ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering
Master's Degree in Computer Science

THESIS' TITLE

Thesis in Cryptography

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Correlatore:
Chiar.mo Prof.
NOME CORRELATORE

Presentata da:
Michele Dinelli

Session IV
Academic Year 2024/2025

To my Mum and Dad

Abstract

Reduction-based proofs are the standard technique for establishing the security of cryptographic constructions in the computational model. However, encoding such proofs within formal systems remains difficult, as they involve probabilistic computation, adversarial interaction, and explicit reasoning about computational efficiency. This thesis studies the use of programming-language techniques for formalizing cryptographic proofs in the computational model. We work within λ**BLL**, a probabilistic lambda calculus whose graded type system characterizes probabilistic polynomial-time computation. To support reasoning about program correctness and cryptographic experiments, we introduce an assertion-annotated extension of λ**BLL** inspired by Hoare logic, together with a lifting procedure that embeds standard typing judgments into the enriched system. Within this framework, we formalize the security of Message Authentication Codes (MACs) constructed from pseudorandom functions by proving existential unforgeability under chosen-message attacks (EUF-CMA). Overall, this work shows that λ**BLL** is expressive enough to represent classical cryptographic reductions while statically enforcing polynomial-time resource bounds. By extending the calculus with Hoare-style assertions, we enable reasoning about invariants during program execution and about stateful references that arise in cryptographic constructions such MACs built from pseudorandom functions. This demonstrates the potential of the framework as a foundation for the formal verification of proofs in computational cryptography.

Sommario

Le prove per riduzione sono la tecnica standard per stabilire la sicurezza dei protocolli crittografici nel modello computazionale. Tuttavia, la loro formalizzazione presenta notevoli difficoltà, dovute alla necessità di modellare computazioni probabilistiche, interazioni avversarie e vincoli di efficienza computazionale. In questa tesi si esplora l'applicazione di tecniche proprie dei linguaggi di programmazione per formalizzare tali dimostrazioni. Il lavoro si basa su λ**BLL**, un lambda calcolo probabilistico il cui sistema di tipi garantisce per costruzione che ogni computazione avvenga in tempo polinomiale. λ**BLL** è stato esteso definendo un nuovo sistema di tipi ispirato alla logica di Hoare ed è stata definita una procedura di lifting per integrare i giudizi di tipizzazione standard nel nuovo sistema. L'efficacia del framework è stata testata formalizzando un esperimento crittografico relativo ai Message Authentication Codes (MACs) ottenuti tramite funzioni pseudorandom, e ne è stata dimostrata la sicurezza in termini di existential unforgeability against chosen message attacks (EUF-CMA). Il lavoro dimostra che λ**BLL** è sufficientemente espressivo per gestire riduzioni crittografiche complesse, garantendo staticamente il rispetto dei limiti di tempo polinomiale. L'introduzione della logica di Hoare permette inoltre di affrontare ragionamenti articolati su invarianti relative allo stato di esecuzione dei termini, ponendo una base per lo sviluppo di strumenti di verifica formale in crittografia computazionale.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography has historically been regarded as form of art, grounded in humanity's need to communicate while concealing information from unauthorized parties. For much of its history, research in cryptography was conducted under conditions of secrecy, primarily because cryptographic techniques conferred substantial military and political advantages. As a result, developments in cryptography were often responses to events or activities that were either secret at the time or became public only much later [Sin99].

The history of cryptography is a continuous battle between codemakers (cipher makers) and codebreakers (cipher breakers). The creation of strong cipher, as well as the decryption of existing ones, heavily relied on creativity, inspiration, and occasional strokes of genius [KL14]. In the absence of a rigorous theoretical foundation, the discipline lacked formal security definitions and systematic criteria to assess whether a given cryptographic construction was secure or vulnerable. These characteristics reinforced the perception of cryptography as an art rather than a formal science for many centuries. However, between the 1970s and 1980s, the spread of the internet and the availability of affordable digital hardware brought cryptography into everyday life. At the same time, theoretical advances [DH22, RSA78, GM84] in information theory and computer science laid the foundations for provably secure cryptosystems, transforming cryptography into a scientific discipline.

Cryptography may thus be seen as a science derived from art, an appealing definition, but one that does not fully capture what cryptography has become today. Modern cryptography is a field at the intersection of mathematics and computer science, concerned with techniques for ensuring data integrity, exchanging secret keys, authenticating users, enabling electronic auctions and elections, performing secure distributed computations, implementing digital cash, establishing server identities, and much more [KL14]. Indeed the scope of modern cryptography is very broad, in contrast with the classical cryptography which is mainly concerned secure communication over insecure channels [GW96].

While cryptography yields concrete mechanisms for addressing these challenges, such mechanisms are more appropriately regarded as applications of the discipline rather than the discipline itself. At its foundation, cryptography aims to resolve precisely formulated problems by abstracting away many of the complexities of real-world environments. This abstraction is essential, as cryptographic guarantees can only be established relative to explicitly specified adversarial models and clear assumptions. Although this may initially seem restrictive, the reliance on rigorous security definitions, formally stated assumptions, and mathematically sound proofs enables cryptography to provide robust and dependable security guarantees [Ros]. This constitutes one of the main accomplishments of theoretical computer science [GW96].

At the foundation of the science of cryptography lies the notion of security. One may ask under which conditions a given cryptographic primitive or protocol can be regarded as secure, and what the term *secure* precisely denotes in this context. The definition of security depends on the underlying formal framework, with two principal paradigms being the computational model and the symbolic model.

In the computational model, adversaries are modeled as probabilistic polynomial-time Turing machines. Cryptographic operations are represented as functions on binary strings, and security properties are expressed in terms of the probability that an adversary succeeds together with the computa-

tional complexity of the attack. Security is achieved when this probability is negligible. By contrast, the symbolic model abstracts away from complexity and probability, and instead relies on classical verification techniques. This makes it particularly suitable for detecting design flaws in cryptographic protocols, and it has led to most of the automated verification tools available today. However, security guarantees obtained in the symbolic model are generally weaker than those provided by the computational model.

While the symbolic model has been widely studied and integrated into formal verification frameworks, the computational model has not reached the same level of development in this respect. A promising direction for enabling the use of classical verification techniques in the computational setting is to apply theories of program equivalence to programming languages designed to capture the central complexity-theoretic notion of probabilistic polynomial-time computation. Within such languages, the notion of computational indistinguishability [Gol01] can be reinterpreted as a form of observational equivalence. This reinterpretation opens the way to studying computational indistinguishability using established tools from programming language theory, such as logical relations.

The objective of this work is to employ a specific language, namely λ**BLL** [LGG24], whose main features are summarized in Chapter 5, in order to formulate and formally prove an important cryptographic result: the security of fixed-length Message Authentication Codes (MACs), in particular existential unforgeability under chosen-message attacks. To this end, we introduce a Hoare logic that extends the type system of λ**BLL** with decorated terms, annotated with propositions that carry stateful meaning with respect to the execution of λ**BLL** programs. By reasoning within this enriched typing system, we also define a lifting procedure that maps typing judgments from the standard λ**BLL** system to the decorated system with Hoare assertions. Furthermore, we extend the set of function symbols and equations of λ**BLL** to reason about equivalences within the assertion-decorated system. In doing so, we provide a fully formal proof of MAC security within the

λ**BLL** framework.

To place this work in context, Chapter 2 presents the necessary background on computational cryptography, in particular on information-theoretic security, pseudorandom generators, pseudorandom functions, and MACs. Chapter 3 briefly surveys automated formal verification tools in the computational model, together with their underlying logics, in order to offer a perspective on the current state of the art in formal verification in the computational setting. In Chapter 4 we review the lambda calculus and its main properties, and in Section 4.2 we recall the simply typed lambda calculus, which provides the foundation for understanding and reasoning about λ**BLL**. Chapter 5 revisits the main features of λ**BLL** and presents its extension via Hoare logic and a new set of typing rules, together with a technique for lifting λ**BLL**-terms into the Hoare-logic extension, yielding assertion-typed terms. Finally, in Chapter 6 we formalize, within the λ**BLL** framework, the proof of MAC security introduced in Section 2.4.1.

Chapter 2

Computational Cryptography

Modern cryptography is concerned with providing security guarantees for information systems, ensuring that malicious users cannot compromise their intended behaviour by corrupting data, stealing information, or disrupting system operation. Security in cryptography is therefore closely tied to formal properties defined and analysed using the language of mathematics. To study and reason about these security properties, two main lines of research exist: the computational model [GW96, KL14] and the symbolic model, also known as the Dolev–Yao model [DY83].

Computational Model The computational model is tightly linked to complexity theory. Within this framework, notions such as secure encryption, unforgeable digital signatures have been placed on solid theoretical foundations. In the computational model, cryptographic primitives, security properties, protocols, and even adversaries are modelled as probabilistic polynomial time (PPT) algorithms, reflecting adversaries that are bounded in computational resources and whose success is quantified probabilistically. To be effective, the computational model must impose reasonable restrictions when defining adversarial capabilities. The success of cryptography depends heavily on how well its security definitions capture realistic and unknown adversaries. Definitions that are too weak fail to give meaningful guarantees,

and overly strong definitions could make secure constructs impossible. For instance, in classical cryptography, a strong security result is achievable using the Vernam cipher under Shannon's definition of perfect secrecy [Sha49, Ros]. However, its use in real-world settings is impractical, due to the fact that the key can be used only once (indeed, the Vernam cipher is also known as the one-time pad) and it must be at least as long as the messages that need encryption. The computational model serves as the reference framework for relatively simple protocols, but its use becomes less practical as protocol complexity increases. This is due to the fact that probabilistic reasoning becomes difficult as protocols grow larger and the number of participating parties increases.

Symbolic Model In contrast, the symbolic model requires very few assumptions about an adversary's capabilities [DY83]. As a result, protocol security can be proved without relying on computational assumptions. When more complex protocols are considered the computational model often reveals its limitations in terms of analysis scalability. In the symbolic model instead, once a protocol is specified, determining whether it is secure can often be automated; indeed, the symbolic model forms the basis for model checking and semi-automated theorem proving [Bla22, MSCB13]. In the symbolic model cryptographic messages are modeled as first-order terms, together with an equational theory that represents attacker capabilities. It also relies on a set of predefined cryptographic primitives that are assumed to be secure and treated as idealized black boxes. Under only these assumptions, the model is effective at identifying logical flaws in protocol designs. A well-known example is the Needham–Schroeder protocol, which was shown to be vulnerable to replay attacks [DS81]. The vulnerability can be detected vert straightforwardly using any symbolic model since it's a protocol flaw and not a cryptographic primitive vulnerability. Several formal frameworks exist for defining symbolic models, depending on the available cryptographic primitives and the underlying theory. One particularly well-studied approach is

based on multi-set rewriting [DLMS04, GK00]. However, tools based on the symbolic model struggle to support cryptographic primitives with rich algebraic properties. For this reason, the security guarantees provided by the symbolic model are often weaker than those offered by the computational model, and a proof of security in the symbolic model does not always imply a corresponding proof in the computational model. For example, secret keys are modeled in the computational model as long bitstrings that are drawn uniformly at random while in the symbolic models they are modeled using abstract names. For example, in the symbolic models, two distinct secret keys are represented by different names, which cannot be equal. However, in the computational model it is possible (but very unlikely) that the sampled bitstrings are equal.

The two models have had very different outcomes with respect to the application of language-based verification techniques. The symbolic model naturally lends itself to the use of classical verification methods. In contrast, applying such techniques within the computational model is significantly more challenging, resulting in less flexibility and automation compared to the symbolic models. The connection between the two approaches (modulo additional assumptions) is established through computational soundness theorems [AR00]. Computational (or complexity theoretic) cryptography relies on the famous $\mathcal{P}$ vs $\mathcal{NP}$ probelm. If $\mathcal{P} \neq \mathcal{NP}$ and one way functions exist, then one can derive most private-key cryptography [IL89]. If $\mathcal{P} = \mathcal{NP}$ then one way functions do not exist and all cryptographic constructions that relies on complexity theoretic security are useless.

This work concentrates mainly on the computational model and its application in the development of formal verification techniques. As already mentioned, cryptography is concerned with the construction of efficient schemes for which it is infeasible to violate the intended security properties. The computational model clarifies the notions of *efficient* and *infeasible computation* by formally defining the computational resources available to legitimate users and adversaries. The computations performed by honest parties

are required to be efficient, while any attempt by an adversary to violate the security guarantees of the scheme should be computationally infeasible. Efficiency is commonly modeled by restricting computations to run in polynomial time in a security parameter, hence adversarial computations, are likewise assumed to be polynomially bounded, although the specific polynomial is not fixed in advance. Under this assumption, adversaries are restricted to the class of polynomial time computations, and any computation requiring super-polynomial resources is considered infeasible in the computational model [GW96].

During first lectures of most cryptography courses it is often introduced the notion of perfect secrecy. The framework being discuessed implies a relaxation of the classical notion of perfect secrecy [Sha49]. Perfect secrecy requires that absolutely no information about the plaintext is leaked by the ciphertext, even to adversaries with unbounded computational power. This strong notion of security must be relaxed in order to obtain efficient protocols that are not perfectly secure in the information-theoretic sense, but are still considered secure in real-world scenarios. In the computational model, protocols may be deemed secure even if they leak a negligible amount of information to computationally bounded adversaries, provided that any such information can be exploited only with negligible probability. Security definitions that restrict adversaries by their computational power are referred to as computational security, in contrast to information-theoretic security, which makes no such assumptions [KL14]. Of course the relaxations need very well definitions in order to capture the right amount of adversaries but still enabling efficient protocols for legitimate users. In the computational model two different approaches exits: the concrete approach and the asymptotic approach.

Concrete Approach In the concrete approach, an encryption scheme is defined (t, ϵ)-secure, if every adversary running for time at most t succeeds in breaking the scheme with probability at most ϵ. Of course, the notion

of breaking a scheme must be precisely defined before such an analysis can be carried out. This approach differs from asymptotic computational security where security is quantified using explicit, concrete parameters rather than asymptotic bounds. Instead of requiring security against all polynomial time adversaries with negligible success probability, the concrete approach provides explicit guarantees relating an adversary's running time to its maximum achievable success probability. The concrete approach is useful in practice because it yields explicit security guarantees for a cryptographic protocol, expressed in terms of the probabilities of concrete events, such as a successful key recovery. However, such guarantees are often difficult to obtain, since probability estimates typically depend on assumptions about the efficiency of the underlying hardware available to the adversary. As an illustrative example, suppose the key length is n, so that the key space has size 2^n, and the key is a uniformly random bitstring of length n. A brute-force attack performed by an adversary running for t CPU cycles succeeds with probability approximately $ct/2^n$, for some constant c depending on the implementation. If $n = 60$ and the adversary uses a desktop computer running at 4GHz (that is, 4×10^9 cycles per second), then a full brute-force search would require approximately $2^{60}/(4 \times 10^9)$ seconds which is about 9 years. A modern exascale supercomputer is capable of performing approximately 2×10^{18} operations per second and would require it roughly $2^{60}/(2 \times 10^{18})$ seconds to break the scheme. It is on the order of half a second [KL14]. Nonetheless a key length of $n = 128$ deters even the most powerful supercomputer since $2^{128}/(2 \times 10^{18}) \approx 1,7 \times 10^{20}$ which is $5,39 \times 10^{12}$ (5,39 trillion years). This clearly show some of the major limitations of the concrete security approach, which often struggles to provide meaningful guarantees without making additional assumptions about an adversary's hardware capabilities or the availability of highly optimized algorithms implemented on specialized hardware for specific tasks.

Asymptotic Approach The asymptotic approach is rooted in complexity theory. All parties involved in a cryptographic protocol are modeled as algorithms that take as input a security parameter, usually denoted by n. The notion of efficiency is formalized using PPT algorithms. An efficient adversary is defined as a probabilistic algorithm whose running time is bounded by a polynomial in the security parameter; that is, there exists a polynomial p such that the adversary runs in time at most $p(n)$. The notion of small *probability of success* is given by the introduction of negligible functions.

Definition 2.1 (Negligible Functions)**.** *A function $f : \mathbb{N} \to \mathbb{R}$ is said to be negligible iff for every polynomial $p : \mathbb{N} \to \mathbb{N}$ there is a N such that for every $n > N$ it holds that $f(n) < 1/p(n)$.*

Lemma 2.2. *The set of all negligible functions $\mathcal{NGL}$ is closed for sum, product and product by an arbitrary polynomial.*

Proof. Let $f(n)$ be a negligible function and let $q(n)$ be any polynomial that is positive for sufficiently large n. We must show that the function $h(n) = q(n)f(n)$ is negligible. Let $p(n)$ be an arbitrary polynomial. Define a new polynomial $p'(n) = p(n)q(n)$. Since the product of two polynomials is itself a polynomial, $p'(n)$ is a polynomial. Because $f(n)$ is negligible, there exists an integer N such that

$$f(n) < \frac{1}{p'(n)} \quad \forall n > N.$$

Substituting $p'(n) = p(n)q(n)$, we obtain

$$f(n) < \frac{1}{p(n)q(n)} \quad \forall n > N.$$

Multiplying both sides by $q(n)$ yields

$$q(n)f(n) < \frac{1}{p(n)} \quad \forall n > N.$$

Because $p(n)$ was arbitrary, it follows that $h(n)$ is negligible. Closure under addition and multiplication of negligible functions follows by similar arguments. □

The definition of negligible functions helps formalize the extent to which an adversary may hope to break a cryptographic scheme being bounded in computational time. At first, however, the role of probability may appear misleading in a cryptographic setting. In this context, probability does not reflect uncertainty in the correctness of cryptographic algorithms such as encryption/decryption or signature/verification which are deterministic in their correctness guarantees. Rather, probability arises because all algorithms (hence parties) in a cryptographic protocol use randomized subroutines that are essential for achieving security. Indeed, without randomness, secrecy cannot be generated [GW96]. For this reason, probabilistic reasoning and randomized computations are indispensable components of the computational model.

In most cases, security proofs in the computational model are reductionist. That is, given an arbitrary but computationally bounded adversary that breaks the security of a cryptographic construction with non-negligible probability, one constructs another computationally bounded adversary that breaks an underlying hardness assumption with comparable probability. This methodology is known as provable security and enables the development of rigorous and detailed mathematical proofs. In practice, the security of a cryptographic scheme Π is often reduced to the security of another scheme or assumption Π'. Reductionist proofs can be viewed as proofs by contraposition since a reduction establishes an implication $(P \Rightarrow Q)$, it is equivalent to proving its contrapositive $(\neg Q \Rightarrow \neg P)$. For example, if P denote the statement that Π is secure, and Q denote the statement that Π' is secure. A proof by reduction shows that if Π' is secure, then Π is secure. In the same way if Π is insecure (i.e., there exists an efficient adversary that breaks Π), then Π' is also insecure. This means that given any adversary A that breaks Π with non-negligible probability, one can construct an adversary B that uses A as a subroutine and breaks Π' with non-negligible probability. Thus, an efficient attack against Π would imply an efficient attack against Π', contradicting the assumed security of Π'.

As cryptography evolves, driven both by increasingly sophisticated adversaries and by the emergence of new technologies, the computational model face several challenges. In particular, it struggles to scale probabilistic reasoning and to apply formal, rigorous mathematical methods to increasingly complex protocols. These difficulties motivate the development of machine-checked frameworks that support the formal construction and automated verification of cryptographic systems. While the symbolic model naturally lends itself to model checking and semi-automated theorem proving, the computational model requires substantially more effort, since reasoning about probabilistic, resource-bounded adversaries becomes increasingly challenging as protocols grow in complexity.

2.1 Perfect Secrecy

The notion of encryption scheme Π in the computational model refers to a triple of algorithms $\Pi = (Gen, Enc, Dec)$ with a message space $\mathcal{M}$ and a key space $\mathcal{K}$ which is the set of all the possible keys that can be produced by Gen. The key-generation algorithm Gen is a probabilistic algorithm that outputs a key $k \in \mathcal{K}$ chosen according to some distribution. The encryption algorithm Enc takes as input a key $k \in \mathcal{K}$ (often compacted to Enc_k) and a message $m \in \mathcal{M}$, and outputs a ciphertext c. Finally $\mathcal{C}$ denote the set of all possible ciphertexts that can be output by $Enc_k(m)$, for all possible choices of $k \in \mathcal{K}$ and $m \in \mathcal{M}$. The algorithm Dec is Given the general definition of an encryption scheme, one may naturally ask whether it is secure. In the computational model, the security of an encryption scheme relies on explicit assumptions about both the computational power of the adversary and the probability with which the adversary may succeed in breaking the scheme while the scheme is still considered secure. Formalizing the notion of security in the computational model therefore constitutes the first and most fundamental set of assumptions. First of all, according to Kerckhoffs' principle, as phrased by Shannon "the enemy knows the system" [Sha49],

the security of an encryption scheme must not rely on the secrecy of its design or algorithms. Relying on hidden algorithms is commonly referred to as security through obscurity, a practice that is widely discouraged. The only piece of information that is always assumed to remain secret is the key k. This convention reflects another formulation of Kerckhoffs' principle: all of the security of a cryptographic system should be concentrated in the secrecy of the key, not in the secrecy of the algorithms themselves [Ros]. If a key is leaked, generating a new one is simple and straightforward given the algorithm Gen, while studying and producing a new cryptographic scheme may take decades. Rather than defining security in terms of informal arguments or abstract intuition, computational cryptography formalizes security through cryptographic experiments (or games). These experiments specify the interaction between a challenger (the scheme) and an adversary and define what it means for the adversary to break a cryptographic scheme. Security is then defined by requiring that any probabilistic polynomial time adversary succeeds in these experiments with at most negligible probability. This contrasts with the perfect-secrecy setting, where there is no notion of negligible functions: security properties rely entirely on the structural design of the scheme and its exact probabilistic behaviour, rather than on computational limitations of the adversary.

In the context of perfect secrecy, the encryption algorithm Enc may be probabilistic, while the decryption algorithm Dec is deterministic, and no other computational limitations are imposed on the algorithms involved. The processes of key generation, message selection, encryption, and decryption are all viewed as parts of a joint probabilistic experiment. By introducing random variables such as $\mathbf{K}$, $\mathbf{M}$ and $\mathbf{C}$, representing respectively the key generated by Gen, the message chosen for encryption, and the resulting ciphertext, it becomes possible to reason formally $\Pr[\mathbf{K} = k]$ where $k \in \mathcal{K}$ is a key, or $\Pr[\mathbf{K} = k \mid \mathbf{M} = m]$ where $m \in \mathcal{M}$.

Definition 2.3 (Perfect Secrecy)**.** *An encryption scheme* $\Pi = (Gen, Enc, Dec)$ *is said to be perfectly secret if for every message* $m \in \mathcal{M}$ *and every ciphertext*

$c \in \mathcal{C}$ *such that* $\Pr[\mathbf{C} = c] > 0$ *it holds that* $\Pr[\mathbf{M} = m \mid \mathbf{C} = c] = \Pr[\mathbf{M} = m]$

The intuition behind this definition is that observing the ciphertext reveals absolutely no information about the plaintext. At the same time the distribution of the ciphertext is independent of the message being encrypted: for any two messages $m, m' \in \mathcal{M}$ the distribution of the ciphertext produced by encrypting m is identical to the distribution of the ciphertext produced by encrypting m'. A scheme that definitely has such property is the Vernam's cipher also known as One-Time Pad (OTP) [1]. OTP is defined as a triple of algorithms Gen, Enc, Dec reported in Figure 1.

$$\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^n \qquad Gen(n) = k \leftarrow \{0,1\}^n;\ \texttt{return}\ k$$
$$Enc_k(m) = \texttt{return}\ m \oplus k \qquad Dec_k(c) = \texttt{return}\ c \oplus k$$

Figure 1: Definition of the OTP encryption scheme.

Where the notation $k \leftarrow \{0,1\}^n$ means to sample uniformly from the set of n-bitstrings. This uniform choice of key is the only randomness in all of the one-time pad algorithms. The scheme is correct because $Dec_k(Enc_k(m)) = (m \oplus k) \oplus k = m \oplus (k \oplus k) = m \oplus 0^n = m$ OTP is perfectly secure in the information-theoretic sense, but it suffers from major limitations: the key length must be greater than or equal to the length of the message, and each key can be used only once.

Encryption schemes can be studied in terms of perfect secrecy, but also in terms of indistinguishability, which is an equivalent and widely used definition of secrecy for encryption schemes. To discuss the indistinguishability property of an encryption scheme, we introduce the experiment $\texttt{PrivK}^{eav}_{A,\Pi}$.

One may ask what the maximum probability is with which an adversary

[1] One-time pad is often called Vernam's cipher after Gilbert Vernam, a telegraph engineer who patented the scheme in 1919. However, an earlier description of one-time pad was recently discovered in a 1882 text by Frank Miller on telegraph encryption [Bel11].

$$
\begin{aligned}
&\texttt{PrivK}^{eav}_{A,\Pi}(n):\\
&\quad m_0, m_1 \leftarrow A\\
&\quad k \leftarrow Gen(1^n)\\
&\quad b \leftarrow \{0,1\}\\
&\quad c \leftarrow Enc_k(m_b)\\
&\quad b^* \leftarrow A(c)\\
&\quad \texttt{return } \neg(b \oplus b^*)
\end{aligned}
$$

Figure 2: Pseudocode for the $\texttt{PrivK}^{eav}$ experiment.

can cause the experiment to output 1, namely

$$\Pr[\texttt{PrivK}^{Eav}_{A,\Pi} = 1.]$$

A natural question is whether it would make sense to require

$$\Pr[\texttt{PrivK}^{Eav}_{A,\Pi} = 1] = 0.$$

This requirement is clearly too strong, since an adversary A can trivially succeed with probability 1/2 by outputting a random guess. Perfect indistinguishability instead requires that no adversary can achieve a success probability strictly greater than 1/2.

Definition 2.4 (Indistinguishable Encryptions)**.** *An encryption scheme* $\Pi = (Gen, Enc, Dec)$ *has indistinguishable encryptions if and only if for every adversary* A*,*

$$\Pr[\texttt{PrivK}^{Eav}_{A,\Pi} = 1] = \frac{1}{2} \tag{1}$$

From the definition of indistinguishable encryptions, Theorem 2.5 follows directly.

Theorem 2.5 (Perfect Secrecy Via Indistinguishability)**.** *An encryption scheme* Π *is perfectly secret if and only if it is perfectly indistinguishable.*

The limitations of perfect secrecy stem from the fact that any perfectly secret encryption scheme must have a key space that is at least as large as the message space. If all keys have the same length and the message space consists of all strings of some fixed length, this implies that the key must be at least as long as the message, and this requirement is unavoidable. If two users wish to privately convey an n-bit message, they must first privately agree on an n-bit key. This also leads to the conclusion that the key length used in the one-time pad is optimal. Another important limitation is that the key can only be used once. A modified scheme, such as the so-called two-time pad, is inevitably insecure and vulnerable to attacks, most notably crib-dragging attacks. Nonetheless, the most fundamental limitation of perfect secrecy is captured by the following theorem.

Theorem 2.6. *Let* $\Pi = (Gen, Enc, Dec)$ *be a perfectly secret encryption scheme over a message space* $\mathcal{M}$ *and a key space* $\mathcal{K}$*. Then* $|\mathcal{K}| \geq |\mathcal{M}|$*.*

This makes perfect secrecy almost impossible in a real world setting. Just imagine the keys required to encrypt videos or very large files.

