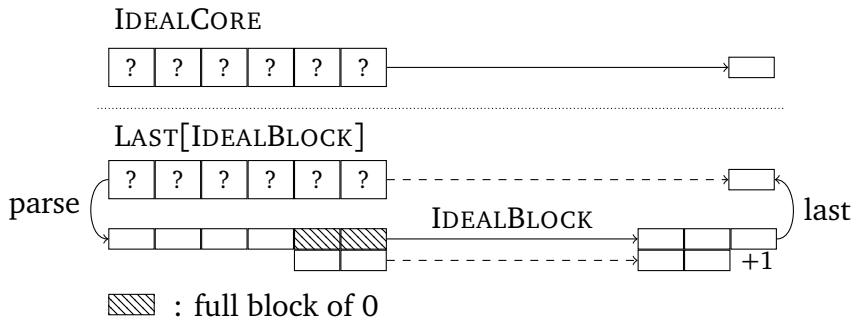


Figure 6.10 – IDEALCORE and LAST[IDEALBLOCK] on input that is a 6-blocks list whose last two blocks are full of 0.

For the real equivalence SQUEEZE ∘ CORESPONGE[RP] ∼ BLOCKSPONGE[RP], since the primitive RP appears deterministic (and in particular returns the same output when queried twice on the same input) it is easy to prove in EasyCrypt that this equivalence holds.

As before, the ideal equivalence here must be expressed and proved over the whole game, and indeed requires relating the states of ideal functionalities and simulator based on queries done on both interfaces. We thus prove in EasyCrypt for all adversaries $\mathscr{A}$

$$\mathscr{A}^{\text{SQUEEZE[IDEALCORE]},\ \text{CORESIM[IDEALCORE]}} \sim \mathscr{A}^{\text{IDEALBLOCK},\ \text{CORESIM[LAST}\circ\text{IDEALBLOCK]}}$$

Once again, this proof leverages the generic lazy-eager sampling theory described in Section 2.3.2 to prove that the intermediate blocks sampled and thrown away by LAST[IDEALBLOCK] can instead not be sampled at all.

On the query cost

In this particular application of the decomposition, the cost in number of permutation queries differs greatly between BLOCKSPONGE[RP] and SQUEEZE ∘ CORESPONGE[RP], resulting in a lost in the security bound. Considering the example shown in Figure 6.9, the cost of that query to BLOCKSPONGE is 6 permutation calls, but the same query to SQUEEZE ∘ CORESPONGE costs $4+5+6=15$ permutation calls. Simply applying the decomposition as described would yield a bound dominated by $O\left(\sigma^4/2^r\right)$.

We therefore also use this transfer result to refine the way in which the cost of a query in CORESPONGE is measured, to avoid double-counting common prefixes. This is done through a construction named PREFIXES applied to BLOCKSPONGE [RP] and CORESPONGE [RP] that stores all resulting states of the absorption phase from all queries's prefixes. For any fresh query, it takes the longest common prefix with all previous queries, and starts with the stored state for this prefix. That way, all calls to RP that follow the same prefix as a previous query's are avoided. Then, it continues squeezing with RP for the rest of the query's blocks.

6.3.4 Core Layer : CoreSponge

All proof steps discussed thus far involve transferring indifferentiability from a simple construction to a more complex one. We now focus on the core of the Sponge construction, a block-oriented construction that, on input a list of blocks, produces a single block of output.

As discussed above, our notion of query cost on this CoreSponge construction differs from the top-level notion, and is defined instead as the length of the query without its longest common prefix with any of the previous queries. We call this cost the *prefix cost* of a query. Considering this notion of cost requires us to carefully design the CoreSponge by integrating the construction Prefixes to ensure that the prefix cost and query cost of the same query align when transferring indifferentiability to BlockSponge. We prove in EasyCrypt that CoreSponge is indifferentiable from IdealCore.

As a first simplifying step, we replace the primitive, a random permutation, with a random function. This simplifies some of the formal reasoning—in particular by removing internal dependencies between samplings in the random permutation—but introduces an additional term $\frac{\sigma^2-\sigma}{2^{b+1}}$ in the bound. We note that all results discussed above transfer indifferentiability *without loss*. Therefore, improving the bound for this simpler functionality would be sufficient in improving the bound for the whole Sponge construction at almost no formal cost beyond that of tightening the bound for CoreSponge.

The idea behind the simulator

Indifferentiability requires us to simulate answers to the permutation, in a way that is consistent with what the adversary may have already observed of the ideal functionality, without knowing which queries the distinguisher has made—or will make—to the functionality. To do so, the simulator CoreSim, shown in Figure 6.11, keeps track of *paths*—which

CoreSim$[\mathscr{F}](x_1, x_2)$

```
1 : if (x1, x2) ∉ m {
2 :   y2 ←$ Z_2^{b−r};
3 :   if x2 ∈ paths {
4 :     (p, v) ← paths[x2];
5 :     y1 ← F(p ||(v ⊕ x1));
6 :     paths[y2] ← (p ||(v ⊕ x1), y1);
7 :   } else {
8 :     y1 ←$ Z_2^r;
9 :   }
10 :  m[(x1, x2)] ← (y1, y2);
11 :  mi[(y1, y2)] ← (x1, x2);
12 : }
13 : return m[(x1, x2)];
```

CoreSim$[\mathscr{F}]^{-1}(x_1, x_2)$

```
if (x1, x2) ∉ mi {
  y1 ←$ Z_2^r;
  y2 ←$ Z_2^{b−r};
  mi[(x1, x2)] ← (y1, y2);
  m[(y1, y2)] ← (x1, x2);
}
return mi[(x1, x2)];
```

Figure 6.11 – The core simulator CoreSim

are sequences of blocks that, when fed through CoreSponge, leave its state with a particular value of the capacity—and uses the functionality to simulate its answer to any query

that makes use of a capacity to which a path is known, that is, a query that extends a known path through CORESPONGE. Queries that are disjoint from path are answered as if by the ideal random function.

When the simulation fails

The indifferentiability between the real game and the ideal game mainly relies on the unicity of capacities that appears in the different absorbtion phases. Therefore, when an event occurs that contradicts this unicity, the simlation fails. More specifically, it fails (and can be distinguished from the true permutation) if either one of the following events occurs:

bcol **:** A capacity that was previously seen as output of the permutation with a block s_a has been output again, associated with a different block s'_a.

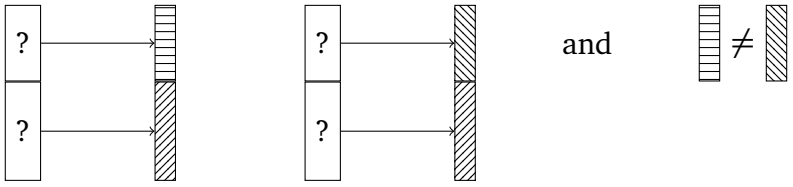


Figure 6.12 – Representation of bcol.

bext **:** The adversary has queried the primitive or its inverse with a capacity that has already been, or is later sampled internally by CORESPONGE, but to which the simulator does not know a path. In other words, it guessed a value of a hidden capacity, i.e. that was or will be an intermediate capacity in an absorbtion phase.

We show in EasyCrypt that these are the only conditions under which the distinguisher can indeed distinguish the construction from the ideal functionality.

Bounding the probability of a simulation failure

It remains to bound the probability that these bad events occur. The collision event bcol is straightforward since it occurs as values are sampled and its probability can be bounded immediately.

On the other hand, bounding the probability that bext occurs in any particular run is much more complex, as it requires identifying that the adversary has guessed a previously sampled value, or will be sampled later on in the run. We note, however, that the event is only triggered when the adversary guesses a value that is independent from its view of the system: indeed CORESPONGE keeps all capacities internal, and the event does not consider the case where the adversary has obtained the capacity's value through a legitimate sequence of calls to the permutation that mimics CORESPONGE's operations. The trick is therefore to delay the sampling of capacities that are used in CORESPONGE until the end of the game (at which point we can sample them all, and easily bound the probability that any one of them was used by the adversary), or until the simulator constructs a path to it, at which point bext is no longer triggered.

In order to deploy the lazy-eager sampling theory described in Section 2.3.2, we must first remove the dependencies between capacities and rates introduced by their joint use in the permutation.

A proof trick: indirection

We deploy an indirection technique similar to that used in [Backes et al., 2012] that proves indifferentiability of Merkle-Damgård. Since only the unicity of capacities is needed in the indifferentiability rather than their value, we deploy an isomorphism (that stays an isomorphism if no failure event happens) that associates a counter (with type handle in Section 6.4) to each capacity.

Concretely, each fresh permutation query (made by the adversary or the functionality) is tagged with its indirection number. The main permutation map (G1.mh) is used to keep track, given an input state, of the rate that was returned, and of the indirection number. An auxiliary map is used to translate indirection numbers into capacities (FRO.m).

This auxiliary map can then be sampled lazily: on direct permutation queries, both rate and capacities are sampled and returned to the distinguisher; on permutation queries made by the construction, the rate is sampled and associated with an indirection number, but the capacity is not sampled. A loop after the distinguisher has finished running samples all remaining capacities — that is, exactly those that have been used in the construction, but not been observed as part of a path — triggering the bad event *a posteriori* if any one of them collides with an adversarial input that is not part of a path. This last transformation makes heavy use of the lazy-eager sampling theory described in Section 2.3.2, including in particular the use of programming queries in some cases.

6.3.5 The final SPONGE simulator

As described in Figure 6.5, the simulator resulting from the proof described above is constructed from the modular layers as:

$$\text{SPONGESIM}[\text{BLOCKSIM}[\text{CORESIM}[\cdot]]]$$

The final step of our proof is to collapse the layered construction into a final simulator, shown in Figure 6.13. This allows us to reduce the cost of simulator queries, aligning them with the announced notion of query cost, and to present the simulator as a single algorithm. However, the layered nature of the simulator can still be seen in this final presentation: the core simulator still appears, keeping track of paths through the simulated permutation that could correspond to functionality calls and extending them as required, or simply simulating the random permutation with a random function. However, the way in which paths are extended in the layered simulator (lines 5-16 in Figure 6.13 replacing the single line 5 in Figure 6.11) reflects the different layers required to turn a path through the core sponge (a well-padded bitstring followed by a number of 0^r blocks) into a valid query to the SPONGE (a bitstring that the SPONGE itself pads and a number of desired output bits), also turning the output of the SPONGE (a list of bits of the requested length) into the expected output for the core simulator (a single block). For top-level simulator queries that would yield invalid queries to any intermediate functionalities (for example because they correspond to paths that do not reflect well-padded inputs), the layered simulator simply simulates an independent random function.

In other words, our final simulator is CORESIM where the ideal functionality IDEALCORE itself is further simulated from the top-level IDEALSPONGE ideal functionality.

SIMULATOR$[\mathscr{F}](x_1, x_2)$

```
1 : if (x1, x2) ∉ m {
2 :   y2 ←$ {0,1}^(b−r);
3 :   if x2 ∈ paths {
4 :     (p, v) ← paths[x2];
5 :     (m, k) ← parse(p||(v ⊕ x1));
6 :     if unpad(m) ≠ false {
7 :       lb ← F(unpad(m), k · r);
8 :       y1 ← last(bits2blocks(lb));
9 :     } else if 0 < k {
10 :      if (m, k − 1) ∉ invalid {
11 :        invalid[m, k − 1] ←$ {0,1}^r;
12 :      }
13 :      y1 ← invalid[m, k − 1];
14 :    } else {
15 :      y1 ← 0^r;
16 :    }
17 :    paths[y2] ← (p||x1 ⊕ v, y1);
18 :  } else {
19 :    y1 ←$ {0,1}^r;
20 :  }
21 :  m[x1, x2] ← (y1, y2);
22 :  mi[y1, y2] ← (x1, x2);
23 : }
24 : return m[(x1, x2)];
```

SIMULATOR$[\mathscr{F}]^{-1}(x_1, x_2)$

```
if (x1, x2) ∉ mi {
  y1 ←$ {0,1}^r;
  y2 ←$ {0,1}^(b−r);
  mi[x1, x2] ← (y1, y2);
  m[y1, y2] ← (x1, x2);
}
return mi[(x1, x2)]
```

Figure 6.13 – The optimized SIMULATOR for SPONGE

6.4 A deeper look at the upto-bad game

In this section, instead of decribing the full formal proof that has been verified in EasyCrypt, I describe the formalization of the game named G1 and the invariant that are both involved in the upto-bad formal proof of the indifferentiability. Placing G1 in the context:

— G1 is indifferentiable from the real game with CORESPONGE and RP:

$$\left|\Pr\left[\mathsf{Real}^{\mathsf{D}}_{b-r,\mathsf{pad},r} = 1\right] - \Pr[\mathrm{G1} = 1]\right| \leq \Pr[\mathrm{G1} : \mathrm{G1.bcol} \vee \mathrm{G1.bcol}] \leq \frac{\sigma^2}{2^{b-r-2}} + \frac{\sigma^2 - \sigma}{2^{b+1}}$$

— and G1 is equivalent to the ideal game with IDEALCORE and CORESIM:

$$\Pr[\mathrm{G1} = 1] = \Pr\left[\mathsf{Ideal}^{\mathsf{D}}_{\mathsf{P}} = 1\right]$$

In this part, I first describe the meaning of G1's global variables, then other global variables and modules, then each procedure, and finally the invariant that holds when proving the indifferentiability upto the bad event G1.bext ∨ G1.bcol.

EasyCrypt

```
type state = block * capacity.
type smap = (state, state) fmap.
type handle = int.
type hstate = block * handle.
type hsmap = (hstate, hstate) fmap.
module G1 (D : DISTINGUISHER) = {
  var m, mi : smap
  var mh, mhi : hsmap
  var chandle : int
  var paths : (capacity, block list * block) fmap
  var bext, bcol : bool
  module M = { proc f(p : block list): block = { (* functionality *) } }
  module S = {
    proc f(x : state): state = { (* primitive *) }
    proc fi(x : state): state = { (* inverse *) }
  }
  proc main(): bool = {
    var b;
    (* initialization *)
    b ← D(M,S).distinguish();
    return b;
}}.
```

The intuition of G1's global variables is as follows:

— G1.chandle: is the next indirection number that is attributed to a new capacity.

— G1.m, G1.mi: represented in Figure 6.14, they are the finite maps that keep track of the simulator's answers to queries, with G1.m for the permutation, and G1.mi for its inverse.

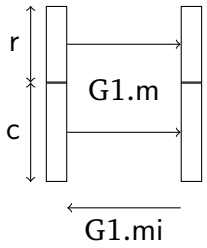


Figure 6.14 – Representation of G1.m and G1.mi

— G1.mh, G1.mhi: represented in Figure 6.15, they are the finite maps that help compute the absorbtion phase where capacities are replaced by their indirection number, with G1.mh for the permutation, and G1.mhi for its inverse.

— G1.paths: represented in Figure 6.16, this is the finite map that keeps track of how each computed capacity was obtained from the absorbtion phase of a block list. The block associated is the one output at the same time as the capacity at the end of the absorbtion.

— G1.bext, G1.bcol: are booleans that identify the triggering of bad events.

