Automated Security Proofs for Almost-Universal Hash for MAC verification*

Martin Gagné[1], Pascal Lafourcade[2], and Yassine Lakhnech[2]

[1] Department of Computer Science, Saarland University, Germany
[2] Université Grenoble 1, CNRS, VERIMAG, France

Abstract. Message authentication codes (MACs) are an essential primitive in cryptography. They are used to ensure the integrity and authenticity of a message, and can also be used as a building block for larger schemes, such as chosen-ciphertext secure encryption, or identity-based encryption. MACs are often built in two steps: first, the 'front end' of the MAC produces a short digest of the long message, then the 'back end' provides a mixing step to make the output of the MAC unpredictable for an attacker. Our verification method follows this structure. We develop a Hoare logic for proving that the front end of the MAC is an almost-universal hash function. The programming language used to specify these functions is fairly expressive and can be used to describe many block-cipher and compression function-based MACs. We implemented this method into a prototype that can automatically prove the security of almost-universal hash functions. This prototype can prove the security of the front-end of many CBC-based MACs (DMAC, ECBC, FCBC and XCBC to name only a few), PMAC and HMAC. We then provide a list of options for the back end of the MAC, each consisting of only two or three instructions, each of which can be composed with an almost-universal hash function to obtain a secure MAC.

1 Introduction

Message authentication codes (MACs) are among the most common primitives in symmetric key cryptography. They ensure the integrity and provenance of a message, and they can be used, in conjunction with chosen-plaintext (CPA) secure encryption, to obtain chosen-ciphertext (CCA) secure encryption. Given the importance of this primitive, it is important that their proofs of security be the object of close scrutiny. The study of the security of MACs is, of course, not a new field. Bellare et al. [5] were the first to prove the security of CBC-MAC for fixed-length inputs. Following this work, a myriad of new MACs secure for variable-length inputs were proposed ([4, 7–9, 17]). None of these protocols' proofs have been verified by any means other than human scrutiny. Automated proofs can provide additional assurance of the correctness of these security proofs by providing an independent proof of complex schemes. This paper presents a method for automatically proving the security of MACs based on block ciphers and hash functions.

Contributions: To analyze the security of MACs, we first decompose the MAC algorithms into two parts: a 'front-end', whose work is to compress long input messages

* This work was partially supported by ANR project ProSe and Minalogic project SHIVA.

into small digests, and a 'back-end', usually a mixing step, which obfuscates the output of the front-end. We present a Hoare logic to prove that the front-ends of block-cipher based and hash based MACs are almost-universal hash functions in the ideal cipher model and random oracle model respectively. We then make a list of operations which, when composed with an almost-universal hash function, yield a secure MAC. We can then attest the security of MACs by first proving the security of the front end using our logic, and then by manually verifying that the back end of the MAC belongs to our list.

Our result differs significantly from previous works that used Hoare logic to generate proofs of cryptographic protocols (such as [12, 15]) because those results proved the security of encryption schemes. Proving the security of MACs proved to be singularly more challenging: the security of encryption schemes could be simply proven by showing that the ciphertext is indistinguishable from a random value, whereas the unforgeability property required of MACs cannot, to our knowledge, be captured by their predicates. As a result, we have to consider the simultaneous execution of the program, define a dedicated semantics to capture these executions, and introduce appropriated predicates that keep track of equality and inequality of values between the two executions.

In contrast to the previous results that only deal with schemes that had fixed-length inputs, we are able to analyze for-loops, which allows us to prove the security of protocols that can take arbitrary strings as an input. We describe two heuristics that can be used to discover stable loop invariants and apply them to one example. These heuristics successfully find stable invariants for all the hash functions analyzed in this paper.

Finally, we implemented our method into a prototype [14] that can be used to verify the security of the front-end of several well-known MACs, such as HMAC [4], DMAC [17], ECBC, FCBC and XCBC [8] and PMAC [9], and could be used to verify the security of other hash functions based on the same primitives. We also give a predicate filter that enables us to discard unnecessary predicates, which speeds up our implementation and facilitates the discovery of loop invariants. Our prototype goes through the programs from beginning to end, instead of the more common backward approach, to avoid an exponential blowout in the number of possibilities to examine, due to the many choices of rules that can cause certain predicates caused by the presence of the logical or connector in our Hoare logic.

Related Work: The idea of using Hoare logic to automatically produce proofs of security for cryptographic protocols is not new. Courant et al. [12] presented a Hoare logic to prove the security of asymmetric encryption schemes in the random oracle model. A Hoare logic was also used by Gagné et al. [15] to verify proofs of security of block cipher modes of encryption. Also worth mentioning is the paper by Corin and Den Hartog [11], which presented a Hoare-style proof system for game-based cryptographic proofs.

Fournet et al. [13] developed a framework for modular code-based cryptographic verification. However, their approach considers interfaces for MACs. In a way, our work is complementary to theirs, as our result, coupled with theirs, could enable a more complete verification of systems.