2.2 Private Key Encryption

As stated in Section 2.1, perfect secrecy requires that absolutely no information about an encrypted message is leaked, even to an eavesdropper with unlimited computational power [KL14]. In computational cryptography, this strong notion is weakened in favor of cryptographic schemes that are still considered secure, but only against efficient adversaries with a low probability of success. The notion of efficiency in the computational model is captured by bounding the running time of adversaries by a polynomial in a security parameter. Recall that an algorithm (or adversary) A runs in polynomial time if there exists a polynomial p such that, for every input $x \in 0, 1^n$, the computation of $A(x)$ terminates within at most $p(n)$ steps. The requirement of a low probability of success is formalized through the notion of negligible functions.

The computational model, which is the *de facto* standard for analysing the security properties of cryptographic schemes using rigorous mathematical tools, relies on concepts from computational complexity theory. In particular, it requires that the size of inputs be well defined. For this reason, it is customary to assume, without loss of generality, that keys, messages, and ciphertexts are represented as bitstrings. Another essential requirement is that all algorithms be probabilistic. This can be viewed as each algorithm being provided, in addition to its input, with access to a uniformly distributed random tape of sufficient length, whose bits it may use as needed throughout its execution. The design choices underlying the computational model, and in particular the asymptotic approach are neither arbitrary nor accidental. One advantage of using (probabilistic) polynomial time as the measure of efficiency is that one doesn't have to precisely specify a concrete model of computation. Indeed, the extended Church-Turing thesis asserts that all reasonable models of computation are polynomially equivalent. If the analysis shows polynomial time execution, then any reasonable implementation will also run in polynomial time. Another advantage of restricting attention to (probabilistic) polynomial time algorithms is that they have desirable closure properties. In particular, an algorithm that makes polynomially many calls to polynomial time subroutines, and performs only polynomially bounded additional computation, will itself run in polynomial time (see proof of Lemma 2.2).

In the setting of private-key encryption, two parties share a secret key and use it to communicate confidentially. Formally, a private-key encryption scheme is a tuple of probabilistic polynomial time algorithms $\Pi = (Gen, Enc, Dec)$, defined over a message space $\mathcal{M}$ and a key space $\mathcal{K}$, such that:

- The key-generation algorithm Gen takes as input a string of the form 1^n, interpreted as the security parameter, and outputs a key k such that $|k| \geq n$.

- The encryption algorithm Enc may be probabilistic; that is, it may

produce different ciphertexts when invoked multiple times on the same message–key pair (m, k).

- The decryption algorithm Dec is deterministic, since correctness would otherwise not be guaranteed.

The scheme is assumed to be correct, meaning that for all keys $k \in \mathcal{K}$ generated by Gen and all messages $m \in \mathcal{M}$ it is true that $Dec_k(Enc_k(m)) = m$. Often, the encryption algorithm Enc_k is defined only for messages whose length is equal to some function $\ell(n)$ of the security parameter n. In this case, the encryption scheme is said to be a fixed-length encryption scheme with length parameter ℓ.

In order to define security for such an encryption scheme let's introduce a modified notion of the experiment $\texttt{PrivK}^{eav}_{A,\Pi}$ reported in Figure 3.

$\texttt{PrivK}^{eav}_{A,\Pi}(n)$:
  $(m_0, m_1) \leftarrow A$
  `if` $|m_0| \neq |m_1|$
    `return` 0
  $k \leftarrow Gen(1^n)$
  $b \leftarrow \{0, 1\}$
  $c \leftarrow Enc_k(m_b)$
  $b^* \leftarrow A(c)$
  `return` $\neg(b \oplus b^*)$

Figure 3: Pseudocode for the modified $\texttt{PrivK}^{eav}$ experiment.

In the experiment $\texttt{PrivK}^{eav}_{A,\Pi}$, the adversary A is given the security parameter as input and outputs a pair of messages (m_0, m_1). If the two messages have different lengths, the experiment immediately returns 0. Otherwise, a key k is generated by running $Gen(1^n)$, and a uniformly random bit $b \in \{0, 1\}$

is sampled. The adversary is then given the ciphertext $c = Enc_k(m_b)$ and it has to output a single bit b^*. The experiment returns 1 if $b^* = b$, meaning that the adversary has successfully guessed which message was encrypted.

The definition of security in the form of a indistinguishability, for a scheme Π with respect of $\texttt{PrivK}^{eav}_{A,\Pi}$ is presented in Definition 2.7.

Definition 2.7 (Security Against Passive Attacks)**.** *An encryption scheme* Π *is said to be secure against passive attacks or secure with respect to* $\texttt{PrivK}^{eav}_{A,\Pi}$ *if and only if for every PPT adversary* A *there exists a negligible function* ϵ *such that*

$$\Pr[\texttt{PrivK}^{eav}_{A,\Pi}] = \frac{1}{2} + \epsilon(n).$$

Once again, adversaries are restricted to probabilistic polynomial time algorithms, and their probability of success is not required to be zero. Such a requirement would be unreasonable, since an adversary can always guess randomly and succeed with probability 1/2. Instead, security requires that any adversary's advantage over random guessing be negligible. The class of attacks considered with respect to the experiment $\texttt{PrivK}^{eav}_{A,\Pi}$ consists of *passive attacks*, namely attacks in which the adversary only observes information without actively interfering with the protocol execution. In particular, the adversary may have access either to a collection of ciphertexts $c_1, \ldots, c_n$, or to a list of message–ciphertext pairs $(m_1, c_1), \ldots, (m_n, c_n)$, where each c_i is the ciphertext corresponding to m_i.

To construct an encryption scheme satisfying this notion of security, one must address the requirement that $|\mathcal{K}| < |\mathcal{M}|$. As seen in the case of perfect secrecy, this condition renders schemes impractical in real-world settings. There is therefore a need for algorithms that use relatively short keys while still inducing a cryptographic scheme that is secure according to Definition 2.7. A fundamental tool for achieving this goal is the pseudorandom generator.

2.2.1 Pseudorandom Generators

A pseudorandom generator is an efficient, deterministic algorithm that transforms a short, uniformly random string, called a *seed*, into a longer string that appears uniform to any efficient adversary. In other words, a pseudorandom generator uses a small amount of true randomness to generate a large amount of pseudorandomness. Let $\ell : \mathbb{N} \to \mathbb{N}$ be a polynomial, called the *expansion factor*, and let G be a deterministic algorithm that, for every input $s \in \{0,1\}^*$, outputs a string $G(s) \in \{0,1\}^{\ell(|s|)}$. Then G is a *pseudorandom generator (PRG)* if and only if the following conditions hold:

- For every $n \in \mathbb{N}$, $\ell(n) > n$.
- G runs in probabilistic polynomial time.
- For every PPT distinguisher D, there exists a negligible function ϵ such that

$$|\Pr[D(G(s)) = 1] - \Pr[D(r) = 1]| \leq \epsilon(n). \tag{2}$$

Here, the first probability is taken over the uniform choice of $s \in \{0,1\}^n$ and the internal randomness of D, while the second probability is taken over the uniform choice of $r \in \{0,1\}^{\ell(n)}$ and the internal randomness of D.

As stated above, a pseudorandom generator is not random in a strict information-theoretic sense. If the expansion factor ℓ is fixed, for example $\ell(n) = 2n$, then the set of all possible bitstrings of length $2n$ has size 2^{2n}, the generator G, being deterministic, can produce at most 2^n distinct outputs one for each possible seed. Thus, the distribution induced by $G(s)$ has support of size at most 2^n, which is clearly smaller than the full space 2^{2n}. Consequently, there always exists a distinguisher D that can separate the output of G from the uniform distribution with high probability. However, such a distinguisher necessarily has exponential complexity. Since the definition of pseudorandomness only considers efficient (polynomial time) distinguishers, this does not contradict Equation 2 and for all PPT distinguishers, the distinguishing advantage remains negligible.

In most cryptography courses, the standard definition of pseudorandom generators and, more generally, pseudorandomness is formulated in terms of intractability, that is, the infeasibility for any efficient algorithm to distinguish a pseudorandom object from a uniformly distributed one. This notion is closely related to the class of assumptions commonly adopted in computational complexity theory, according to which one can derive cryptographic primitives from problems believed to be computationally hard, for example inverting one-way functions [Gol08]. Such assumptions constitute the fundamental building blocks of modern computational cryptography.

Given the notion of a pseudorandom generator, one can construct a PRG-induced encryption scheme that is secure against passive (eavesdropping) attacks. Formally, this scheme is a triple of algorithms $\Pi_G = (Gen, Enc, Dec)$ defined as follows:

- The algorithm Gen on input 1^n outputs each string of length n with same probability $1/2^n$.
- The algorithm Enc is defined as $Enc_k(m) = m \oplus G_k$.
- The algorithm Dec is defined as $Dec_k(c) = c \oplus G_k$.

The scheme Π_G is correct because G is deterministic. Indeed, for every key k and message m it is true that $Dec_k(Enc_k(m)) = Dec_k(m \oplus G_k) = ((G_k \oplus m) \oplus G_k) = m$. The scheme Π_G is defined for messages of fixed length $\ell(n)$, where ℓ is the expansion factor of the pseudorandom generator G.

In the context of the experiment $\mathtt{PrivK}^{eav}_{A,\Pi}$, the considered attacks are *passive*. This means that the adversary cannot actively interfere with the execution of the protocol, but can only observe its surroundings. In particular, the communicating parties transmit a single ciphertext, which may be observed by an eavesdropper. It would be desirable, however, for the communicating parties to be able to exchange multiple ciphertexts generated using the same secret key even in the presence of an eavesdropper who can observe all of them. To model such scenarios an encryption scheme is required to

remains secure under the encryption of multiple messages, and has to behave accordingly with respect of a new security experiment, shown in Figure 4.

$\texttt{PrivK}^{mult}_{A,\Pi}(n)$:
  $(\vec{M}_0, \vec{M}_1) \leftarrow A$
  if $\exists i.|m_{0,i}| \neq |m_{1,i}|$
    return 0
  $k \leftarrow Gen(1^n)$
  $b \leftarrow \{0,1\}$
  $\vec{C} \leftarrow Enc_k(\vec{M}_b)$
  $b^* \leftarrow A(c)$
  return $\neg(b \oplus b^*)$

Figure 4: Pseudocode for the $\texttt{PrivK}^{mult}$ experiment.

In the experiment $\texttt{PrivK}^{mult}_{A,\Pi}$, the adversary is allowed to output two vectors of messages of length t, namely $\vec{M}_0 = (m_{0,0}, \ldots, m_{0,t-1})$ and $\vec{M}_1 = (m_{1,0}, \ldots, m_{1,t-1})$. A random bit b is chosen, and the challenger encrypts the entire vector $\vec{M}_b$ under the same key k, producing a ciphertext vector $\vec{C} = (c_0, \ldots, c_{t-1})$ which is then given to the adversary.

Definition 2.8 (Security Against Multiple Encryptions)**.** *A scheme* Π *is said to be secure against multiple encryptions if for every PPT adversary* A *there exists a negligible function* ϵ *such that*

$$\Pr[\texttt{PrivK}^{mult}_{A,\Pi} = 1] \leq \tfrac{1}{2} + \epsilon(n). \tag{3}$$

It immediately follows that:

Lemma 2.9. *The scheme* Π_G *is not secure with respect to* $\texttt{PrivK}^{mult}_{A,\Pi}$*, even if* G *is a pseudorandom generator.*

What at first appears to be a strong limitation is in fact fundamental.

Theorem 2.10. *If the encryption algorithm Enc is stateless and deterministic, then the scheme* $\Pi = (Gen, Enc, Dec)$ *cannot be secure with respect to* $\mathtt{PrivK}^{mult}_{A,\Pi}$.

Another important class of attacks consists of *active* attacks, in which adversaries are allowed to play an active role. In this setting, the adversary may obtain encryptions of messages of its choice by interacting with an encryption oracle $Enc_k(\cdot)$ (chosen-plaintext attacks), or may even have access to a decryption oracle $Dec_k(\cdot)$ (chosen-ciphertext attacks). In both cases, the secret key k is unknown to the adversary. Chosen-plaintext attacks (CPA) are particularly useful for introducing a stronger and more realistic security notion. In this scenario, the adversary A is allowed to interact with the encryption oracle and query it on messages of its choice, for a polynomial number of queries in the security parameter n. This interaction is formalized by the experiment shown in Figure 5. In the experiment $\mathtt{PrivK}^{cpa}_{A,\Pi}$, the adver-

$\mathtt{PrivK}^{cpa}_{A,\Pi}(n)$:
  $k \leftarrow \{0,1\}^n$
  $(m_0, m_1) \leftarrow A^{Enc_k(\cdot)}(1^n)$
  $b \leftarrow \{0,1\}$
  $c \leftarrow Enc_k(m_b)$
  $g \leftarrow A(c, Enc_k(\cdot))$
  `return` $(b = g)$

Figure 5: Pseudocode for $\mathtt{PrivK}^{cpa}$ experiment.

sary first obtains access to the encryption oracle and outputs two messages m_0 and m_1 of equal length. A random bit b is then sampled, and the challenge ciphertext $c = Enc_k(m_b)$ is returned to the adversary. The adversary

continues to have access to the encryption oracle and must output a guess g for the value of b. When the algorithm Enc is randomized, the oracle uses fresh randomness each time it answers a query.

Security against chosen-plaintext attacks requires that no efficient adversary can guess the value of b with probability significantly greater than 1/2. This is captured by the following definition.

Definition 2.11 (CPA Security)**.** *An encryption scheme* Π *is said to be secure against chosen-plaintext attacks if and only if for every PPT adversary* A *there exists a negligible function* ϵ *such that*

$$\Pr[\mathtt{PrivK}^{cpa}_{A,\Pi}(n) = 1] \leq \frac{1}{2} + \epsilon(n) \tag{4}$$

where the probability is taken over the randomness used by A*, as well as the randomness used in the experiment.*

An immediate consequence of the definition of security with respect to $\mathtt{PrivK}^{cpa}_{A,\Pi}$ is that the encryption algorithm must be probabilistic. This observation reinforces the need to rely on cryptographic primitives that introduce randomness into the encryption.

Lemma 2.12. *Any encryption scheme* $\Pi = (Gen, Enc, Dec)$ *that is secure with respect to* $\mathtt{PrivK}^{cpa}$ *must have a probabilistic encryption algorithm Enc.*

This requirement also provides a solution to the problem of security under multiple encryptions, which was shown to be unachievable when relying only on pseudorandom generators. Indeed, probabilistic encryption achieve security even when the same key is used to encrypt multiple messages, as captured by the following theorem.

Theorem 2.13. *Every encryption scheme that is CPA-secure is also secure in the presence of multiple encryptions.*

2.3 Pseudorandom Functions

To construct a CPA-secure encryption scheme, it is necessary to introduce the notion of a *pseudorandom function* (PRF). Informally, a pseudorandom function captures the idea of pseudorandomness for a distribution over functions. Such a distribution is induced by considering *keyed functions*, defined as follows. A keyed function is a two-input function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ where the first input is the key and is denoted by k. Typically a key k is chosen once and fixed. One then considers the single-input function $F : \{0,1\}^* \to \{0,1\}^*$ denoted by F_k. By choosing a key $k \in \{0,1\}^n$ uniformly at random, the keyed function F induces a distribution over single-input functions F_k, so a PRF is not a single function, but a family of functions indexed by a key, for each key fixed there is a function. The intuition is that F is pseudorandom if the distribution is computationally indistinguishable from the uniform distribution over all functions with the same domain and range. Meaning that no efficient adversary should be able to tell whether it is interacting with F_k for a uniformly chosen key k, or with a truly random function f.

A very important consideration concerns the size of the set of truly random functions from $\{0,1\}^n$ to $\{0,1\}^n$. The number of all such functions is 2^{n2^n} since for each of the 2^n possible inputs one may independently choose any of the 2^n possible outputs and is usually denoted by Func_n. When fixing a key $k \in \{0,1\}^n$ for a keyed function F, the resulting function F_k is just a single deterministic function. As k ranges over all n-bit strings, the family F_k contains at most 2^n distinct functions. This induces a distribution over functions supported on at most 2^n functions if all keys define distinct functions, then each occurs with probability $1/2^n$. Requiring that no efficient adversary be able to distinguish a truly random function from a pseudorandom function amounts to an extremely strong requirement. It asks the adversary to be unable to distinguish between a uniform distribution over an astronomically large set of size 2^{n2^n} where each function has probability $1/2^{n2^n}$ and a distribution supported on "only" 2^n functions, each occurring with prob-

ability $1/2^n$. These two distributions are vastly different. Nevertheless, the defining property of a pseudorandom function is that this difference cannot be exploited by any PPT adversary.

To formally define pseudorandom function let's introduce also the following notions

- A binary partial function is a partial function from $\{0,1\}^n$ to $\{0,1\}^n$
- A binary partial function F is length-preserving iif $F(k,x)$ is defined iif $|k| = |x|$ and in that case $|F(k,x)| = |x|$.
- A binary partial function efficient if there exists a PPT algorithm that computes it.

Definition 2.14. *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. F is a pseudorandom function if for all PPT distinguishers D, there is a negligible function ϵ such that:*

$$\left|\Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1]\right| \leq \epsilon(n) \tag{5}$$

where k is uniformly chosen among all strings of length n and f is uniformly chosen among all functions from $\{0,1\}^n$ to $\{0,1\}^n$.

Given a pseudorandom function, one can construct a private-key encryption scheme that is secure with respect to the experiment $\texttt{PrivK}^{cpa}$. The resulting scheme, denoted by $(\Pi_F = (Gen, Enc, Dec))$, is defined as follows.

- The algorithm Gen on input 1^n outputs each string of length n with same probability $1/2^n$.
- Given a pseudorandom function F, the algorithm $Enc_k(m)$ is defined as the (binary encoding of the) pair $\langle r, F_k(r) \oplus m \rangle$, where r is a random string $|k|$ bits long.
- $Dec_k(c)$ returns $F_k(r) \oplus s$ anytime c is the (binary encoding of the) pair $\langle r, s \rangle$.

Theorem 2.15. *If F is a PRF, then Π_F is secure against CPA attacks.*

2.4 Message Authentication Codes

Message Authentication Codes (MACs) address the problem of message authentication over insecure channels. In this setting, messages are not required to be secret, but it must be possible to authenticate them, namely, each party should be able to verify that a message it receives was indeed sent by the party claiming to be the sender. In other words, MACs provide integrity and authenticity rather than confidentiality.

Schemes based solely on pseudorandom generators are insufficient to solve this problem. For instance, consider an encryption scheme Π_G based on a pseudorandom generator G, where encryption is defined as $Enc_k(m) = G_k \oplus m$ and decryption as $Dec_k(m) = G_k \oplus c$. An adversary who observes a ciphertext can easily construct a valid ciphertext for a related message, hence impersonating the legitimate sender. Even more directly, flipping any bit in the ciphertext $c = G_k \oplus m$ results in the same bit being flipped in the plaintext after decryption. Thus, given a ciphertext c that encrypts a message m, it is possible to generate a modified ciphertext c' such that $m' = Dec_k(c')$ is equal to m except for one or more flipped bits. This illustrates that pseudorandomness alone does not guarantee message integrity. Authentication requires dedicated mechanisms beyond encryption.

Two users who wish to communicate in an authenticated manner begin by generating and securely sharing a secret key k prior to communication. When one party wants to send a message m to the other, she computes a MAC tag (or simply a tag) t based on the message and the shared key, and sends the pair (m, t) to the receiver. The tag is computed using a tag-generation algorithm, denoted by Mac. That is, the sender computes the tag t as $Mac_k(m)$ and transmits (m, t). Upon receiving (m, t), the receiver checks whether t is a valid tag for the message m with respect to the key k. Verification is performed by a deterministic algorithm$Vrfy$, which takes as input the key, the message, and the tag, and outputs a bit indicating acceptance or rejection of the tag on the message.

Formally, a MAC is a triple $\Pi = (Gen, Mac, Vrfy)$ of PPT algorithms

such that:

- Gen takes as input a unary representation of the security parameter 1^n and outputs a secret key k, where typically $|k| \geq n$;
- Mac takes as input a key k and a message m, and outputs a tag t;
- $Vrfy$ takes as input a key k, a message m, and a tag t, and outputs a bit b, with $b = 1$ meaning valid and $b = 0$ meaning invalid.

A MAC is said to be correct if for all keys k generated by Gen and all messages m, it holds that

$$Vrfy_k(m, Mac_k(m)) = 1$$

Correctness ensures that honestly generated tags are always accepted by the verification algorithm.

The security goal for message authentication codes (MACs) can be stated informally as follows: no efficient adversary should be able to forge a valid tag for any message that was not previously sent and authenticated by one of the legitimate communicating parties. Equivalently, even after interacting with the system, the adversary should not be able to produce a new message–tag pair that will be accepted as valid. More precisely, the adversary is given oracle access to $Mac_k(\cdot)$ and may query the oracle to obtain valid pairs (m, t), where m is a message and t is its corresponding tag. Any pair obtained directly from the oracle is, by definition, not considered a forgery. In practice, this models an adversary who passively observes the communication between honest parties and is able to see all transmitted messages together with their associated MAC tags. The security requirement is that, despite this access, any PPT adversary cannot generate a valid tag for any new message that was not previously authenticated. This can be formalized through an experiment called `MacForge` which is described in Figure 6.

Given the experiment `MacForge` the notion of security for a scheme MAC Π is captured by Definition 2.16.

$\texttt{MacForge}_{A,\Pi}(n)$:
  $k \leftarrow Gen(1^n)$
  $(m, t) \leftarrow A^{Mac_k(\cdot)}(1^n)$
  $\mathbb{Q} \leftarrow \{m \mid A \text{ queries } Mac_k(m)\}$
  return $(m \notin \mathbb{Q} \wedge Vrfy_k(m, t) = 1)$

Figure 6: Pseudocode for `MacForge` experiment.

Definition 2.16 (MAC security)**.** *A MAC* Π *is secure if and only if for every PPT adversary* A *there exists a negligible function* ϵ *such that*

$$\Pr[\texttt{MacForge}_{A,\Pi} = 1] \leq \epsilon(n) \tag{6}$$

This definition of MAC security is simultaneously strong and weak. On the one hand, it is strong because it considers the scheme broken even if the adversary forges a valid tag for a message that is meaningless. Any successful forgery, regardless of the message content, translates to a violation of security. On the other hand, the definition is weak in that it does not protect against replay attacks. An adversary who records a previously transmitted and authenticated pair (m, t) can resend the same pair at a later time, and it will still be accepted as valid by the receiver. Since replayed messages are not considered new forgeries under the standard definition, preventing such attacks requires additional mechanisms, such as nonces, timestamps, or sequence numbers, which go beyond basic MAC security.

2.4.1 MAC Security From Pseudorandom Function

A well-known result in cryptography states that pseudorandom functions can be used to construct secure message authentication codes. Given a pseudorandom function F, the construction MAC^F yields a secure MAC. Formally, let $\Pi_F = (Gen, Mac, Vrfy)$ be the triple of algorithms defined as

follows:

- The key-generation algorithm Gen takes as input the security parameter 1^n and outputs a uniformly random bitstring $k \in 0, 1^n$, each with probability $1/2^n$;
- The tag generation algorithm is defined as $Mac_k(m) = F_k(m)$;
- The verification algorithm is defined as $Vrfy_k(m, t) = (F_k(m) \stackrel{?}{=} t)$.

Theorem 2.17 (Security of MAC^F)**.** *If F is a pseudorandom function, then the MAC scheme Π_F is a secure fixed-length message authentication code for messages of length n.*

Proof. Let A be a PPT adversary. Consider the message authentication code $\widetilde{\Pi}$, which is mostly identical to MAC^F except that it uses a truly random function f instead of the pseudorandom function F_k. The algorithm $\widetilde{Gen}(1^n)$ samples uniformly a truly random function from the set of all functions $\{0,1\}^n \to \{0,1\}^n$, and the algorithm $\widetilde{Mac}(m)$ computes the tag $t = f(m)$.

It is clear that

$$\Pr[\texttt{MacForge}_{A,\widetilde{\Pi}}(n) = 1] \leq \frac{1}{2^n} \tag{7}$$

since for any message $m \notin \mathcal{Q}$, the value $t = f(m)$ is uniformly distributed in $\{0,1\}^n$ and independent of all previous oracle responses. Because 2^{-n} is a negligible function, the MAC scheme $\widetilde{\Pi}$ is secure with respect to the experiment `MacForge`.

The goal of the proof is to show that

$$\left|\Pr[\texttt{MacForge}_{A,\Pi_F}(n) = 1] - \Pr[\texttt{MacForge}_{A,\widetilde{\Pi}}(n) = 1]\right| \leq \epsilon(n) \tag{8}$$

for some negligible function ϵ and where Π_F is MAC^F construction. That is, the probability that any PPT adversary succeeds in forging a valid MAC under MAC^F differs only negligibly from the probability that the same adversary succeeds when the MAC is computed using a truly random function. Combining Equation 8 with Equation 7, we obtain

$$\Pr[\texttt{MacForge}_{A,\Pi_F}(n) = 1] \leq \frac{1}{2^n} + \epsilon(n) \tag{9}$$

which proves the theorem because of the closure of negligible functions (see Lemma 2.2).

To prove Equation 8, we construct from any adversary A that succeeds in the $\texttt{MacForge}$ experiment, a distinguisher D_A for the pseudorandom function F_k. The distinguisher D_A is given oracle access to a function $\mathcal{O} : \{0,1\}^n \to \{0,1\}^n$ and must determine whether $\mathcal{O}$ is a pseudorandom function of the form F_k for a uniformly random key $k \in \{0,1\}^n$, or a truly random function f. The distinguisher $D_A^{\mathcal{O}}$ emulates the MAC-forgery experiment for A in such a way that $D_A^{F_k(\cdot)}$ behaves exactly like $\texttt{MacForge}_{A,\Pi_F}$, while $D_A^{f(\cdot)}$ behaves like $\texttt{MacForge}_{A,\widetilde{\Pi}}$, where $\widetilde{\Pi}$ is the MAC scheme based on a truly random function. The distinguisher is defined as follows:

$D_A^{\mathcal{O}}$:

1. Run $A(1^n)$. Whenever A queries its MAC oracle on a message $m \in \{0,1\}^n$, answer the query by returning $t \leftarrow \mathcal{O}(m)$. Record the queried message m in the set $\mathcal{Q}$.
2. Eventually, A outputs a pair (m^*, t^*).
3. If $m^* \notin \mathcal{Q}$ and $t^* = \mathcal{O}(m^*)$, then output 1 and 0 otherwise.

The distinguisher $D_A^{\mathcal{O}}$ runs in probabilistic polynomial time, since it only runs A and makes a polynomial number of oracle queries. There are two cases:

- If $\mathcal{O} = F_k$, then the simulation provided by D_A is distributed identically to the view of A in the experiment $\texttt{MacForge}_{A,\Pi_F}$, and therefore D outputs 1 exactly when $\texttt{MacForge}_{A,\Pi}(n) = 1$

$$\Pr[D_A^{F_k(\cdot)}(1^n) = 1] = \Pr[\texttt{MacForge}_{A,\Pi_F}(n) = 1] \tag{10}$$

- If instead $\mathcal{O} = f$, where f is a truly random function, then the simulation is distributed identically to the view of A in the experiment

$\texttt{MacForge}_{A,\widetilde{\Pi}}$, and hence D outputs 1 exactly when $\texttt{MacForge}_{A,\widetilde{\Pi}}(n) = 1$

$$\Pr[D_A^{f(\cdot)}(1^n) = 1] = \Pr[\texttt{MacForge}_{A,\widetilde{\Pi}}(n) = 1] \tag{11}$$

where $f \in \mathsf{Func}_n$ is chosen uniformly.