The other global variables and modules that appear in G1 or in the invariant are:

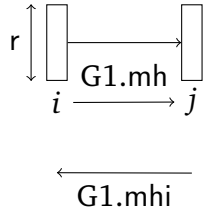


Figure 6.15 – Representation of G1.mh and G1.mhi

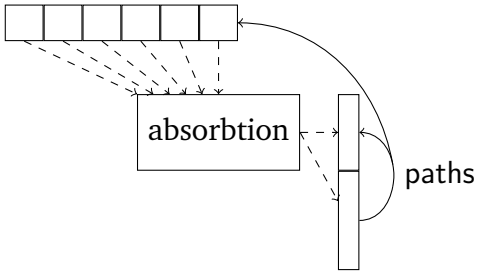


Figure 6.16 – Representation of G1.paths

— C.queries: represented in Figure 6.17, this is the finite map that keeps track of all the answers of the functionnality. It prevents the adversary to ask again its first oracle the non-fresh queries, and helps compute the maximum size of commom prefixes with previous queries.

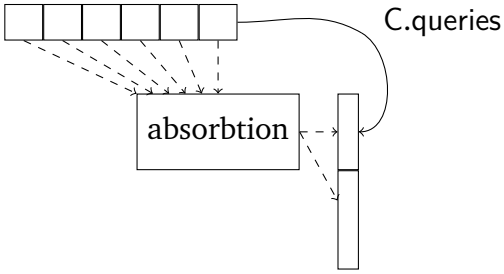


Figure 6.17 – Representation of C.queries

— FRO.m: is the finite map that keeps track of the value of the capacity that is associated to each indirection number. It is also associated to the flag that states if the capacity is in the adversary's view: Known or Unknown.

— F.RO: is the module that represents the ideal random function IDEALCORE.

— P.m, P.mi: are the finite maps of the permutation in the real setting that keeps track of all the permutation's answers to queries made either by the adversary or during the absorbtion, with m for the permutation, and mi for its inverse.

— Redo.prefixes: represented in Figure 6.18, this is the map that keeps track of all the states that are a result of an absortion of one of the queries' prefixes.

Invariant

The invariant that holds when none of the bad events has been triggered is a conjonction of the following ten properties:

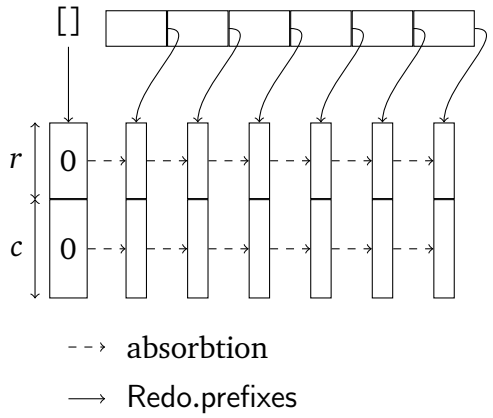


Figure 6.18 – Representation of Redo.prefixes

1. In the map FRO.m, unique capacities are associated to each indirection number, the indirection number 0 is associated to the full-0 capacity (Known), and all indirection numbers are positive and less than chandle (the next indirection number to set).
2. The maps G1.mh and G1.mhi are inverse to each other.
3. The maps P.m and P.mi are inverse to each other.
4. Represented in Figure 6.19, if a state is in P.m, then the corresponding pair is in G1.mh, following the associations of indirection number and capacity provided by FRO.m.

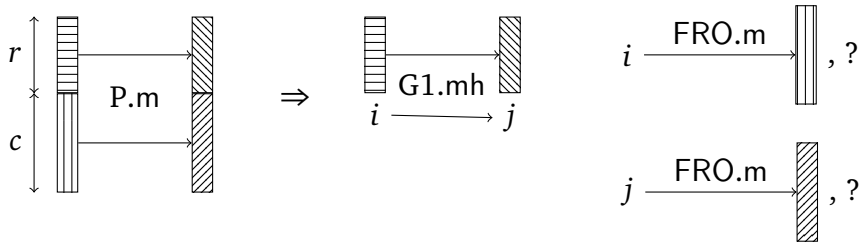


Figure 6.19 – Representation of the first property of 4.

Inversely, if a pair is in G1.mh, the corresponding state is in P.m, following FRO.m. This is represented in Figure 6.20.

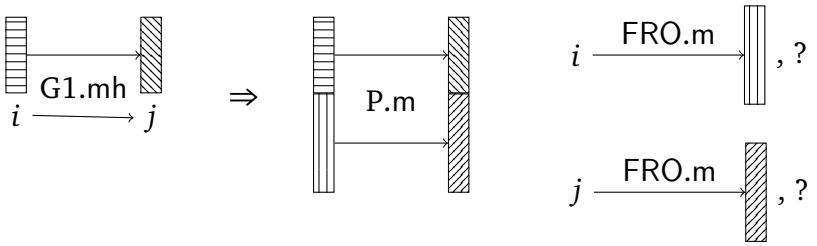


Figure 6.20 – Representation of the second property of 4.

5. Same as 4 but for P.mi, G1.mhi and FRO.m.
6. The map G1.m is included in the map P.m.
7. The map G1.mi is included in the map P.mi.

8. This property is about building paths and is divided into a conjonction of three sub-properties:
   (a) Represented in Figure 6.21, if some pair is in G1.mh, then either the flag corresponding to both input and output's indirection numbers is Known and G1.m has the corresponding entries, or the resulting flag is Unknown and the absorbtion path exists upto this point.

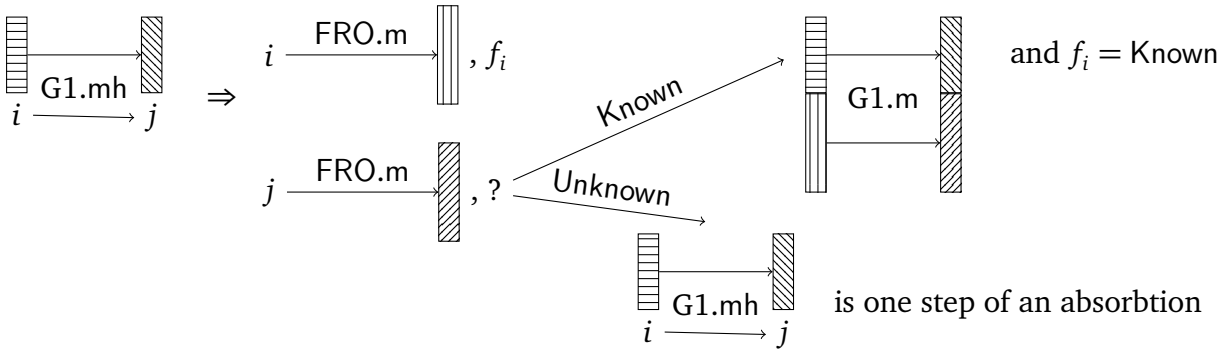


Figure 6.21 – Representation of the property 8a.

   (b) There is an equivalence between the belonging of a list to F.RO's map and building the full absorbtion path with G1.mh, as represented in Figure 6.22.

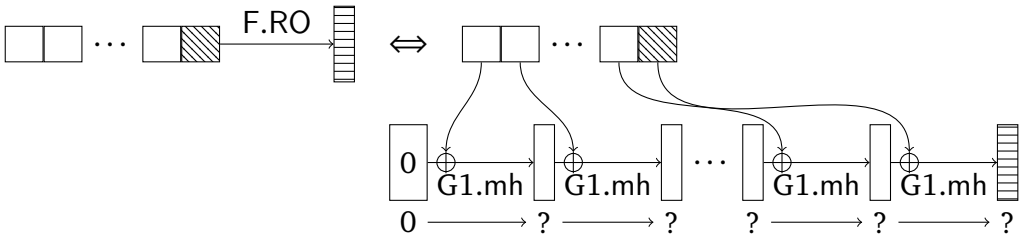


Figure 6.22 – Representation of the property 8b.

   (c) While building paths using G1.mh, the resulting indirection number identifies a unique list and a unique resulting block, as represented in Figure 6.23.
9. For every entry in the map G1.paths, the absorbtion path built from G1.mh exists in full, and the resulting indirection number associated to this resulting capacity has the flag Known in FRO.m.
10. This property is about the prefix map and is divided into a conjonction of three sub-properties:
    (a) Each computation of absorbtion through P.m are stored in Redo.prefixes.
    (b) The map C.queries is a sub-map of Redo.prefixes.
    (c) If a list of blocks is in Redo.prefixes, then it is a prefix a previous query stored in C.queries.

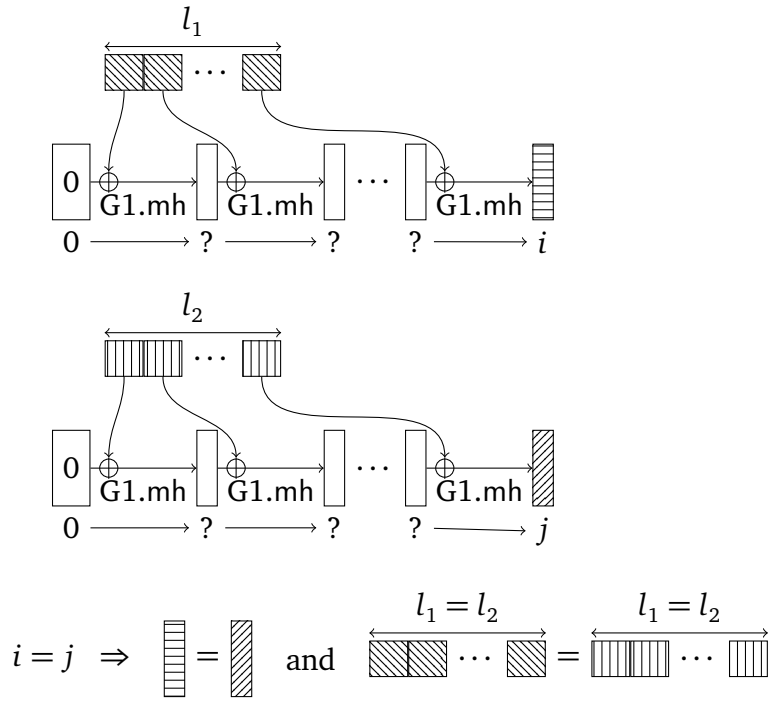


Figure 6.23 – Representation of the property 8c.

Initialization

EasyCrypt

```
module G1(D:DISTINGUISHER) = {
  (* global variables, module M, module S *)
  proc main(): bool = {
    var b;
    (* initializing all the global variables *)
    F.RO.m ← empty;
    m ← empty;
    mi ← empty;
    mh ← empty;
    mhi ← empty;
    bext ← false;
    bcol ← false;
    C.queries← empty[[] ← b0];
    (* the empty path is initially known by the adversary to lead to capacity 0^c *)
    FRO.m ← empty[0 ← (c0, Known)];
    paths ← empty[c0 ← ([<:block>],b0)];
    chandle ← 1;
    (* call the distinguisher and outputs its answer *)
    b ← D(M,S).distinguish();
    return b;
  }
}.
```

Listing 6.1 – Initialization of global variables

Functionality

The submodule M implements the functionnality that is functionnally equivalent to the ideal random function previously named IDEALCORE and implemented by F.RO. It keeps an absorbtion structure with the indirection described in the previous section. The map that associates states to states P.m is replaced by the indirection map G1.mh that associates a pair composed of a block and a indirection number to a pair composed of a block and another indirection number.

EasyCrypt

```
module G1(D:DISTINGUISHER) = {
  (* global variables *)
  module M = {
    proc f(p : block list): block = {
      var sa, sa', sc;
      var h, i, counter ← 0; (* all three variables are initialized to 0 *)
      (sa,sc) ← (b0,c0); (* initial state *)
      while (i < size p) {
        if ((sa +^ nth witness p i, h) ∈ mh) (* if the path is already known *) {
          (sa, h) ← oget mh[(sa +^ nth witness p i, h)]; (* follow it *)
        } else {
          (* check if the prefix cost is respected (helpful when proving the invariant) *)
          if (counter < size p − prefix p (get_max_prefix p (elems (fdom C.queries)))) {
            sc ←$ cdistr; (* sample the next capacity *)
            (* does it triggeres a bad event ? *)
            bcol ← bcol ∨ hinv FRO.m sc ≠ None;
            (* ask to the random oracle for the next block *)
            sa' ← F.RO.get(take (i+1) p);
            (* xors the message's block with the last block of the absorbtion *)
            sa ← sa +^ nth witness p i;
            (* updates the indirection map and its inverse *)
            mh[(sa,h)] ← (sa', chandle);
            mhi[(sa',chandle)] ← (sa, h);
            (* adds the correspondance of the indirection number to its capacity *)
            FRO.m[chandle] ← (sc,Unknown);
            (* updates iterative variables *)
            (sa,h) ← (sa',chandle);
            chandle ← chandle + 1;
            counter ← counter + 1;
          }
        } (* end if (···, h) ∈ mh *)
        i ← i + 1;
      } (* end while *)
      (* ask for the block associated to the full message *)
      sa ← F.RO.get(p);
      return sa;
    }
  }
  (* module S *)
  (* proc main *)
}.
```

Listing 6.2 – Algorithm of the functionnality in G1

In this procedure, the bad event bcol can be triggered using the operator hinv. It looks into the map given as its first argument to find a indirection number that may be associated to the capacity given as its second argument, that may be associated with the flag Known or Unknown. If the bad event is triggered, this means that either there is a collision of capacities with two different messages (two different absorbtion paths), or the adversary correctly guessed a capacity before it was sampled.

Simulation of the permutation

The other sub-module of G1 is named S and is given to the adversary as the primitive. We show in EasyCrypt that the sub-module S is equivalent to the simulator in Figure 6.11. The next EasyCrypt's code outlines the inverse permutation that is simpler than the direct permutation, followed by the direct permutation.

EasyCrypt

```
module G1(D:DISTINGUISHER) = {
  (* global variables *)
  (* module M *)
  module S = {
    proc f(x : state): state = { (* ··· *) }
    proc fi(x : state): state = {
      var y, y1, y2, hx2, hy2;
      if (x \notin mi) {
        (* has the adersary guessed one of the sampled non−revealed capacities *)
        bext ← bext ∨ rng FRO.m (x.`2, Unknown);
        (* updates the maps associating indirection numbers to capacities (and knwledgable flags) *)
        if (¬(rng FRO.m (x.`2, Known))) {
          FRO.m[chandle] ← (x.`2, Known);
          chandle ← chandle + 1;
        }
        (* obtain the indirection number *)
        hx2 ← oget (hinvK FRO.m x.`2);
        (* sample the output *)
        y1 ←$ bdistr;
        y2 ←$ cdistr;
        y ← (y1,y2);
        (* if the input block is associated to the obtained indirection number, and the capacity
        associated to the next indirection number is unknown to the adversary, follow the path *)
        if ((x.`1,hx2) ∈ mhi ∧ in_dom_with FRO.m (oget mhi[(x.`1,hx2)]).`2 Unknown) {
          (y1,hy2) ← oget mhi[(x.`1, hx2)];
          y ← (y.`1, (oget FRO.m[hy2]).`1);
          FRO.m[hy2] ← (y.`2, Known);
          (* update the maps of the simulator with the computed values *)
          mi[x] ← y;
          m[y] ← x;
        } else {
          (* the bad event is triggered if the capacity sampled already has appeared *)
          bcol ← bcol ∨ hinv FRO.m y.`2 ≠ None;
          (* update the maps to keep the absorbing structure *)
          hy2 ← chandle;
          chandle ← chandle + 1;
          FRO.m[hy2] ← (y.`2, Known);
          mi[x] ← y;
```

```
        mhi[(x.`1, hx2)] ← (y.`1, hy2);
        m[y] ← x;
        mh[(y.`1, hy2)] ← (x.`1, hx2);
       }
      } else {
       y ← oget mi[x];
      }
      return y;
     }
    }
    (* proc main *)
}.
```

Listing 6.3 – Inverse permutation that is equivalent to the simulator

The direct permutation is the only procedure that uses the global variable paths. That way, if the adversary asks a query that is part of an absorbtion computation, the variable paths will keep track of the message from which it is a absorbtion, and the simulator will asks F.RO for the next block. In any case, the capacity is sampled in the same way as for the inverse procedure.