In [1], the authors introduce a general logic for proving the security of cryptographic primitives. This framework can easily be extended using external results, such as [12], to add to its power. Our result could also be added to this framework to further extend it.

Other tools, such as Cryptoverif [10] and EasyCrypt [3, 2], can be used to verify the security of cryptographic schemes, but they are far less convenient than our method for proving the security of MACs. Cryptoverif does not support loop constructs, which are an important part of our result, and is generally used for proving the security of higher level protocols, assuming the security of primitives such as MACs. As for Easycrypt, it relies on a game-based approach and requires human assistance to enter the sequence of games. Our result is complementary to these approaches: we offer a method for proving the security of MACs that are assumed to be secure low level primitive in such tools. Combining our results would enable a more complete analysis of cryptographic protocols. Moreover, our method requires only the description of the program as input, and automatically outputs a proof, removing the need for human assistance.

Outline: In Section 2, we introduce cryptographic background. The following section introduces our grammar, semantics and assertion language. In Section 4, we present our Hoare logic and method for proving the security of almost-universal hash functions, and we discuss our implementation of this logic and treatment of loops in Section 5. We then obtain a secure MAC by combining these with one of the back-end options described in Section 6. Finally, we conclude in Section 7.

2 Cryptographic Background

In this section, we introduce a few notational conventions, and we recall a few cryptographic concepts.

Notation and Conventions

We assume that all variables range over domains whose cardinality is exponential in the security parameter η and that all programs have length polynomial in η. We say that a function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if, for any polynomial p, there exists a positive integer n_0 such that for all $n \geq n_0$, $f(n) \leq \frac{1}{|p(n)|}$.

For a probability distribution $\mathcal{D}$, we denote by $x \xleftarrow{\$} \mathcal{D}$ the operation of sampling a value x according to distribution $\mathcal{D}$. If S is a finite set, we denote by $x \xleftarrow{\$} S$ the operation of sampling x uniformly at random among the values in S.

MAC Security

A message authentication code ensures the authenticity of a message m by computing a small tag τ, which is sent together with the message to the intended receiver. Upon receiving the message and the tag, the receiver recomputes the tag τ' using the message and his own copy of the key, and he accepts the message as authentic if $\tau = \tau'$. More formally:

Definition 1 (MAC). *A* message authentication code *is a triple of polynomial-time algorithms* (K, MAC, V)*, where* $K(1^\eta)$ *takes a security parameter* 1^η *and outputs a secret key* sk*,* $MAC(sk, m)$ *takes a secret key and a message* m*, and outputs a* tag*, and* $V(sk, m, tag)$ *takes a secret key* sk*, a message* m *and a tag, and outputs a bit: 1 for a correct tag, 0 otherwise.*

We say that a MAC is secure, or *unforgeable* if it is impossible to compute a new valid message-tag pair for anybody who does not know the secret key, even when given access to oracles that can compute and verify the MACs. This way, when one receives a valid message-tag pair, he can be certain that the message was sent by someone who possesses a copy of his secret key.

Definition 2 (Unforgeability [5]). *A MAC* (K, Mac, V) *is* unforgeable under a chosen-message attack *(UNF-CMA) if for every polynomial-time algorithm* $\mathcal{A}$ *that has oracle access to the MAC and verification algorithm and whose output message* m^* *is different from any message it sent to the* Mac *oracle, the following probability is negligible*

$$\mathsf{Pr}[sk \xleftarrow{\$} K(1^\eta); (m^*, tag^*) \xleftarrow{\$} \mathcal{A}^{Mac(sk,\cdot),V(sk,\cdot,\cdot)} : V(sk, m^*, tag^*) = 1]$$

A standard method for constructing MACs is to apply a pseudo-random function, or some other form of 'mixing' step, to the output of an almost-universal hash function [18, 19]. We assume that a MAC is constructed in this way.

Definition 3 (Almost-Universal Hash). *A family of functions* $\mathcal{H} = \{h_i\}$ *indexed with key* $i \in \{0,1\}^\eta$ *is a* family of almost-universal hash functions *if for any two distinct strings* M *and* M', $\mathsf{Pr}_{h_i \in \mathcal{H}}[h_i(M) = h_i(M')]$ *is negligible, where the probability is taken over the choice of* h_i *in* $\mathcal{H}$.

It is much easier to work with this definition than with the unforgeability definition because of the absence of an adaptive adversary, and the collision probability is taken over all possible choices of key.

Block Cipher Security

Many MAC constructions are based on block cipher, so we quickly recall the definition of block ciphers and their security definition.