Since F is a pseudorandom function, by definition the advantage of any PPT distinguisher against F is negligible (and D runs in polynomial time), there exists a negligible function ϵ such that

$$\left|\Pr[D_A^{F_k(\cdot)}(1^n) = 1] - \Pr[D_A^{f(\cdot)}(1^n) = 1]\right| \leq \epsilon(n) \tag{12}$$

This implies Equation 8 and therefore completes the reduction. □

Chapter 3

Formal Verification Tools in the Computational Model

Automated proof systems are machine-checked frameworks that enable cryptographers to reason directly within the computational model or the symbolic model while providing a formal and machine-verifiable environment for proving the security of cryptographic protocols. While many such frameworks exist for the symbolic model of cryptography, comparatively little work has been done in the computational model, largely due to the complexity of probabilistic reasoning and reduction-based proofs. Over the last two decades, however, several tools have been developed to address this gap, including EasyCrypt [BDG+14], Squirrel [BDJ+24] and CryptoVerif [Bla05]. EasyCrypt performs Hoare-style proofs of programs, CryptoVerif relies on game transformations while Squirrel reasons over execution traces of protocols.

3.1 EasyCrypt

In EasyCrypt, security goals and hardness assumptions are modeled as probabilistic programs, called experiments (or games), which include unspecified adversarial code. EasyCrypt supports common reasoning patterns from

the game-based approach, in which reductionist security proofs are decomposed into a sequence or, more generally, a tree of transformations called *game hops* [BR04, Sho04]. Each hop introduces a small change to the initial game, whose impact on the adversary's distinguishing advantage can be formally bounded, making the overall proof easier to understand, structure, and check. Since each game hop relates two probabilistic programs, EasyCrypt relies on probabilistic relational Hoare logic (pRHL) to reason about pairs of programs. pRHL defines judgments of the form

$$[c_1 \sim c_2 : \Phi \Longrightarrow \Psi],$$

where c_1 and c_2 are probabilistic programs, and Φ and Ψ are relational assertions on two memories. An example is $x\langle 1\rangle = x\langle 2\rangle$, which states that the value of x coincides in both memories. pRHL judgments do not explicitly refer to probabilities but probabilistic statements can still be derived from valid judgments. The semantics of pRHL is based on a notion of *lifting* which allows one to derive equalities and inequalities between probabilities of events in related programs. For instance, from pRHL judgment of the form

$$[c_1 \sim c_2 : \Phi \Longrightarrow E\langle 1\rangle \to F\langle 2\rangle],$$

one can derive that

$$\Pr[c_1, m_1 : E] \leq \Pr[c_2, m_2 : F],$$

for all initial memories m_1 and m_2 that are related by Φ, and for events E and F.

A concrete example illustrating the use of pRHL involves reasoning about two probabilistic experiments (games), shown in Figure 7. The goal is to prove that the two games induce the same joint distribution over pairs (m, c). This captures the perfect secrecy of the one-time pad, the ciphertext produced by encrypting a message with a uniformly random key is itself uniformly random and independent of the message.

In pRHL, this equivalence can be expressed by the judgment

$$[\texttt{OTP} \sim \texttt{Uniform} : \Phi \Longrightarrow m\langle 1\rangle = m\langle 2\rangle \wedge c\langle 1\rangle = c\langle 2\rangle],$$

$\texttt{OTP}(n)$:

$m \leftarrow \mathcal{M}$

$k \leftarrow \{0,1\}^n$

$c \leftarrow m \oplus k$

$\texttt{return}(m, c)$

$\texttt{Uniform}(n)$:

$m \leftarrow \mathcal{M}$

$c \leftarrow \{0,1\}^n$

$\texttt{return}(m, c)$

Figure 7: Example of `OTP` and `Uniform` cryptographic games.

where the precondition Φ relates the initial memories of the two games. In this case, since both programs start from empty memories and no inputs are assumed, can simply be taken as ($\Phi = \top$). By the soundness of pRHL, a judgment of the form

$$[\texttt{OTP} \sim \texttt{Uniform} : \Phi \Longrightarrow res\langle 1\rangle = res\langle 2\rangle] \Longrightarrow \texttt{OTP} \equiv \texttt{Uniform},$$

implies that the two programs are observationally equivalent. To establish the judgment, we proceed as follows. Both programs begin by sampling a message m using the same distribution $\mathcal{M}$ which is assumed to be the set containing all the bit strings of length n. Therefore, we can couple the two samplings so that $m\langle 1\rangle = m\langle 2\rangle$. Next the `OTP` game samples a key $k \leftarrow \{0,1\}^n$ uniformly at random, while the `Uniform` game samples a ciphertext $c \leftarrow \{0,1\}^n$ uniformly at random. We define a coupling between these samplings by setting

$$c\langle 2\rangle = m \oplus k,$$

and this is valid because, for any fixed message m m, the distribution of $m \oplus k$ is uniform whenever k is uniformly distributed. Finally, the `OTP` game sets

$$c\langle 1\rangle = m \oplus k,$$

which implies $c\langle 1\rangle = c\langle 2\rangle$. The postcondition

$$(m\langle 1\rangle = m\langle 2\rangle) \wedge (c\langle 1\rangle = c\langle 2\rangle),$$

holds and we conclude that

$$\texttt{OTP} \equiv \texttt{Uniform}.$$

This shows that pRHL can formally capture standard cryptographic arguments by reasoning about couplings between probabilistic programs rather than directly manipulating probability distributions.

Additionally in pRHL, given a valid judgment of the form

$$\left[c_1 \sim c_2 : \Phi \Longrightarrow \bigwedge_{i=1}^{n} x_i\langle 1\rangle = x_i\langle 2\rangle \right],$$

it holds that $\Pr[c_1, m_1 : A] = \Pr[c2, m_2 : A]$ for every initial memories m_1 and m_2 that are related by Φ and event A that only depends on $\{x_1, \ldots, x_n\}$. A further generalization of observational equivalence is observational equivalence up to a failure event F

$$\left[c_1 \sim c_2 : \Phi \Longrightarrow \neg F\langle 2\rangle \rightarrow \bigwedge_{i=1}^{n} x_i\langle 1\rangle = x_i\langle 2\rangle \right].$$

It follows that from the above judgment, for every initial memories m_1 and m_2 that are related by Φ and event A that only depends on $\{x_1, \ldots, x_n\}$ it holds

$$\Pr[c_1, m_1 : A] \leq Pr[c_2, m_2 : A] + \Pr[c_2, m_2 : F].$$

This is useful to take into account upper bounds on the probability of some event (typically a failure).

The second key component of EasyCrypt is a probabilistic Hoare Logic (pHL) that enables probabilistic reasonging about certain events in a game. Judgments in pHL have the form

$$[c : \varsigma \Longrightarrow \varphi] \diamond p$$

, where c is a probabilistic program, ς and φ are (non-relational) assertions, $\diamond$ is a comparison operator and p is a probability expression.

In EasyCrypt, both pRHL and pHL are embedded into a higher-order logic, and reasoning within this logic is supported by a core proof engine

inspired by SsReflect, an extension of Coq [GM10]. To support compositional reasoning, which is essential when analyzing cryptographic systems, as they are typically constructed by composing multiple cryptographic primitives, EasyCrypt provides a module system. It enables the development of modular proofs that reflect the compositional structure of cryptographic constructions. In addition, it make easier the reasoning, organization and management of large and complex proofs involving numerous game hops and reductions at different levels of abstraction.

3.2 Squirrel

Squirrel is a proof assistant tool that enables automated verification techniques for the computational model. The system is based on the Computationally Complete Symbolic Attacker (CCSA) approach, which retains a symbolic reasoning framework while avoiding some of the limitations of traditional symbolic models. Rather than modeling an adversary's capabilities by explicitly specifying all actions the adversary can perform, the CCSA approach characterizes the adversary by specifying what it cannot do. Starting from the assumed security properties of cryptographic primitives, one derives inference rules that express which pairs of message sequences are computationally indistinguishable. Rules are proven sound with respect to an interpretation of terms as Probabilistic Polynomial-Time Turing Machines (PPTM). Any security proof carried out using the rules implies security in the computational model, given that the underlying cryptographic assumptions hold.

The CCSA approach reasons about cryptographic protocols viewed as concurrent programs (or games) that rely on cryptographic primitives while executing in a malicious environment. A central concept in this setting is computational indistinguishability: two games are said to be indistinguishable if any probabilistic polynomial-time adversary has at most a negligible probability of distinguishing between them. The CCSA framework uses first-

order logic to establish computational indistinguishability by first modeling the messages generated during a game's execution as first-order terms, and then reasoning about indistinguishability within first-order logic using inference rules derived from the assumed security properties of the underlying cryptographic primitives. Describing cryptographic games as probabilistic imperative programs allows cryptographers to rely on an intuitive understanding of their semantics and avoids the introduction of overly complex execution models. Formally reasoning about such programs can be challenging, as it requires handling statefulness, loop invariants, and probabilistic dependencies. Moreover, while encoding complex protocols as imperative programs is possible, it is unnatural and can significantly complicate security proofs.

The CCSA approach represents only the messages computed during a game, modeling them as first-order terms. These terms are pure in the functional programming sense and are therefore easier to reason about than full game descriptions that maintain mutable state. To illustrate this idea, let us consider a simple game.

$$
\begin{aligned}
&\mathtt{G}(n): \\
&\quad k \leftarrow \{0,1\}^n \\
&\quad m \leftarrow A() \\
&\quad t \leftarrow Enc_k(m) \\
&\quad x \leftarrow A(t) \\
&\quad \mathtt{return}\ x
\end{aligned}
$$

Figure 8: Example of a cryptographic game `G` to discuss a Squirrel proof.

A secret key k of length n (Squirrel normally uses η instead of n) is randomly sampled. The attacker A is asked to provide a message m which is then encrypted using k and sent back to A, which uses it to return a second

message x which is the final output of the game.

The sampling of the secret key can be modeled by a name symbol k. The first interaction with the adversary A is represented by the adversarial function symbol $\mathtt{att}_0(\cdot)$. Although the encryption function Enc_k may be probabilistic may be probabilistic, Squirrel explicitly tracks probabilistic dependencies by modeling the randomness used by the encryption algorithm. This is achieved by introducing a fresh name symbol $\mathtt{r}$ and representing the encryption as the term $\mathtt{enc}(\mathtt{att}_0(\cdot), \mathtt{r}, \mathtt{sk})$. The second interaction with the adversary is then modeled by another adversarial function symbol, for instance $\mathtt{att}_1(\mathtt{enc}(\mathtt{att}_0(\cdot), \mathtt{r}, \mathtt{sk}))$. This representation models two successive calls to the stateful adversary A in the game using two pure function symbols $\mathtt{att}_0$ and $\mathtt{att}_1$. Any state computed by the adversary during the first interaction and used in the second can be recomputed when modeling the second interaction as a pure function. This allows CCSA to avoid explicit state while preserving the adversary's capabilities.

n Squirrel, terms are parameterized by a security parameter $\eta \in \mathbb{N}$ and by sources of randomness provided by a pair of random tapes $\rho = (\rho_h, \rho_a)$, each filled with random bits. A model $\mathbb{M}$ of Squirrel's logic provides the following interpretations:

- For each honest function symbol $\mathtt{f}$ its interpretation as PPTM $\mathcal{M}_{\mathtt{f}}$
- Associate to each name $\mathtt{n}$ a unique sub-sequence of η bits in ρ_h using a PPTM $\mathcal{M}_{\mathtt{n}}$ with access to ρ_h, such that distinct names use disjoint parts of ρ_h;
- Interpret any adversarial function symbol $\mathtt{att}$ as a PPTM $\mathcal{M}_{\mathtt{att}}$ which can only access the random tape ρ_a and not ρ_h.

A term $\mathtt{t}$ is interpreted as a function that associates to each value of the security parameter η and random tapes ρ, a concrete value denoted by $[\![t]\!]^{\eta,\rho}_{\mathbb{M}}$. The mapping $\eta, \rho \mapsto [\![t]\!]^{\eta,\rho}_{\mathbb{M}}$ is computable by a PPTM.

As one of the most commonly used notions to define cryptographic properties is computational indistinguishability, Squirrel's logic relies on the symbol

$\sim$ to represent it. Formally, for two terms $\mathtt{u}$, $\mathtt{v}$, the predicate $\mathtt{u} \sim \mathtt{v}$ holds in a model $\mathbb{M}$ if $\mathtt{G}_{\mathtt{u}}^{\mathbb{M}} \approx \mathtt{G}_{\mathtt{v}}^{\mathbb{M}}$, and $\mathtt{G}_{\mathtt{t}}^{\mathbb{M}}$ is the game where the adversary is provided with $[\![t]\!]_{\mathbb{M}}^{\eta,\rho}$ and must produce a bit b, returned by the game. $\mathtt{u}$, $\mathtt{v}$ holds in $\mathbb{M}$ if exists a negligible function ϵ such that

$$\forall A \in \mathrm{PPTM}.\eta \mapsto |\Pr_\rho(A([\![u]\!]_{\mathbb{M}}^{\eta,\rho}, \rho_a) = 1) - \Pr_\rho(A([\![v]\!]_{\mathbb{M}}^{\eta,\rho}, \rho_a) = 1)| \leq \epsilon(\eta).$$

$\mathtt{G}_{\mathtt{k}}(\eta)$:
  $k \leftarrow \{0,1\}^\eta$
  $m \leftarrow A()$
  `if` $m = sk$ `then`
    `return` 0
  `else`
    `return` 1

Figure 9: Example of a cryptographic game $\mathtt{G}_{\mathtt{k}}$ to discuss computational indistinguishability in Squirrel.

Let's consider another game $\mathtt{G}_{\mathtt{k}}$ reported in Figure 9. The message returned at the end of the game can be represented by the term

$$\phi_k = \mathtt{if}\ (\mathtt{att}_0 = sk)\ \mathtt{then}\ 0\ \mathtt{else}\ 1,$$

which outputs 0 if the adversary correctly guesses the secret key k and 1 otherwise. Checking the indistinguishability judgment ($\phi_k \sim 1$) amounts to verify that the corresponding game $\mathtt{G}_{\mathtt{k}}$ is computationally indistinguishable from the trivial game (`return` 1) i.e., $G_k \approx$ (`return` 1).

The inference rules over $\sim$ in Squirrel represented in Figure 10. The key rule is the second one that states that if given u indistinguishable from v, then u can be replaced with v inside any context C without changing the overall indistinguishability from w.

$$\frac{}{u \sim u}\ \text{REF} \qquad \frac{(u = v) \sim \texttt{true} \quad C[v] \sim w}{C[u] \sim w}\ \text{REW} \qquad \frac{}{(t \neq \texttt{n}) \sim \texttt{true}}\ \text{FRESH}$$

Figure 10: Inference rules over $\sim$ in Squirrel.

One can verify that it is true that $(\texttt{if}\ (\texttt{att}_0(\cdot) = k)\ \texttt{then}\ 0\ \texttt{else}\ 1) \sim 1$ as follows

$$\dfrac{\dfrac{\dfrac{}{(\texttt{att}_0(\cdot) \neq k) \sim \texttt{true}}\ \text{FRESH}}{((\texttt{att}_0(\cdot) = k) = \texttt{false}) \sim \texttt{true}}\ \text{SIMPL}_{\equiv} \qquad \dfrac{\dfrac{}{1 \sim 1}\ \text{REF}}{(\texttt{if false then } 0 \texttt{ else } 1) \sim 1}\ \text{SIMPL}_{\equiv}}{(\texttt{if}\ (\texttt{att}_0(\cdot) = k)\ \texttt{then}\ 0\ \texttt{else}\ 1) \sim 1}\ \text{REW}$$

Here it is assumed an additional proof rule $\text{SIMPL}_{\equiv}$ that allows to replace a term with another one equal to it with probability one.

3.3 CryptoVerif

CryptoVerif is a mechanized proof assistant designed for the computational model of cryptography. It constructs security proofs as sequences of games, following the standard methodology of game-based reasoning in computational cryptography. The tool supports the formal verification of secrecy, authentication, and indistinguishability properties, and provides a generic framework for specifying and reasoning about the security of cryptographic primitives, including message authentication codes, symmetric and public-key encryption schemes, digital signatures, hash functions, and Diffie–Hellman key agreement protocols. For each proof, CryptoVerif derives an explicit bound on the probability of a successful attack.

The central notion used to express security properties in CryptoVerif is observational equivalence. Proofs proceed by transforming an initial game, representing the real protocol, into a final game in which the desired security property is evident. Each transformation step corresponds either to

a syntactic rewriting or to the application of a security assumption for a cryptographic primitive. The difference in the adversary's success probability between successive games must be negligible, ensuring that the overall transformation preserves security up to a negligible function.

CryptoVerif uses a formal probabilistic process calculus for representing cryptographic experiments. In this calculus, messages are modeled as bitstrings, and cryptographic primitives are represented as functions over bitstrings. Games are expressed as probabilistic processes that execute in polynomial time, while the adversary is modeled as a probabilistic Turing machine, thus faithfully capturing the standard computational assumptions underlying cryptographic security definitions.

In order to define oracles, CryptoVerif uses the following grammar:

$$
\begin{aligned}
Q ::= & \\
& \mid 0 \\
& \mid Q \mid Q' \\
& \mid \texttt{foreach}\, i \leq N\, \texttt{do}\, Q \\
& \mid O(x_1 : T_1, \ldots, x_m : T_m) := P,
\end{aligned}
$$

where the symbol 0 denotes termination, $Q \mid Q'$ denotes parallel computation, and the `foreach` construct denotes replication up to N times. The construct $O(x_1 : T_1, \ldots, x_m : T_m) := P$ defines an oracle with parameters $x_1, \ldots, x_m$ and body P. The oracle body can have the following forms:

$$
\begin{aligned}
P ::= & \\
& \mid \texttt{yield} \\
& \mid \texttt{return}(M_1, \ldots, M_m); Q \\
& \mid \texttt{event}\, e(M_1, \ldots, M_m); P \\
& \mid x \xleftarrow{R} T; P \\
& \mid x : T \leftarrow M; P \\
& \mid \texttt{if}\, M\, \texttt{then}\, P\, \texttt{else}\, P'
\end{aligned}
$$

$$\mid \texttt{insert}\, L(M_1, \ldots, M_m); P$$
$$\mid \texttt{get}\, L(M_1, \ldots, M_m)\, \texttt{suchthat}\, M\, \texttt{in}\, P\, \texttt{else}\, P'.$$

Here, `yield` denotes termination of the oracle execution, while the expression $\texttt{return}(M_1, \ldots, M_m); Q$ outputs a tuple of messages and continues with process Q. The construct $\texttt{event}\, e(M_1, \ldots, M_m); P$ records an event e and then proceeds with P. The command $x \xleftarrow{R} T; P$ denotes uniform random sampling of a value of type T binding the result to x, whereas $x : T \leftarrow M; P$ represents deterministic assignment. The conditional $\texttt{if}\, M\, \texttt{then}\, P\, \texttt{else}\, P'$ branches on the boolean value of M. Finally, $\texttt{insert}\, L(M_1, \ldots, M_m); P$ adds an entry to a list L, while $\texttt{get}\, L(M_1, \ldots, M_m)\, \texttt{suchthat}\, M\, \texttt{in}\, P\, \texttt{else}\, P'$ performs a lookup in L satisfying the condition M, proceeding with P if successful and with P' otherwise.

The processes in CryptoVerif are expressed using the syntax introduced above. These processes specify the oracles that an adversary may invoke, and their execution is bounded with respect to a security parameter, denoted by η. In particular, processes are bounded both in the number of oracle invocations and in the length of the messages provided as inputs to these oracles. Let $I_\eta(n)$ denote the interpretation of a bound n for a given value of the security parameter η. Then $I_\eta(n)$ is required to be an efficiently computable function of η that is bounded by a polynomial in η.

At the type level, the CryptoVerif calculus is parameterized by the security parameter η. For each η, every type T is interpreted as a subset $I_\eta(T)$ of $\mathit{Bitstring} \cup \{\bot\}$, where $\mathit{Bitstring}$ denotes the set of all finite bitstrings and $\bot$ is a special symbol. The set $I_\eta(T)$ must be recognizable in polynomial time, hence there exists an algorithm that, on input x, decides whether $x \in I_\eta(T)$ in time polynomial in the length of x and in the value of η.

A type is called *fixed-length* if T is such that $I_\eta(T)$ consists exactly of all bitstrings of a certain length $\ell(\eta)$, where ℓ is a polynomially bounded function of η while a type is called *large* if its cardinality is sufficiently high so that collisions between elements sampled uniformly at random from $I_\eta(T)$

occur only with negligible probability, while still allowing the tool to keep explicit track of the residual collision probability; formally, $\frac{1}{|I_\eta(T)|}$ is negligible as a function of η.

The boolean type is predefined as $bool = \{\,\texttt{true}, \texttt{false}\,\}$, with the convention $\texttt{true} = 1$ and $\texttt{false} = 0$. Moreover, the predefined type *bitstring* is interpreted as $I_\eta(bitstring) = Bitstring$, and the type $bitstring_\perp$ is interpreted as $I_\eta(bitstring_\perp) = Bitstring \cup \{\perp\}$.

The calculus further assumes a finite set of function symbols f, each equipped with a type declaration of the form

$$f : T_1 \times \cdots \times T_m \to T.$$

Each such symbol f denotes a function, also written f, from $T_1 \times \cdots \times T_m$ to T, such that $f(x_1, \ldots, x_m)$ is computable within time t_f, where t_f is bounded by a function of the lengths of the inputs $x_1, \ldots, x_m$. Function symbols f correspond to functions computable by deterministic Turing machines that always terminate. Some function symbols are written in infix notation, for example: $M = N$ denotes the equality test, which takes two values of the same type T and returns a value of type *bool* and $M \wedge N$ denotes the boolean conjunction, which takes and returns values of type *bool*.

Recalling Chapter 2, a MAC consists of a function $\mathsf{mac}(m, k)$ that takes as input a message m and a secret key k, together with a verification function $Vrfy(m, k, t)$ such that

$$Vrfy(m, k, mac(m, k)) = \texttt{true}$$

An authenticated encryption scheme can be constructed using the *encrypt-then-MAC* paradigm:

$$enc'(m, (k, mk)) = c_1 \,\|\, mac(c_1, mk),$$

where $c_1 = enc(m, k)$ and enc is a secure symmetric encryption algorithm. This construction can be modeled in Cryptoverif, using the following terms.

$$\texttt{letfun}\, full\_enc(m : bitstring, k : key, mk : mkey) =$$

$c_1 \leftarrow enc(m, k);$

$\texttt{concat}(c_1, mac(c_1, mk)).$

$\texttt{letfun}\ full\_dec(c : bitstring, k : key, mk : mkey) =$

$\texttt{let}\ \texttt{concat}(c_1, mac1) = c\ \texttt{in}$

$(\texttt{if}\ \texttt{verify}(c_1, mk, mac1)\ \texttt{then}\ \texttt{dec}(c_1, k)\ \texttt{elsebottom})$

$\texttt{else}$

$\texttt{bottom}.$

In this construction, the MAC is assumed to be SUF-CMA (strongly unforgeable under chosen-message attacks). Even an adversary with access to both the MAC and verification oracles has only negligible probability of producing a fresh valid tag not previously output by the MAC oracle. The encryption scheme is assumed to be IND-CPA (indistinguishable under chosen-plaintext attacks) that is, no efficient adversary can distinguish encryptions of two equal-length messages with more than negligible advantage.

Definition 3.1 (Indistinguishability under Chosen Plaintext Attacks)**.**

$$\mathrm{Succ}^{ind-cpa}_{SE}(t, q_e, l) = \max_{\mathcal{A}} 2\Pr\left[b \xleftarrow{R} \{0,1\}; k \xleftarrow{R} key; b' \xleftarrow{R} A^{enc(LR(.,.,b),k)} : b' = b\right] - 1$$

where $\mathcal{A}$ runs in time at most t, calls $enc(LR(.,.,b),k)$ at most q_e times on messages of length at most l, the left-or-right oracle LR is defined as $LR(x,y,0) = x$, $LR(x,y,1) = y$, and $LR(x,y,b)$ and is defined only when x and y have the same length.

Theorem 3.2. *The encrypt-then-MAC scheme is IND-CPA.*

Proof. We formalize the IND-CPA experiment for the encrypt-then-MAC scheme in CryptoVerif. We begin by defining an initialization oracle *Ostart* that is invoked by the adversary. The oracle samples a random bit b and generates fresh keys for both the encryption scheme and the MAC scheme.

$$Ostart() := b \xleftarrow{R} bool; k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \texttt{return}.$$

Next, we define the left-or-right encryption oracle.

$$\begin{aligned}
&\texttt{let } QencLR(b : bool, k : key, mk : mkey) = \\
&\quad \texttt{foreach } i \leq qEnc \texttt{ do} \\
&\quad Oenc(m_1 : bitstring, m_2 : bitstring) := \\
&\quad \texttt{if } Z(m_1) = Z(m_2) \texttt{ then} \\
&\quad m_0 \leftarrow \texttt{if } b \texttt{ then } m_1 \texttt{ else } m_2; \\
&\quad \texttt{return } full_enc(m_0, k, mk).
\end{aligned}$$

The construct `foreach` $i \leq qEnc$ represents $qEnc$ independent oracle instances, indexed by $i \in \{1, \ldots, qEnc\}$, so that the adversary may issue at most $qEnc$ encryption queries. The oracle takes two messages m_1 and m_2 as input and first checks that they have the same length. The condition $Z(m_1) = Z(m_2)$ ensures that m_1 and m_2 are of equal length, where $Z(x)$ denotes the bitstring of the same length as x containing only zeroes. If the test succeeds, the oracle selects $m_0 = LR(m_1, m_2, b)$ and returns $full\_enc(m_0, k, mk)$.

Finally, we consider the following process:

$$\begin{aligned}
&\texttt{process} \\
&\quad Ostart() := b \xleftarrow{R} bool; k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \texttt{return}; \\
&\quad \texttt{run } QencLR(b, k, mk).
\end{aligned}$$

CryptoVerif proves secrecy of the bit b in this process under the assumptions that the encryption scheme is IND-CPA secure and the MAC is SUF-CMA secure. Secrecy of b is equivalent to stating that no polynomial-time adversary can distinguish whether encryptions correspond to the left or the right messages with non-negligible advantage. Therefore, the encrypt-then-MAC construction is IND-CPA secure. A full example of this proof is available online at proverif24.paris.inria.fr/cryptoverif.php. □

Chapter 4

Lambda Calculus

The lambda calculus is a formal system introduced in the 1920s by Alonzo Church [Chu41], in which all computation is reduced to the basic operations of function definition and application. It can be viewed as a simple, but still Turing complete programming language in which computations are mathematical objects that can be used for proofs [Pie02]. The key idea is that computation consists of simplifying expressions; when an expression can no longer be reduced, the computation ends. lambda calculus can be enriched in many ways. Section 4.1 will present the basis of untyped lambda calculus while Section 4.2 will introduce the typed lambda calculus.

4.1 Untyped Lambda Calculus

Functional abstraction is one of the main ingredients of programming. It avoids code repetition and isolates meaningful pieces of code. When discussing functions in programming, particularly in functional programming, which is the focus here, common examples include the factorial function and the Fibonacci function. These examples are frequently used because of their recursive nature. For instance, the value of `5!` can be expressed in terms of previous factorials as `5! = 5 x 4!`. Similarly, the Fibonacci sequence is defined recursively: the value returned by the function that calculates the

n-th Fibonacci number, for $n > 2$ and denoted by `fib` satisfies

```
fib(n) = fib(n-1) + fib(n-2)
```

Rather than reasoning in terms of an algorithm that explicitly manipulates memory through loops and termination conditions, the same computation can be expressed by composing previously defined functions, allowing one to reason in a more mathematical and declarative way. If `λx.` is the function λ that applied to `x` returns something, then one can write

```
fib = λx. if x <= 1 then 1 else fib(x-1) + fib(x-2)
```

as the recursive function that calculates the n-th Fibonacci number.