EasyCrypt

```
module G1(D:DISTINGUISHER) = {
  (* global variables *)
  (* module M *)
  module S = {
    proc f(x : state): state = {
      var p, v, y, y1, y2, hy2, hx2;
      (* has this state not already been queried to S.f or was an output of S.fi ? *)
      if (x \notin m) {
        (* is this capacity output from an absorbing phase ? *)
        if (x.`2 ∈ paths) {
          (* if so, get the message and output block associated to this capacity *)
          (p,v) ← oget paths[x.`2];
          (* call the ideal random function as if this query was part of an absorbtion phase *)
          y1 ← F.RO.get (rcons p (v +^ x.`1));
          (* sample the capacity normally *)
          y2 ←$ cdistr;
        } else {
          (* sample both block and capacity normally *)
          y1 ←$ bdistr;
          y2 ←$ cdistr;
        }
        y ← (y1, y2);
        (* bad event : was the queried capacity part of a previous absorbtion phase ? *)
        bext ← bext ∨ rng FRO.m (x.`2, Unknown);
        (* updates the map associating indirection numbers to capacities and knowledgable flags *)
        if (¬(rng FRO.m (x.`2, Known))) {
          FRO.m[chandle] ← (x.`2, Known);
          chandle ← chandle + 1;
        }
        (* obtain the indirection number associated to the queried capacity *)
        hx2 ← oget (hinvK FRO.m x.`2);
        (* test if the indirection number that is next after (y.`1,hx2) is already associated to a capacity *)
```

```
      if ((x.`1, hx2) ∈ mh ∧ in_dom_with FRO.m (oget mh[(x.`1,hx2)]).`2 Unknown) {
        (* then erase the sampled capacity and output the associated capacity *)
        hy2 ← (oget mh[(x.`1, hx2)]).`2;
        y ← (y.`1, (oget FRO.m[hy2]).`1);
        FRO.m[hy2] ← (y.`2, Known);
        m[x] ← y;
        mi[y] ← x;
      } else { (* otherwise, associate this capacity to the indirection number with the flag Known *)
        (* bad event triggered if the capacity is already associated to a previous indirection number *)
        bcol ← bcol ∨ hinv FRO.m y.`2 ≠ None;
        (* update the maps and global variables *)
        hy2 ← chandle;
        chandle ← chandle + 1;
        FRO.m[hy2] ← (y.`2, Known);
        m[x] ← y;
        mh[(x.`1, hx2)] ← (y.`1, hy2);
        mi[y] ← x;
        mhi[(y.`1, hy2)] ← (x.`1, hx2);
      }
      (* same test as the beginning *)
      if (x.`2 ∈ paths) {
        (* update the known path of a given capacity *)
        (p,v) ← oget paths[x.`2];
        paths[y.`2] ← (rcons p (v +^ x.`1), y.`1);
      }
    } else {
      y ← oget m[x];
    }
    return y;
  }
  proc fi(x : state): state = { (* ··· *) }
 }
 (* proc main *)
}.
```

Listing 6.4 – Permutation that is equivalent to the simulator

Part IV

Confidentiality and Authentication

Chapter 7

Security of Authenticated Encryption

In cryptography, *encryption* is the process of encoding information, often encoded as a message, in such a way that only authorized parties can access it while non-authorized others cannot. In an encryption scheme, the intended message, referred to as *plaintext*, is encrypted using an encryption algorithm — a *cipher* — generating a *ciphertext* that can be read — whose meaning can be retrieved — only if decrypted. Sender and recipient share keys that can encrypt and decrypt messages. When the encryption key is the same as the decryption key, the scheme is called *symmetric encryption* (or *private key encryption*), otherwise it is called *asymmetric encryption* (or *public key encryption*). In my manuscript, I focus on the security of symmetric encryption.

The objective of encryption is to deny the intelligible content to a would-be interceptor. This security property is called *data confidentiality*. This means that any entity that does not possess the secret key is unable to retrieve information on plaintext from ciphertext.

7.1 Authenticated Encryption

Data confidentiality has been one of main goals of cryptography for quite a long time, while the need to combine *data integrity* with confidentiality came from the banking field.

Example

Integrity of banking transactions, e.g. the price agreed upon, is a main concern, especially when using substitution ciphers. In this setting, an attacker does not need to decrypt the ciphertext, but just flip a coin and change the amount in the transaction without knowing the secret key. This concern may be extended to any encryption scheme, even without the malleability of substitution ciphers; banks did not want to transmit encrypted data that would allow an attacker to change messages undetected.

Authenticated encryption (AE) and *authenticated encryption with associated data* (AEAD) are forms of encryption which simultaneously aim to provide the confidentiality and authenticity of data, the associated data being a non-encrypted part of the message that is nonetheless authenticated (e.g. packet header for transportation across the network).

The easiest way to create a AE scheme is to combine an existing encryption scheme with an existing message authentication code (MAC) scheme with two different secret keys. In [Bellare and Namprempre, 2000], Bellare and Namprempre analyze the security of three general compositions of encryption schemes and MAC schemes: 1. Encrypt-then-MAC 2. Encrypt-and-MAC 3. MAC-then-Encrypt. In their work, they prove that, under the security

assumption that the encryption scheme and the MAC scheme are secure, only the general approach Encrypt-then-MAC provides a secure authenticated encryption scheme.

This means that the other two are in general insecure. Indeed, securely combining separate confidentiality and authentication block cipher operation modes can be difficult and error prone. « In fact, efficient integration of encryption, authentication, and message integrity using a *single shared key* is an important and intensely studied problem in network security [Kaufman et al., 1995, Perlman et al., 2016, Preneel, 1998] which is solved by the use of [an authenticated] encryption scheme. » ([Katz and Yung, 2000b]) This issue has been raised for decades, and a recent example of an attempt to answer it is the CAESAR competition [CAE, 2013] that aimed to determine the best AE scheme through a long process of public competitive peer review.

7.2 Security Definitions

Against a secure encryption scheme, any adversary should learn no information from seeing a ciphertext. Therefore, the adversary should be able to do no better than if it guessed randomly. The following definition encompasses this security notion.

An encryption scheme is considered secure if no adversary, given an encryption of a message randomly chosen from a two-element message space determined by the adversary, can identify the message choice with probability significantly better than that of random guessing ($\frac{1}{2}$). If any adversary can succeed in distinguishing the chosen ciphertext with a probability significantly greater than $\frac{1}{2}$, then this adversary is considered to have an *advantage* in distinguishing the ciphertext, and the scheme is not considered secure.

Security in terms of indistinguishability has several definitions, depending on assumptions made about the adversary's capabilities. The most common definitions used in provable security — first in the public-key setting [Bellare et al., 1997, Bellare et al., 1998a], then extended into the private-key setting [Katz and Yung, 2000a] — are *indistinguishability under chosen plaintext attack* (IND-CPA), *indistinguishability under (non-adaptive) chosen ciphertext attack* (IND-CCA1 [Naor and Yung, 1990]), and *indistinguishability under adaptive chosen ciphertext attack* (IND-CCA2 [Rackoff and Simon, 1991]).

Security under either of the latter definition implies security under the previous ones: a scheme which is IND-CCA1 secure is also IND-CPA secure, and a scheme which is IND-CCA2 secure is both IND-CCA1 and IND-CPA secure. Thus, IND-CCA2 is the strongest of the three definitions of security.

Intuitively, the IND-CPA security is the security definition relative to data confidentiality; information about plaintexts cannot be retrieved from ciphertexts without knowing the secret key. Whereas the IND-CCA2 security also includes data integrity and authentication; that is, in addition, no valid ciphertext can be produced without the secret key.

Example

In [Garman et al., 2016], Garman et al. found an adaptive chosen ciphertext attack (showing the IND-CCA2 insecurity) against Apple's iMessage in which they achieve to break the security of a ciphertext if they have access to 2^{18} decryption oracle queries after the ciphertext they want to break is obtained. « The practical implication of these attacks is that any party who gains access to iMessage ciphertexts may potentially decrypt them remotely and after the fact. » ([Garman et al., 2016])

7.2.1 IND-CPA

A chosen-plaintext attack (CPA) is an attack model for cryptanalysis which presumes that the attacker can obtain the ciphertexts for arbitrary plaintexts. The goal of the attack is to gain information that reduces the security of the encryption scheme.

In a chosen-plaintext attack, the adversary can adaptively ask for the encryption of arbitrary plaintext messages. This is formalized by allowing the adversary to interact with an encryption oracle, viewed as a black box that encrypts plaintexts using the secret key. The attacker's goal is to retrieve any information or pattern about encrypted plaintexts.

The game IND-CPA is works as follows.

1. The game initializes the secret key of the encryption scheme.
2. The adversary interacts with the encryption oracle by querying adaptive plaintexts.
3. The adversary submits a challenge that consists of a pair of two plaintexts m_0, m_1.
4. The game randomly samples a bit $b \in \{0,1\}$ and sends to the adversary the ciphertext associated to m_b.
5. The adversary may ask for new queries to the encryption oracle.
6. The adversary outputs a bit b' that represents its guess of the bit b.

The adversary has won if it guessed correctly, i.e. $b = b'$.

EasyCrypt

```
module type Enc = {
  proc key_gen () : key
  proc enc (_: key * plaintext) : ciphertext
  proc dec (_: key * ciphertext) : plaintext
}.
```

```
module type CPA_Oracles = {
  proc enc (_: plaintext) : ciphertext
}.
```

```
module Mem (E : Enc) : CPA_Oracles = {
  var key : key
  proc enc (m : plaintext) = {
    var c;
    c ← E.enc(Mem.key, m);
    return c;
}}.
```

```
module type Adv_CPA (E : CPA_Oracles) = {
  proc challenge() : plaintext * plaintext { E.enc }
  proc guess(_: ciphertext) : bool { E.enc }
  (* remove E.enc when the adversary is not
     allowed to ask for further encryptions *)
}.
```

```
module IND_CPA (A : Adv_CPA, E : Enc) = {
  proc game() : bool = {
    var b, b', m0, m1, c;
    Mem.key ← E.key_gen();
    (m0, m1) ← A(Mem(E)).challenge();
    b $← {0,1};
    c ← Mem(E).enc(b ? m1 : m0);
    b' ← A(Mem(E)).guess(c);
    return b = b';
}}.
```

The *IND-CPA advantage* of an adversary $\mathcal{A}$ against an encryption scheme E is defined as

$$\mathrm{Adv}_{E}^{\mathrm{ind-cpa}}(\mathcal{A}) := \left| \Pr[b \leftarrow \mathrm{IND\_CPA}(\mathcal{A}, E).\mathrm{game}() : b = 1] - \frac{1}{2} \right|$$

7.2.2 IND-CCA1 and IND-CCA2

Indistinguishability under non-adaptive and adaptive Chosen Ciphertext Attack (IND-CCA1, IND-CCA2) uses a definition similar to that of IND-CPA. However, in addition to

the encryption oracle, the adversary is given access to a decryption oracle which decrypts arbitrary ciphertexts at the adversary's request, returning either the plaintext or an error message stating that the ciphertext provided is not well-formed. In the non-adaptive definition, the adversary is allowed to query the decryption oracle only up to receiving the challenge ciphertext. In the adaptive definition, the adversary may continue to query the decryption oracle even after receiving the challenge ciphertext, with the caveat that it may not ask the challenge ciphertext for decryption (otherwise, the definition would be trivial).

The game IND-CCA works as follows, where the difference between IND-CCA1 and IND-CCA2 is respectively either the absence or presence of the decryption oracle in step 5.

1. The game initializes the secret key of the encryption scheme.
2. The adversary interacts with both the encryption and decryption oracles by querying adaptative plaintexts and ciphertexts.
3. The adversary submits a challenge that consists of a pair of two plaintexts m_0, m_1.
4. The game randomly samples a bit $b \in \{0,1\}$ and sends to the adversary the ciphertext associated to m_b.
5. The adversary may ask for new queries to the encryption (and decryption) oracles (with the restriction of not asking for the decryption of the challenge's ciphertext).
6. The adversary outputs a bit b' that represents its guess of the bit b.

The adversary has won if it guessed correctly, i.e. $b = b'$.

IND-CCA1

The *IND-CCA1 advantage* of an adversary $\mathcal{A}$ against an encryption scheme E is defined as

$$\mathrm{Adv}^{\mathrm{ind-cca1}}_{\mathrm{E}}(\mathcal{A}) := \left|\Pr[b \leftarrow \mathrm{IND\_CCA1}(\mathcal{A}, \mathrm{E}).\mathrm{game}() : b = 1] - \frac{1}{2}\right|$$

EasyCrypt

```
module type CCA_Oracles = {
  proc enc (_: plaintext) : ciphertext
  proc dec (_: ciphertext) : plaintext option
}.
module Mem (E : Enc) : CPA_Oracles = {
  var key : key
  proc enc (m : plaintext) = {
    var c;
    c ← E.enc(Mem.key, m);
    return c;
  }
  proc dec (c : ciphertext) = {
    var p;
    p ← E.dec(Mem.key, c);
    return p;
}}.
```

```
module type Adv_CCA1
   (E : CCA_Oracles) = {
  proc challenge()
    : plaintext * plaintext { E.enc E.dec }
  proc guess(_: ciphertext)
    : bool { E.enc } (* E.dec is absent *)
}.
module IND_CCA1 (A : Adv_CCA1, E : Enc) = {
  proc game() : bool = {
    var b, b', m0, m1, c;
    Mem.key ← E.key_gen();
    (m0, m1) ← A(Mem(E)).challenge();
    b $← {0,1};
    c ← Mem(E).enc(b ? m1 : m0);
    b' ← A(Mem(E)).guess(c);
    return b = b';
}}.
```

IND-CCA2

The *IND-CCA2 advantage* of an adversary $\mathscr{A}$ against an encryption scheme E is defined as

$$\mathsf{Adv}^{\mathrm{ind-cca2}}_{\mathrm{E}}(\mathscr{A}) := \left|\Pr[b \leftarrow \mathsf{IND\_CCA2}(\mathscr{A},\mathrm{E}).\mathsf{game}() : b = 1] - \frac{1}{2}\right|$$

EasyCrypt

```
module Mem (E : Enc)
     : CPA_Oracles = {
  var key : key
  var cipher : ciphertext option
  proc enc (m : plaintext) = {
    var c;
    c ← E.enc(Mem.key, m);
    return c;
  }
  proc dec (c : ciphertext) = {
    var p;
    if (Some c = Mem.cipher) { p ← None; }
    else { p ← E.dec(Mem.key, c); }
    return p;
}}.
module type Adv_CCA2 (E : CCA_Oracles) = {
  proc challenge()
    : plaintext * plaintext { E.enc E.dec }
  proc guess(_: ciphertext): bool { E.enc E.dec }
}.
```

```
module IND_CCA2 (A : Adv_CCA2, E : Enc) = {
  proc game() : bool = {
    var b, b', m0, m1, c;
    Mem.key ← E.key_gen();
    Mem.cipher ← None;
    (m0, m1) ← A(Mem(E)).challenge();
    b $← {0,1};
    c ← Mem(E).enc(b ? m1 : m0);
    Mem.cipher ← Some c;
    b' ← A(Mem(E)).guess(c);
    return b = b';
}}.
```

Remark

In the following, whenever we talk about IND-CCA, we refer to IND-CCA2 unless stated otherwise.

7.2.3 INT-CTXT : Integrity of the CipherTeXT

First named *unforgeable encryption* in [Katz and Yung, 2000b], *integrity of ciphertexts* (INT-CTXT) is a security requirement necessary for any protocol that wants to detect any unauthentic ciphertext. That is, no one without the secret key should be able to produce a ciphertext that can be successfully decrypted, even if valid ciphertexts can be asked for chosen plaintexts and several attempts can be made to forge a valid ciphertext.

In the INT-CTXT game, the adversary can ask for encryption queries and it wins if it succeeds in querying the decryption query with a ciphertext that was not an output of the encryption oracle that can be successfully decrypted. The advantage of such an adversary is defined as the probability of winning this particular game:

$$\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathrm{E}}(\mathscr{A}) := \Pr[\mathsf{INTCTXT}(\mathscr{A},\mathrm{E}).\mathsf{main}() : \exists \mathsf{c}, \mathsf{c} \in \mathsf{Mem.lc} \wedge \mathsf{dec}_{\mathrm{E}}(\mathsf{Mem.k},\mathsf{c}) \neq \mathsf{None}]$$

Here, Mem.lc is the list that keeps track of all decryption queries and $\mathsf{dec}_{\mathrm{E}}$ refers to the decryption algorithm of the scheme E.

EasyCrypt

In EasyCrypt, an event in a probability formula cannot contain procedures, but it may contain operators and global variables. While procedures are more expressive than operators — they allow random sampling and updates of internal states — EasyCrypt can prove equivalences between operators and stateless deterministic procedures. Since symmetric encryption generally has stateless deterministic procedures, they can be expressed as operators.

```
type key, plaintext, ciphertext.
op enc : key → plaintext → ciphertext.
op dec : key → ciphertext → plaintext option.
```

```
module type StLOrcls = {
  proc * init () : unit
  proc kg () : key
}.
```

```
module Mem = {
  var k   : key
  var log : (ciphertext, plaintext) fmap
  var lc  : ciphertext list
}.
```

```
module type INTCTXT_Oracles = {
  proc enc(_: plaintext): ciphertext
  proc dec(_: ciphertext): plaintext option
}.
```

```
module type Adv_INTCTXT
    (O : INTCTXT_Oracles) = {
  proc main(): bool
}.
```

```
module INTCTXTOrcls (S : StLOrcls)
    : INTCTXT_Oracles = {
  proc init () = {
    S.init();
    Mem.log ← empty;
    Mem.lc ← [];
    Mem.k ← S.kg();
  }
  proc enc (p : plaintext) = {
    var c;
    c ← enc Mem.k p;
    Mem.log[c] ← p;
    return c;
  }
  proc dec (c : ciphertext) = {
    (* note that the operator dec is not used in
       the ''decryption oracle'' *)
    Mem.lc ← if c ∈ Mem.log then Mem.lc
             else c :: Mem.lc;
    return Mem.log[c];
}}.
```

```
module INTCTXT
    (A : Adv_INTCTXT, StL : StLOrcls) = {
  module O = INTCTXTOrcls(StL)
  proc main () = {
    var b;
    O.init();
    b ← A(O).main();
    return b;
}}.
```

7.3 Relations between IND-CCA, IND-CPA and INT-CTXT

In EasyCrypt, the difference between the games IND-CPA and IND-CCA is the set of procedures accessible by the adversary. In particular, the decryption oracle is only accessible in the IND-CCA game.

When in the game IND-CCA the encryption scheme is modified so that the decryption algorithm always answers None unless the query was the output of a previous encryption, then IND-CCA security is equivalent to IND-CPA security for this scheme. Using this observation, we define a module that transforms an IND-CCA adversary into an IND-CPA adversary.

EasyCrypt

The formalization uses the global variables from the module Mem. It transforms a module of module type Enc into the module type accepted by an IND-CCA adversary and hides the secret key inside Mem.k.

```
module Mem (S : Enc) = {
  var k   : key
  var log :  (ciphertext, plaintext) fmap
  proc init () = { Mem.k ← S.keygen(); }
  proc enc(p:plaintext) = {
    var c;
    c ← S.enc(Mem.k,p);
    return c;
  }
  proc dec(c:ciphertext) = {
    var p;
    p ← S.dec(Mem.k,c);
    return p;
}}.
```

Given an IND-CCA adversary $\mathscr{A}$ and an IND-CPA encryption oracle, we built the IND-CPA adversary by giving to $\mathscr{A}$ a direct acces to the encryption oracle, and by simulating the decryption oracle using the information contained in the state of Mem.

```
module CPA_CCA_Orcls(O:CPA_Oracles)
   : CCA_Oracles = {
  proc init () = {
    Mem.log ← empty;
  }
  proc enc(p:plaintext) = {
    var c;
    c ← O.enc(p);
    Mem.log[c] ← p;
    return c;
  }
  proc dec(c:ciphertext) = {
    return Mem.log[c];
}}.
```

Finally, the construction of an IND-CCA adversary is named CPA_CCA_Adv.

```
module CPA_CCA_Adv
   (A:CCA_Adv, O:CPA_Oracles) = {
  proc main () = {
    var b;
    CPA_CCA_Orcls(O).init();
    b ← A(CPA_CCA_Orcls(O)).main();
    return b;
}}.
```

7.3.1 IND-CCA from IND-CPA and INT-CTXT

The game IND-CPA is equivalent to IND-CCA when the decryption procedure always answers None. Therefore, when in IND-CCA the adversary asks for a decryption query that is correctly decrypted (and was not an output from the encryption oracle), this event *distinguishes* IND-CPA from IND-CCA. This distinguishing event is exactly what INT-CTXT accounts for. This means that the IND-CCA advantage is bounded by the IND-CPA advantage plus the INT-CTXT advantage, as first stated in [Bellare and Namprempre, 2000].

Theorem 7.3.1. *For all IND-CCA adversary $\mathscr{A}$, its IND-CCA advantage is bounded by the IND-CPA advantage of* CPA_CCA_Adv($\mathscr{A}$) *plus the INT-CTXT advantage of $\mathscr{C}(\mathscr{A})$:*

$$\mathsf{Adv}_{\mathrm{E}}^{\mathrm{ind-cca}}(\mathscr{A}) \leq \mathsf{Adv}_{\mathrm{E}}^{\mathrm{ind-cpa}}(\mathsf{CPA\_CCA\_Adv}(\mathscr{A})) + \mathsf{Adv}_{\mathrm{E}}^{\mathrm{int-ctxt}}(\mathscr{C}(\mathscr{A}))$$

when the adversary $\mathscr{C}(\mathscr{A})^{\mathsf{O}}$ is defined as the simulation of the game IND-CCA with $\mathscr{A}$ and O*, and outputs the result of the game.*

Proof. The proof is a simple application of a failure event: the forgery of a valid decryption. □

EasyCrypt

Furthermore, this remains true for more general games than IND-CPA and IND-CCA, that plays two rounds challenge and guess. We name them CCA and CPA, that use the module Mem from Section 7.3. The main difference between those games is the module type of the adversary that differs in the procedures allowed.

```
module type CCA_Adv (O: CCA_Oracles) = {
  proc main () : bool { O.enc O.dec }
}.
module CCA (A : CCA_Adv, E : Enc) = {
  proc game () : bool = {
    var b;
    Mem(E).init();
    b ← A(Mem(E)).main();
    return b;
}}.
```

```
module type CPA_Adv (O: CCA_Oracles) = {
  proc main () : bool { O.enc }
}.
module CPA (A : CPA_Adv, E : Enc) = {
  proc game () : bool = {
    var b;
    Mem(E).init();
    b ← A(Mem(E)).main();
    return b;
}}.
```

Theorem 7.3.2. *For all CCA adversary $\mathscr{A}$, its CCA probability is bounded by the CPA probability of* CPA_CCA_Adv($\mathscr{A}$) *plus the INT-CTXT advantage of $\mathscr{A}$.*

$$\begin{aligned}&\Pr[b \leftarrow \mathsf{CCA}(\mathscr{A}, \mathsf{E}).\mathsf{game}() : b = 1] \\ \leq &\Pr[b \leftarrow \mathsf{CPA}(\mathsf{CPA\_CCA\_Adv}(\mathscr{A}), \mathsf{E}).\mathsf{game}() : b = 1] + \mathsf{Adv}_{\mathsf{E}}^{\mathrm{int-ctxt}}(\mathscr{A})\end{aligned}$$

Proof. The proof is a simple application of a failure event: the forgery of a valid decryption. □

7.3.2 Ideal Encryption Scheme

The encryption scheme described in the next chapter in a *nonce-based* AEAD scheme [Rogaway, 2004b]. This means that the user supplies not only a symmetric key k, the message m, and associated data a, but also a *nonce* n. The concept of a nonce is something that takes on a new value with every message one encrypts. This idea was introduced (partly) in order to randomize the encryption of the same message over time.

The module we name Ideal is the unrealistic module that captures this idea that every fresh encryption query will output a truly random ciphertext, and every non-encrypted ciphertext will never be authenticated and decrypted. The IND-CPA, IND-CCA and INT-CTXT advantages of this "scheme" are proven to be equal to 0:

$$\begin{aligned}\forall \mathscr{A}, \quad &\Pr[b \leftarrow \mathsf{IND\_CPA}(\mathscr{A}, \mathsf{Ideal}).\mathsf{game}() : b = 1] = \frac{1}{2} \\ \forall \mathscr{A}, \quad &\Pr[b \leftarrow \mathsf{IND\_CCA}(\mathscr{A}, \mathsf{Ideal}).\mathsf{game}() : b = 1] = \frac{1}{2} \\ \forall \mathscr{A}, \quad &\mathsf{Adv}_{\mathsf{Ideal}}^{\mathrm{int-ctxt}}(\mathscr{A}) = 0\end{aligned}$$

Therefore, any nonce-based AEAD scheme that is indistinguishable from Ideal, is IND-CPA, IND-CCA and INT-CTXT secure, under some nonce-respecting condition expressed in further details in Chapter 8.

EasyCrypt

In the case of an encryption scheme that operates on a message space that can be interpreted as the set of lists of bytes, the Ideal scheme ressembles to the following.

The decryption always answers None, and the encryption outputs a list of random bytes that has the same overall size as the plaintext given as input, and a random tag.

```
type key, nonce, associated_data, tag, block,
  byte.
type ad = associated_data.
type message = byte list.
type bytes = byte list.
type plaintext =
    nonce * associated_data * message.
type ciphertext =
    nonce * associated_data * message * tag.
```

```
op bytes_of_block : block → byte list.
op dtag : tag distr.
op dblock : block distr.
```

```
module Ideal : Enc = {
  proc key_gen () : key = { return witness; }
  proc cc(n:nonce, p:message) : bytes = {
    var z, c;
    p ← List.map (fun _ ⇒ witness<:byte>) p;
    c ← [];
    while (p ≠ []) {
      z ←$ dblock;
      c ←
        c ++ take (size p) (bytes_of_block  z);
      p ← drop block_size p;
    }
    return c;
  }
  proc enc (k: key, nap : plaintext)
    : ciphertext = {
    var n, a, p, c, t;
    (n,a,p) ← nap;
    c ← cc(n,p);
    t ←$ dtag;
    return (n,a,c,t);
  }
  proc dec (_: key * ciphertext)
    : plaintext option = {
    return None;
}}.
```

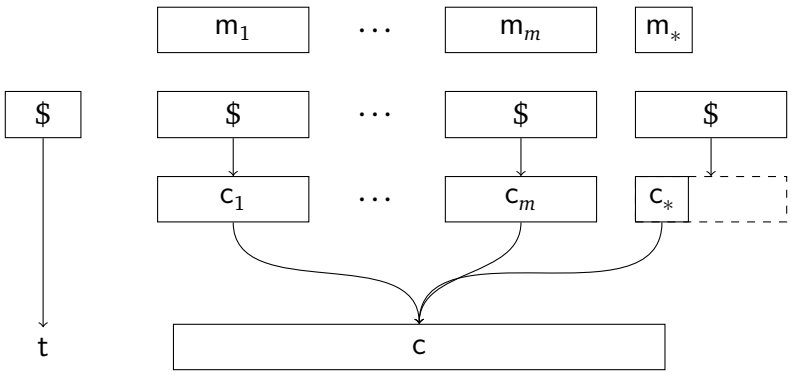


Figure 7.1 – Representation of the encryption of Ideal.

Chapter 8

ChaCha20-Poly1305's Formal Proof

The Internet Engineering Task Force (IETF) is an open standard organization, which develops and promotes voluntary Internet standards, in particular the standards that comprise the Internet protocol suite (TCP/IP). Due to concern over the reliance of existing IETF protocols on AES and the risk that advances in the cryptanalysis of AES could leave users without a good choice for a symmetric cryptographic primitive, there has been a proposal to the CFRG (Crypto Forum Research Group of the Internet Research Task Force (IRTF)) to consider a combination of ChaCha20 and Poly1305 for inclusion in future IETF protocols [Nir and Langley, 2015].

Remark

A similar concern led to the SHA-3 competition; improvements to attacks against SHA-1 [Stevens et al., 2017, Leurent and Peyrin, 2019] lead NIST to transition to the SHA-2 family of hash functions and to announce the SHA-3 competition, with the aim of choosing an alternative hash function to SHA-2, so that it improves the robustness of NIST overall hash algorithm toolkit' [Kayser, 2007].

Furthermore, ChaCha20 [Bernstein, 2008a] and Poly1305 [Bernstein, 2005] have both been designed for high performance in software implementations. They typically admit a compact implementation that uses few resources and inexpensive operations, which makes them suitable on a wide range of architectures. They have also been designed to minimize leakage of information through side-channels. Some attacks [Al Fardan and Paterson, 2013] used the particularities of the CBC-mode cipher suites in TLS, as well as issues with the only supported stream cipher RC4 [Isobe et al., 2013]. While the existing AEAD cipher suites —based on AES-GCM [McGrew and Viega, 2004a], whose security proof was flawed and repaired in [Iwata et al., 2012]— address some of these issues, there are concerns about their performance and ease of software implementation.

Therefore, a new stream cipher to replace RC4 and address all the previous issues has been proposed. The construction named ChaCha20-Poly1305 [Langley et al., 2016] aims to provide a secure stream cipher for TLS [Nir and Langley, 2015] that is comparable to RC4 in speed on a wide range of platforms and can be implemented easily without being vulnerable to software side-channel attacks. Indeed, in [Almeida et al., 2019a], Almeida et al. proposed a fast implementation of ChaCha20-Poly1305 that is resistant to some side-channel attacks. My contribution is to provide a IND-CCA security proof of the AEAD scheme that is functionally equivalent to the one implemented in [Almeida et al., 2019a].

8.1 Description of ChaCha20-Poly1305

The confidentiality of the AEAD scheme is provided by ChaCha20 [Bernstein, 2008a], a stream cipher based on the Salsa family [Bernstein, 2008b]. The encryption ressembles the counter mode of operation for which the block cipher is replaced by a PRF with signature $\mathsf{CC} : \{0,1\}^{256} \times \{0,1\}^{128} \to \{0,1\}^{512}$, i.e. 32-byte key, 16-byte input, and 64-byte output.

As opposed to a MAC scheme, the data intregrity is provided by a *one-time authenticator* named Poly1305, meaning that keys should only be used once. Poly1305 is a universal hash function based on polynomial evaluation [Carter and Wegman, 1977, Bernstein, 2005].