The lambda calculus codifies these ideas in their purest form [Pie02]. It is a formalism in which every computation is expressed using anonymous unary functions, that is, functions that accept exactly one argument and yields a single output. Functions in lambda calculus may also take other functions as argument and return them as results. Since functions are anonymous, they are not associated with explicit names bound to the computations they represent.

The syntax of the lambda calculus consists solely of lambda-terms. The set of lambda-terms Λ is built up from an infinite set of variables $V = \{v_1, v_2, \dots, \}$ using application and function abstraction:

$$V ::= v \mid V' \tag{13}$$

$$\Lambda ::= v \mid \Lambda\Lambda \mid \lambda V.\Lambda \tag{14}$$

Arbitrary lambda-terms are commonly represented by $M, N, L, \dots$ while $x, y, z, \dots$ usually denote arbitrary variables. In 14 one possible form of a lambda-term is v, which represents a variable. The term $\Lambda\Lambda$ denotes the application of the first term to the second. The expression $\lambda V.\Lambda$ is called a lambda-abstraction (or simply an abstraction) and it represents the explicit definition of an anonymous function that binds its formal parameter V and has body Λ. In conventional mathematical notation, such a function might be written as $x \mapsto M$ or $f(x) = t$ if the the lambda-abstraction were given the name f and if it were written as $\lambda x.M$.

There are some conventions for writing lambda-expressions. The first is that application is left-associative. For example, the expression $(t\,s\,u)$ is the same as $(t\,s)u$. The second one is that application has higher precedence than abstraction. Hence, the expression $\lambda x.tt$ is to be interpreted as $\lambda x.(tt)$, and not as $(\lambda x.t)t$. The third convention is that the body of an abstraction extends as far to the right as possible. For example, $\lambda x.\lambda y.x\,y\,z$ is the same as $\lambda x.(\lambda y.((x\,y)z))$. In the last expression $(\lambda x.\lambda y.x\,y\,z)$, the lambda-term z does not appear as one of the arguments of the abstractions. In this case the lambda-term z is said to be a free variable. An occurrence of the variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$. The λ is also called *binder*, because it said it binds its argument (x in the last example) in its body t. In this context t is called the scope of the abstraction. When a lambda-term appear in an expression and there are no binders capturing it, then it's a free variable. For example in the expression

$$x\lambda x.\lambda y.w\,x\,y$$

the first instance of x is free, while the second one is captured by the binder. w is also free. The set of free variables of a lambda-term t is often denoted by $FV(t)$. $FV(t)$ is defined inductively by

$$\begin{aligned} FV(x) &= x \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

Figure 11: Definition of free variables in the untyped lambda calculus

Let's consider the expression $\lambda x.x\,y(\lambda y.y\,z)$. Applying the definition of FV the set of the free variables of $\lambda x.x\,y(\lambda y.y\,z)$ is

$$\begin{aligned} &FV(\lambda x.xy(\lambda y.yz)) = \\ &= FV(xy(\lambda y.yz) \setminus \{x\}) = (FV(x) \cup FV(y) \cup FV(\lambda y.yz)) \setminus \{x\} \end{aligned}$$

$$= (\{x\} \cup \{y\} \cup (FV(yz) \setminus \{y\})) \setminus \{x\} = \{x, y, z\} \setminus \{x\} = \{y, z\}$$

Whenever a lambda-term M has $FV(M) = \emptyset$ then it is called combinator (or closed lambda-term).

The computation in lambda calculus is performed only through function application. Applying functions to each other means rewriting the abstractions by substituting the bounded variables in the body. But substitution is not always safe. Let $M[N/x]$ denote the term obtained by substituting all free occurrences of the variable x in M with the term N. The substitution operation is defined inductively as shown in Figure 12 The last case could be

$$\begin{aligned} x[N/x] &= N \\ y[N/x] &= y \\ (ML)[N/x] &= (M[N/x])(L[N/x]) \\ (\lambda x.M)[N/x] &= \lambda x.M \\ (\lambda y.M)[N/x] &= \lambda y.(M[N/x]) \text{ if } x \neq y \end{aligned}$$

Figure 12: Definition of substitution in the untyped lambda calculus.

extended to prevent variable capture: the bound variable y may be first renamed to a *fresh* variable z before performing the substitution. The freshness condition ensures that z does not occur free in either M or N. Given a term M, the set of its free variables $FV(M)$ is finite, whereas the set of terms that are alpha-equivalent to M, denoted by $\overset{\alpha}{\equiv}$, is infinite (by 13). Consequently, there always exists a term t such that $M \overset{\alpha}{\equiv} t$, allowing bound variables to be renamed as needed to avoid capture.

In the lambda calculus, each step of a computation consists of rewriting an application whose left-hand side is an abstraction, by substituting the free occurrences of the bound variable with the argument. For example the expression $(\lambda x.\lambda y.x\, y)(\lambda z.z)(\lambda z.z)$ evaluates to $(\lambda z.z)$. However, there are multiple ways to reduce the original term, depending on which reducible

expression is chosen at each step. A term of the form $(\lambda x.M)$ beta-reduces to a term u, written by $(\lambda x.M) \rightarrow_\beta u$ if u is the result of substituting N for the free occurrences of x in M, that is $u = M[N/x]$. For example $(\lambda y.x\,y)x \rightarrow_\beta \lambda x.x\,x$. An expression of the form $(\lambda x.M)N$ is called a redex (short for reducible-expression) [Chu41]. The binary relation $\rightarrow_\beta$ is defined using the following inference rules

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]} \qquad \frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}$$

$$\frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

Figure 13: Inference rules for the binary relation $\rightarrow_\beta$ untyped lambda calculus.

Given the formal definition of beta-reduction, and lambda calculus' operational semantics, the binary relations $\twoheadrightarrow_\beta$ *and* $=_\beta$ on Λ are defined inductively as follows

1. (a) $M \twoheadrightarrow_\beta M$;
   (b) $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$;
   (c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$.
2. (a) $M \twoheadrightarrow_\beta N \Rightarrow M =_\beta N$;
   (b) $M =_\beta N \Rightarrow N =_\beta M$;
   (c) $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$.

The relation $\twoheadrightarrow_\beta$ is the transitive reflexive closure of $\rightarrow_\beta$, while $=_\beta$ is a congruence relation [Bar13]. Beta-reduction introduces non-determinism into computation, since the order in which redexes are reduced may lead to different reduction sequences. This choice is known as an evaluation strategy.

Several strategies have been studied: full beta reduction, where any redex may be reduced at any time; normal order, where the leftmost outermost redex is reduced first; call by name, which is similar to normal order but forbids reduction inside abstractions; and call by value, where only outermost redexes are reduced and a redex is contracted only when its argument has been reduced to a normal form [Pie02].

A term M is said to be in *normal form* if M contains no beta-redex; that is, no subterm of the form $(\lambda x.P)\,Q$.

Let M be a term, M is *weakly normalisable* if there exists some M' in normal form such that $M \twoheadrightarrow_\beta N'$. We say that M is *strongly normalisable* if there does not exist an infinite beta-reduction path starting from M. There exist terms in the untyped lambda calculus which are neither strongly nor weakly normalisable, for example $(\lambda x.xx)(\lambda x.xx)$, as well as terms which are weakly but not strongly normalisable such as $(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))$. However in the simply typed lambda calculus every lambda-term is strongly normalisable [Gir89].

Up to this point, the lambda calculus may appear promising as a practical programming language. However, since every term is an anonymous unary function, two questions may arise: how can functions that take multiple arguments be expressed, and how can iteration in the form of recursion be encoded? In the lambda calculus, a binary function $f(x, y) = g(x, y)$ is represented as $\lambda x.\lambda y.g\,x\,y$. More generally, functions of arbitrary arity are encoded by using a sequence of abstractions, each taking one argument at a time. This representation also enables partial function application. For example, the expression $((\lambda x.\lambda y.x+y)1)$ reduces to $\lambda y.1+y$, which denotes a function that always increments its argument by `1`. Regarding the encoding of recursion in the lambda calculus, let's consider a set A and a function $f : A \to A$. An element $x \in A$ is called a fixed point of f if and only if $f(x) = x$. In mathematics, functions do not in general admit fixed points; for example, the function $f(x) = x+1$ has no fixed points. In computer science, however, the notion of a function differs from the classical mathematical

definition. While a mathematical function associates exactly one output with each input in its domain, computational functions may be partial or divergent, meaning that they may fail to return a result for some inputs. In the lambda calculus, a lambda term t is a fixed point of f if and only if $f(t) =_\beta t$. Unlike the classical mathematical setting, fixed points always exist in the untyped lambda calculus. For example, the term

$$(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

is a fixed point of f since

$$f((\lambda x.f(x\,x))(\lambda x.f(x\,x))) =_\beta (\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

A lambda term Y is called a fixed-point operator if, for every lambda term f, the term Yf is a fixed point of f. A standard example of such an operator is

$$Y \equiv \lambda f.((\lambda x.f(x\,x))(\lambda x.f(x\,x)))$$

This allows Y to generate fixed points, where Yf is equal to itself under the function f, creating recursion without explicit self-reference (since lambda-terms are anonymous). In this way, the untyped lambda calculus models recursion through self-application.

The lambda calculus can also encode primitive data types commonly found in most programming languages, such as booleans. These can be represented as follows:

$$\mathsf{true} = \lambda x.\lambda y.x$$
$$\mathsf{false} = \lambda x.\lambda y.y$$

Given a boolean value B, a conditional expression of the form

```
if B then M else N
```

can be expressed by the lambda term BMN since ($\mathsf{true}\,MN = M$) and ($\mathsf{false}\,MN = N$). In a similar way one can construct ordered pairs using

$$[M, N] = \lambda z.zMN,$$

and retrieve the first or the second element using $([M, N]\,\mathsf{true} = M)$ and $([M, N]\,\mathsf{false} = N)$.

4.2 Simply Typed Lambda Calculus

In mathematics, functions are usually defined on a fixed domain and range, there is no corresponding requirement in type-free lambda calculus [Sø07]. The lambda calculus can be extended with type annotations that specify the kinds of inputs a term accepts and the kinds of outputs it produces. There are two distinct approaches to introducing types into the lambda calculus: in Church-style typing [Chu40], variables and abstractions are explicitly annotated with their types, and each well-typed term has a unique type. In contrast, Curry-style typing [Cur34] does not require type annotations on variables; instead, types are assigned to terms after they are constructed, and typable expressions admit a unique most general type. In Church's approach, a term's type is a built-in part of the term itself, the modern analogy would be a programmer explicitly writing the types for all variables used in the program as in, e.g., Pascal. In Curry's approach, a term's type is inferred after the term has been defined, the programmer writes functions, and it is then the job of the compiler to infer the types of variables, as e.g., in ML and Haskell [vR09, Sø07]. The aim of this section is to introduce the simply typed lambda calculus following Curry's approach, hence first defining terms, how they behave and then defining a basic type system.

At the end of Section 4.1, the expression $((\lambda x.\lambda y.x + y)1)$ was introduced as an example of a term that reduces to another function, namely $\lambda y.1 + y$, which increments its argument by 1. However, one may ask what happens if the formal argument y is instantiated with a function instead of a number, for example in the expression $(\lambda y.1 + y)(\lambda x.x)$. In this case, the meaning of applying the + operator to a number and a function is unclear. This is a common situation in which type systems capture errors early, preventing

erroneous behaviour. Such errors correspond to terms that cannot be reduced and therefore become stuck, since no reduction rules can be applied to them [Pie02].

Given the syntax of terms

$$V ::= v \mid V'$$
$$\Lambda ::= v \mid \Lambda\Lambda \mid \lambda V.\Lambda$$

let T denote a denumerably infinite alphabet whose members will be called type variables, the set T of simple types is the set of strings defined by the grammar

$$\Pi ::= U \mid (\Pi \to \Pi)$$

Usually α, β, γ denotes type variables, and σ, τ arbitrary types. $(\Pi \to \Pi)$ denotes the type of functions that accept an argument of type Π and return a result of type Π, conventionally $\to$ is right-associative. The set C of contexts is the set of all sets of pairs of the form

$$C ::= \{x_1 : \tau_1, \ldots x_n : \tau_n\}$$

where $\{x_1, \ldots x_n\} \in \Lambda$ and $\{\tau_1, \ldots x_n\} \in T$ and $x_i \neq x_j$ for $i \neq j$. The domain of a context $\Gamma = \{x_1 : \tau_1, \ldots x_n : \tau_n\}$ is defined by:

$$\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$$

The possible forms of the contexts $\Gamma, \Delta, \ldots$ are given by

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

where $\emptyset$ denotes an empty context and $\Gamma, x : \tau$ represents the extension of the context Γ with a new variable x of type τ. The typability relation on $C \times \Lambda \times T$ is defined by the inference rules shown in Figure 14.

The first and second rule require $x \notin \mathrm{dom}(\Gamma)$. If a variable x is assigned the type τ in the context Γ, then x has type τ. To type an abstraction $\lambda x.M$, a type σ is assumed for the argument x and added to the context; if the body

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Figure 14: Typability relation on $C \times \Lambda \times T$ in the simply typed lambda calculus.

M can then be assigned type τ, the abstraction has type $\sigma \to \tau$. Finally, if a function M has type $\sigma \to \tau$ and an argument N has type σ, then applying the function to the argument yields a term of type τ.

The simply typed lambda calculus à la Curry is often denoted by $\Lambda\!\to$ and is the triple $(\Lambda, T, \vdash)$.

If $\Gamma \vdash M : \sigma$ then M has type σ in Γ. $M \in \Lambda$ is typable if there are Γ and σ such that $\Gamma \vdash M : \sigma$ and is a ternary relation. Here, Γ is called the typing context and consists of a finite set of assumptions about the free variables of M, containing the association of each variable with its type. A formal proof of $\Gamma \vdash M : \tau$ is given by a finite tree whose nodes are labelled by pairs of form $(\Gamma', M' : \tau')$, which will also be written satisfying the following conditions:

- The root node is $\Gamma \vdash M : \tau$;
- All leaves are labelled by axioms;
- The label of each father node is obtained from the labels of the sons using one of the rules.

Let's consider the expression $(\lambda f.\lambda x.fx)$ and apply the typing rules to derive its type in the empty context ϵ.

$$\cfrac{\cfrac{\cfrac{f : \sigma \to \tau, x : \sigma \vdash f : \sigma \to \tau \qquad f : \sigma \to \tau, x : \sigma \vdash x : \sigma}{f : \sigma \to \tau, x : \sigma \vdash fx : \tau}}{f : \sigma \to \tau \vdash \lambda x.fx : \sigma \to \tau}}{\vdash \lambda f.\lambda x.fx : (\sigma \to \tau) \to \sigma \to \tau}$$

Thus the term $(\lambda f.\lambda x.fx)$ has type $(\sigma \to \tau) \to \sigma \to \tau$ in the empty context. However, not all terms are typable in simply typed lambda calculus. Attempting to type the term $(\lambda x.xx)$ (the self-application combinator ω) leads to a requirement for an infinite type. This is intentional, as it prevents the construction of general recursion ensuring that all typable terms in the system are strongly normalizing hence guaranteed to converge to a normal form.

While the untyped lambda calculus is a universal model of computation, its expressiveness has some drawbacks. In an untyped setting, any term may be applied to any other term, which allows the formation of meaningless expressions and non-terminating computations, such as the ω combinator $(\lambda x.xx)$ or the ill term $(\lambda y.1 + y)(\lambda x.x)$. Type systems impose a formal structure on terms in order to address these issues, with two objectives: logical consistency and guarantee on termination. Early formulations of the lambda calculus were shown to be logically inconsistent when interpreted as foundations for mathematics [KR35]. Type systems, such as Church's simply typed lambda calculus, were introduced to restore consistency by restricting the class of valid terms and eliminating problematic forms as the self-reference. Another benefit of many typed systems is the property of strong normalization, which guarantees that every well-typed term reduces to a normal form. In the simply typed lambda calculus, self-application (such as ω) is untypable, preventing the construction of terms that lead to infinite reduction sequences. As a result, every typable computation is guaranteed to terminate, ensuring that evaluation always produces a normal form [Gir89].

Chapter 5

Lambda-BLL

This chapter introduces the λ**BLL** calculus [LGG24], a typed probabilistic lambda calculus for modeling polynomial-time cryptographic computation. Its design and language features capture the essentials for expressing cryptographic experiments, adversaries, and reductions.

λ**BLL** provides a programming-language framework in which computational complexity is enforced statically through the type system. In particular, it supports probabilistic polynomial-time computation, bounded interaction with oracles, shared mutable state, and higher-order reasoning about cryptographic constructions.

The type system of λ**BLL** is linear and graded, closely related to Bounded Linear Logic [GSS92] and graded calculi [RP10]. Resource usage is tracked by polynomial grades, generated by the grammar

$$p ::= 1 \mid i \mid p + p \mid p * p \qquad \text{(Polynomials)}$$

where the variable i represents the security parameter.

The calculus follows a call-by-push-value (CBPV) discipline [Lev12], distinguishing between positive types and general types. This separation permits to restrict function application to positive arguments only.

The base of the type system consists of ground types:

$$\mathbb{U} \mid \mathbb{B} \mid \mathbb{S}[p]$$

representing unit, booleans, and fixed-length bitstrings of polynomially bounded size. These types correspond to the fundamental data manipulated in cryptographic protocols, such as nonces, messages, and keys.

Positive types extend ground types with

$$P \otimes P \mid !_p A,$$

namely pairs formed with the tensor products $P \otimes P$, and the graded comonadic modality $!_p A$.

In a linear type system, a value must normally be used exactly once. The modality $!_p A$ relaxes linearity by allowing a value of type A to be used up to p times. This is central to the cryptographic interpretation of the calculus because it enables a static enforcement of polynomial-time bounds. For example, an oracle of type $!_{q(i)}\mathit{Oracle}$ may be queried at most $q(i)$ times by an adversary for a polynomial q. Any attempt to exceed this bound results in an ill-typed term, ensuring that polynomial-time constraints are enforced by typing rather than by operational reasoning. As a result, the proof fails at the type level rather than at runtime.

$$\begin{aligned} G &::= \mathbb{U} \mid \mathbb{B} \mid \mathbb{S}[p] && \text{(Ground types)} \\ P &::= G \mid P \otimes P \mid !_p A && \text{(Positive types)} \\ A &::= P \mid P \multimap A && \text{(Types)} \end{aligned}$$

Figure 15: Types of λ**BLL**.

λ**BLL** extends the pure lambda calculus with other features, necessary for modelling cryptographic computation, in particular it defines a collection of function symbols $\mathcal{F}$. Each function symbol $f \in \mathcal{F}$ has a type denoted by $\mathsf{typeof}(f)$ of the form $G_1 \times \cdots \times G_m \to G$ where $G_1, \ldots, G_m$ and G are ground types. Then, for each polynomial p, in λ**BLL** there is a corresponding constructor f_p representing a resource-bounded invocation of f, with the grade p accounting for its computational cost within the type system.

Values represent data that can be passed, stored or paired but not directly executed, while computations represent effectful execution. Ground values include $\star$, booleans, and bitstrings. Positive values include variables, pairs, and suspended computations of the form $!M$. Computations follow the CBPV structure so values are injected using `return`, suspended computations are forced using `der`, and applications is defined only on positive arguments.

$$
\begin{aligned}
W &::= \star \mid \mathbf{t} \mid \mathbf{f} \mid s && \text{(Ground values)} \\
Z &::= x \mid W \mid \langle Z, Z \rangle \mid !M && \text{(Positive values)} \\
\mathcal{V} \ni V &::= Z \mid \lambda x.M && \text{(Values)} \\
\Lambda \ni M &::= \mathtt{return}\, V \mid \mathtt{der}(Z) \mid MZ && \text{(Computations)} \\
&\quad \mid \mathtt{let}\, x = N\, \mathtt{in}\, M \mid \mathtt{let}\, \langle x, y \rangle = Z\, \mathtt{in}\, M \\
&\quad \mid \mathtt{if}\, Z\, \mathtt{then}\, M\, \mathtt{else}\, N \mid \mathtt{loop}\, V\, p\, \mathtt{times}\, \mathtt{from}\, M \\
&\quad \mid \mathtt{set}\, r\, Z \mid \mathtt{get}\, r \\
&\quad \mid f_p(Z_1, \ldots, Z, m)
\end{aligned}
$$

Figure 16: Syntax of λ**BLL**.

To model cryptographic experiments involving shared state, λ**BLL** includes global references. References store ground values (and ground only) and are manipulated via $\mathtt{set}\, r\, Z$ and $\mathtt{get}\, r$. This ensures that stateful effects do not introduce higher-order behavior beyond what is explicitly admitted by the type system. The language provides also control constructs such as conditionals, let-bindings, and pattern matching on pairs. Bounded iteration is expressed using the construct $\mathtt{loop}\, V\, p\, \mathtt{times}\, \mathtt{from}\, M$, which executes the computation M at most p times.

Since variables in λ**BLL** are classified according to their types, a variable is said to be ground if its type contains no function arrows, preventing ground

terms from encoding higher-order behavior thus being dangerous as global references. To track higher-order dependencies, the calculus distinguishes the set of free higher-order variables of a term M, denoted $HOFV(M)$. This set contains exactly those free variables whose types include at least one occurrence of the function type constructor $(\multimap)$.

The typing rules for λ**BLL** are presented in Figure 17 and formalize its type system through a set of rules that assign types to terms while respecting linearity and resource constraints. λ**BLL** considers two different typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, x : P \qquad \text{(Variable context)}$$
$$\Theta ::= \emptyset \mid \Theta, r : P \qquad \text{(Reference context)}$$

The context Γ contains term variables with positive types only, which are not computations, while the reference context Θ contains references which are can only be ground types. Fixed a reference context Θ then there are two typing judgments with respect to Θ.

$$\Gamma \vdash_v^{\Theta} V : A \qquad \text{(Value typing)}$$
$$\Gamma \vdash_c^{\Theta} M : A \qquad \text{(Computation typing)}$$

Computations may access references, but references never contain computations. The notation $\pi \triangleright \Gamma \vdash_v^{\Theta} M : A$ and $\pi \triangleright \Gamma \vdash_c^{\Theta} M : A$ indicates that π is a type derivation witnessing, respectively, the value or computation typing judgment under variable context Γ and reference context Θ. Type derivations are preserved under the substitution of the security parameter with a polynomial which is crucial for modelling polynomial-time computations and for reasoning about families of terms that are indexed by the security parameter.

The polynomial addition induces a binary partial operation $\boxplus$ on positive types defined by induction

$$G \boxplus G := G$$

$$(P \otimes Q) \boxplus (R \otimes S) := (P \boxplus R) \otimes (Q \boxplus S)$$
$$(!^{\Theta}_{p} A) \boxplus (!^{\Theta}_{q} A) := !^{\Theta}_{p+q} A$$

The partial operation $\boxplus$ can be extended as a total operation on contexts

$$\varnothing \boxplus \varnothing := \varnothing$$
$$(\Gamma, x : P) \boxplus \Delta := \begin{cases} \Gamma, x : P \boxplus \Delta & \text{if } x \text{ does not occur in } \Delta \\ x : P \boxplus Q, \Gamma \boxplus \Sigma & \text{if } \Delta = \Sigma, x : Q \end{cases}$$

And to account for polynomial multiplication, for every polynomial $p \in \mathbb{N}_{\geq 1}[i]$ there is a total unary operation on positive types by induction:

$$p * G := G$$
$$p * (P \otimes Q) := (p * P) \otimes (p * Q)$$
$$p * (!^{\Theta}_{q} A) := !^{\Theta}_{p \times q} A$$

The operation $p * (-)$ on positive types can also be extended to a total operation on term variables contexts:

$$p * \varnothing := \varnothing$$
$$p * (x : P, \Gamma) := (x : p * P), p * \Gamma$$

The notion of substitution in λ**BLL** is defined with respect to the security parameter i. Given a polynomial $p \in \mathbb{N}_{\geq 1}[i]$ and a type A, the type Ap is the type obtained by substituting every occurrence of the parameter i in A with the polynomial p, that is, $A[p/i]$. Similarly, for a term M, the type Mp to denote the term obtained by substituting p for i in all polynomial annotations occurring in M, namely $M[p/i]$.

A computation in λ**BLL** can be probabilistic and can read or modify the global reference context. In order to model probabilistic effects together with mutable references, λ**BLL** combines the distribution monad with the state monad.

To account for probabilistic effects, λ**BLL** models probabilistic computation using finite probability distributions. Given a set X, a probability

$$\frac{}{x : P \vdash_v^\Theta x : P}\ \text{VAR} \qquad \frac{}{\vdash_v^\Theta \mathbf{t} : \mathbb{B}}\ \text{TRUE} \qquad \frac{}{\vdash_v^\Theta \mathbf{f} : \mathbb{B}}\ \text{FALSE}$$

$$\frac{\mathsf{typeof}(f) = G_1 \times \cdots \times G_m \to G \quad (\Gamma_k p \vdash_v^\Theta Z_k : G_k p)_{1 \le k \le m} \quad p \in \mathbb{N}_{\ge 1}[i]}{\boxplus_k \Gamma_k p; \Theta \vdash_c^\Theta f_p(Z_1, \ldots, Z_m) : Gp}\ \text{FUN}$$

$$\frac{s \in \{0,1\}^c \quad p : i \mapsto c \text{ is a constant polynomial}}{\Gamma \vdash_v^\Theta s : \mathbb{S}[p]}\ \text{STRING} \qquad \frac{\Gamma \vdash_v^\Theta Z_1 : P \quad \Delta \vdash_v^\Theta Z_2 : Q}{\Gamma \boxplus \Delta \vdash_v^\Theta \langle Z_1, Z_2 \rangle : P \otimes Q}\ \text{TENSOR}$$

$$\frac{\Gamma \vdash_v^\Theta Z : P \otimes Q \quad \Delta, x : P, y : Q \vdash_c^\Theta M : A}{\Gamma \boxplus \Delta \vdash_c^\Theta \mathtt{let}\, \langle x, y \rangle = Z \,\mathtt{in}\, M : A}\ \text{LET-PAIR} \qquad \frac{}{\vdash_v^\Theta \star : \mathbb{U}}\ \text{UNIT}$$

$$\frac{\Gamma \vdash_c^\Theta M : A}{p * \Gamma \vdash_v^\Theta\, !M : !_p^\Theta A}\ \text{BANG} \qquad \frac{\Gamma \vdash_v^\Theta Z : !_q^\Theta A}{\Gamma \vdash_c^\Theta \mathtt{der}(Z) : A}\ \text{DER} \qquad \frac{\Gamma \vdash_c^\Theta M : P \overset{\Theta}{\multimap} A \quad \Delta \vdash_v^\Theta Z : P}{\Gamma \boxplus \Delta \vdash_c^\Theta MZ : A}\ \text{APP}$$

$$\frac{\Gamma, x : P \vdash_c^\Theta M : A}{\Gamma \vdash_v^\Theta \lambda x.M : P \multimap A}\ \text{LAM} \qquad \frac{\Gamma \vdash_v^\Theta V : A}{\Gamma \vdash_c^\Theta \mathtt{return}\, V : A}\ \text{ETA}$$

$$\frac{\Gamma \vdash_v^\Theta Z : P \otimes Q \quad \Delta, x : P, y : Q \vdash_c^\Theta M : A}{\Gamma \boxplus \Delta \vdash_c^\Theta \mathtt{let}\, \langle x, y \rangle = Z \,\mathtt{in}\, M : A}\ \text{LET} \qquad \frac{\Gamma \vdash_v^\Theta V : P \multimap P \quad \Delta \vdash_c^\Theta M : P}{(p * \Gamma) \boxplus \Delta \vdash_c^\Theta \mathtt{loop}\, V\, p\, \mathtt{times\, from}\, M : P}\ \text{LOOP}$$

$$\frac{\Gamma \vdash_v^\Theta Z : G \quad r : G \in \Theta}{\Gamma \vdash_c^\Theta \mathtt{set}\, r\, Z : \mathbb{U}}\ \text{SET} \qquad \frac{r : G \in \Theta}{\vdash_c^\Theta \mathtt{get}\, r : G}\ \text{GET} \qquad \frac{\Gamma \vdash_v^\Theta Z : \mathbb{B} \quad \Delta \vdash_c^\Theta M : A \quad \Delta \vdash_c^\Theta N : A}{\Gamma \boxplus \Delta \vdash_c^\Theta \mathtt{if}\, Z \,\mathtt{then}\, M \,\mathtt{else}\, N : A}\ \text{CASE}$$

$$\frac{\Gamma \vdash_v^\Theta V : A \quad x \text{ not free in } \Gamma}{\Gamma, x : P \vdash_v^\Theta V : A}\ \text{WEAK} \qquad \frac{\Gamma \vdash_c^\Theta M : A \quad x \text{ not free in } \Gamma}{\Gamma, x : P \vdash_c^\Theta M : A}\ \text{WEAK}$$

Figure 17: Typing rules of λ**BLL**.

distribution over X is a function $\mu : X \to [0,1]$ with finite support and total mass equal to 1. Any such distribution can be written as a finite weighted sum of Dirac distributions δ_x, which represent deterministic outcomes. This construction induces a monad $(\mathbf{D}, \eta_{\mathbf{D}}, \gg=_{\mathbf{D}})$ on the category **Set** of sets and functions. The unit maps a value x to the Dirac distribution δ_x, embedding pure values into probabilistic computations having components $\eta_X : x \mapsto \delta_x$, while the bind operation models sequential probabilistic computation by combining distributions:

$$\gg=: \mathbf{D}(X) \times \mathbf{Set}(X, \mathbf{D}(Y)) \to \mathbf{D}(Y).$$

Here, $\mathbf{D}(X)$ denotes the set of all probability distributions over X.

To model mutable references, λ**BLL** uses a state monad whose state space is given by the set of stores associated with a fixed reference context Θ. A store assigns to each reference a ground value of the appropriate type, thereby ensuring that memory remains well-typed throughout execution. Reading a reference retrieves its current value from the store, while writing to a reference produces a new store in which the corresponding location is updated.

The full semantics of computations in λ**BLL** is obtained by combining probabilistic choice with state. For a fixed reference context Θ, computations returning values in a set X are interpreted as functions from stores to probability distributions over pairs of values and stores. For every closed reference context Θ, there is a corresponding monad $(\mathbf{T}_\Theta, \eta_\Theta, \gg\!=_\Theta)$ on **Set**, defined as the tensor product of the distribution monad with the state monad, namely

$$\mathbf{T}_\Theta := \big(\mathbf{D}(X \times \mathrm{St}_\Theta)\big)^{\mathrm{St}_\Theta}$$

The unit of $\mathbf{T}_\Theta$ has components $\eta_\Theta : X \to \mathbf{D}(X \times \mathrm{St}_\Theta)^{\mathrm{St}_\Theta}$, mapping a value $x \in X$ and a store $e \in \mathrm{St}_\Theta$ to the Dirac distribution $\delta_{(x,e)}$, corresponding to a deterministic computation that returns x without modifying the store. The bind operator

$$\gg\!=_\Theta \colon \mathbf{T}_\Theta X \times \mathbf{Set}(X, \mathbf{T}_\Theta Y) \to \mathbf{T}_\Theta Y$$

maps $\varphi \in \mathbf{T}_\Theta X$ and $f : X \to \mathbf{T}_\Theta Y$ to the function $\boldsymbol{\lambda} e.\, \big(\varphi(e) \gg\!=_{\mathbf{D}} \mathbf{eval} \circ (f \times \mathrm{id}_{\mathrm{St}_\Theta})\big)$, where $\boldsymbol{\lambda}$ denotes currying and **eval** is the evaluation map induced by the Cartesian closed structure of **Set**.

In λ**BLL**, application is written as MZ, and the constructor `return` lifts values into computations. The syntactic sugar

$$V\,Z \triangleq (\texttt{return}\, V)\, Z \tag{15}$$

simplifies expressions by implicitly promoting values to computations, reducing the need for nested `return` constructs. Moreover, the syntactic sugar

$$M; N \triangleq \texttt{let}\ \_ = M\ \texttt{in}\ N \tag{16}$$

is used to compose effectful computations, such as the `set` command, without binding a variable. Finally, the sugar

$$\texttt{skip} \triangleq \texttt{return}\star \tag{17}$$

denotes a computation that performs no effect.

5.1 Useful Functional Symbols and Equations

Functional Symbols We introduce a collection of function symbols for λ**BLL**. These symbols capture fundamental computational patterns such as randomness, bitstring manipulation, and memory operations and they extend the set of function symbols presented in [LGG24], which are recalled in Appendix A, Table A.1.

Table 5.1 presents the new function symbols, for clarity they are organized into two categories according to their intended purpose.

Function Symbol	Type	Interpretation
Function Symbols for Manipulating Bitstrings		
`eq`	$\mathbb{S}[i] \times \mathbb{S}[i] \to \mathbb{B}$	Function that takes as input two strings of length i, and returns true if they are equals and false otherwise.
Function Symbols for Ledger Manipulation		
$\texttt{initQ}_q$	$\mathbb{S}[q \times (i+1)]$	Function that initializes a query ledger by generating a string of length $q \times (i+1)$ filled entirely with zeros.
$\texttt{contains}_q$	$\mathbb{S}[q \times (i+1)] \times \mathbb{S}[i] \to \mathbb{B}$	Function that, given a query ledger and an input string of length i, searches for a row in the ledger whose first i bits are equal to the input string and whose final bit is set to 1 and returns a boolean indicating whether the substring was found or not.

$\texttt{modify}_q$	$\mathbb{S}[q \times (i+1)] \times \mathbb{S}[i] \times \mathbb{S}[i] \rightarrow \mathbb{S}[q \times (i+1)]$	Function that, given a query ledger and two input strings of length i, identifies the first inactive row in the ledger, inserts into that row the concatenation of the two strings followed by a bit set to 1 (indicating activation), and returns the resulting updated ledger.
$\texttt{coherent}_{q,p}$	$\mathbb{S}[q \times (i+1)] \times \mathbb{S}[p \times (2i+1)] \rightarrow \mathbb{B}$	Function that, given two ledgers, it verifies that every message present in the first ledger exists as a key in the the second ledger, and conversely, that no keys exist in the second ledger that are not also recorded in the first one.

Table 5.1: Table of function symbols and their interpretations. We consider ledgers modelled by strings of length $(q \times (i+1))$ and $(p \times (2i+1))$, where p and q are polynomials in $\mathbb{N}_{\geq 1}[i]$. This string represents a table with q (resp., p) rows, where each row consists of two strings of length i (resp., $2i$) followed by a single bit, which is set to 1 if the corresponding row is active, and 0 otherwise.

The first group of function symbols concerns operations on bitstrings. In particular, this group contains only the function symbol `eq`, which denotes a function that, given two bitstrings of length i, returns a boolean value indicating whether they are equal.

The second group consists of function symbols used to interact with a ledger, a structure that behaves as a finite table. The ledger is represented as a flat binary string segmented into rows of length $i + 1$. Each row is composed of a bitstring of length i together with an additional bit that indicates whether the row is active (set to 1) or inactive (set to 0). The total size of the ledger is chosen to fit q such rows, where q is a polynomial in $\mathbb{N}_{\geq 1}[i]$. The number of rows must be polynomially bounded otherwise the

resulting string could exceed the size constraints imposed by λ**BLL**.

The function symbol `initQ` constructs an empty ledger by returning a bitstring initialized entirely with zeros, thus representing a table with no active entries. The symbol `contains` checks whether a given input string is already present in the ledger. It scans the ledger to determine whether there exists a row whose first i bits match the input string and whose final bit is set to 1. If such a row is found, the input string is considered to be defined in the ledger and the function returns $\mathbf{t}$. The function symbol `modifyQ` enables the insertion of a new string into the ledger: it searches for the first inactive row, writes the string into the first i positions, and sets the final bit to 1, thereby marking the row as active. Finally, the function symbol `coherent` checks whether every active entry in a ledger of size $q \times (i+1)$ also appears as an active entry in a ledger of size $p \times (2i+1)$. More precisely, for every row in the first ledger whose last bit is set to 1, there exists a corresponding active row in the second ledger such that the first i bits of its $2i$-bit payload coincide with the i-bit string stored in the first ledger ensuring that all bitstrings recorded as active in the first ledger are consistently reflected in the second one.

Equations These equations extend the set introduced in [LGG24]. For completeness, these equations are recalled in Appendix A, Figure 27. They serve as semantic constraints specifying how the function symbols behave under evaluation.

$$\texttt{let}\, x = \texttt{random}\, \texttt{in}\, \texttt{eq}(x, t) \sim_{\Xi \subseteq \Upsilon} \texttt{return}\, \mathbf{f} \tag{randF}$$

$$\begin{pmatrix} \texttt{if}\, b\, \texttt{then}\, \texttt{return}\, v \\ \texttt{else}\, \texttt{return}\, v \end{pmatrix} \sim_{\Xi \subseteq \Upsilon} \texttt{return}\, v \tag{ifConst}$$

Equation randF formalizes a recurring pattern in which a freshly sampled random value is compared with an independently provided value. In

this setting, a fresh random variable x is generated via the primitive `random` and is subsequently compared with some value t using the equality predicate `eq`. Because x is drawn uniformly at random and independently of all values already contained in the program state, the probability that x coincides with the fixed value t is negligible. Consequently, the equality test $\texttt{eq}(x, t)$ evaluates to $\mathbf{f}$ except with negligible probability. This equation plays a central role in the analysis of cryptographic constructions. For example, when an adversary attempts to guess a randomly generated nonce or challenge, the probability that the adversary's guess equals the freshly sampled value is negligible. Equation randF will therefore be a key ingredient in the security proof of existential unforgeability under chosen-message attacks for PRF-induced MACs.

Equation ifConst formalizes a simple simplification pattern for conditional expressions. In this equation, both branches of the conditional expression evaluate to the same value v. Consequently, the overall behavior of the program is invariant with respect to the truth value of the boolean guard b: whether b evaluates to $\mathbf{t}$ or $\mathbf{f}$, the result is always v. Therefore, within any program context, the conditional expression is observationally equivalent to the expression v itself. This equation is particularly useful in the simplification of programs generated during program transformations, where conditional constructs may be introduced and subsequently found to produce identical results in both branches.

5.2 Hoare Logic

We introduce a Hoare logic [Hoa69], which will be fundamental to formalize the proof in Chapter 6 of existential unforgeability under chosen-message attacks for PRF-induced MACs in the λ**BLL** framework.

Following Hoare's original formalism, program behavior is specified by

partial correctness specifications, written as triples of the form

$$\{P\}\, e\, \{Q\}^{1}.$$

Here, e denotes a program, and P and Q are logical propositions representing the precondition and postcondition, respectively. The intended meaning of such a triple is that, if execution of e starts in a program state satisfying P, then every terminating execution of e produces a state satisfying Q.

A Hoare triple of the form $\{P\}\, e\, \{\lambda v.\, Q(v)\}$ is referred to as a partial correctness specification. It is termed partial because, if the execution of e diverges, the triple imposes no additional constraints and does not assert any property about the non-terminating behavior. A stronger form of specification is a total correctness specification, which asserts that whenever e is executed in a state satisfying P, the execution of e is guaranteed to terminate and, upon termination, the postcondition Q holds. Since λ**BLL** is a strongly normalizing calculus, we adopt the standard notation for partial correctness Hoare triples using curly braces.

Since Q is a predicate on values namely, on the value to which e evaluates, we write the Hoare triple $\{P\}\, e\, \{\lambda v.\, Q(v)\}$, where v denotes the result of evaluating e. When a postcondition does not depend on the value returned by the expression, we write $\{P\}\, e\, \{\lambda \_.\, Q\}$ denoting that the result of the computation e is ignored.michi

Propositions used in Hoare triples are generated by the following grammar:

$$\Phi ::= \phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$$

where ϕ ranges over atomic first-order formulas such as $\mathbf{t}$, $\mathbf{f}$, $t_1 = t_2$, $t_1 \in t_2$, or $\texttt{coherent}(t_1, t_2)$. These formulas constitute the assertion language of the logic, while semantic implication $P \models Q$ is defined at the meta-level.

[1] Actually in the original paper the notation is $P\,\{Q\}\,R$ and has to be interpreted as "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion"

Given a typing judgment of the form $\vdash_c^{\Theta} e$, propositions are restricted to ground values that appear in e or in Θ, and possibly to the result of the computation e when it has ground type.

Using the notion of Hoare triples, we introduce a new set of typing rules for λ**BLL**, presented in Figure 18. These rules assign types to terms while respecting linearity and resource constraints, and they equip λ**BLL** with a simple Hoare logic.

To relate the original definition of λ**BLL** with its extension equipped with Hoare logic, we define a translation from standard types to types decorated with propositions. Given a standard type A and an invariant P, the lifting μ produces a decorated type in which P is threaded through every computation. The map μ is defined inductively as follows:

$$\begin{aligned}
&\mu(\mathbb{U}, P) = \mathbb{U} \\
&\mu(\mathbb{B}, P) = \mathbb{B} \\
&\mu(\mathbb{S}[p], P) = \mathbb{S}[p] \\
&\mu(A \otimes B, P) = \mu(A, P) \otimes \mu(B, P) \\
&\mu(!_q A, P) = !_q^{P,P} \mu(A, P) \\
&\mu(A \multimap B, P) = \mu(A, P) \underset{P,\lambda x.P}{\multimap} \mu(B, P)
\end{aligned} \tag{18}$$

We now state a lemma that lifts a typing judgment to a Hoare typing judgment.

Lemma 5.1. *Let P be a proposition over a set of references $\mathcal{R}(P)$. If $\Gamma \vdash^{\Theta} M : A$ then $\mu(\Gamma, P) \vdash^{\Xi} M : \mu(A, P)$ whenever $\Xi \supseteq \Theta$ and $\mathcal{R}(P) \cap \Theta = \emptyset$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash^{\Theta} M : A$. □

In addition to the introduction of Hoare-style reasoning in λ**BLL**, we also present a set of equivalences between terms in the language λ**BLL** extended with Hoare triples. In particular, in Figure 19 we define two equations.

The two equations state that, whenever the precondition entails the guard of a conditional, the conditional term collapses to the corresponding branch,

$$\frac{\Gamma \vdash_v^\Theta V : A}{\Gamma \vdash_c^\Theta \{P[V/v]\}\,\mathtt{return}\,V\,\{\lambda v.P(v)\} : A}\ \text{H-ETA}$$

$$\frac{\Gamma \vdash_c^\Theta \{P\}M\{\lambda v.Q(v)\} : A \quad Q \models P}{p * \Gamma \vdash_v^\Theta\, !M : !_p^{P,Q} A}\ \text{H-BANG}$$

$$\frac{\Gamma \vdash_c^\Theta \{P\}M\{\lambda f.Q(f)\} : A \underset{R,S}{\multimap} B \quad \Delta \vdash_v^\Theta Z : A \quad Q \models R}{\Gamma \boxplus \Delta \vdash_c^\Theta \{P\}\,MZ\,\{\lambda v.S(v)\} : B}\ \text{H-APP}$$

$$\frac{\Gamma, x : A \vdash_c^\Theta \{P(x)\}\,M\,\{\lambda v.Q(x,v)\} : B}{\Gamma \vdash_v^\Theta \lambda x.M : A \underset{\lambda x.P(x),\lambda x.\lambda v.Q(x,v)}{\multimap} B}\ \text{H-LAM}$$

$$\frac{\mathsf{typeof}(f) = G_1 \times \cdots \times G_m \to G \quad (\Gamma_k p \vdash_v^\Theta Z_k : G_k p)_{1\le k\le m} \quad p \in \mathbb{N}_{\ge 1}[i]}{\boxplus_k \Gamma_k \vdash_c^\Theta \{P[f(Z_1,\ldots,Z_m)/v]\}\,f_p(Z_1,\ldots,Z_m)\,\{\lambda v.P(v)\} : Gp}\ \text{H-FUN}$$

$$\frac{r : G \in \Theta}{\Gamma \vdash_c^\Theta \{P(r)\}\,\mathtt{get}\,r\,\{\lambda x.P(x)\} : G}\ \text{H-GET} \qquad \frac{\Gamma \vdash_v^\Theta Z : G \quad r : G \in \Theta}{\Gamma \vdash_c^\Theta \{P[Z/r]\}\,\mathtt{set}\,r\,Z\,\{P\} : \mathbb{U}}\ \text{H-SET}$$

$$\frac{\Gamma \vdash_c^\Theta \{P\}\,M\,\{\lambda v.Q(v)\} : A \quad \Delta, x : A \vdash_c^\Theta \{Q(x)\}\,N\,\{S\} : B}{\Gamma \boxplus \Delta \vdash_c^\Theta \{P\}\,\mathtt{let}\,x = M\,\mathtt{in}\,N\,\{S\} : B}\ \text{H-LET}$$

$$\frac{\Gamma \vdash_v^\Theta Z : \mathbb{B} \quad \Delta \vdash_c^\Theta \{P \wedge (Z = \mathbf{t})\}\,M\,\{R\} : A \quad \Delta \vdash_c^\Theta \{P \wedge (Z = \mathbf{f})\}\,N\,\{R\} : A}{\Gamma \boxplus \Delta \vdash_c^\Theta \{P\}\,\mathtt{if}\,Z\,\mathtt{then}\,M\,\mathtt{else}\,N\,\{R\} : A}\ \text{H-CASE}$$

$$\frac{P' \models P \quad Q \models Q' \quad \Gamma \vdash_c^\Theta \{P\}\,M\,\{\lambda v.Q(v)\} : A}{\Gamma \vdash_c^\Theta \{P'\}\,M\,\{\lambda v.Q'(v)\} : A}\ \text{H-CONS}$$

Figure 18: Typing rules of λ**BLL** extended with Hoare Logic.

$$\{P\}\,\texttt{if}\,b\,\texttt{then}\,M\,\texttt{else}\,N\,\{Q\} \sim \{P\}\,M\,\{Q\} \qquad \text{(ifTrue)}$$
$$\text{if } P \models b$$

$$\{P\}\,\texttt{if}\,b\,\texttt{then}\,M\,\texttt{else}\,N\,\{Q\} \sim \{P\}\,N\,\{Q\} \qquad \text{(ifFalse)}$$
$$\text{if } P \models \neg b$$

Figure 19: Equations in λ**BLL** with Hoare extension.

since the alternative branch is unreachable under the given precondition. Under the precondition P such that $P \models b$, the behaviour of the conditional term $\texttt{if}\,b\,\texttt{then}\,M\,\texttt{else}\,N$ is equivalent to that of M, since the guard is always true and the branch N is never executed. Symmetrically, if $P \models \neg b$, the effect of the conditional is equivalent to that of N, as the guard is always false and the branch M becomes unreachable.

Chapter 6

Proving PRF-Induced MAC Security Equationally

This chapter demonstrates that existential unforgeability under chosen message attacks for pseudorandom function based MACs can be proved within the λ**BLL** framework. We formalize the standard reduction from MAC forgery to pseudorandom function indistinguishability by encoding the `MacForge` experiment, the adversary, and the MAC oracle as λ**BLL** terms. By reconstructing the proof of Theorem 2.4.1 using the equational reasoning principles of the calculus, and by exploiting its extension with Hoare logic and assertion-typed terms, we demonstrate that λ**BLL** is sufficiently expressive to internalize classical security proofs while simultaneously enforcing strict polynomial-time constraints via its graded type system.

We begin by recalling the MAC forgery experiment, `MacForge`, as defined in Figure 20. To bridge the gap between pseudocode and formalization, we specify the types for each component as they appear in λ**BLL**. The concrete implementation of the terms modeling the key generation algorithm Gen and the MAC oracle $Mac_k(\cdot)$ depends on the specific MAC scheme under consideration. For this reason, these terms are left abstract and will be instantiated only after fixing a scheme Π. In the following paragraphs, we make this instantiation explicit for the schemes Π^F and $\widetilde{\Pi}$ which are used in

the proof of Theorem 2.17.

In λ**BLL** typing discipline, the key generation algorithm Gen is a probabilistic computation that yields a secret key of polynomial length. The adversary, Adv, is a higher-order computation having access to an oracle. The oracle itself is controlled by the graded modality $!_q$, which enforces a polynomial bound on the number of queries.

$$
\begin{aligned}
&\texttt{MacForge}_{A,\Pi}(n) : \\
&\quad k \leftarrow Gen(1^n) \\
&\quad (m, t) \leftarrow A^{Mac_k(\cdot)}(1^n) \\
&\quad \mathbb{Q} \leftarrow \{m \mid A \text{ queries } Mac_k(m)\} \\
&\quad \texttt{return } (m \notin \mathbb{Q} \wedge Vrfy_k(m, t) = 1)
\end{aligned}
$$

Figure 20: Pseudocode for `MacForge` experiment.

We now describe how the components of the experiment are typed in λ**BLL**. The key generation algorithm Gen is modeled as a probabilistic computation that, given the security parameter, produces a secret key whose length is polynomial in i. The adversary is modeled as a higher-order computation that receives oracle access to the MAC function and outputs a candidate forgery. The MAC oracle itself is represented as a term whose use is controlled by a graded modality, enforcing a polynomial bound on the number of queries.

$$
\begin{aligned}
&\vdash_v^{\Theta} Gen : \mathbb{U} \multimap \mathbb{S}[i] \\
&\vdash_v^{\Theta} Mac : (\mathbb{S}[i] \otimes \mathbb{S}[i]) \multimap \mathbb{S}[i] \\
&\vdash_c^{\Theta} Oracle : \mathbb{S}[i] \multimap !_q\big(\mathbb{S}[i] \multimap \mathbb{S}[i]\big) \\
&\vdash_c^{\varnothing} Adv : !_q\big(\mathbb{S}[i] \multimap \mathbb{S}[i]\big) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i])
\end{aligned}
$$

The typing judgments for the main components of the experiment are

given above. Here, i is a polynomial variable and q denote a polynomial in the security parameter. The oracle type $\mathbb{S}[i] \multimap !_q(\mathbb{S}[i] \multimap \mathbb{S}[i])$ models a MAC oracle which, once instantiated with a key, provides a function mapping messages of length i to tags of the same length. The graded modality $!_q$ enforces a polynomial upper bound on the number of oracle queries that the adversary may perform. In addition to answering MAC queries, the oracle maintains a record of all messages queried by the adversary. This information encodes the set

$$\mathbb{Q} \leftarrow \{\, m \mid A \text{ queries } Mac_k(m) \,\}$$

as presented in Figure 20. To model this behavior in λ**BLL**, we rely on the notion of a *ledger*, which allows mutable storage with a statically bounded size. The reference Q has type $Q : \mathbb{S}[q \times (i+1)]$ and can be viewed as a table with q rows, where each row consists of a message of length i together with a final bit indicating whether the row is active. The size of the ledger reflects the fact that the oracle can be queried at most q times. At each oracle invocation on a message m, the oracle updates the ledger stored in Q as follows: it scans the ledger for an active row containing m; if none exists, it writes m into the first inactive row and marks that row as active. In this way, the reference Q always encodes exactly the set $\mathbb{Q}$ of messages previously queried by the adversary. To inspect the adversary's query history, we introduced in Section 5.1 some useful functions symbols. In particular in Table 5.1 we defined the function `contains`, whose purpose is to determine whether a given message occurs in the encoding of the set $\mathbb{Q}$ stored in the reference context. More precisely, `contains` checks whether a bitstring of type $\mathbb{S}[i]$ appears in a *query ledger* of type $\mathbb{S}[q \times (i+1)]$. It takes as input the stored query history together with a message and returns a boolean value.

The adversary *Adv* takes as input the closure obtained by currying and encapsulating the key within the *Oracle* term. Based on its interaction with the oracle, it produces a candidate forgery (m, t) of type $(\mathbb{S}[i] \otimes \mathbb{S}[i])$.

The verification function *Vrfy* is not introduced since the validity of a proposed tag can be checked just by recomputing the MAC on the message

and comparing it with the supplied tag.

It is worth noting that the type derivations of the terms introduced so far are parameterized by a reference context Θ, which varies depending on the cryptographic experiment considered. In the model for the scheme Π_F, the context Θ contains the reference Q used to record oracle queries so the types of the terms in Figure 22 must be adapted by defining a reference context $\Xi = \{\, Q : \mathbb{S}[(q \times (i+1)] \,\}$ and setting Θ accordingly. In contrast, in the model for the scheme $\widetilde{\Pi}$, the context Θ is instantiated with a reference context containing also the ledger used to model the random function f, namely $\Upsilon = \{\, rand : \mathbb{S}[(q+1) \times (2i+1)], Q : \mathbb{S}[(q \times (i+1)] \,\}$.

We define the λ**BLL** term corresponding to the forgery experiment, shown in Figure 21. The experiment begins by generating a fresh key k and constructing the corresponding oracle o. The adversary Adv is then given access to o and returns an attempted forgery (m, t). The experiment first checks whether the message m appears in the query set $\mathbb{Q}$ by reading the reference Q containing the query ledger and applying the predicate `contains`. If verification succeeds the experiment immediately returns `false`, otherwise if it succeeds, the experiment tests if the tag provided by the adversary is equal to the recomputed tag on the same message obtained through the oracle. The experiment returns **t** if m was not previously queried, and **f** otherwise. If verification fails, the experiment immediately returns **f**.

In order to underline the similarity with the pseudocode given in Figure 20, we apply the syntactic sugar introduced in Equation 15 to the first line of the term shown in Figure 21, observing that Gen is a value.

We have to define a λ**BLL** term for the experiment using the encryption scheme Π_F which in Proof 2.4.1 corresponds to the experiment $\texttt{MacForge}_{A,\Pi_F}$ and one for the experiment using the encryption scheme $\widetilde{\Pi}$ which corresponds to $\texttt{MacForge}_{A,\widetilde{\Pi}}$. In both the cryptographic experiment and the distinguisher, we need to construct two instances of the λ**BLL** term *MacForge* (Figure 21) that models these components, namely one that interacts with the pseudorandom function F and one that interacts with a truly random function f.

$$
\begin{aligned}
&MacForge \triangleq \\
&\quad \texttt{let}\, k = Gen \star\, \texttt{in} \\
&\quad \texttt{let}\, o = Oracle\, k\, \texttt{in} \\
&\quad \texttt{let}\, \langle m, tag \rangle = Adv\, o\, \texttt{in} \\
&\quad \texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
&\quad \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
&\quad \texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
&\quad \texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m\, \texttt{in}\, \texttt{eq}(x, tag)
\end{aligned}
$$

Figure 21: λ**BLL** term for the `MacForge` experiment.

The model for the experiment $\texttt{MacForge}_{A,\Pi_F}$, which we denote by $MacForge^F$, is obtained by instantiating the generic term *MacForge* with the components Gen^F, Mac^F, and $Oracle^F$ defined in Figure 22. Unlike the pseudocode presented in Figure 20.

The key generation algorithm of Π_F is modeled by the term Gen^F, which samples a fresh key using the function symbol `random`, whose type is $\mathbb{S}[i]$. The term Gen^F is additionally responsible for the initialization of query ledger which is instantiated by setting the value of the reference Q using the function symbol `initQ` which generates a string of length $q \times (i + 1)$ filled entirely with zeros.