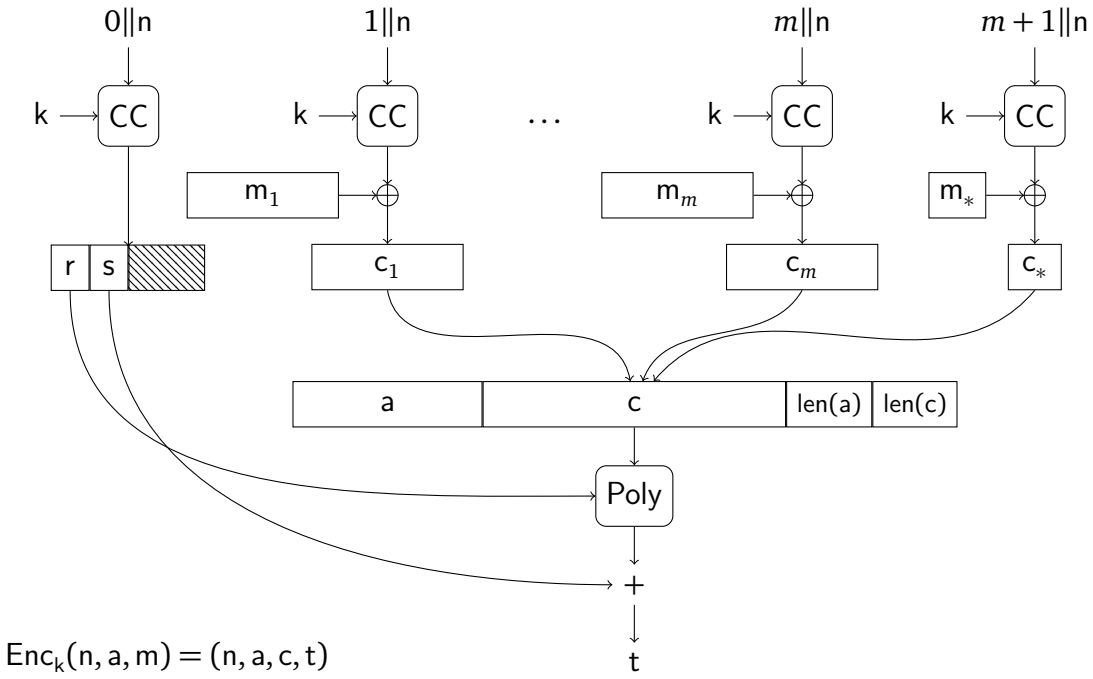


Figure 8.1 – Representation of the encryption algorithm of ChaCha20-Poly1305.

As represented in Figure 8.1, the counter mode for ChaCha20 is used with the initial counter 1, instead of the usual counter 0 which is used for the single use authentication key for Poly1305. The polynomial to be evaluated for Poly1305 is an interpretation of the composition of the associated data and ciphertext as a sequence of polynomial coefficients: integers modulo $2^{130} - 5$. This polynomial is evaluated in $\mathbb{F}_{2^{130}-5}$ at the key $r \in \{0,1\}^{128}$: the first 128-bits of the ChaCha20 block derivation with the counter 0. The final tag of the AEAD scheme is the addition modulo 2^{128} of the polynomial evaluation's output plus the next 128-bits of the same ChaCha20 block derivation with the counter 0. The decryption algorithm is analogous to the encryption, with a check on the validity of the tag.

EasyCrypt

In the following, CC refers to the function described in the standard ChaCha20 that given a ChaCha20 key ($\{0,1\}^{256}$), a nonce ($\{0,1\}^{96}$) and a counter ($\{0,1\}^{32}$) outputs a block ($\{0,1\}^{512}$). The computation of c from m, the nonce N and the ChaCha20 key is called ChaCha (parametrized by the underlying CC function), whereas the authenticated encryption scheme — also parametrized by the CC function — is called CCPoly.

The formalization of the formal security proof of ChaCha20-Poly1305 is based on a general finite type named byte that represents what is commonly called with the same name.

We give here a simplified version, since the objective is not to argue that this is a strong formalization, but to explain as clearly as possible our formal security proof.

```
type key, nonce, byte, tag, associated_data,
  block, polynomial, poly_key.
type message = byte list.
type plaintext =
  nonce * associated data * message.
type ciphertext =
  nonce * associated data * message * tag.
op dkey : key distr.
op dtag : tag distr.
op dpoly_key : poly_key distr.
op bytes_of_block : block → byte list.
op extend : message → block.
op (+^) : block → block → block.
op block_size : int.
op parse_block : block → poly_key * tag.
op topol : associated_data → message → polynomial.
op eval_poly : poly_key → polynomial → tag.
op (+) : tag → tag → tag.
theory C.
  type counter.
  op ofint : int → counter.
end C.
module type CC = {
  proc cc (k:key, n:nonce, c: C.counter) : block
}.
module type FCC = {
  proc * init () : unit
  include CC
}.
```

```
module ChaCha(CC:CC) = {
  proc enc(k:key, n:nonce, p:message)
    : message = {
    var z, c, l;
    c ← [];
    while (p ≠ []) {
      z ← CC.cc(k, n, C.ofint i);
      l ← bytes_of_block (extend p +^ z);
      c ← c ++ take (size p) l;
      p ← drop block_size p;
    }
    return c;
  }
}.
```

```
module CCPoly (CC : CC) : Encryption = {
  proc keygen () : key = {
    var k;
    k ←$ dkey;
    return k;
  }
  proc enc (k : key, nap : plaintext) = {
    var n, a, p, c, b, r, s, t;
    (n,a,p) ← nap;
    c ← ChaCha(CC).enc(k,n,p);
    b ← CC.cc(k,n,C.ofint 0);
    (r,s) ← parse_block b;
    t ← (eval_poly r (topol a c)) + s;
    return (n,a,c,t);
  }
  proc dec (k : key, nact : ciphertext) = {
    var n, a, c, t, b, r, s, p, ret;
    (n,a,c,t) ← nact;
    b ← CC.cc(k,n,C.ofint 0);
    (r,s) ← parse_block b;
    if (t = (eval_poly r (topol a c)) + s) {
      p ← ChaCha(CC).enc(k,n,c);
      ret ← Some p;
    } else { ret ← None; }
    return ret;
  }
}.
```

The formal proof may be found (as of December 15, 2020) in the branch deploy — chachapoly in the GitHub repository at https://github.com/EasyCrypt/easycrypt.git.

8.2 Security of ChaCha20-Poly1305

This work extends the work of [Almeida et al., 2019a] in which Almeida et al. present an optimized implementation of ChaCha20 and Poly1305. Their work uses EasyCrypt to prove in particular the functional correctness of their implementation according to the reference.

ChaCha20-Poly1305 is a slightly modified version of Encrypt-then-MAC with the schemes ChaCha20 and Poly1305. In [Bellare and Namprempre, 2000], the security proof of Encrypt-then-MAC assumes the keys of the cipher and MAC scheme to be sampled independently.

In Chacha20&Poly1305, since the key of Poly1305 is derived from ChaCha20, Bellare and Namprempre's proof cannot be applied in this setting.

In [Procter, 2014], Procter proposes a simple security reduction for ChaCha20-Poly1305 in the nonce-respecting adversarial model. In the effort to formalize their proof, we noticed that some reasoning flaws and corrected them in our proof. We first present the nonce-respecting adversarial model, then we state our verified security bound and present our formal security proof.

8.2.1 Adversarial Model

The security proof of ChaCha20-Poly1305 is placed in the *nonce-respecting* adversarial model. This means that the adversary respects the uniqueness of nonces in its encryption queries. Moreover, there is *no restriction* on nonces asked with decryption queries.

Along with the formalization in EasyCrypt of a nonce-respecting adversary, we include bounds in the number of oracle queries it can make: at most q_e encryption queries and at most q_d decryption queries.

EasyCrypt

We formalize this adversarial model by restricting oracle access using a module named BNR (Bounded Nonce-Respecting) which provide an interface that restricts oracle calls from the adversary. A bounded nonce-respecting adversary is formalized using the module BNR_Adv.

```
op check_plaintext (lenc:nonce list) (p:plaintext) =
  let (n, a, m) = p in ¬ n ∈ lenc ∧ valid_topol a m ∧ size lenc < qenc.
op check_cipher (ndec:int) (c:ciphertext) =
  let (n, a, m, t) = c in valid_topol a m ∧ ndec < qdec.
```

The variable BNR.lenc is the list of all nonces already queried by the adversary and it encodes both nonce respecting and the number of encryption queries. The variable BNR.ndec encodes the number of decryption queries.

```
module BNR (O:CCA_Oracles) = {
  var lenc : nonce list
  var ndec : int
  proc init () = { lenc ← []; ndec ← 0; }
  proc enc (p:plaintext) = {
    var c ← witness;
    if (check_plaintext lenc p) {
      c ← O.enc(p);
      lenc ← p.`1 :: lenc;
    }
    return c;
  }
  proc dec (c:ciphertext) = {
    var p ← None;
    if (check_cipher ndec c) {
      p ← O.dec(c);
      ndec ← ndec + 1;
    }
    return p;
}}.
```

```
module BNR_Adv(A:CCA_Adv,
      O:CCA_Oracles) = {
  proc main() = {
    var b;
    BNR(O).init();
    b ← A(BNR(O)).main();
    return b;
}}.
```

8.2.2 Security Statement

The AEAD scheme named ChaCha20-Poly1305 is secure under the assumption that the ChaCha20 function is indistinguishable from a random function when the ChaCha20 key is randomly sampled and kept secret. A few papers have attacked *reduced round* versions for ChaCha20 (e.g. [Aumasson et al., 2008, Ishiguro, 2012, Shi et al., 2012]). This analysis has not contradicted the security assumption about the PRF security of the *full round* version for ChaCha20.

Theorem 8.2.1. *For any nonce-respecting adversary $\mathscr{A}$ making at most q_d decryption queries and at most q_e encryption queries, its advantage in* CCA *to distinguish ChaCha20-Poly1305 from* Ideal *is bounded by a small probability plus the distinguishing advantage of $\mathscr{B}(\mathscr{A})$ between* CC *and a random function* \$*, where the algorithm $\mathscr{B}(\mathscr{A})^{\mathsf{O}}$ runs once $\mathscr{A}^{\mathsf{CCPoly(O)}}$ and outputs the same result.*

$$\mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(CC),Ideal}}(\mathscr{A}) \leq q_d \cdot \max(p_0, p_1) + q_d \cdot p_1 + \mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CC,\$}}(\mathscr{B}(\mathscr{A}))$$

Let L be the maximum byte length of messages, when we instantiate all the specifications of the standard ChaCha20-Poly1305: $p_0 = 2^{-128}$ and $p_1 = \frac{8\lceil L/16 \rceil}{2^{106}}$.

8.3 Security Proof

First Step - Security Assumption - ChaCha20 is a PRF

The first step of the security proof is to transform ChaCha20-Poly1305 into the same construction built upon a random function instead of CC. This transformation introduces a particular instance of an adversary playing the game to distinguish CC (when the key is sampled randomly) from a truly random function PRF. Therefore, the security of the scheme ChaCha20-Poly1305 relies on the indistinguishability of CC from a random function.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(CC),Ideal}}(\mathscr{A}) &\leq \mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(CC),CCPoly(PRF)}}(\mathscr{A}) + \mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(PRF),Ideal}}(\mathscr{A}) \\
&= \mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CC,PRF}}(\mathscr{B}(\mathscr{A})) + \mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(PRF),Ideal}}(\mathscr{A})
\end{aligned}$$

Here $\mathscr{B}(\mathscr{A})^{\mathsf{O}}$ is defined as the algorithm that, given O, returns the output of $\mathscr{A}^{\mathsf{CCPoly(O)}}$.

Second Step - IND-CCA from IND-CPA and INT-CTXT

The second step is the application of the general lemma 7.3.2 that states that any probability of a CCA game can be bound by the probability of a CPA game plus the INT-CTXT advantage with the same AE scheme.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{dist}}_{\mathsf{CCPoly(PRF),Ideal}}(\mathscr{A}) &= |\Pr[\mathsf{b} \leftarrow \mathsf{CCA}(\mathscr{A}, \mathsf{CCPoly(PRF)}) : \mathsf{b}] - \Pr[\mathsf{b} \leftarrow \mathsf{CCA}(\mathscr{A}, \mathsf{Ideal}) : \mathsf{b}]| \\
&\leq |\Pr[\mathsf{b} \leftarrow \mathsf{CPA}(\mathsf{CPA_CCA_Adv}(\mathscr{A}), \mathsf{CCPoly(PRF)}) : \mathsf{b}] \\
&\quad - \Pr[\mathsf{b} \leftarrow \mathsf{CCA}(\mathscr{A}, \mathsf{Ideal}) : \mathsf{b}]| + \mathsf{Adv}^{\mathsf{int-ctxt}}_{\mathsf{CCPoly(PRF)}}(\mathscr{A})
\end{aligned}$$

The third step modifies the calls to PRF in CCPoly(PRF), in both the games CPA and INT-CTXT. In CPA, the final goal is to prove that CCPoly(PRF) is equivalent to Ideal under the assumption that the adversary is bounded and nonce-respecting. For INT-CTXT, those manipulations are aimed at bounding the event.

Third Step - Modifications of PRF

In CPA and INT-CTXT, all decryption queries output None, whether the tag is valid or not. Therefore, all calls to PRF in the decryption oracle can be delayed in both games without changing the behavior of the adversary. This first modification is done using the lazy-eager library described in Section 2.3.2, and has different implications for CPA and INT-CTXT:

— In CPA, calls can be erased, since they do not affect the event of the probability we are interested in.

— In INT-CTXT, the event of the probability we are interested in is affected by those calls. This modification also delays the update of the event to *after* the end of the interactions with the adversary. This operation is achieved by archiving all decryption queries in a list when interacting with the adversary, and then afterwards, iterating on each decryption query to test if the event is triggered for this particular query.

EasyCrypt

The loop after the interactions with the adversary iterates on each unique nonce that was asked in a decryption query. This is motivated by the fact that one random sampling is involved for all decryption queries that share the same nonce: the first is random, whereas the others are fixed values. When several decryption queries share the same nonce, we choose to group them into only one iteration.

```
op get_nonce (x : α * β * γ * 'd) = x.`1.
op same_nonce (a : α) (x : α * β * γ * 'd) : bool = x.`1 = a.
op compute_s (r : poly_key) (c:ciphertext) = c.`4 − poly_eval r (topol c.`2 c.`3).
```

```
local module G1 (A : CCA_Adv) = {
  var forged : bool
  proc distinguish () = {
    var b, ns, i, n, bl, r, s;
    forged ← false;
    b ← CPA(CPA_CCA_Adv(BNR_Adv(A)),CCPoly(PRF)).main();
    if (size Mem.lc ≤ qdec) {
      ns ← undup (List.map get_nonce Mem.lc);
      i ← 0;
      while (i < size ns) {
        n ← nth witness ns i;
        bl ← PRF.get(n,C.ofint 0);
        (r,s) ← parse_block bl;
        forged ← forged || s ∈ List.map (compute_s r) (List.filter (same_nonce n) Mem.lc)
        i ← i + 1;
    }}
    return b;
}}.
```

We have:

$$\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathsf{CCPoly(PRF)}}(\mathscr{A}) = \Pr[\mathsf{b} \leftarrow \mathsf{G1}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G1.forged}]$$

and

$$\Pr[\mathsf{b} \leftarrow \mathsf{CPA}(\mathsf{CPA\_CCA\_Adv}(\mathscr{A}), \mathsf{CCPoly}(\mathsf{PRF})) : \mathsf{b}] = \Pr[\mathsf{b} \leftarrow \mathsf{G1}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{b}]$$

The second modification of PRF separates its domain into the encryption domain and the authentication domain using the library SplitDom described in Section 2.3.3. The random oracle PRF is split into two random oracles : PRF_1 and PRF_2, as shown in Figure 8.2.