The pseudorandom function used by the scheme is abstracted as a generic value F, typed as $\vdash_v^{\Theta} F : (\mathbb{S}[i] \otimes \mathbb{S}[i]) \multimap S[i]$ reflecting the fact that the construction is parameterized by an arbitrary pseudorandom function which can be instantiated in many ways.

The tag generation algorithm of Π_F is modeled by the term Mac^F. Given a key k and a message m it computes the tag t by evaluating the pseudorandom function F on the pair (k, m), finally returning a tag.

The oracle $Oracle^F$ encapsulates the secret key and provides access to

$Gen^F \triangleq$

$\lambda u.$

$\texttt{let}\, q = \texttt{initQ}\, \texttt{in}$

$\texttt{set}\, Q\, q;$

$\texttt{let}\, k = \texttt{random}\, \texttt{in}$

$\texttt{return}\, k$

$Mac^F \triangleq$

$\lambda x.$

$\texttt{let}\, \langle k, m\rangle = x\, \texttt{in}$

$\texttt{return}\, F\langle k, m\rangle$

$Oracle^F \triangleq$

$\texttt{return}\, (\lambda k.\texttt{return}\, !(\texttt{return}\, \lambda m.$

$\texttt{let}\, q = \texttt{get}\, Q\, \texttt{in}$

$\texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in}$

$\texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m)$

$\texttt{else}\, \texttt{skip};$

$Mac^F\langle k, m\rangle)))$

Figure 22: Models for subterms of $MacForge^F$.

the MAC function through the graded modality !. In addition to answering MAC queries, the oracle maintains a record of all messages queried by the adversary. This record is stored in the reference Q thus introducing a non-empty reference context. At each oracle invocation on a message m, the current contents of Q are retrieved, updated to include m, and written back to the reference before computing and returning the corresponding MAC value. In order to retrieve and update the query ledger, we rely on several function symbols such as `initQ`, `modifyQ` and `contains`, which are reported in Table 5.1. To support this behavior, the typing of the terms in Figure 22 is carried out in a reference context $\Xi = \{\, Q : \mathbb{S}[q \times (i+1)] \,\}$ which represents the ledger used to store the adversary's oracle queries. The context Θ in the

corresponding type derivations is instantiated with Ξ. In the term $Oracle^F$ in Figure 22 we use the syntactic sugar of Equation 15 since Mac^F is a value.

The model for $\texttt{MacForge}_{A,\widetilde{\Pi}}$, denoted by $\widetilde{MacForge}$, is obtained from the term *MacForge* from Figure 21 instantiating the terms Mac, Gen and $Oracle$ according to $\widetilde{\Pi}$, and is represented in Figure 23. The model for $\widetilde{MacForge}$ relies on a truly random function, which is characterized by the property that each distinct input is deterministically associated with a uniformly random output upon its first evaluation, and that this output remains fixed for all subsequent evaluations of the random function on the same input. To model this behavior within λ**BLL**, we follow the same ledger based approach introduced in [LGG24], which is conceptually analogous to the mechanism used to record oracle queries. We introduce an additional ledger that can be manipulated through a collection of function symbols, detailed in Appendix A. As shown in Figure 23, this ledger is stored in the reference *rand*, with type $\mathbb{S}[(q+1)\times(2i+1)]$. The ledger can be viewed as a table with $q+1$ rows, where each row contains an input-output pair of the random function together with a final bit indicating whether the row is active. The size of the ledger reflects the fact that the random function f can be queried at most $q+1$ times: once by the experiment itself and up to q times by the adversary.

The key generation algorithm of $\widetilde{\Pi}$ is modeled by the term $\widetilde{Gen}$. Its role is to initialize the ledger representing the random function by setting the reference *rand* to the output of the function symbol `initTable`, which generates a string of length $(q+1)\times(2i+1)$ filled with zeros, corresponding to an empty table. It also initializes the ledger representing the query history by setting the value of the reference Q similarly to Gen^F. The term then returns a dummy value generated using the function symbol `zero`, as the key is not used in the experiment $\widetilde{MacForge}$.

The term $\widetilde{Mac}$ models the MAC algorithm of $\widetilde{\Pi}$ built from the random function f. Given an input message m, it samples a fresh random value r and inspects the ledger stored in *rand* to determine whether r has already been used as an input to the random function. This check is performed

$\widetilde{Gen} \triangleq$

$\lambda u.$
$\texttt{let}\, q = \texttt{initQ}\, \texttt{in}$
$\texttt{set}\, Q\, q;$
$\texttt{let}\, tab = \texttt{initTable}\, \texttt{in}$
$\texttt{set}\, rand\, tab;$
$\texttt{let}\, z = \texttt{zero}\, \texttt{in}$
$\texttt{return}\, z$

$\widetilde{Mac} \triangleq$

$\lambda e.$
$\texttt{let}\, \langle k, m\rangle = e\, \texttt{in}$
$\texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in}$
$\texttt{let}\, b = \texttt{isdefined}(tab, m)\, \texttt{in}$
$\texttt{if}\, b\, \texttt{then}$
  $\texttt{let}\, x = \texttt{getValue}(tab, m)\, \texttt{in}$
  $\texttt{return}\, x$
$\texttt{else}$
  $\texttt{let}\, x = \texttt{random}\, \texttt{in}$
  $\texttt{let}\, newtab = \texttt{modify}(tab, m, x)\, \texttt{in}$
  $\texttt{set}\, rand\, newtab;$
  $\texttt{return}\, x$

$\widetilde{Oracle} \triangleq$

$\texttt{return}\, (\lambda k.\texttt{return}\, !(\texttt{return}\, \lambda m.$
  $\texttt{let}\, q = \texttt{get}\, Q\, \texttt{in}$
  $\texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in}$
  $\texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, m)$
  $\texttt{else skip};$
  $\widetilde{Mac}\langle k, m\rangle))$

Figure 23: Models for subterms of $\widetilde{MacForge}$.

using the predicate `isdefined` which is reported in Table A.1. If r is already present in the ledger, the corresponding output value x is retrieved using `getValue`, and the resulting tag is computed deterministically. Otherwise, if r is a fresh input, the term samples a new uniformly random value x, records the association (r, x) in the ledger using `modify`, updates the reference *rand*, and then returns the corresponding tag. In this way, the ledger ensures that repeated evaluations of f on the same input yield consistent outputs.

As in the case of the query ledger, the algorithms of the scheme $\widetilde{\Pi}$ operate directly on the ledger stored in the reference *rand*. Accordingly, the typing of the terms in Figure 23 is carried out in a reference context $\Upsilon = \{\, rand : \mathbb{S}[(q+1) \times (2i+1)], Q : \mathbb{S}[q \times (i+1)] \,\}$ and the context Θ appearing in the corresponding typing judgments is instantiated with Υ. It is worth observing that the underlying function f can be queried at most $q+1$ times: up to q times by the adversary through its oracle access, and once by the experiment itself during verification via an oracle evaluation on the candidate message generated by the adversary. The size of the ledger stored in *rand* reflects this bound, as it contains exactly $q+1$ rows, corresponding to the maximum number of distinct messages on which f may be evaluated throughout the execution of the experiment.

In the term $\widetilde{Oracle}$ in Figure 23 we use the syntactic sugar of Equation 15 since $\widetilde{Mac}$ is a value.

6.1 Modeling The Distinguishers

The distinguisher D_A constructed from any adversary that succeeds in the `MacForge` experiment, and used in the first step of the proof of Theorem 2.17, emulates the experiment `MacForge`. The model for the distinguisher is very similar to that of the experiment, however in this case the oracle o is taken as an explicit input. This allows the same term to be used to model both $D_A^{F_k(\cdot)}$ and $D_A^{f(\cdot)}$, namely the distinguisher that receives oracle access to a pseudorandom function F_k and the one that receives oracle access to a truly

random function f. The λ**BLL** term defining the distinguisher is shown in Figure 24.

$$
\begin{aligned}
&D \triangleq \\
&\quad \texttt{return}\,(\lambda o. \\
&\quad \texttt{let}\,\langle m, tag\rangle = \mathit{Adv}\,o\,\texttt{in} \\
&\quad \texttt{let}\,\mathit{queries} = \texttt{get}\,Q\,\texttt{in} \\
&\quad \texttt{let}\,c = \texttt{contains}(\mathit{queries}, m)\,\texttt{in} \\
&\quad \texttt{if}\,c\,\texttt{then return}\,\mathbf{f} \\
&\quad \texttt{else let}\,x = \texttt{der}(o)\,m\,\texttt{in eq}(x, tag))
\end{aligned}
$$

Figure 24: λ**BLL** term for the `D` distinguisher.

The distinguisher invokes the adversary with oracle access to o, obtains a candidate forgery (m, t), and checks its validity using a verification procedure identical to the one used in the `MacForge` in Figure 21. As in the `MacForge` experiment, the reference Q is shared between the oracle and the distinguisher and records all oracle queries performed by the adversary. The distinguisher returns $\mathbf{t}$ if (m, t) is a valid fresh forgery, and $\mathbf{f}$ otherwise.

We briefly justify that the distinguisher D is well typed in λ**BLL**. The term D is a value returning a function that takes an oracle o as input which has type $!_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i])$, matching the oracle type expected by the adversary Adv. Since Adv is typed as $!_q(\mathbb{S}[i] \multimap \mathbb{S}[i]) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i])$ for a polynomial q the application $\mathit{Adv}\,o$ is well typed and yields a candidate forgery (m, t) with type $(\mathbb{S}[i] \otimes \mathbb{S}[i])$. The distinguisher then reads the reference Q using $\texttt{get}\,Q$. As in the `MacForge` experiment, the oracle o and the distinguisher D operate within the same reference context Θ, which contains the reference $Q : \mathbb{S}[q \times (i+1)]$. This ensures that the query history recorded by the oracle during the execution of Adv is available to the distinguisher when performing the check for the freshness of the message. It is important to

note that modifying the reference state would not correspond to a meaningful strategy for distinguishing a pseudorandom function from a truly random one. The verification step is carried out by checking if the message generated by the adversary belongs to the set of registered queries, if it does not then D evaluates the oracle on the message checking if the tag generated by the adversary and the tag generated by the oracle are equals. The type for the distinguisher follows the type derivation $\vdash_c^{\Theta} !_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i]) \multimap \mathbb{B}$

We can now model the distinguishers $D^{F_k(\cdot)}$ and $D^{f(\cdot)}$ as follows:

$$D^F \triangleq \mathtt{let}\, k = Gen^F \star \mathtt{in}\, \mathtt{let}\, o = M^F k\, \mathtt{in}\, \mathtt{let}\, e = MacOracle\, o\, \mathtt{in}\, D\, e \quad (19)$$

$$\widetilde{D} \triangleq \mathtt{let}\, k = \widetilde{Gen} \star \mathtt{in}\, \mathtt{let}\, o = \widetilde{M} k\, \mathtt{in}\, \mathtt{let}\, e = MacOracle\, o\, \mathtt{in}\, D\, e \quad (20)$$

where the subterms M^F and $\widetilde{M}$ defined in Figure 25 model the pseudorandom function F and the random function f, respectively.

The terms M^F and $\widetilde{M}$ are typed under the following type derivations:

$$\vdash_c^{\Xi} M^F : \mathbb{S}[i] \multimap !_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i])$$
$$\vdash_c^{\Upsilon} \widetilde{M} : \mathbb{S}[i] \multimap !_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i])$$

These judgments are parameterized by two distinct reference contexts, denoted by Ξ and Υ, which correspond to the two different oracles that may be provided to the distinguisher D. When the distinguisher is given access to the oracle M^F, modeling a pseudorandom function, the reference context Θ is instantiated as $\Xi = \{\, Q : \mathbb{S}[q \times (i+1)] \,\}$ where the reference Q is used to record the messages queried to the oracle. In contrast, when the distinguisher is given access to the term $\widetilde{M}$, modeling a truly random function f, the reference context Θ is instantiated as $\Upsilon = \{\, rand : \mathbb{S}[(q+1)\times(2i+1)],\, Q : \mathbb{S}[q\times(i+1)] \,\}$. In this case, the additional reference *rand* is required to store previously generated random outputs which models the truly random function.

The oracle *MacOracle*, defined in Equation 21, returns a function that is parameterized by an underlying function oracle o. In the present setting, o abstracts either a pseudorandom function or a truly random function through terms in Figure 25 and is used to model the MAC computation. Upon

$M^F \triangleq$

```
return (λk.return (!(return (λm.
  let q = get Q in
  let t = contains(q, m) in
  if not(t) then
    set Q modifyQ(q, m)
  else skip;
  return F⟨k, m⟩
```

(a) Subterm with pseudorandom function.

$\widetilde{M} \triangleq$

```
return (λk.return (!(return (λr.
  let q = get Q in
  let t = contains(q, r) in
  if not(t) then set Q modifyQ(q, r)
  else skip;
  let tab = get rand in
  let b = isdefined(tab, r) in
  if b then
    let x = getValue(tab, r) in
    return x
  else
    let x = random in
    let newtab = modify(tab, r, x) in
    set rand newtab;
    return x))))
```

(b) Subterm with truly random function.

Figure 25: Models for function oracles used by the distinguisher.

receiving a message m, the returned function simply invokes the oracle o on m and returns the resulting value, which is interpreted as the corresponding tag. The term is well typed under the judgment $\vdash_c^{\varnothing} \mathit{MacOracle} : !_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i]) \multimap !_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i])$ where the graded modality accounts for the single use of the underlying oracle within the returned function.

$$\begin{aligned} \mathit{MacOracle} \triangleq\ & \mathtt{return}\,(\lambda o.\mathtt{return}\,(!(\mathtt{return}\,(\lambda m. \\ & \quad \mathtt{return}\,(\mathtt{der}(\mathtt{o})\,m) \\ &))))\end{aligned} \tag{21}$$

6.2 Formalization of the Proof

The security of Π^F is established via a proof by reduction (Theorem 2.17), proceeding contrapositively: from any hypothetical adversary Adv against Π^F we obtain a distinguisher D for F which is designed to distinguish the pseudorandom function F from a truly random function f. If D is shown to be successful whenever Adv succeeds, we conclude that Π^F is secure provided F is a pseudorandom function where security is characterized by the non existence of efficient adversaries of the appropriate kind.

In the context of λ**BLL**, the distinguisher D is represented as the term defined in Figure 24, having type $!_{q+1}(\mathbb{S}[i] \multimap \mathbb{S}[i]) \multimap \mathbb{B}$. We consider two specific instances of this distinguisher: D^F, which interacts with the pseudorandom function F, and $\widetilde{D}$, which interacts with a truly random function f. Both F and f are modeled using stateful computations. Specifically, a ledger Q records the query history, and in the case of the truly random function, an additional ledger rand is used to provide consistent random outputs. To complete the formalization, we defined an ideal MAC scheme $\widetilde{\Pi}$ that is structurally identical to Π^F but uses a truly random function. The components of Π^F are specified in Figure 22, while those of $\widetilde{\Pi}$ are detailed in Figure 23.

By defining the reference contexts $\Xi = \{Q : \mathbb{S}[q \times (i+1)]\}$ and $\Upsilon = \{\mathit{rand} : \mathbb{S}[(q+1) \times (2i+1)], Q : \mathbb{S}[q \times (i+1)]\}$, the security of $\mathit{MacForge}^F$ is

expressed by the equation:

$$MacForge^F \sim_{\Xi \subseteq \Upsilon} \mathbf{f} \tag{22}$$

where $\sim_{\Xi \subseteq \Upsilon}$ denotes the Θ-contextual indistinguishability relation as defined in [LGG24]. This equation asserts that the experiment $MacForge^F$ reveals nothing to the adversary, even when accounting for the stateful references stored in the context Υ. While this seems a strong requirement, and indeed it is, we show that it is possible to structure a reduction proof by combining equational reasoning with reasoning via Hoare logic and assertion-typed terms. This allows us to derive a proof of Equation 22 hence obtaining a formal version of the proof of Theorem 2.17. The diagram shows that the left component ($MacForge^F$) can be proved insecure only if the lower relation in the diagram is also violated. Since we assume that the pseudorandom function hypothesis holds (see Definition 2.14), it follows that the corresponding MAC is secure.

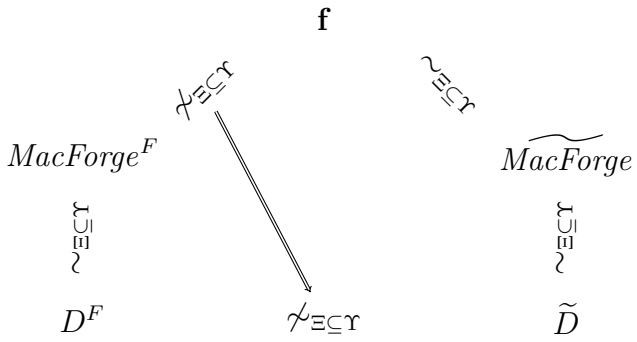

Figure 26: Outline Proof of Security for `MacForge`.

Consider the diagram in Figure 26. Our objective is to demonstrate that, on both the left-hand and right-hand sides of the diagram, the relation $\sim_{\Xi \subseteq \Upsilon}$ can be used to relate the corresponding terms. On the left-hand side, we must establish that the contextual equivalence $D^F \sim_{\Xi \subseteq \Upsilon} MacForge^F$ holds. Similarly, on the right-hand side, we must show that $\widetilde{D} \sim_{\Xi \subseteq \Upsilon} \widetilde{MacForge}$ holds. These two equivalences correspond respectively to Equation 10 and Equation 11 in Proof 2.4.1.

6.2.1 From $\widetilde{D}$ to $\widetilde{MacForge}$

We begin by focusing on the bottom-right side of the diagram. The first step is to establish the relation $\texttt{let}\, o = \widetilde{M}\, k\, \texttt{in}\, MacOracle\, o \sim_{\Xi \subseteq \Upsilon} \widetilde{Oracle}\, k$, which is stated as Lemma 6.1 and proved immediately after. Once this equivalence is established, the relation $\widetilde{D} \sim_{\Xi \subseteq \Upsilon} \widetilde{MacForge}$ follows by an application of Equation `letAss` and Lemma 6.1, and by concluding with a reduction using Equation `reduc2`.

Lemma 6.1.

$$\texttt{let}\, o = \widetilde{M}\, k\, \texttt{in}\, MacOracle\, o \sim_{\Xi \subseteq \Upsilon} \widetilde{Oracle}\, k$$

Proof. We begin by unfolding the definition of $\widetilde{Oracle}$ and applying Equation `reduc2`. We then unfold the definition of $\widetilde{Mac}$, after which the proof is completed by straightforward applications of Equations `reduc4`, `reduc3`, and finally `reduc2`. Observe that the equations used, which are reported in Figure 27, are parameterized by two contexts, Θ and Ξ. In this case, $\Theta = \Xi = \{\, Q : \mathbb{S}[q \times (i+1)] \,\}$ and $\Xi = \Upsilon = \{\, rand : \mathbb{S}[(q+1) \times (2i+1)], Q : \mathbb{S}[q \times (i+1)] \,\}$, since both the references Q and $rand$ are accessed during the reduction.

$$
\begin{aligned}
\widetilde{Oracle}\, k \quad \triangleq \quad & \texttt{return}\, (\lambda k.\texttt{return}\,!(\texttt{return}\, \lambda m. \\
& \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
& \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\
& \quad \texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, m) \\
& \quad \texttt{else skip}; \\
& \quad \widetilde{Mac}\langle k, m \rangle \\
&))\, k
\end{aligned}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if}\,\texttt{not}(t)\,\texttt{then}\\
\quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else}\,\texttt{skip};\\
\quad \widetilde{Mac}\langle k,m\rangle\\
)))
\end{array}
\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}}
\triangleq
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if}\,\texttt{not}(t)\,\texttt{then}\\
\quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else}\,\texttt{skip};\\
\quad (\lambda e.\\
\quad\quad \texttt{let}\,\langle k,m\rangle = e\,\texttt{in}\\
\quad\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad\quad \texttt{let}\,b = \texttt{isdefined}(tab,m)\,\texttt{in}\\
\quad\quad \texttt{if}\,b\,\texttt{then}\\
\quad\quad\quad \texttt{let}\,x = \texttt{getValue}(tab,m)\,\texttt{in}\\
\quad\quad\quad \texttt{return}\,x\\
\quad\quad \texttt{else}\\
\quad\quad\quad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
\quad\quad\quad \texttt{let}\,newtab = \texttt{modify}(tab,m,x)\,\texttt{in}\\
\quad\quad\quad \texttt{set}\,rand\,newtab;\\
\quad\quad\quad \texttt{return}\,x\\
\quad)\langle k,m\rangle)))
\end{array}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,\langle k,m\rangle = \langle k,m\rangle\,\texttt{in}\\
\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad \texttt{let}\,b = \texttt{isdefined}(tab,m)\,\texttt{in}\\
\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}} \quad \texttt{if}\,b\,\texttt{then}\\
\quad\quad \texttt{let}\,x = \texttt{getValue}(tab,m)\,\texttt{in}\\
\quad\quad \texttt{return}\,x\\
\quad \texttt{else}\\
\quad\quad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
\quad\quad \texttt{let}\,newtab = \texttt{modify}(tab,m,x)\,\texttt{in}\\
\quad\quad \texttt{set}\,rand\,newtab;\\
\quad\quad \texttt{return}\,x\\
)))
\end{array}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q, m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q, m)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad \texttt{let}\,b = \texttt{isdefined}(tab, m)\,\texttt{in}\\
\overset{\texttt{reduc5}}{\sim_{\Xi \subseteq \Upsilon}} \quad \texttt{if}\,b\,\texttt{then}\\
\quad\quad \texttt{let}\,x = \texttt{getValue}(tab, m)\,\texttt{in}\\
\quad\quad \texttt{return}\,x\\
\quad \texttt{else}\\
\quad\quad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
\quad\quad \texttt{let}\,newtab = \texttt{modify}(tab, m, x)\,\texttt{in}\\
\quad\quad \texttt{set}\,rand\,newtab;\\
\quad\quad \texttt{return}\,x\\
)))
\end{array}
\tag{23}
$$

The remaining goal is to show that the term $\texttt{let}\,o = \widetilde{M}\,k\,\texttt{in}\,MacOracle\,o$ is Θ-contextually indistinguishable from the term shown in Equation 23. We begin by unfolding the definition of *MacOracle*, as given in Equation 21. This term is a curried function that, when applied to an oracle o, returns the corresponding MAC computation. We then apply Equation `reduc2` to eliminate the lambda abstraction and simplify the application of *MacOracle* to o. Next, we unfold the definition of $\widetilde{M}$ from Figure 25b. Applying $\widetilde{M}$ to a key k yields a function that, on input m, inspects the ledger to determine whether a value has already been associated with m. If so, the stored value is returned; otherwise, fresh randomness is sampled, the reference storing the ledger is updated, and the newly sampled value is returned. We then apply Equation `reduc2` to evaluate this application. Finally, the term is simplified by successive applications of Equations `reduc4`, `reduc3`, and `reduc2`.

$$\texttt{let}\, o = \widetilde{M}\, k\, \texttt{in}\, MacOracle\, o \quad \triangleq \quad \begin{array}{l} \texttt{let}\, o = \widetilde{M}\, k\, \texttt{in} \\ \texttt{return}\,(\lambda o.\texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der(o)}\, m) \\ ))))\, o \end{array}$$

$$\overset{\texttt{reduc2}}{\sim_{\Xi \subseteq \Upsilon}} \quad \begin{array}{l} \texttt{let}\, o = \widetilde{M}\, k\, \texttt{in} \\ \texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der(o)}\, m) \\ ))) \end{array} \quad \triangleq \quad \texttt{let}\, o = \left(\begin{array}{l} \texttt{return}\,(\lambda k.\texttt{return}\,(!(\texttt{return}\,(\lambda r. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, r)\, \texttt{in} \\ \quad \texttt{if not}(t)\, \texttt{then} \\ \quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, r) \\ \quad \texttt{else skip}; \\ \quad \texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in} \\ \quad \texttt{let}\, b = \texttt{isdefined}(tab, r)\, \texttt{in} \\ \quad \texttt{if}\, b\, \texttt{then} \\ \quad\quad \texttt{let}\, x = \texttt{getValue}(tab, r)\, \texttt{in} \\ \quad\quad \texttt{return}\, x \\ \quad \texttt{else} \\ \quad\quad \texttt{let}\, x = \texttt{random}\, \texttt{in} \\ \quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, r, x)\, \texttt{in} \\ \quad\quad \texttt{set}\, rand\, newtab; \\ \quad\quad \texttt{return}\, x \\ )))) \end{array} \right) k\, \texttt{in}$$

$$\texttt{return}\,(!(\texttt{return}\,(\lambda m.\texttt{return}\,(\texttt{der(o)}\, m)))$$

$$\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}} \quad \texttt{let}\, o = \left(\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda r.\\
\quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in}\\
\quad \texttt{let}\, t = \texttt{contains}(q, r)\, \texttt{in}\\
\quad \texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, r)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in}\\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, r)\, \texttt{in}\\
\quad \texttt{if}\, b\, \texttt{then}\\
\qquad \texttt{let}\, x = \texttt{getValue}(tab, r)\, \texttt{in}\\
\qquad \texttt{return}\, x\\
\quad \texttt{else}\\
\qquad \texttt{let}\, x = \texttt{random}\, \texttt{in}\\
\qquad \texttt{let}\, newtab = \texttt{modify}(tab, r, x)\, \texttt{in}\\
\qquad \texttt{set}\, rand\, newtab;\\
\qquad \texttt{return}\, x\\
)))
\end{array}\right) \texttt{in}$$

$$\texttt{return}\,(!(\texttt{return}\,(\lambda m.\texttt{return}\,(\texttt{der(o)}\, m)))$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \overset{\text{reduc4}}{\sim_{\Xi\subseteq\Upsilon}}\ \texttt{return}\ \left(\begin{array}{l}
\texttt{der}\ \left(!\ \left(\begin{array}{l}
\texttt{return}\,(\lambda r.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,r)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q,r)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad \texttt{let}\,b = \texttt{isdefined}(tab,r)\,\texttt{in}\\
\quad \texttt{if}\,b\,\texttt{then}\\
\qquad \texttt{let}\,x = \texttt{getValue}(tab,r)\,\texttt{in}\\
\qquad \texttt{return}\,x\\
\quad \texttt{else}\\
\qquad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
\qquad \texttt{let}\,newtab = \texttt{modify}(tab,r,x)\,\texttt{in}\\
\qquad \texttt{set}\,rand\,newtab;\\
\qquad \texttt{return}\,x)
\end{array}\right)\right)\\
m
\end{array}\right)\\
)))
\end{array}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m. \\
\quad \overset{\text{reduc3}}{\sim_{\Xi \subseteq \Upsilon}} \quad \texttt{return} \left(\begin{array}{l}
\texttt{return}\,(\lambda r. \\
\quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, r)\, \texttt{in} \\
\quad \texttt{if}\, \texttt{not}(t)\, \texttt{then} \\
\quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, r) \\
\quad \texttt{else}\, \texttt{skip}; \\
\quad \texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, r)\, \texttt{in} \\
\quad \texttt{if}\, b\, \texttt{then} \\
\quad\quad \texttt{let}\, x = \texttt{getValue}(tab, r)\, \texttt{in} \\
\quad\quad \texttt{return}\, x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\, x = \texttt{random}\, \texttt{in} \\
\quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, r, x)\, \texttt{in} \\
\quad\quad \texttt{set}\, rand\, newtab; \\
\quad\quad \texttt{return}\, x)
\end{array} \right) m \\
)))
\end{array}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q, m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q, m)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad \texttt{let}\,b = \texttt{isdefined}(tab, m)\,\texttt{in}\\
\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}} \quad \texttt{if}\,b\,\texttt{then} \qquad\qquad = \quad \text{Term in Equation 23}\\
\quad\quad \texttt{let}\,x = \texttt{getValue}(tab, m)\,\texttt{in}\\
\quad\quad \texttt{return}\,x\\
\quad \texttt{else}\\
\quad\quad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
\quad\quad \texttt{let}\,newtab = \texttt{modify}(tab, m, x)\,\texttt{in}\\
\quad\quad \texttt{set}\,rand\,newtab;\\
\quad\quad \texttt{return}\,x\\
)))
\end{array}
$$