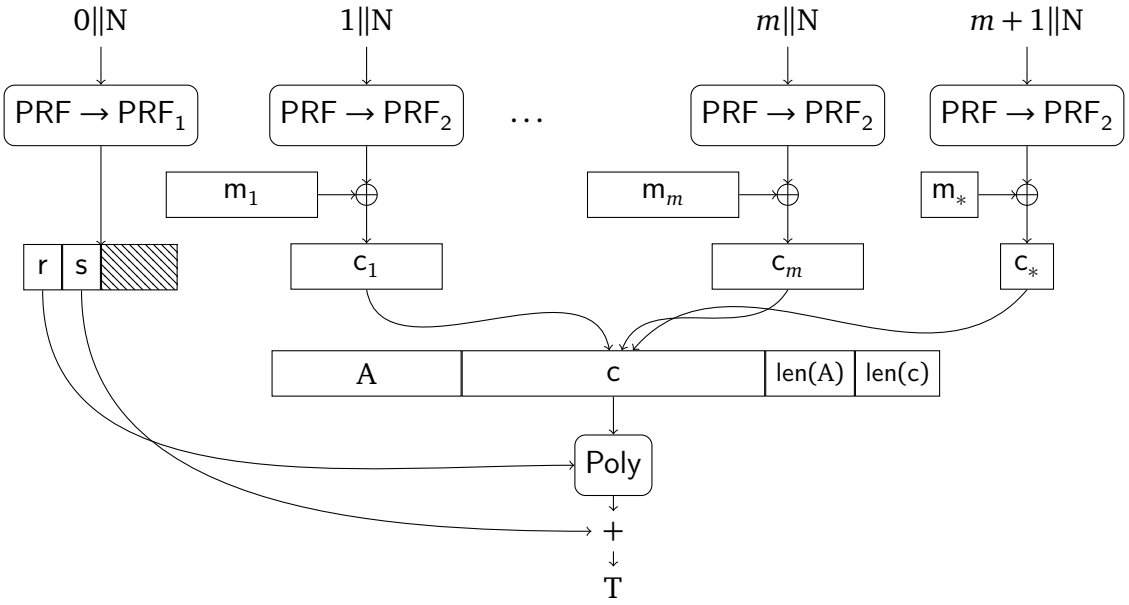


Figure 8.2 – Split Dom of PRF

The third modification of PRF transforms the co-domain of PRF_1 into three parts : the r part, the s part, and the non-used part. We use the library SplitCodom (twice) described in Section 2.3.3, as shown in Figure 8.3 :

1. split the co-domain into the useful part PRF_u and the useless part $\mathsf{PRF}_\emptyset$,
2. split the useful PRF_u into PRF_r and PRF_s.

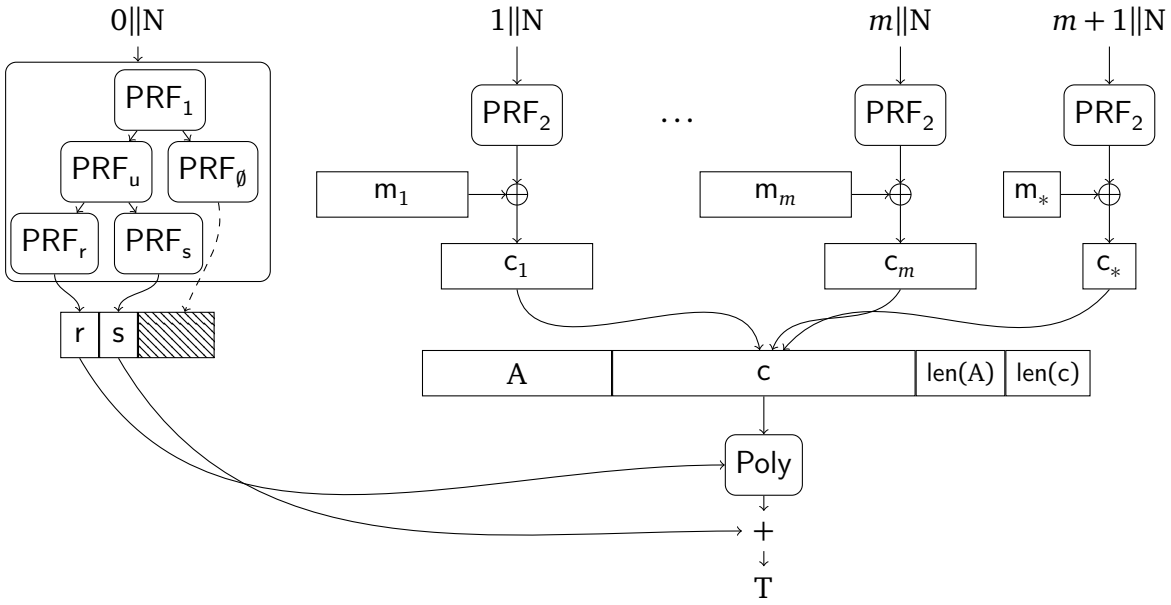


Figure 8.3 – Split Codom of PRF_1

The last PRF modification erases all calls to $\mathsf{PRF}_\emptyset$, since its outputs are never used or shown to the adversary.

EasyCrypt

```
local module G2 (A : CCA_Adv) = {
  module O = {
    proc keygen = CCPoly(PRF2).keygen
    proc enc (k : key, nap : plaintext) = {
      var n, a, p, c, t, r, s;
      c ← ChaCha(PRF2).enc(k,n,p);
      r ← PRFr.get(n,C.ofint 0);
      s ← PRFs.get(n,C.ofint 0);
      t ← (poly_eval r (topol a c)) + s;
      return (n,a,c,t);
    }
    proc dec (k :  key, nact : ciphertext) = { return None; }
  }
  proc distinguish () = {
    var b, ns, i, n, r, s;
    G1.forged ← false;
    PRFs.init(); PRFr.init(); PRF2.init();
    b ← CCA(BNR_Adv(A),O).main();
    if (size Mem.lc ≤ qdec) {
      ns ← undup (List.map get_nonce Mem.lc);
      i ← 0;
      while (i < size ns) {
        n ← nth witness ns i;
        r ← PRFr.get(n,C.ofint 0);
        s ← PRFs.get(n,C.ofint 0);
        G1.forged ← G1.forged || s ∈ List.map (compute_s r) (List.filter (same_nonce n) Mem.lc)
        i ← i + 1;
    }}
    return b;
}}.
```

We have:

$$\mathrm{Adv}^{\mathrm{int-ctxt}}_{\mathrm{CCPoly(PRF)}}(\mathscr{A}) = \Pr[\mathsf{b} \leftarrow \mathsf{G2}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G1.forged}]$$

and

$$\Pr[\mathsf{b} \leftarrow \mathsf{CPA}(\mathsf{CPA\_CCA\_Adv}(\mathscr{A}), \mathsf{CCPoly(PRF)}) : \mathsf{b}] = \Pr[\mathsf{b} \leftarrow \mathsf{G2}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{b}]$$

Fourth Step - Application of the BNR restriction in CPA

A nonce-respecting adversary may only ask fresh nonces for its encryption queries. Therefore, in G2 with a BNR adversary, inputs to PRFs, PRF2 are fresh in the encryption oracle, and calls to those random functions may be replaced by direct random samplings (when we are only interested in the return value of the adversary, not in the value of G1.forged).

When the encryption oracle is queried in G2, say with input (n, a, p) (the secret key being hidden to the adversary), each portion of c is computed by xoring the corresponding portion of p with a uniform random value. Using the bijectivity of the xor operation (⊕), the value of each portion of c may be directly sampled from the uniform distribution.

Furthermore, the output tag t is computed from the values of r and s which are now uniform random samplings. Since the function $a + \cdot$ is bijective (for all values of a) and

the intermediate value s is not revealed to the adversary, the output tag t can be directly sampled and output to the adversary, without using r.

Both modifications of the computation of the output of the encryption oracle prove that under the BNR restrictions, both the encryption oracle and decryption oracle of the game G2 have the same distribution as Ideal in CCA, as represented in Figure 8.4:

$$\begin{aligned}\Pr[b \leftarrow \mathsf{CPA}(\mathsf{CPA\_CCA\_Adv}(\mathscr{A}), \mathsf{CCPoly}(\mathsf{PRF})) : b] &= \Pr[b \leftarrow \mathsf{G2}(\mathscr{A}).\mathsf{distinguish}() : b] \\ &= \Pr[b \leftarrow \mathsf{CCA}(\mathscr{A}, \mathsf{Ideal}) : b]\end{aligned}$$

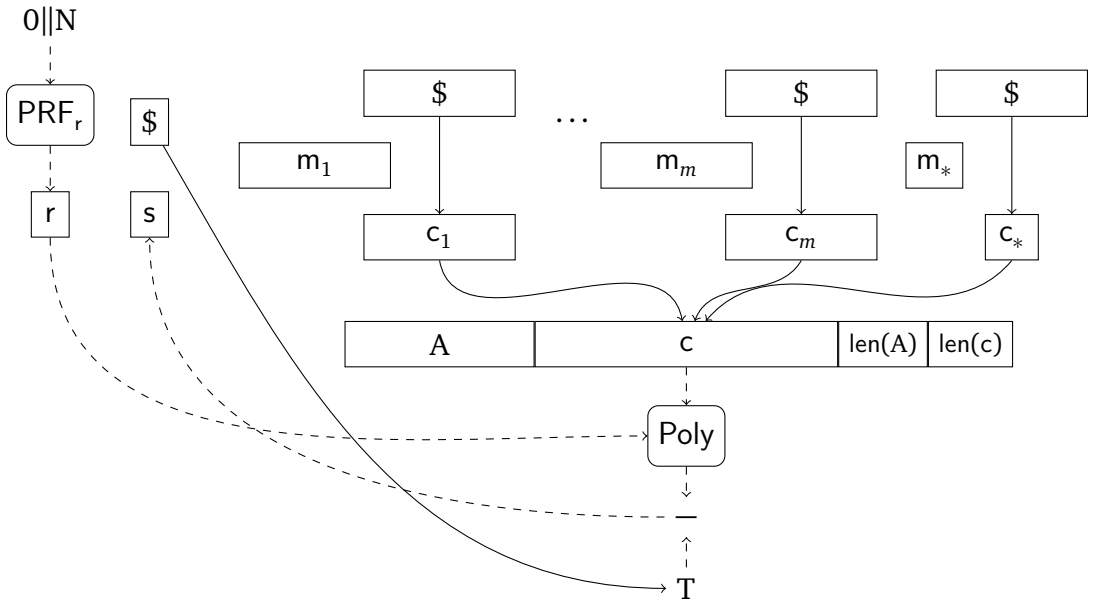


Figure 8.4 – Forth step

We can conclude from this transition that:

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathsf{CCPoly(CC)},\mathsf{Ideal}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{dist}}_{\mathsf{CC},\mathsf{PRF}}(\mathscr{B}(\mathscr{A})) + \Pr[b \leftarrow \mathsf{G2}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G1.forged}]$$

This transition changes the polynomial used in the computation of the event G1.forged. Therefore, the application of the BNR restriction cannot replace calls to the random functions by direct random samplings to formally bound the event G1.forged.

Fifth step - Modify G2 **to bound** G1.forged **using the** BNR **restrictions**

To bound the event G1.forged, we need to be more careful about how we use the bijections and the BNR restrictions. We do not care about calls to PRF_2, since they do not appear in the computation of G1.forged, so they may be replaced by direct random samplings. However, the calls to PRFr, PRFs cannot be replaced by direct random samplings since their is no restriction on the nonces asked to the decryption, we cannot ensure that the occurrences that happen in the loop have fresh inputs or that they remain consistent with oracle outputs.

Remark

The event of finding a polynomial root can be easily bounded when the point of evaluation is sampled uniformly at random. But, when a decryption query is asked with a nonce already used in a previous query, the key r of the polynomial is already set and is not random anymore. Therefore, the probability of the forgery event for this decryption query can only be bounded by 1 (very far from being negligible).

Generally, this absence of randomness that comes from the BNR adversarial model (nonce-repeating in decryption queries) is often forgotten, as in [Procter, 2014]. Our formal analysis has shown this reasoning flaw in their security proof, but this does not lead to any security issue. Indeed, the technique we use (that is presented in the following) allows us to bound the INT-CTXT advantage with a similar bound.

We still want to modify G2 into a game G3 in which all the calls to PRFs, PRFr in G2 are replaced by random samplings. We need to specify the event(s) in G3 that fully capture the event G1.forged in G2. We end up with a disjonction of three events that capture more than the event G1.forged in G2, but since we want an upper bound, this is still relevant.

The first event that we split is for the case when the nonce iterated on (after the interaction with the adversary) has not been asked to the encryption oracle. We call G3.e2 the event that is the conjonction of this case with the event G1.forged.

Left with all the nonces that were asked at some point to the encryption oracle, the event needs further treatment to be bounded. We want to delay all the calls to PRFr in the encryption oracle, so that all the calls in the loop have fresh inputs, and then use the probability of *sampling* a root of a non-zero polynomial. In order to ensure that all polynomials we test in G1.forged are non-zero, we ensure that all outputs (n, a, c, t) of the encryption oracle have never been asked to the decyption oracle by introducing a failure event named G3.e1. The rest of the event G1.forged that is not covered by G3.e1 or G3.e2 is called G3.e3.