□

Theorem 6.2.

$$\widetilde{D} \sim_{\Xi\subseteq\Upsilon} \widetilde{MacForge}$$

Proof. We begin with the definition of $\widetilde{D}$ given in Equation 20. It describes a computation in which a key k is generated by $\widetilde{Gen}$, after which the function oracle $\widetilde{M}$ is applied to k to produce an oracle o. This oracle is then passed to the term *MacOracle* to produce a MAC oracle e, which is finally given as input to the distinguisher D to obtain a boolean outcome. We first apply Equation `letAss` to restructure the expression which allows us to isolate the application of *MacOracle*. Next, we replace the term $\widetilde{M}$ with $\widetilde{Oracle}$, since these two terms are Θ-contextually indistinguishable by Lemma 6.1. After this, we apply α-equivalence, as recalled in Chapter 4, to rename bound variables and align the resulting term with the expected structure of the

distinguisher. We then unfold the definition of D from Figure 24 and finally, we apply Equation reduc2 to eliminate redundant bindings and Equation letCom to reorder computations, yielding a term that is syntactically equal to $\widetilde{MacForge}$, which concludes the proof.

$$
\begin{aligned}
\widetilde{D} \triangleq\ & \mathtt{let}\, k = \widetilde{Gen} \star\, \mathtt{in}\, \mathtt{let}\, o = \widetilde{M} k\, \mathtt{in}\, \mathtt{let}\, e = MacOracle\, o\, \mathtt{in}\, D\, e \\
\overset{\text{letAss}}{\sim_{\Xi\subseteq\Upsilon}}\ & \mathtt{let}\, k = \widetilde{Gen} \star\, \mathtt{in}\, \mathtt{let}\, e = (\mathtt{let}\, o = \widetilde{M}\, k\, \mathtt{in}\, MacOracle\, o)\, \mathtt{in}\, D\, e \\
\overset{\text{Lemma 6.1}}{\sim_{\Xi\subseteq\Upsilon}}\ & \begin{array}{l} \mathtt{let}\, k = \widetilde{Gen} \star\, \mathtt{in} \\ \mathtt{let}\, e = \widetilde{Oracle}\, k\, \mathtt{in} \\ D\, e \end{array} \\
=_{\alpha}\ & \begin{array}{l} \mathtt{let}\, k = \widetilde{Gen} \star\, \mathtt{in} \\ \mathtt{let}\, o = \widetilde{Oracle}\, k\, \mathtt{in} \\ D\, o \end{array} \\
\triangleq\ & \mathtt{let}\, k = \widetilde{Gen} \star\, \mathtt{in}\, \mathtt{let}\, o = \widetilde{Oracle}\, k\, \mathtt{in} \\
& \left(\begin{array}{l} \mathtt{return}\, (\lambda o. \\ \quad \mathtt{let}\, \langle m, tag\rangle = Adv\, o\, \mathtt{in} \\ \quad \mathtt{let}\, queries = \mathtt{get}\, Q\, \mathtt{in} \\ \quad \mathtt{let}\, c = \mathtt{contains}(queries, m)\, \mathtt{in} \\ \quad \mathtt{if}\, c\, \mathtt{then}\, \mathtt{return}\, \mathbf{f} \\ \quad \mathtt{else}\, \mathtt{let}\, x = \mathtt{der}(o)\, m\, \mathtt{in}\, \mathtt{eq}(x, tag)) \end{array}\right) o
\end{aligned}
$$

$$
\begin{array}{lll}
 & \texttt{let}\, k = \widetilde{Gen}\, \star\, \texttt{in} & \\
 & \texttt{let}\, o = \widetilde{Oracle}\, k\, \texttt{in} & \\
 & \texttt{let}\, \langle m, tag\rangle = Adv\, o\, \texttt{in} & \\
\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}} & \texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} & = \quad \widetilde{MacForge} \\
 & \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} & \\
 & \texttt{if}\, c\, \texttt{then return f} & \\
 & \texttt{else let}\, x = \texttt{der}(o)\, m\, \texttt{in eq}(x, tag))
\end{array}
$$

□

6.2.2 From D^F to $MacForge^F$

For the left side of the diagram, we first establish the relation $M^F \sim_{\Xi\subseteq\Upsilon} Oracle^F$ in Lemma 6.3. Then, we prove the desired relation $D^F \sim_{\Xi\subseteq\Upsilon} MacForge^F$ in Theorem 6.4. The proofs of these statements follow the same structure and reasoning as those presented for the right side of the diagram.

Lemma 6.3.

$$\texttt{let}\, o = M^F\, \texttt{in}\, MacOracle\, o \sim_{\Xi\subseteq\Upsilon} Oracle^F\, k$$

Proof. The proof follows a similar approach to the proof of Lemma 6.1.

$$
\begin{array}{lll}
 & & \texttt{return}\,(\lambda k.\texttt{return!}(\texttt{return}\,(\lambda m. \\
 & & \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
 & & \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\
Oracle^F\, k & \triangleq & \quad \texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, m) \\
 & & \quad \texttt{else skip;} \\
 & & \quad Mac^F\langle k, m\rangle \\
 & &)))\, k
\end{array}
$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad Mac^F\langle k,m\rangle\\
)))
\end{array}
\overset{\text{reduc2}}{\sim_{\Xi\subseteq\Upsilon}}
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then}\\
\quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,\langle k,m\rangle = \langle k,m\rangle\,\texttt{in}\\
\quad \texttt{return}\,F\langle k,m\rangle\\
)))
\end{array}
\overset{\text{reduc5}}{\sim_{\Xi\subseteq\Upsilon}}
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then}\\
\quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad \texttt{return}\,F\langle k,m\rangle\\
)))
\end{array}
\triangleq
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad (\lambda x.\\
\quad\quad \texttt{let}\,\langle k,m\rangle = x\,\texttt{in}\\
\quad\quad \texttt{return}\,F\langle k,m\rangle\\
\quad)\langle k,m\rangle\\
)))
\end{array}
\tag{24}
$$

Now, the goal is to show that the term $\texttt{let}\,o = M^F\,k\,\texttt{in}\,MacOracle\,o$ is Θ-

contextual indistinguishable from the term above in Equation 24.

$$\texttt{let}\, o = M^F\, k\, \texttt{in}\, MacOracle\, o \quad \triangleq \quad \begin{array}{l} \texttt{let}\, o = M^F\, k\, \texttt{in} \\ \texttt{return}\,(\lambda o.\texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der}(\texttt{o})\, m) \\ ))))\, o \end{array}$$

$$\begin{array}{l} \texttt{let}\, o = M^F\, k\, \texttt{in} \\ \texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der}(\texttt{o})\, m) \\ ))) \end{array} \overset{\text{reduc2}}{\sim_{\Xi \subseteq \Upsilon}} \triangleq \quad \begin{array}{l} \texttt{let}\, o = \left(\begin{array}{l} \texttt{return}\,(\lambda k.\texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\ \quad \texttt{if not}(t)\, \texttt{then} \\ \quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else skip}; \\ \quad \texttt{return}\, F\langle k, m\rangle \\ )))) \end{array} \right) k\, \texttt{in} \\ \texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der}(\texttt{o})\, m) \\ ))) \end{array}$$

$$\begin{array}{l} \texttt{let}\, o = \left(\begin{array}{l} \texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\ \quad \texttt{if not}(t)\, \texttt{then} \\ \quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else skip}; \\ \quad \texttt{return}\, F\langle k, m\rangle \\ ))) \end{array} \right) \texttt{in} \\ \texttt{return}\,(!(\texttt{return}\,(\lambda m. \\ \quad \texttt{return}\,(\texttt{der}(\texttt{o})\, m) \\ ))) \end{array} \overset{\text{reduc2}}{\sim_{\Xi \subseteq \Upsilon}}$$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{return}\left(\begin{array}{l}\texttt{der}\left(!\left(\begin{array}{l}\texttt{return}\,(\lambda m.\\ \quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\ \quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\ \quad \texttt{if not}(t)\,\texttt{then}\\ \quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\ \quad \texttt{else skip};\\ \quad \texttt{return}\,F\langle k,m\rangle)\end{array}\right)\right)\\ m\end{array}\right)\\
)))
\end{array}
$$

$\overset{\texttt{reduc4}}{\sim}_{\Xi\subseteq\Upsilon}$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \left(\begin{array}{l}\texttt{return}\,(\lambda m.\\ \quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\ \quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\ \quad \texttt{if not}(t)\,\texttt{then}\\ \quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\ \quad \texttt{else skip};\\ \quad \texttt{return}\,F\langle k,m\rangle\end{array}\right) m\\
)))
\end{array}
$$

$\overset{\texttt{reduc3}}{\sim}_{\Xi\subseteq\Upsilon}$

$$
\begin{array}{l}
\texttt{return}\,(!(\texttt{return}\,(\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then}\\
\quad\quad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad \texttt{return}\,F\langle k,m\rangle\\
)))
\end{array}
\quad = \quad \text{Term in Equation 24}
$$

$\overset{\texttt{reduc2}}{\sim}_{\Xi\subseteq\Upsilon}$

□

Theorem 6.4.

$$D^F \sim_{\Xi \subseteq \Upsilon} MacForge^F$$

Proof. The proof follows a similar approach to the proof of Theorem 6.2.

$$
\begin{aligned}
D^F \quad &\triangleq \quad \texttt{let}\, k = Gen^F \star \,\texttt{in}\, \texttt{let}\, o = M^F k \,\texttt{in}\, \texttt{let}\, e = MacOracle\, o \,\texttt{in}\, D\, e \\
&\overset{\text{letAss}}{\sim_{\Xi \subseteq \Upsilon}} \quad \texttt{let}\, k = Gen^F \star \,\texttt{in}\, \texttt{let}\, e = (\texttt{let}\, o = M^F\, k \,\texttt{in}\, MacOracle\, o) \,\texttt{in}\, D\, e \\
&\overset{\text{Lemma 6.3}}{\sim_{\Xi \subseteq \Upsilon}} \quad \begin{array}{l} \texttt{let}\, k = Gen^F \star \,\texttt{in} \\ \texttt{let}\, e = Oracle^F\, k \,\texttt{in} \\ D\, e \end{array} \\
&=_{\alpha} \quad \begin{array}{l} \texttt{let}\, k = Gen^F \star \,\texttt{in} \\ \texttt{let}\, o = Oracle^F\, k \,\texttt{in} \\ D\, o \end{array} \\
&\triangleq \quad \begin{array}{l} \texttt{let}\, k = Gen^F \star \,\texttt{in} \\ \texttt{let}\, o = Oracle^F\, k \,\texttt{in} \\ \left(\begin{array}{l} \texttt{return}\, (\lambda o. \\ \quad \texttt{let}\, \langle m, tag \rangle = Adv\, o \,\texttt{in} \\ \quad \texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\ \quad \texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\ \quad \texttt{if}\, c \,\texttt{then}\, \texttt{return}\, \mathbf{f} \\ \quad \texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m \,\texttt{in}\, \texttt{eq}(x, tag)) \end{array} \right) o \end{array} \\
&\overset{\text{reduc2}}{\sim_{\Xi \subseteq \Upsilon}} \quad \begin{array}{l} \texttt{let}\, k = Gen^F \star \,\texttt{in} \\ \texttt{let}\, o = Oracle^F\, k \,\texttt{in} \\ \texttt{let}\, \langle m, tag \rangle = Adv\, o \,\texttt{in} \\ \texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\ \texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\ \texttt{if}\, c \,\texttt{then}\, \texttt{return}\, \mathbf{f} \\ \texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m \,\texttt{in}\, \texttt{eq}(x, tag) \end{array} \quad = \quad MacForge^F
\end{aligned}
$$

□

6.2.3 From $\widetilde{MacForge}$ to f

To complete the overall security proof, it remains to establish the relation $\widetilde{MacForge} \sim_{\Xi \subseteq \Upsilon} \mathbf{f}$, which corresponds to the right part of the diagram in Figure 26.

We introduce assertions, in the style of standard Hoare proofs, which are indicated in blue. The rules are composed using Rule H-LET and correspond to derivation trees that establish the propositions annotated after each line. We state two lemmas, namely, Lemma 6.5 and Lemma 6.6, that are used to decorate the term $\widetilde{MacForge}$ and to show that a given assertion holds at a specific point in the term. Once the assertion is established, we use it to simplify part of the term using equations introduced in Section 5.2. Finally, we conclude by standard equational reasoning of $\lambda\mathbf{BLL}$, namely Θ-contextual indistinguishability, which is applied to the term obtained after the rewriting through Hoare logic.

Lemma 6.5. *The following Hoare triple holds:*

$$\{\top\}\,\widetilde{Gen} \star \{\lambda\_.\texttt{coherent}(Q, rand)\}$$

Proof. Since we evaluate $\widetilde{Gen}$ at $\star$, we first eliminate the abstraction using H-LAM and H-APP. Then we reason about the body of $\widetilde{Gen}$ and we need to prove the triple for the body. Sequential composition is justified by repeated

applications of H-LET.

$\{\top\}$

$\texttt{let}\, q = \texttt{initQ}\, \texttt{in}$

$\{\texttt{coherent}(q, rand)\}$ H-FUN, H-CONS

$\texttt{set}\, Q\, q;$

$\{\texttt{coherent}(Q, rand)\}$ H-SET, H-CONS

$\texttt{let}\, tab = \texttt{initTable}\, \texttt{in}$

$\{\texttt{coherent}(Q, tab)\}$ H-FUN, H-CONS

$\texttt{set}\, rand\, tab;$

$\{\texttt{coherent}(Q, rand)\}$ H-SET

$\texttt{let}\, z = \texttt{zero}\, \texttt{in}$

$\{\texttt{coherent}(Q, rand)\}$ H-FUN

$\texttt{return}\, z$

$\{\lambda z.\texttt{coherent}(Q, rand)\}$ H-ETA

□

Lemma 6.6. *Let M be the term*

$$
\begin{aligned}
M \triangleq\ & !(\texttt{return}\,\lambda m.\\
& \quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
& \quad \texttt{let}\,t = \texttt{contains}(q, m)\,\texttt{in}\\
& \quad \texttt{if not}(t)\,\texttt{then}\\
& \quad\quad \texttt{let}\,q' = \texttt{modifyQ}(q, m)\,\texttt{in}\\
& \quad\quad \texttt{set}\,Q\,q'\\
& \quad \texttt{else skip};\\
& \quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
& \quad \texttt{let}\,b = \texttt{isdefined}(tab, m)\,\texttt{in}\\
& \quad \texttt{if}\,b\,\texttt{then}\\
& \quad\quad \texttt{let}\,x = \texttt{getValue}(tab, m)\,\texttt{in}\\
& \quad\quad \texttt{return}\,x\\
& \quad \texttt{else}\\
& \quad\quad \texttt{let}\,x = \texttt{random}\,\texttt{in}\\
& \quad\quad \texttt{let}\,newtab = \texttt{modify}(tab, m, x)\,\texttt{in}\\
& \quad\quad \texttt{set}\,rand\,newtab;\\
& \quad\quad \texttt{return}\,x)
\end{aligned}
$$

The following Hoare triple holds:

$$\{\texttt{coherent}(Q, rand)\}\ M\ \{\lambda v.\texttt{coherent}(Q, rand)\}$$

Proof. The proof follows the same structure as that of Lemma 6.5. We extend the derivation with assertions justified by the rules introduced in Figure 18. Sequential composition is handled by repeated applications of Rule H-LET, while Rule H-FUN is applied to function symbols. We reason on the body of the term, implicitly applying Rule H-BANG, followed by Rule H-ETA, and finally Rule H-LAM to eliminate the abstraction. The states of the ledgers Q and $rand$ are described using standard set notation. For instance,

$Q \setminus m$ denotes the ledger obtained from Q by removing the element m, more formally $Q \setminus \{m\} := \{\, x \in Q \mid x \neq m \,\}$. The ledger *rand* is specified in assertions as a set of pairs. The notation $rand \cup \{(m, x)\}$ denotes the ledger obtained by adding a new pair (m, x), where m and x are bitstrings of length $\mathbb{S}[i]$. In the concrete representation, such a pair is modeled as a single entry of type $\mathbb{S}[2i + 1]$, where the additional bit indicates whether the entry is active. The status bit is handled by the underlying logic of the function symbol `modify`. We use syntactic sugar of Equation 16 and 17 to avoid unnecessary binding and to simplify computations that do not add effects.

$\{\texttt{coherent}(Q, rand)\}$

$\texttt{let}\, q = \texttt{get}\, Q\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q)\}$ H-GET

$\texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge t = (m \in Q)\}$ H-FUN, H-CONS

$\texttt{if not}(t)\, \texttt{then}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge t = (m \in Q) \wedge (t = \mathbf{f})\}$ H-CASE

$\Longrightarrow \{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin Q)\}$ H-CONS

$\texttt{let}\, q' = \texttt{modifyQ}(q, m)\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin Q) \wedge q' = \texttt{modifyQ}(q, m)\}$ H-FUN

$\Longrightarrow \{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin Q) \wedge q' = (q \cup m)\}$ H-CONS

$\Longrightarrow \{\texttt{coherent}(q' \setminus \{m\}, rand) \wedge (q = Q) \wedge (m \notin Q) \wedge q' = (q \cup m)\}$ H-CONS

$\texttt{set}\, Q\, q'$

$\{\texttt{coherent}(Q \setminus \{m\}, rand) \wedge (m \in Q)\}$ H-SET

$\texttt{else}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \in Q)\}$ H-CASE, H-CONS

$\texttt{skip};$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \in Q)\}$ H-ETA

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \in Q)\} \Longrightarrow$

$\{\texttt{coherent}(Q \setminus \{m\}, rand) \wedge (m \in Q)\}$ H-CONS

$\texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in}$

$\{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand)\}$ H-GET

$\texttt{let}\, b = \texttt{isdefined}(tab, m)\, \texttt{in}$

$\{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge b = (m \in tab)\}$ H-FUN

$\texttt{if}\, b\, \texttt{then}$

$\{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge b = (m \in tab) \wedge (b = \mathbf{t})\}$ H-CASE

$\Longrightarrow \{\texttt{coherent}(Q, tab) \wedge (tab = rand)\}$ H-CONS

$\texttt{let}\, x = \texttt{getValue}(tab, m)\, \texttt{in}$

$\texttt{return}\, x$

$\{\lambda x.\texttt{coherent}(Q, rand)\}$ H-CONS, H-FUN, H-ETA

$\texttt{else}$

$\{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge b = (m \in tab) \wedge (b = \mathbf{f})\}$ H-CASE

$\Longrightarrow \{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge (m \notin tab)\}$ H-CONS

$\texttt{let}\, x = \texttt{random}\, \texttt{in}$

$\texttt{let}\, newtab = \texttt{modify}(tab, m, x)\, \texttt{in}$

$\{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge (m \notin tab) \wedge newtab = \texttt{modify}(tab, m, x)\}$ H-FUN

$\Longrightarrow \{\texttt{coherent}(Q \setminus \{m\}, tab) \wedge (m \in Q) \wedge (tab = rand) \wedge (m \notin tab) \wedge newtab = (tab \cup \{(m, x)\})\}$ H-CONS

$\Longrightarrow \{\texttt{coherent}(Q \setminus \{m\}, tab \cup \{(m, x)\}) \wedge (m \in Q) \wedge (tab = rand) \wedge (m \notin tab) \wedge newtab = (tab \cup \{(m, x)\})\}$ H-CONS

$\texttt{set}\, rand\, newtab;$

$\{\texttt{coherent}(Q \setminus \{m\}, rand) \wedge (m \in Q) \wedge (m \in rand)\} \Longrightarrow \{\texttt{coherent}(Q, rand)\}$ H-SET, H-CONS

$\texttt{return}\, x$

$\{\lambda x.\texttt{coherent}(Q, rand)\}$ H-ETA

□

Finally we state and prove Theorem 6.7 that completes the proof.

Theorem 6.7.

$$\widetilde{MacForge} \sim_{\Xi \subseteq \Upsilon} \mathbf{f}$$

Proof.

$$
\begin{aligned}
\widetilde{MacForge} \triangleq\ & \texttt{let}\, k = \widetilde{Gen}\, \star\, \texttt{in} \\
& \texttt{let}\, o = \widetilde{Oracle}\, k\, \texttt{in} \\
& \texttt{let}\, \langle m, tag \rangle = Adv\, o\, \texttt{in} \\
& \texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
& \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
& \texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
& \texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m\, \texttt{in}\, \texttt{eq}(x, tag)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{let}\, k = \widetilde{Gen}\, \star\, \texttt{in} \\
& \texttt{let}\, o = \left(\begin{array}{l} \texttt{return}\, (\lambda k.\texttt{return}\, !(\texttt{return}\, \lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m \rangle)) \end{array} \right) k\, \texttt{in} \\
\triangleq\ & \texttt{let}\, \langle m, tag \rangle = Adv\, o\, \texttt{in} \\
& \texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
& \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
& \texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
& \texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m\, \texttt{in}\, \texttt{eq}(x, tag)
\end{aligned}
$$

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen}\, \star\, \texttt{in} \\
\texttt{let}\, o = \left(\begin{array}{l}
\texttt{return}\,!(\texttt{return}\, \lambda m. \\
\quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\
\quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\, \texttt{skip}; \\
\quad \widetilde{Mac}\langle k, m \rangle)
\end{array} \right) \texttt{in} \\
\overset{\texttt{reduc2}}{\sim_{\Xi \subseteq \Upsilon}} \\
\texttt{let}\, \langle m, tag \rangle = Adv\, o\, \texttt{in} \\
\texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
\texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
\texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else}\, \texttt{let}\, x = \texttt{der}(o)\, m\, \texttt{in}\, \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen}\, \star\, \texttt{in} \\
\texttt{let}\, \langle m, tag\rangle = Adv \left(\begin{array}{l} !(\texttt{return}\, \lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m\rangle) \end{array} \right) \texttt{in} \\
\texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
\overset{\texttt{reduc4}}{\sim_{\Xi \subseteq \Upsilon}} \quad \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
\texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else}\, \texttt{let}\, x = \texttt{der} \left(\begin{array}{l} !(\texttt{return}\, \lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m\rangle) \end{array} \right) m\, \texttt{in} \\
\texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen} \star \,\texttt{in} \\
\texttt{let}\, \langle m, tag\rangle = Adv \left(\begin{array}{l} !(\texttt{return}\, \lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m\rangle) \end{array} \right) \,\texttt{in} \\
\texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\
\overset{\text{reduc3}}{\sim_{\Xi \subseteq \Upsilon}} \quad \texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\
\texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else}\, \texttt{let}\, x = \left(\begin{array}{l} \texttt{return}\, (\lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m\rangle) \end{array} \right) m \,\texttt{in} \\
\texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen} \star \,\texttt{in} \\
\texttt{let}\, \langle m, tag\rangle = Adv \left(\begin{array}{l} !(\texttt{return}\, \lambda m. \\ \quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\ \quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\ \quad \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \quad \texttt{else}\, \texttt{skip}; \\ \quad \widetilde{Mac}\langle k, m\rangle) \end{array} \right) \,\texttt{in} \\
\overset{\texttt{reduc2}}{\sim_{\Xi \subseteq \Upsilon}} \texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\
\texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\
\texttt{if}\, c\, \texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else}\, \texttt{let}\, x = \left(\begin{array}{l} \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\ \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\ \texttt{if}\, \texttt{not}(t)\, \texttt{then}\, \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\ \texttt{else}\, \texttt{skip}; \\ \widetilde{Mac}\langle k, m\rangle \end{array} \right) \,\texttt{in} \\
\texttt{eq}(x, tag)
\end{array}
$$

At this stage, the adversary is given access to a term that can be evaluated for at most a polynomial number of queries q, and it must output a candidate forgery. In the else branch, the oracle is reduced to an expression that applies a sequence of operations to the query ledger and returns the output of $\widetilde{Mac}$. We focus on the lower part of the term, namely the else branch. This branch is unrolled by expanding the definition of $\widetilde{Mac}$. By repeated applications of Equation `letAss`, the resulting term is rearranged. Applying Equation `iflet` pushes the binding of the variable x inside the conditional. The resulting term simplifies reasoning in terms of Hoare assertions.

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen} \star\, \texttt{in} \\
\texttt{let}\, \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\, \lambda m. \\
\quad \texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\
\quad \texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad (\lambda e. \\
\quad \texttt{let}\, \langle k, m\rangle = e\, \texttt{in} \\
\quad \texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, m)\, \texttt{in} \\
\quad \texttt{if}\, b\, \texttt{then} \\
\quad\quad \texttt{let}\, x = \texttt{getValue}(tab, m)\, \texttt{in} \\
\quad\quad \texttt{return}\, x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\, x = \texttt{random}\, \texttt{in} \\
\quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x)\, \texttt{in} \\
\quad\quad \texttt{set}\, rand\, newtab; \\
\quad\quad \texttt{return}\, x)\langle k, m\rangle)
\end{array}\right) \texttt{in} \\
\triangleq \quad \texttt{let}\, queries = \texttt{get}\, Q\, \texttt{in} \\
\quad \texttt{let}\, c = \texttt{contains}(queries, m)\, \texttt{in} \\
\quad \texttt{if}\, c\, \texttt{then return}\, \mathbf{f} \\
\quad \texttt{else let}\, x = \left(\begin{array}{l}
\texttt{let}\, q = \texttt{get}\, Q\, \texttt{in} \\
\texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in} \\
\texttt{if not}(t)\, \texttt{then set}\, Q\, \texttt{modifyQ}(q, m) \\
\texttt{else skip}; \\
(\lambda e. \\
\texttt{let}\, \langle k, m\rangle = e\, \texttt{in} \\
\texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in} \\
\texttt{let}\, b = \texttt{isdefined}(tab, m)\, \texttt{in} \\
\texttt{if}\, b\, \texttt{then} \\
\quad \texttt{let}\, x = \texttt{getValue}(tab, m)\, \texttt{in} \\
\quad \texttt{return}\, x \\
\texttt{else} \\
\quad \texttt{let}\, x = \texttt{random}\, \texttt{in} \\
\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x)\, \texttt{in} \\
\quad \texttt{set}\, rand\, newtab; \\
\quad \texttt{return}\, x)\langle k, m\rangle
\end{array}\right) \texttt{in} \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen}\ \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then}\ \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\ \texttt{skip}; \\
\quad \texttt{let}\ \langle k, m\rangle = \langle k, m\rangle\ \texttt{in} \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\overset{\text{reduc2}}{\sim_{\Xi \subseteq \Upsilon}}\ \texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then}\ \texttt{return}\ \mathbf{f} \\
\texttt{else}\ \texttt{let}\ x = \left(\begin{array}{l}
\texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\texttt{else}\ \texttt{skip}; \\
\texttt{let}\ \langle k, m\rangle = \langle k, m\rangle\ \texttt{in} \\
\texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\texttt{if}\ b\ \texttt{then} \\
\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad \texttt{return}\ x \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad \texttt{set}\ rand\ newtab; \\
\quad \texttt{return}\ x
\end{array}\right)\ \texttt{in} \\
\texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then}\ \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\ \texttt{skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\qquad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\qquad \texttt{return}\ x \\
\quad \texttt{else} \\
\qquad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\qquad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\qquad \texttt{set}\ rand\ newtab; \\
\qquad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\overset{\text{reduc5}}{\sim_{\Xi \subseteq \Upsilon}}\ \texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then}\ \texttt{return}\ \mathbf{f} \\
\texttt{else}\ \texttt{let}\ x = \left(\begin{array}{l}
\texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\texttt{else}\ \texttt{skip}; \\
\texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\texttt{if}\ b\ \texttt{then} \\
\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad \texttt{return}\ x \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad \texttt{set}\ rand\ newtab; \\
\quad \texttt{return}\ x
\end{array}\right)\ \texttt{in} \\
\texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen}\ \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\ \texttt{skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\overset{\texttt{letAss}}{\sim_{\Xi \subseteq \Upsilon}}\ \texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then}\ \texttt{return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\ \texttt{skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{let}\ x = \left(\begin{array}{l}
\texttt{if}\ b\ \texttt{then} \\
\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad \texttt{return}\ x \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad \texttt{set}\ rand\ newtab; \\
\quad \texttt{return}\ x
\end{array}\right)\ \texttt{in} \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen} \star \,\texttt{in} \\
\texttt{let}\, \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\, \lambda m. \\
\quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\
\quad \texttt{if}\, \texttt{not}(t) \,\texttt{then} \\
\quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\, \texttt{skip}; \\
\quad \texttt{let}\, tab = \texttt{get}\, rand \,\texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, m) \,\texttt{in} \\
\quad \texttt{if}\, b \,\texttt{then} \\
\quad\quad \texttt{let}\, x = \texttt{getValue}(tab, m) \,\texttt{in} \\
\quad\quad \texttt{return}\, x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\, x = \texttt{random} \,\texttt{in} \\
\quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x) \,\texttt{in} \\
\quad\quad \texttt{set}\, rand\, newtab; \\
\quad\quad \texttt{return}\, x)
\end{array}\right) \,\texttt{in} \\
\overset{\texttt{iflet}}{\sim_{\Xi \subseteq \Upsilon}} \texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\
\texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\
\texttt{if}\, c \,\texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\
\quad \texttt{if}\, \texttt{not}(t) \,\texttt{then} \\
\quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\, \texttt{skip}; \\
\quad \texttt{let}\, tab = \texttt{get}\, rand \,\texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, m) \,\texttt{in} \\
\quad \texttt{if}\, b \,\texttt{then} \\
\quad\quad \texttt{let}\, x = \texttt{getValue}(tab, m) \,\texttt{in} \\
\quad\quad \texttt{eq}(x, tag) \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\, x = \texttt{random} \,\texttt{in} \\
\quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x) \,\texttt{in} \\
\quad\quad \texttt{set}\, rand\, newtab; \\
\quad\quad \texttt{eq}(x, tag)
\end{array}
$$

Now to complete the proof we need to refine the typing of the term Adv with respect of the typing rules introduced in Figure 18. The original type

designated for Adv was $\vdash_c^{\varnothing} Adv : !_q\big(\mathbb{S}[i] \multimap \mathbb{S}[i]\big) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i])$, now we lift the type of Adv in the new Hoare typing system using the map μ defined in Equation 18. By fixing a proposition P as $\texttt{coherent}(Q, rand)$ and by applying Lemma 5.1, the type of Adv is lifted as follows:

$$\Gamma \vdash_c^{\varnothing} Adv : !_q\big(\mathbb{S}[i] \multimap \mathbb{S}[i]\big) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i])$$

and

$$\mu(\Gamma, P) \vdash_c^{\Xi} Adv : \mu(!_q\big(\mathbb{S}[i] \multimap \mathbb{S}[i]\big) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i])).$$

Which expands as

$$\begin{aligned}
&\mu\big(!_q(\mathbb{S}[i] \multimap \mathbb{S}[i]) \multimap (\mathbb{S}[i] \otimes \mathbb{S}[i]), P\big) \\
&= \mu\big(!_q(\mathbb{S}[i] \multimap \mathbb{S}[i]), P\big) \underset{P,\lambda_.P}{\multimap} \mu(\mathbb{S}[i] \otimes \mathbb{S}[i], P) \\
&= !_q^{P,P}\mu(\mathbb{S}[i] \multimap \mathbb{S}[i], P) \underset{P,\lambda_.P}{\multimap} (\mathbb{S}[i] \otimes \mathbb{S}[i]) \\
&= !_q^{P,P}\Big(\mathbb{S}[i] \underset{P,\lambda_.P}{\multimap} \mathbb{S}[i]\Big) \underset{P,\lambda_.P}{\multimap} (\mathbb{S}[i] \otimes \mathbb{S}[i]).
\end{aligned}$$

The application of Lemma 5.1 is justified by the type of Adv and by the fact that the typing derivation is performed in the empty context, so that $\mathcal{R}(P) \cap \varnothing = \emptyset$. Moreover, $\Theta = \varnothing$ and $\Xi = \varnothing$. Under these conditions, the type of Adv guarantees that P is preserved as an invariant. By applying the lifted typing judgment for Adv and combining it with Lemma 6.6, we can propagate the assertion $\texttt{coherent}(Q, rand)$ down into the term.

$\{\top\}$

$\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in}$

$\{\texttt{coherent}(Q, rand)\}$ Lemma 6.5

$$\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l} !(\texttt{return}\ \lambda m. \\ \quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\ \quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\ \quad \texttt{if not}(t)\ \texttt{then} \\ \quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\ \quad \texttt{else skip}; \\ \quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\ \quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\ \quad \texttt{if}\ b\ \texttt{then} \\ \quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\ \quad\quad \texttt{return}\ x \\ \quad \texttt{else} \\ \quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\ \quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\ \quad\quad \texttt{set}\ rand\ newtab; \\ \quad\quad \texttt{return}\ x) \end{array}\right) \texttt{in}$$

$\{\texttt{coherent}(Q, rand)\}$ Lemmatas 5.1, 6.6

$\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in}$

$\{\texttt{coherent}(queries, rand) \wedge (queries = Q)\}$ H-GET

$\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in}$

$\{\texttt{coherent}(queries, rand) \wedge (queries = Q) \wedge c = (m \in queries)\}$ H-FUN, H-CONS

$\texttt{if}\ c\ \texttt{then}$

$\{\texttt{coherent}(queries, rand) \wedge (queries = Q) \wedge (m \in queries)\}$ H-CASE, H-CONS

$\texttt{return}\ \mathbf{f}$

$\{\lambda b.\texttt{coherent}(Q, rand) \wedge (m \in Q)\}$ H-CONS, H-ETA

$\texttt{else}$

$\{\texttt{coherent}(queries, rand) \wedge (queries = Q) \wedge (m \notin queries)\}$ H-CASE, H-CONS

$\texttt{let}\ q = \texttt{get}\ Q\ \texttt{in}$

$\{\texttt{coherent}(queries, rand) \wedge (queries = Q) \wedge (m \notin queries) \wedge (q = Q)\}$ H-GET, H-CONS

$\Longrightarrow\ \{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin q)\}$ H-CONS

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin q)\}$ H-CONS

$\texttt{let}\, t = \texttt{contains}(q, m)\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin q) \wedge t = (m \in q)\}$ H-FUN

$\quad \Longrightarrow \{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin q) \wedge t = \mathbf{f}\}$ H-CONS

$\texttt{if not}(t)\, \texttt{then}$

$\quad \{\texttt{coherent}(q, rand) \wedge (q = Q) \wedge (m \notin q)\}$ H-CASE, H-CONS

$\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m)$

$\quad \{\texttt{coherent}(q, rand) \wedge (Q = q \cup \{m\}) \wedge (m \notin q)\}$ H-FUN, H-SET, H-CONS

$\texttt{else}$

$\quad \texttt{skip};$

$\{\texttt{coherent}(q, rand) \wedge (Q = q \cup \{m\}) \wedge (m \notin q)\}$

$\texttt{let}\, tab = \texttt{get}\, rand\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (Q = q \cup \{m\}) \wedge (m \notin q) \wedge (tab = rand)\}$ H-GET

$\texttt{let}\, b = \texttt{isdefined}(tab, m)\, \texttt{in}$

$\{\texttt{coherent}(q, rand) \wedge (Q = q \cup \{m\}) \wedge (m \notin q) \wedge (tab = rand) \wedge (b = (m \in tab))\}$ H-FUN

$\quad \Longrightarrow \{b = \mathbf{f}\}$ H-CONS

$\texttt{if}\, b\, \texttt{then}$

$\quad \texttt{let}\, x = \texttt{getValue}(tab, m)\, \texttt{in}$

$\quad \texttt{eq}(x, tag)$

$\texttt{else}$

$\quad \texttt{let}\, x = \texttt{random}\, \texttt{in}$

$\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x)\, \texttt{in}$

$\quad \texttt{set}\, rand\, newtab;$

$\quad \texttt{eq}(x, tag)$

At this point we do not introduce any further assertions. We observe instead that two assertion entails that the guard of a conditional is invariably evaluated to false or true. In particular, the conditional determines whether the oracle outputs a uniformly random bitstring upon receiving a fresh message generated by Adv, or whether it returns a predetermined bitstring stored in the ledger $rand$.

The message produced by the adversary must be fresh hence it cannot appear in the ledger $rand$, which records all outputs of the random oracle. Moreover, it cannot appear in the set Q at this stage either, because the adversary's objective is precisely to produce a fresh message–tag pair. Since the oracle first updates the query ledger and only then invokes $\widetilde{Mac}$ to recompute the tag and verify that the tag provided by the adversary is valid, by

tracking the initial query ledger, under the precondition that it is consistent with the *rand* ledger and that it does not contain the message m, we can conclude that m does not appear in the *rand* ledger as well. Consequently, the conditional collapses to the branch that returns a random bitstring. We apply Equation ifFalse and Equation ifTrue to further rewrite the term.

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen}\ \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then} \\
\quad \texttt{return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \{t = \mathbf{f}\} \\
\quad \texttt{if not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else} \\
\quad\quad \texttt{skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \{b = \mathbf{f}\} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{eq}(x, tag) \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\underset{\sim}{\overset{\text{ifFalse ifTrue}}{}} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m); \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad \texttt{set}\ rand\ newtab; \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

At this point, in order to conclude, we leave assertions aside and we start to simplify the term going back to the notion of Θ-contextual indistinguishability of [LGG24].

$$
\begin{array}{l}
\texttt{let}\, k = \widetilde{Gen} \star \,\texttt{in} \\
\texttt{let}\, \langle m, tag \rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\, \lambda m. \\
\quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\
\quad \texttt{if}\, \texttt{not}(t) \,\texttt{then} \\
\quad\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m) \\
\quad \texttt{else}\, \texttt{skip}; \\
\quad \texttt{let}\, tab = \texttt{get}\, rand \,\texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, m) \,\texttt{in} \\
\quad \texttt{if}\, b \,\texttt{then} \\
\quad\quad \texttt{let}\, x = \texttt{getValue}(tab, m) \,\texttt{in} \\
\quad\quad \texttt{return}\, x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\, x = \texttt{random} \,\texttt{in} \\
\quad\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x) \,\texttt{in} \\
\quad\quad \texttt{set}\, rand\, newtab; \\
\quad\quad \texttt{return}\, x)
\end{array} \right) \texttt{in} \\
\texttt{let}\, queries = \texttt{get}\, Q \,\texttt{in} \\
\texttt{let}\, c = \texttt{contains}(queries, m) \,\texttt{in} \\
\texttt{if}\, c \,\texttt{then}\, \texttt{return}\, \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\, q = \texttt{get}\, Q \,\texttt{in} \\
\quad \texttt{let}\, t = \texttt{contains}(q, m) \,\texttt{in} \\
\quad \texttt{set}\, Q\, \texttt{modifyQ}(q, m); \\
\quad \texttt{let}\, tab = \texttt{get}\, rand \,\texttt{in} \\
\quad \texttt{let}\, b = \texttt{isdefined}(tab, m) \,\texttt{in} \\
\quad \texttt{let}\, x = \texttt{random} \,\texttt{in} \\
\quad \texttt{let}\, newtab = \texttt{modify}(tab, m, x) \,\texttt{in} \\
\quad \texttt{set}\, rand\, newtab; \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$\sim_{\Xi \subseteq \Upsilon}$
letCom

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag \rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array} \right) \texttt{in} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m); \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad \texttt{set}\ rand\ newtab; \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\qquad \sim_{\Xi \subseteq \Upsilon}^{\texttt{letAss}} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{let}\ \_ = \left(\begin{array}{l}
\texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\texttt{set}\ Q\ \texttt{modifyQ}(q, m); \\
\texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\texttt{set}\ rand\ newtab
\end{array}\right)\ \texttt{in} \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ k = \widetilde{Gen} \star\ \texttt{in} \\
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\qquad \underset{\texttt{safeDisc1}}{\sim_{\Xi\subseteq\Upsilon}} \\
\texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\texttt{if}\ c\ \texttt{then return}\ \mathbf{f} \\
\texttt{else} \\
\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right) \texttt{in} \\
\sim_{\Xi \subseteq \Upsilon} \\
\texttt{safeDisc1} \\
\quad \texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\quad \texttt{if}\ c\ \texttt{then return}\ \mathbf{f} \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{eq}(x, tag)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\ \lambda m. \\
\quad \texttt{let}\ q = \texttt{get}\ Q\ \texttt{in} \\
\quad \texttt{let}\ t = \texttt{contains}(q, m)\ \texttt{in} \\
\quad \texttt{if}\ \texttt{not}(t)\ \texttt{then} \\
\quad\quad \texttt{set}\ Q\ \texttt{modifyQ}(q, m) \\
\quad \texttt{else skip}; \\
\quad \texttt{let}\ tab = \texttt{get}\ rand\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{isdefined}(tab, m)\ \texttt{in} \\
\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad \texttt{let}\ x = \texttt{getValue}(tab, m)\ \texttt{in} \\
\quad\quad \texttt{return}\ x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ x = \texttt{random}\ \texttt{in} \\
\quad\quad \texttt{let}\ newtab = \texttt{modify}(tab, m, x)\ \texttt{in} \\
\quad\quad \texttt{set}\ rand\ newtab; \\
\quad\quad \texttt{return}\ x)
\end{array}\right)\ \texttt{in} \\
\qquad \texttt{let}\ queries = \texttt{get}\ Q\ \texttt{in} \\
\qquad \texttt{let}\ c = \texttt{contains}(queries, m)\ \texttt{in} \\
\qquad \texttt{if}\ c\ \texttt{then return f} \\
\qquad \texttt{else return f}
\end{array}
$$

$\sim_{\Xi \subseteq \Upsilon}$
randF

$$
\underset{\text{ifConst}}{\sim_{\Xi\subseteq\Upsilon}}
\begin{array}{l}
\texttt{let}\,\langle m, tag\rangle = Adv \left(\begin{array}{l}
!(\texttt{return}\,\lambda m.\\
\quad \texttt{let}\,q = \texttt{get}\,Q\,\texttt{in}\\
\quad \texttt{let}\,t = \texttt{contains}(q,m)\,\texttt{in}\\
\quad \texttt{if not}(t)\,\texttt{then}\\
\qquad \texttt{set}\,Q\,\texttt{modifyQ}(q,m)\\
\quad \texttt{else skip};\\
\quad \texttt{let}\,tab = \texttt{get}\,rand\,\texttt{in}\\
\quad \texttt{let}\,b = \texttt{isdefined}(tab,m)\,\texttt{in}\\
\quad \texttt{if}\,b\,\texttt{then}\\
\qquad \texttt{let}\,x = \texttt{getValue}(tab,m)\,\texttt{in}\\
\qquad \texttt{return}\,x\\
\quad \texttt{else}\\
\qquad \texttt{let}\,x = \texttt{random in}\\
\qquad \texttt{let}\,newtab = \texttt{modify}(tab,m,x)\,\texttt{in}\\
\qquad \texttt{set}\,rand\,newtab;\\
\qquad \texttt{return}\,x)
\end{array}\right) \texttt{in}\\
\texttt{let}\,queries = \texttt{get}\,Q\,\texttt{in}\\
\texttt{let}\,c = \texttt{contains}(queries,m)\,\texttt{in}\\
\texttt{return}\,\mathbf{f}
\end{array}
$$

$$
\underset{\texttt{safeDisc1}}{\sim_{\Xi\subseteq\Upsilon}} \quad \texttt{return}\,\mathbf{f}
$$

□

Summing up, we return to the diagram in Figure 26, which provides a structured view of the overall security proof in the λ**BLL** framework. We first establish the key equivalences involving the distinguishers. In particular, Theorem 6.2 shows that $\widetilde{D} \sim \widetilde{MacForge}$, while Theorem 6.4 establishes that $D^F \sim_{\Xi\subseteq\Upsilon} MacForge^F$. The final step of the proof is to show that $\widetilde{MacForge} \sim_{\Xi\subseteq\Upsilon}$ $\texttt{return}\,\mathbf{f}$, corresponding to the top-right part of the diagram. To obtain this result, we first prove that certain invariants hold

throughout the subterms of the experiment. This is done using the Hoare logic introduced in Section 5.2, which extends the original typing system of λ**BLL**. Decorating the terms with assertions allows us to establish properties about the state of the experiment. These properties enable the application of the equations introduced in Section 5.2, which collapse unreachable conditional branches. Using Lemma 5.1, we then reason about the typing of the adversary. Once we show in Lemma 6.6 that the oracle with which the adversary interacts satisfies a given invariant, the type of the adversary ensures that the invariant is preserved. Using the typing rules for assertions, we propagate the invariant through the term and apply the equations mentioned above. Finally, we return to the original contextual indistinguishability reasoning. By mapping the behavior of each component in the proof to its counterpart through the logical relation, we conclude that the scheme satisfies the required indistinguishability guarantees.

Conclusions

And then it concludes here.

Bibliography

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00, page 3–22, Berlin, Heidelberg, 2000. Springer-Verlag.

[Bar13] H. P. Barendregt. *Lambda Calculus with Types*. Cambridge University Press, New York, 2013.

[BDG+14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. *EasyCrypt: A Tutorial*, pages 146–166. Springer International Publishing, Cham, 2014.

[BDJ+24] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Joseph Lallemand. The Squirrel Prover and its Logic. *ACM SIGLOG News*, 11(2), April 2024.

[Bel11] Steven M. Bellovin. Frank miller: Inventor of the one-time pad. *Cryptologia*, 35(3):203–222, 2011.

[Bla05] Bruno Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models*, Paris, France, June 2005.

[Bla22] Bruno Blanchet. The security protocol verifier proverif and its horn clause resolution algorithm. *Electronic Proceedings in Theoretical Computer Science*, 373:14–22, November 2022.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, 2004:331, 2004.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941.

[Cur34] H. B. Curry. Functionality in combinatory logic*. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[DH22] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography*, page 365–390. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022.

[DLMS04] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, August 1981.

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

[Gir89] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, New York, 1989.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In David McAllester, editor, *Automated Deduction - CADE-17*, pages 271–290, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, Jan. 2010.

[Gol01] Oded Goldreich. *Foundations of Cryptography. Vol. 1: Basic tools*. Cambridge University Press, 2001.

[Gol08] Oded Goldreich. Computational complexity: a conceptual perspective. *SIGACT News*, 39(3):35–39, September 2008.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[GW96] Oded Goldreich and Avi Wigderson. Theory of computing: a scientific perspective. *ACM Comput. Surv.*, 28(4es):218–es, December 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[IL89] R. Impagliazzo and M. Luby. One-way functions are essential for complexity based cryptography. In *30th Annual Symposium on Foundations of Computer Science*, pages 230–235, 1989.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[Lev12] Paul Blain Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer Science & Business Media, 2012.

[LGG24] Ugo Dal Lago, Zeinab Galal, and Giulia Giusti. On computational indistinguishability and logical relations, 2024.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 696–701, Berlin, Heidelberg, 2013. Springer-Verlag.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[Ros] Mike Rosulek. The joy of cryptography. https://joyofcryptography.com.

[RP10] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 157–168, New York, NY, USA, 2010. Association for Computing Machinery.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949.

[Sho04] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332, 01 2004.

[Sin99] Simon Singh. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography.* Doubleday, USA, 1st edition, 1999.

[Sø07] Morten Heine Sørensen. *Lectures on the Curry-Howard Isomorphism.* Elsevier, Boston, 2007.

[vR09] Femke van Raamsdonk. Lambda-calculus and combinators, an introduction, 2nd edition, j. roger hindley and jonathan p. seldin, cambridge university press, 2008. hardback, isbn 9780521898850. *Theory Pract. Log. Program.*, 9:239–243, 2009.

Appendix A

Useful Functional Symbols and Equations in lambda-BLL

Table A.1 introduces a collection of function symbols operating over boolean values and bitstrings, specifically tailored to facilitate random generation, basic data manipulation, and operations on a specialized ledger structure. While individually simple, these symbols collectively provide a robust framework for modeling both pure computation and stateful interaction within λ**BLL**. The symbols in Table A.1 are divided in three groups based on their functional roles. It's important to note that λ**BLL**, each function symbol in $\mathcal{F}$ denotes a mapping into a probability distribution over values of its respective output type, as established in Section 1 of [LGG24]. Consequently, when a function symbol is interpreted as returning a specific value v, it corresponds to the Dirac distribution centered at v (denoted δ_v), which assigns probability 1 to v and 0 to all other outcomes.

Function Symbol	**Type**	**Interpretation**
Function Symbols Generating Random Values		
`random`	$\mathbb{S}[i]$	Function generating a random bitstring in $\{0,1\}^i$.
Function Symbols for Manipulating Bitstrings and Boolean Values		
`not`	$\mathbb{B} \to \mathbb{B}$	Function computing the negation of a boolean value.

Function Symbol	Type	Interpretation
`xor`	$\mathbb{S}[i] \to \mathbb{S}[i]$	Function computing the bitwise exclusive-or of two binary strings with the same length.
`zero`	$\mathbb{S}[i]$	Function that returns a bitstring consisting solely of zeros.
Function Symbols for Ledger Manipulation		
$\texttt{initTable}_p$	$\mathbb{S}[p \times (2i+1)]$	Function that initializes a ledger by generating a string of length $p \times (2i+1)$ filled entirely with zeros.
$\texttt{isdefined}_\texttt{p}$	$\mathbb{S}[p \times (2i+1)] \times \mathbb{S}[i] \to \mathbb{B}$	Function that takes as input a ledger and a string of length i, and returns true if the row, whose first i bits match the input string, is active and false otherwise.
$\texttt{getValue}_\texttt{p}$	$\mathbb{S}[p \times (2i+1)] \times \mathbb{S}[i] \to \mathbb{S}[i]$	Function that, given a ledger and an input string of length i, searches for a row in the ledger whose first i bits are equal to the input string and whose final bit is set to 1 (indicating that the row is active), and returns the substring spanning from position $i+1$ to position $2i$ within the selected row. If no matching active row is found, it returns a dummy value.
$\texttt{modify}_p$	$\mathbb{S}[p \times (2i+1)] \times \mathbb{S}[i] \times \mathbb{S}[i] \to \mathbb{S}[p \times (2i+1)]$	Function that, given a ledger and two input strings of length i, identifies the first inactive row in the ledger, inserts into that row the concatenation of the two strings followed by a bit set to 1 (indicating activation), and returns the resulting updated ledger.

Table A.1: Table of function symbols and their interpretations as presented in [LGG24].

The equations presented in Figure 27 establish a set of equivalences

between λ**BLL** terms, which are sound with respect to the notion of Θ-contextual indistinguishability introduced in [LGG24] and denoted here by $\sim_{\Theta\subseteq\Xi}$. These equations characterize behavioral equivalence for terms that remain indistinguishable to any probabilistic polynomial-time context, restricted by a specific set of observable references.

$$\mathtt{let}\, x = M\, \mathtt{in}\, \mathtt{return}\, x \sim_{\Theta\subseteq\Xi} \mathtt{return}\, M \qquad (\mathtt{reduc1})$$

$$(\mathtt{return}\, \lambda x.M) Z \sim_{\Theta\subseteq\Xi} M[Z/x] \qquad (\mathtt{reduc2})$$

$$\mathtt{der}(!M) \sim_{\Theta\subseteq\Xi} M \qquad (\mathtt{reduc3})$$

$$\mathtt{let}\, x = \mathtt{return}\, V\, \mathtt{in}\, M \sim_{\Theta\subseteq\Xi} M[V/x] \qquad (\mathtt{reduc4})$$

$$\mathtt{let}\, \langle x, y\rangle = \langle Z, Z'\rangle\, \mathtt{in}\, M \sim_{\Theta\subseteq\Xi} M[Z/x, Z'/y] \qquad (\mathtt{reduc5})$$

$$\begin{pmatrix} \mathtt{let}\, r = \mathtt{random}\, \mathtt{in} \\ \mathtt{let}\, z = \mathtt{isdefined}(y, r)\, \mathtt{in} \\ \mathtt{if}\, z\, \mathtt{then}\, M\, \mathtt{else}\, N \end{pmatrix} \sim_{\Theta\subseteq\Xi} \mathtt{let}\, r = \mathtt{random}\, \mathtt{in}\, N \qquad (\mathtt{randT})$$

$$\mathtt{let}\, x = M_1\, \mathtt{in}\, \mathtt{let}\, y = M_2\, \mathtt{in}\, N \sim_{\Theta\subseteq\Xi} \mathtt{let}\, y = (\mathtt{let}\, x = M_1\, \mathtt{in}\, M_2)\, \mathtt{in}\, N \qquad (\mathtt{letAss})$$
$$\text{if } x \notin FV(N)$$

$$\begin{pmatrix} \mathtt{if}\, b\, \mathtt{then}\, \mathtt{let}\, x = M\, \mathtt{in}\, N \\ \mathtt{else}\, \mathtt{let}\, x = L\, \mathtt{in}\, N \end{pmatrix} \sim_{\Theta\subseteq\Xi} \mathtt{let}\, x = (\mathtt{if}\, b\, \mathtt{then}\, M\, \mathtt{else}\, L\, \mathtt{in}\, N) \qquad (\mathtt{iflet})$$

$$\mathtt{let}\, x_1 = M_1\, \mathtt{in}\, \mathtt{let}\, x_2 = M_2\, \mathtt{in}\, N \sim_{\Theta\subseteq\Xi} \mathtt{let}\, x_2 = M_2\, \mathtt{in}\, \mathtt{let}\, x_1 = M_1\, \mathtt{in}\, N \qquad (\mathtt{letCom})$$

$$\mathtt{let}\, x = M\, \mathtt{in}\, N \sim_{\Theta\subseteq\Xi} N \qquad (\mathtt{safeDisc1})$$
$$\text{if } x \notin FV(N) \text{ and } \mathcal{R}(M) \cap \mathcal{R}(N) = \emptyset \text{ and}$$
$$\mathcal{R}(M) \cap \Theta = \emptyset \text{ and} HOFV(M) = \emptyset$$

$$\mathtt{let}\, \langle x, y\rangle = M\, \mathtt{in}\, N \sim_{\Theta\subseteq\Xi} N \qquad (\mathtt{safeDisc2})$$
$$\text{if } x, y \notin FV(N) \text{ and } \mathcal{R}(M) \cap \mathcal{R}(N) = \emptyset \text{ and}$$
$$\mathcal{R}(M) \cap \Theta = \emptyset \text{ and} HOFV(M) = \emptyset$$

Figure 27: Equations in λ**BLL** as presented in [LGG24]. Let $\mathcal{R}(M)$ be the set of references accessed by the term M. The equations in this figure hold for all reference contexts Θ, Ξ such that $\Theta \subseteq \Xi$, except where stated otherwise.