EasyCrypt

The event G3.e1 is defined as the event in which the tag t computed in an encryption query in the game G3 is equal to one of the previous inputs for decryption contained in Mem.lc that share the same nonce.

```
local module G2 (A : CCA_Adv) = {

  module O = { (* ··· keygen ··· *)
    proc enc (k : key, nap : plaintext) = {
      (* ··· *)
      r ← PRFr.get(n,C.ofint 0);
      s ← PRFs.get(n,C.ofint 0);
      t ← (poly_eval r (topol a c)) + s;

      return (n,a,c,t);
    }
    proc dec (* ··· *)
  }
```

```
local module G3 (A : CCA_Adv) = {
  var e1 : bool
  var e2 : bool
  var e3 : bool
  var log : (nonce,
    associated_data * message * tag) fmap
  module O = { (* ··· keygen ··· *)
    proc enc (k : key, nap : plaintext) = {
      (* ··· *)
      t ←$ dtag;
      e1 ← e1 || t ∈ map get_tag
          (List.filter (same_nonce n) Mem.lc);
      log[n] ← (a,c,t);
      return (n,a,c,t);
    }
    proc dec (* same as G2 *)
  }
```

```
proc distinguish () = {
  var b, ns, i, n, r, s;
  G1.forged ← false;
  PRFs.init(); PRFr.init(); PRF2.init();
  b ← CCA(BNR_Adv(A),O).main();
  if (size Mem.lc ≤ qdec) {
    ns ← undup (List.map get_nonce Mem.lc);
    i ← 0;
    while (i < size ns) {
      n ← nth witness ns i;
      r ← PRFr.get(n,C.ofint 0);
      s ← PRFs.get(n,C.ofint 0);
      G1.forged ← G1.forged ||
        s ∈ List.map (compute_s r)
          (List.filter (same_nonce n) Mem.lc)
      i ← i + 1;
  }}
  return b;
}}. (* end G2 *)
```

```
proc distinguish () = {
  var b, ns, i, n, r, s;
  e1 ← false; e2 ← false; e3 ← false;
  log ← empty; PRF2.init();
  b ← CCA(BNR_Adv(A),O).main();
  if (size Mem.lc ≤ qdec) {
    ns ← undup (List.map get_nonce Mem.lc);
    i ← 0;
    while (i < size ns) {
      n ← nth witness ns i;
      if (n ∈ BNR.lenc) {
        r ←$ dpoly_key;
        e3 ← e3 ||
          test_poly_key n Mem.lc r (oget log[n])
      } else {
        r ←$ dpoly_key;
        s ←$ dtag;
        e2 ← e2 || s ∈ map (compute_s r)
          (List.filter (same_nonce n) Mem.lc);
      }
      i ← i + 1;
  }}
  return b;
}}. (* end G3 *)
```

```
op valid_topol_cipher (c : ciphertext) = valid_topol c.`2 c.`3.
op test_poly_key (n : nonce) (lc : ciphertext list) (r : poly_key)
     (amt: associated_data * message * tag) =
  let (a,m,t) = amt in
  let p = topol a m in
  let pts =
    List.map (fun (c : ciphertext) ⇒ (topol c.`2 c.`3, c.`4))
      (List.filter (same_nonce n) (List.filter valid_topol_cipher lc)) in
  valid_topol a m ∧
  List.has (fun (pt : polynomial * tag) ⇒
      pt.`1 ≠ p ∧ pt.`2 = t + (poly_eval r pt.`1 − poly_eval r p)) pts.
```

We have:

$$\begin{aligned}\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathrm{CCPoly(PRF)}}(\mathscr{A}) &= \Pr[b \leftarrow \mathrm{G2}(\mathscr{A}).\mathrm{distinguish}() : \mathrm{G1.forged}] \\ &\leq \Pr[b \leftarrow \mathrm{G3}(\mathscr{A}).\mathrm{distinguish}() : \mathrm{G3.e1}] + \\ &\quad \Pr[b \leftarrow \mathrm{G3}(\mathscr{A}).\mathrm{distinguish}() : \mathrm{G3.e2} \vee \mathrm{G3.e3}]\end{aligned}$$

Remark

Let us recall that all values that are added into the list of decryption queries Mem.lc are checked for not having been answered by the encryption oracle up to this point. Therefore, the only remaining possibility to trigger G3.e1 is that the encryption query is asked after the decryption queries. We use the randomness in the encryption oracle to bound G3.e1 in G3.

Remark

The event G3.e1 is larger than only ensuring that the tested polynomial in G1.forged is non-zero. In our formal analysis, we obtain a smaller bound for G3.e1 than for the disjonction G3.e2 ∨ G3.e3. This is sufficient to conclude of the security of the scheme, but someone interested in making the bound more accurate may use the randomness of the parts of c. This would introduce the use of some restriction encryption calls, whether it is q_e or a maximum communication complexity on encryption queries.

At first, we have bounded those events in a very loose way that have yielded in a quadratic bound in the number of oracle calls. The short argument is the following. Since the size of the list Mem.lc is bounded by q_d, and there are at most q_e calls to trigger G3.e1, and q_d calls to trigger either G3.e2 or G3.e3, we can bound the probability of these event by:

— $\Pr[\mathsf{b} \leftarrow \mathsf{G3}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G3.e1}] \leq q_e q_d \cdot p_1$,

— $\Pr[\mathsf{b} \leftarrow \mathsf{G3}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G3.e2}] \leq q_d^2 \cdot p_1$, and

— $\Pr[\mathsf{b} \leftarrow \mathsf{G3}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G3.e3}] \leq q_d^2 \cdot p_0$.

Remark

We noticed that each event does not use the full list Mem.lc, but only the part filtered by the nonce value. Furthermore, a nonce used to trigger each event may not be used to trigger the same event a second time. Therefore, for each event, each nonce may only trigger once each event, reducing the quadratic bound to a linear one. The following explains our argument a little further.

We choose to bound the events G3.e2 and G3.e3 together. They are located in the loop that iterates on each unique nonce that appears in Mem.lc, which have a fixed value after the interaction with the adversary that we name lc. The formula to obtain n from lc and the iteration counter i is n := nth witness ulc i, where ulc := List.undup (List.map get_nonce lc). The probability of each iteration to trigger either G3.e2 or G3.e3 is bounded by

$$\mathsf{size}\ (\mathsf{List.filter}\ (\mathsf{same\_nonce}\ \mathsf{n})\ \mathsf{lc}) \cdot \max(p_0, p_1)$$

where n is the unique nonce iterated on. We use the fel tactic [1] to bound the event G3.e2 ∨ G3.e3 by :

$$\begin{aligned}
&\Pr[\mathsf{b} \leftarrow \mathsf{G3}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G3.e2} \vee \mathsf{G3.e3}] \\
&\leq \sum_{i=0}^{\mathsf{size\ ulc}-1} \mathsf{size}\ (\mathsf{List.filter}\ (\mathsf{same_nonce}\ \mathsf{n})\ \mathsf{lc}) \cdot \max(p_0, p_1) \\
&\leq \max(p_0, p_1) \cdot \sum_{i=0}^{\mathsf{size\ ulc}-1} \mathsf{size}\ (\mathsf{List.filter}\ (\mathsf{same_nonce}\ \mathsf{n})\ \mathsf{lc})
\end{aligned}$$

We prove that $\mathsf{size\ lc} = \sum_{i=0}^{\mathsf{size\ ulc}-1} \mathsf{size}\ (\mathsf{List.filter}\ (\mathsf{same\_nonce}\ \mathsf{n})\ \mathsf{lc})$. Then, using the bound $\mathsf{size\ lc} \leq q_d$, we can bound the full event G3.e2 ∨ G3.e3 in G3 by:

$$\Pr[\mathsf{b} \leftarrow \mathsf{G3}(\mathscr{A}).\mathsf{distinguish}() : \mathsf{G3.e2} \vee \mathsf{G3.e3}] \leq q_d \cdot \max(p_0, p_1)$$

1. Refer to the manual of EasyCrypt for more details.

Sixth step - Bound G3.e1 **in** G3

We would use the same technique to bound G3.e1, but the list that appears in the expression of the bound to trigger G3.e1 is not constant. Indeed, the list Mem.lc has no fixed value during the interation with the adversary, preventing us to use the tactic fel in this context. Therefore, we use another technique that splits the event G3.e1 into q_d parts. The definition of these parts comes with the game G4.

EasyCrypt

In the game G4, we build the list composed of all the couples of tags that are tested to trigger G3.e1. This list is named G4.lt and the way it is constructed enforces the following invariants. For each element (t, t') of G4.lt:

— it is associated to a nonce n,

— a decryption query has been recorded in Mem.lc that contains both n and t', and,

— at the end of the encryption, t is the tag associated to n in G3.log.

```
local module G4 (A : CCA_Adv) = {
  var lt : (tag * tag) list
  module O = { (* ··· keygen ··· *)
    proc enc (k : key, nap : plaintext) = {
      (* ··· *)
      t <$- dtag;
      lt ← List.map (fun c ⇒ (t, get_tag c))
        (List.filter (same_nonce n) Mem.lc)
      return (n,a,c,t);
    }
    proc dec (* same as G3 *)
  }
  proc distinguish () = {
    var b, ns, i, n, r, s;
    lt ← [];
    PRF2.init();
    b ← CCA(BNR_Adv(A),O).main();
    (* the loop is not relevant for G3.e1 *)
    return b;
}}.
```

We prove in EasyCrypt that:

$$\begin{aligned}&\Pr[b \leftarrow G3(\mathscr{A}).\text{distinguish}() : G3.e1]\\ \leq &\Pr\left[b \leftarrow G4(\mathscr{A}).\text{distinguish}() : \exists (t,t') \in G4.lt, t = t'\right]\end{aligned}$$

Using the invariants of G4, we prove that the size of G4.lt is bounded by q_d. Therefore, we can partition the event $\exists (t,t') \in G4.lt, t = t'$ into q_d parts, where each part is indexed by $0 \leq i < q_d$ and says that this event is triggered at the position i of the list. The library EventPartitioning allows us to use apply this partitioning and obtain the following result:

$$\begin{aligned}&\Pr\left[b \leftarrow G4(\mathscr{A}).\text{distinguish}() : \exists (t,t') \in G4.lt, t = t'\right]\\ \leq &\sum_{i=0}^{q_d-1} \Pr[b \leftarrow G4(\mathscr{A}).\text{distinguish}() : (\text{nth witness } G4.lt\ i).`1 = (\text{nth witness } G4.lt\ i).`2]\end{aligned}$$

We have defined another game that only triggers the bad event for the i-th position in the list. Then we have proven the equivalence of this game and G4 when in relation to this

event. Then we proved that for all indexes, the probability in this new game to set bad is bounded by p_1:

$$\forall i, \Pr[\mathsf{b} \leftarrow \mathsf{G4}(\mathscr{A}).\mathsf{distinguish}() : (\mathsf{nth\ witness\ G4.lt\ i}).`1 = (\mathsf{nth\ witness\ G4.lt\ i}).`2] \leq p_1$$

Finally, we have:

$$\Pr\Big[\mathsf{b} \leftarrow \mathsf{G4}(\mathscr{A}).\mathsf{distinguish}() : \exists (\mathsf{t}, \mathsf{t}') \in \mathsf{G4.lt}, \mathsf{t} = \mathsf{t}'\Big] \leq \sum_{i=0}^{q_d - 1} p_1 \leq q_d \cdot p_1$$

which concludes the security proof of ChaCha20-Poly1305.

Conclusion

In this thesis, we formalize and prove concrete security bounds for several cryptographic constructions: a standard MAC scheme named CMAC [Baritel-Ruet et al., 2018], a standard hash function named SHA-3 [Almeida et al., 2019b], and a standard nonce-based AEAD scheme named ChaCha20-Poly1305.

For the last two standards, the formal security proof is accompanied by an implementation [Almeida et al., 2019a, Almeida et al., 2019b] in Jasmin [Almeida et al., 2017] that includes formal functional correctness, an optimized implementation, and protection against some side-channel attacks. Even though I did not participate in the Jasmin developments, this collaboration proposes an approach that not only delivers an optimized version of the standard that is functionally secure and side-channel resistant but also links the functional correctness with the formal security proof.

In a related work [Protzenko et al., 2019], Protzenko et al. expose EverCrypt and fast implementation of a library of cryptographic primitives in C and/or assembly code that includes functional correctness and side-channel resistance. It is based on F* [Swamy et al., 2016] and HACL* [Zinzindohoué et al., 2017] and represents an extensive work. It is an important step in the direction of high-assurance high-speed cryptography, but the link with formal security proofs is still missing. The combination of EasyCrypt and Jasmin is (for now) the only one that provides end-to-end security to implement cryptographic schemes.

Standardization

The final product of the work in this thesis has many advantages for cryptography and the process of standardization, but its development has some downsides to consider when such a formalization effort is considered. Computer security is generally about evaluating the risks versus the cost to deploy a security solution. I do not argue that every user should use only formally proven cryptographic constructions. My argument is that, in a standardization process, which aims at providing the best cryptosystems in terms of security and/or efficiency, organizers should consider including a phase in which formal security proofs are required so that no more cases like OCB2 [Inoue et al., 2019] happen in the future.

To emphasis this argument a little further, in each security proof that we formalized from an existing pen-and-paper proof, we spotted a few fallacies in the reasoning. They do not question the overall security of the constructions, since for each, we found another formal argument to make the security bound hold. However, no construction is free from such a fallacy to effectively harm its security, unless they are formally verified.

Indeed, once a formal security proof is provided, all reasoning steps have been carefully

described and related to simple axioms and local security assumptions. There is no leap of faith to convince of the correctness of a formal security proof, except for the correctness of the proof assistant itself. In my point of view, the level of detail and the verification provided by the tool both increase the trustworthiness of security proof that are formalized.

Furthermore, the verification provided by the proof assistant allows the costs of verifying each reasoning to be offloaded. This cognitive discharge allows scientific reviewers to redirect their attention from checking the accuracy of evidence to the definitions of security, from adversarial models and security assumptions. Since cognitive effort is limited, this improves the quality of criticism, which in turn improves the quality of the peer-review process. In the long term, security proofs will hopefully be more critical, more accurate and more practical for real-world applications.

The high value of formal proofs of security must be mitigated because the cost of the formalization effort is also high. The time and work needed to formalize and prove the security of a cryptographic construction are much more expensive than for its pen-and-paper equivalent, even for an expert. The EasyCrypt user needs to know how to communicate to the proof assistant the properties he needs to prove and how he is going to do it. Such work requires the understanding and mastery of both cryptography and formal proofs, two different high-level fields in each of which a research career is commonplace. The lack of such experts in both fields is one of the challenges of producing libraries composed exclusively of formally secure cryptographic implementations.

Perspectives

EasyCrypt is very expressive in terms of the results it can prove. It should be possible to extend the domain of applications of EasyCrypt, e.g. post-quantum cryptography, and to model stronger security properties, e.g. other side-channel resistances. Its application domain is not restricted to cryptography or computer security and may include the study of more general randomized algorithms, provided that the cost of formalization is worth the effort. Indeed, formalization effort remains an issue and future work may also include the reduction of its cost, e.g. by including more automation in EasyCrypt.

For the implementation of cryptographic standards, the link between formal security proofs and secure fast implementation that includes formal functional security and resistance against side-channel attacks should be encouraged and reinforced. A first approach has been implemented using EasyCrypt and Jasmin, and future work may extend each framework, their link, or may introduce alternatives.

List of Figures

Bibliography

[Coq, 1984] (1984). The coq proof assistant. https://coq.inria.fr/. Accessed: December 15, 2020.

[Lea, 2012] (2012). Lean, theorem prover. https://leanprover.github.io/. Accessed: December 15, 2020.

[CAE, 2013] (2013). CAESAR competition. https://competitions.cr.yp.to/caesar.html. Accessed: December 15, 2020.

[Affeldt et al., 2007] Affeldt, R., Tanaka, M., and Marti, N. (2007). Formal proof of provable security by game-playing in a proof assistant. In *International Conference on Provable Security*, pages 151–168. Springer.

[Al Fardan and Paterson, 2013] Al Fardan, N. J. and Paterson, K. G. (2013). Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE.

[Almeida et al., 2017] Almeida, J. B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., and Strub, P.-Y. (2017). Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823. ACM.

[Almeida et al., 2019a] Almeida, J. B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., and Strub, P.-Y. (2019a). The last mile: High-assurance and high-speed cryptographic implementations. *arXiv preprint arXiv:1904.04606*.

[Almeida et al., 2019b] Almeida, J. B., Baritel-Ruet, C., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Oliveira, T., Stoughton, A., and Strub, P.-Y. (2019b). Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1622.

[Aumasson et al., 2008] Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., and Rechberger, C. (2008). New features of latin dances: analysis of salsa, chacha, and rumba. In *International Workshop on Fast Software Encryption*, pages 470–488. Springer.

[Backes et al., 2012] Backes, M., Barthe, G., Berg, M., Grégoire, B., Kunz, C., Skoruppa, M., and Zanella-Béguelin, S. (2012). Verified security of merkle-Damgård. pages 354–368.

[Baritel-Ruet et al., 2018] Baritel-Ruet, C., Dupressoir, F., Fouque, P.-A., and Grégoire, B. (2018). Formal security proof of cmac and its variants. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 91–104. IEEE.

[Barthe et al., 2010] Barthe, G., Grégoire, B., and Béguelin, S. Z. (2010). Programming language techniques for cryptographic proofs. In *International Conference on Interactive Theorem Proving*, pages 115–130. Springer.

[Barthe et al., 2011a] Barthe, G., Grégoire, B., Heraud, S., and Béguelin, S. Z. (2011a). Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer.

[Barthe et al., 2011b] Barthe, G., Grégoire, B., Lakhnech, Y., and Béguelin, S. Z. (2011b). Beyond provable security verifiable ind-cca security of oaep. In *Cryptographers' Track at the RSA Conference*, pages 180–196. Springer.

[Barthe et al., 2009] Barthe, G., Grégoire, B., and Zanella Béguelin, S. (2009). Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 90–101.

[Bellare, 1997] Bellare, M. (1997). Practice-oriented provable-security. In *International Workshop on Information Security*, pages 221–231. Springer.

[Bellare et al., 1997] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. (1997). A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE.

[Bellare et al., 1998a] Bellare, M., Desai, A., Pointcheval, D., and Rogaway, P. (1998a). Relations among notions of security for public-key encryption schemes. In *Annual International Cryptology Conference*, pages 26–45. Springer.

[Bellare et al., 1998b] Bellare, M., Garay, J. A., and Rabin, T. (1998b). Fast batch verification for modular exponentiation and digital signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 236–250. Springer.

[Bellare et al., 1994] Bellare, M., Kilian, J., and Rogaway, P. (1994). The security of cipher block chaining. In *Advances in Cryptology—CRYPTO'94*, pages 341–358. Springer.

[Bellare and Namprempre, 2000] Bellare, M. and Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 531–545. Springer.

[Bellare et al., 2005] Bellare, M., Pietrzak, K., and Rogaway, P. (2005). Improved security analyses for CBC MACs. In *Crypto*, volume 3621, pages 527–545. Springer.

[Bellare and Rogaway, 1994] Bellare, M. and Rogaway, P. (1994). Optimal asymmetric encryption. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 92–111. Springer.

[Bellare and Rogaway, 1995] Bellare, M. and Rogaway, P. (1995). Random oracles are practical: a paradigm for designing e cient protocols. *Proc. of the 1st CCS*, pages 62–73.

[Bellare and Rogaway, 2006] Bellare, M. and Rogaway, P. (2006). The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 409–426. Springer.

[Bernstein, 2005] Bernstein, D. J. (2005). The poly1305-aes message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer.

[Bernstein, 2008a] Bernstein, D. J. (2008a). Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5.

[Bernstein, 2008b] Bernstein, D. J. (2008b). The salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer.

[Bertoni et al., 2008] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2008). On the indifferentiability of the sponge construction. pages 181–197.

[Bertot, 2006] Bertot, Y. (2006). Coq in a hurry. *arXiv preprint cs/0603118*.

[Black and Rogaway, 2005] Black, J. and Rogaway, P. (2005). CBC MACs for arbitrary-length messages: The three-key constructions. *Journal of Cryptology*, 18(2):111–131.

[Blanchet, 2013] Blanchet, B. (2013). Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of Security Analysis and Design VII*, pages 54–87. Springer.

[Boneh and Franklin, 2003] Boneh, D. and Franklin, M. (2003). Identity-based encryption from the weil pairing. *SIAM journal on computing*, 32(3):586–615.

[Bosselaers and Preneel, 1995] Bosselaers, A. and Preneel, B. (1995). *Integrity Primitives for Secure Information Systems: Final Ripe Report of Race Integrity Primitives Evaluation*. Number 1007. Springer Science & Business Media.

[Carter and Wegman, 1977] Carter, J. L. and Wegman, M. N. (1977). Universal classes of hash functions (extended abstract. In *in STOC'77: Proceedings of the ninth annual ACM symposium on Theory of computing*. Citeseer.

[Chakraborty et al., 2015] Chakraborty, D., Hernandez-Jimenez, V., and Sarkar, P. (2015). Another look at xcb. *Cryptography and Communications*, 7(4):439–468.

[Cogliati et al., 2018] Cogliati, B., Dodis, Y., Katz, J., Lee, J., Steinberger, J. P., Thiruvengadam, A., and Zhang, Z. (2018). Provable security of (tweakable) block ciphers based on substitution-permutation networks. pages 722–753.

[Coron, 2002] Coron, J.-S. (2002). Optimal security proofs for pss and other signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 272–287. Springer.

[Coron et al., 2005] Coron, J.-S., Dodis, Y., Malinaud, C., and Puniya, P. (2005). Merkle-Damgård revisited: How to construct a hash function. pages 430–448.

[Coron and Naccache, 1999] Coron, J.-S. and Naccache, D. (1999). On the security of rsa screening. In *International Workshop on Public Key Cryptography*, pages 197–203. Springer.

[Dachman-Soled et al., 2016] Dachman-Soled, D., Katz, J., and Thiruvengadam, A. (2016). 10-round feistel is indifferentiable from an ideal cipher. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 649–678. Springer.

[Dachman-Soled et al., 2016] Dachman-Soled, D., Katz, J., and Thiruvengadam, A. (2016). 10-round Feistel is indifferentiable from an ideal cipher. pages 649–678.

[Daemen and Rijmen, 1999] Daemen, J. and Rijmen, V. (1999). Aes proposal: Rijndael.

[Daemen and Rijmen, 2013] Daemen, J. and Rijmen, V. (2013). *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

[Dai and Steinberger, 2016a] Dai, Y. and Steinberger, J. (2016a). Indifferentiability of 8-round feistel networks. In *Annual International Cryptology Conference*, pages 95–120. Springer.

[Dai and Steinberger, 2016b] Dai, Y. and Steinberger, J. P. (2016b). Indifferentiability of 8-round Feistel networks. pages 95–120.

[Dalal et al., 2010] Dalal, N., Shah, J., Hisaria, K., and Jinwala, D. (2010). A comparative analysis of tools for verification of security protocols. *International Journal of Communications, Network and System Sciences*, 3(10):779.

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

[Dworkin, 2015] Dworkin, M. J. (2015). Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report.

[Dworkin, 2016] Dworkin, M. J. (2016). Recommendation for block cipher modes of operation: The CMAC mode for authentication. *Special Publication (NIST SP)-800-38B*.

[Ehrsam et al., 1978] Ehrsam, W. F., Meyer, C. H., Smith, J. L., and Tuchman, W. L. (1978). Message verification and transmission error detection by block chaining. US Patent 4,074,066.

[Galindo, 2005] Galindo, D. (2005). Boneh-franklin identity based encryption revisited. In *International Colloquium on Automata, Languages, and Programming*, pages 791–802. Springer.

[Garman et al., 2016] Garman, C., Green, M., Kaptchuk, G., Miers, I., and Rushanan, M. (2016). Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 655–672.

[Gaži and Maurer, 2009] Gaži, P. and Maurer, U. (2009). Cascade encryption revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 37–51. Springer.

[Goldwasser and Micali, 1984] Goldwasser, S. and Micali, S. (1984). Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299.

[Goldwasser et al., 1988] Goldwasser, S., Micali, S., and Rivest, R. L. (1988). A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308.

[Gordon, 1988] Gordon, M. J. (1988). Hol: A proof generating system for higher-order logic. In *VLSI specification, verification and synthesis*, pages 73–128. Springer.

[Hales, 2008] Hales, T. C. (2008). Formal proof. *Notices of the AMS*, 55(11):1370–1380.

[Herzog, 2005] Herzog, J. (2005). A computational interpretation of dolev–yao adversaries. *Theoretical Computer Science*, 340(1):57–81.

[Impagliazzo and Rudich, 1989] Impagliazzo, R. and Rudich, S. (1989). Limits on the provable consequences of one-way permutations. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 44–61. ACM.

[Inoue et al., 2019] Inoue, A., Iwata, T., Minematsu, K., and Poettering, B. (2019). Cryptanalysis of ocb2: Attacks on authenticity and confidentiality. *IACR Cryptology ePrint Archive*, 2019:311.

[Ishiguro, 2012] Ishiguro, T. (2012). Modified version of" latin dances revisited: New analytic results of salsa20 and chacha". *IACR Cryptology ePrint Archive*, 2012:65.

[Isobe et al., 2013] Isobe, T., Ohigashi, T., Watanabe, Y., and Morii, M. (2013). Full plaintext recovery attack on broadcast rc4. In *International Workshop on Fast Software Encryption*, pages 179–202. Springer.

[Iwata and Kurosawa, 2003] Iwata, T. and Kurosawa, K. (2003). OMAC: One-key CBC MAC. In *FSE*, volume 2887, pages 129–153. Springer.

[Iwata et al., 2012] Iwata, T., Ohashi, K., and Minematsu, K. (2012). Breaking and repairing GCM security proofs. In *Crypto*, volume 7417, pages 31–49. Springer.

[Jha and Nandi, 2016] Jha, A. and Nandi, M. (2016). Revisiting structure graphs: Applications to cbc-mac and emac. *Journal of Mathematical Cryptology*, 10(3-4):157–180.

[Jonsson et al., 2001] Jonsson, B., Yi, W., and Larsen, K. G. (2001). Probabilistic extensions of process algebras. In *Handbook of process algebra*, pages 685–710. Elsevier.

[Kakvi and Kiltz, 2012] Kakvi, S. A. and Kiltz, E. (2012). Optimal security proofs for full domain hash, revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 537–553. Springer.

[Katz and Lindell, 2014] Katz, J. and Lindell, Y. (2014). *Introduction to modern cryptography*. CRC press.

[Katz and Yung, 2000a] Katz, J. and Yung, M. (2000a). Complete characterization of security notions for probabilistic private-key encryption. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 245–254.

[Katz and Yung, 2000b] Katz, J. and Yung, M. (2000b). Unforgeable encryption and chosen ciphertext secure modes of operation. In *International Workshop on Fast Software Encryption*, pages 284–299. Springer.

[Kaufman et al., 1995] Kaufman, C., Perlman, R., and Speciner, M. (1995). Network security: private communication in a public world.

[Kayser, 2007] Kayser, R. F. (2007). Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (sha-3) family. *Federal Register*, 72(212):62.

[Kerckhoffs, 1883] Kerckhoffs, A. (1883). La cryptographic militaire. *Journal des sciences militaires*, pages 5–38.

[Krawczyk, 2005] Krawczyk, H. (2005). Hmqv: A high-performance secure diffie-hellman protocol. In *Annual International Cryptology Conference*, pages 546–566. Springer.

[Langley et al., 2016] Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and Josefsson, S. (2016). Chacha20-poly1305 cipher suites for transport layer security (tls). *RFC 7905*, (10).

[Leurent and Peyrin, 2019] Leurent, G. and Peyrin, T. (2019). From collisions to chosen-prefix collisions application to full sha-1. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 527–555. Springer.

[Maurer et al., 2004] Maurer, U. M., Renner, R., and Holenstein, C. (2004). Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. pages 21–39.

[McGrew and Viega, 2004a] McGrew, D. and Viega, J. (2004a). The galois/counter mode of operation (gcm). *Submission to NIST Modes of Operation Process*, 20.

[McGrew and Fluhrer, 2007] McGrew, D. A. and Fluhrer, S. R. (2007). The security of the extended codebook (xcb) mode of operation. In *International Workshop on Selected Areas in Cryptography*, pages 311–327. Springer.

[McGrew and Viega, 2004b] McGrew, D. A. and Viega, J. (2004b). The security and performance of the galois/counter mode (GCM) of operation. In *Indocrypt*, volume 3348, pages 343–355. Springer.

[Menezes, 2007] Menezes, A. (2007). Another look at hmqv. *Mathematical Cryptology JMC*, 1(1):47–64.

[Morris et al., 1977] Morris, R., Sloane, N., and Wyner, A. D. (1977). Assessment of the national bureau of standards proposed federal data encryption standard. *Cryptologia*, 1(3):281–291.

[Nandi, 2014] Nandi, M. (2014). Xls is not a strong pseudorandom permutation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 478–490. Springer.

[Naor and Yung, 1990] Naor, M. and Yung, M. (1990). Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 427–437.

[Nir and Langley, 2015] Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. RFC 7539.

[Oechslin, 2003] Oechslin, P. (2003). Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer.

[Owre et al., 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer.

[Paulson, 1994] Paulson, L. C. (1994). *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media.

[Perlman et al., 2016] Perlman, R., Kaufman, C., and Speciner, M. (2016). *Network security: private communication in a public world*. Pearson Education India.

[Preneel, 1998] Preneel, B. (1998). Cryptographic primitives for information authentication—state of the art. In *State of the Art in Applied Cryptography*, pages 49–104. Springer.

[Procter, 2014] Procter, G. (2014). A security analysis of the composition of chacha20 and poly1305. *IACR Cryptology ePrint Archive*, 2014:613.

[Protzenko et al., 2019] Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., et al. (2019). Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 634–653.

[Rackoff and Simon, 1991] Rackoff, C. and Simon, D. R. (1991). Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer.

[Regenscheid, 2016] Regenscheid, A. R. (2016). Nist cryptographic standards and guidelines development process. Technical report.

[Ristenpart and Rogaway, 2007] Ristenpart, T. and Rogaway, P. (2007). How to enrich the message space of a cipher. In *International Workshop on Fast Software Encryption*, pages 101–118. Springer.

[Rivest et al., 1983] Rivest, R. L., Shamir, A., and Adleman, L. (1983). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26(1):96–99.

[Rogaway, 2004a] Rogaway, P. (2004a). Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 16–31. Springer.

[Rogaway, 2004b] Rogaway, P. (2004b). Nonce-based symmetric encryption. In *International Workshop on Fast Software Encryption*, pages 348–358. Springer.

[Rogaway, 2009] Rogaway, P. (2009). Practice-oriented provable security and the social construction of cryptography. *Unpublished essay*.

[Shannon, 1949] Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715.

[Shi et al., 2012] Shi, Z., Zhang, B., Feng, D., and Wu, W. (2012). Improved key recovery attacks on reduced-round salsa20 and chacha. In *International Conference on Information Security and Cryptology*, pages 337–351. Springer.

[Shoup, 1998] Shoup, V. (1998). *Why chosen ciphertext security matters*, volume 57. IBM TJ Watson Research Center.

[Shoup, 2002] Shoup, V. (2002). Oaep reconsidered. *Journal of Cryptology*, 15(4):223–249.

[Shoup, 2004] Shoup, V. (2004). Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332.

[Song et al., 2006] Song, J., Poovendran, R., Lee, J., and Iwata, T. (2006). The AES-CMAC algorithm. Technical report, RFC 4493, June.

[Stern et al., 2002] Stern, J., Pointcheval, D., Malone-Lee, J., and Smart, N. P. (2002). Flaws in applying proof methodologies to signature schemes. In *Annual International Cryptology Conference*, pages 93–110. Springer.

[Stevens et al., 2017] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer.

[Swamy et al., 2016] Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., et al. (2016). Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270.

[Vaudenay, 2000] Vaudenay, S. (2000). Decorrelation over infinite domains: the encrypted CBC-MAC case. In *Selected Areas in Cryptography*, volume 2012, pages 189–201. Springer.

[Yao, 1982] Yao, A. C. (1982). Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 80–91. IEEE.

[Zinzindohoué et al., 2017] Zinzindohoué, J.-K., Bhargavan, K., Protzenko, J., and Beurdouche, B. (2017). Hacl*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security (CCS)*.