A block cipher is a *family of permutations* $\mathcal{E} : \{0,1\}^{K(\eta)} \times \{0,1\}^\eta \to \{0,1\}^\eta$ indexed with a key $k \in \{0,1\}^{K(\eta)}$ where $K(\eta)$ is a polynomial. A block cipher is *secure* if, for a randomly sampled key, the block cipher is indistinguishable from a permutation sampled at random from the set of all permutations of $\{0,1\}^\eta$. However, since random permutations of $\{0,1\}^\eta$ and random functions from $\{0,1\}^\eta$ to $\{0,1\}^\eta$ are statistically close, and that random functions are often more convenient for proof purposes, it is common to assume that secure block ciphers are pseudo-random functions.

Definition 4 (Pseudo-Random Functions). *Let* $P : \{0,1\}^{K(\eta)} \times \{0,1\}^\eta \to \{0,1\}^\eta$ *be a family of functions and let* $\mathcal{A}$ *be an algorithm that takes an oracle and returns a bit. The* prf-advantage *of* $\mathcal{A}$ *is defined as follows.*

$$\mathsf{Adv}^{prf}_{\mathcal{A},P} = \left| \mathsf{Pr}[k \xleftarrow{\$} \{0,1\}^{K(\eta)}; \mathcal{A}^{P(k,\cdot)} = 1] - \mathsf{Pr}[R \xleftarrow{\$} \Phi_\eta; \mathcal{A}^{R(\cdot)} = 1] \right|$$

where Φ_η *is the set of all functions from* $\{0,1\}^\eta$ *to* $\{0,1\}^\eta$. *We say that* P *is a family of pseudo-random functions if for every polynomial-time adversary* $\mathcal{A}$, $\mathsf{Adv}^{prf}_{\mathcal{A},P}$ *is a negligible function in* η.

Since all the schemes in this paper require only one key for the block cipher, to simplify the notation, we write only $\mathcal{E}(m)$ instead of $\mathcal{E}(k, m)$, but it is understood that a key was selected at the initialization of the scheme, and remains the same throughout.

Random Oracle Model

For MACs that make use of a hash function, we assume that the hash function behaves like a random oracle. That is, we assume that the hash function is picked at random among all possible functions from the given domain and range, and that every algorithm participating in the scheme, including all adversaries, has oracle access to this random function. This is a fairly common assumption to provide a heuristic argument for the security of cryptographic protocols [6].

Indistinguishable Distributions

Given two distribution ensembles $X = \{X_\eta\}_{\eta \in \mathbb{N}}$ and $X' = \{X'_\eta\}_{\eta \in \mathbb{N}}$, an algorithm $\mathcal{A}$ and $\eta \in \mathbb{N}$, we define the *advantage* of $\mathcal{A}$ in distinguishing X_η from X'_η as the following quantity:

$$\mathsf{Adv}(\mathcal{A}, \eta, X, X') = \left| \mathsf{Pr}[x \xleftarrow{\$} X_\eta : \mathcal{A}(x) = 1] - \mathsf{Pr}[x \xleftarrow{\$} X'_\eta : \mathcal{A}(x) = 1] \right| .$$

We say that X and X' are *indistinguishable*, denoted by $X \sim X'$, if $\mathsf{Adv}(\mathcal{A}, \eta, X, X')$ is negligible as a function of η for every probabilistic polynomial-time algorithm $\mathcal{A}$.

3 Model

In this section, we introduce the grammar for the programs describing almost-universal hash function. We present the semantics of each commands, and introduce the assertion language that will be used in for our Hoare logic.

3.1 Grammar

We consider the language defined by the BNF grammar below, where p and q are positive integers.

$$\begin{aligned}\mathsf{cmd} ::= {} & x := \mathcal{E}(y) \mid x := \mathcal{H}(y) \mid x := y \mid x := y \oplus z \mid x := y \| z \mid x := \rho(i, y) \\ & \mid \mathsf{for}\ l = p\ \mathsf{to}\ q\ \mathsf{do}\text{: } [\mathsf{cmd}_l] \mid \mathsf{cmd}_1; \mathsf{cmd}_2\end{aligned}$$

We refer to individual instructions as *commands* and to lists of commands as *programs*. Each command has the following effect:

- $x := \mathcal{E}(y)$ denotes application of the block cipher $\mathcal{E}$ to the value of y and assigning the result to x.
- $x := \mathcal{H}(y)$ denotes the application of the hash function $\mathcal{H}$ to the value of y and assigning the result to x.
- $x := y$ denotes the assignment to x of the values of y.
- $x := y \oplus z$ denotes the assignment to x of the xor or the values of y and z.
- $x := y||z$ denotes the assignment to x of the concatenation of the values of y and z.
- $x := \rho(i, y)$ denotes the computation of the function ρ on input i (an integer) and the value of y and assigning the result to x.
- $c_1; c_2$ is the sequential composition of c_1 and c_2.

– for $l = p$ to q do: $[\mathsf{cmd}_l]$ denotes the successive execution of $\mathsf{cmd}_p; \mathsf{cmd}_{p+1}; \dots;$ cmd_q when $p \leq q$. If $p > q$, the command has no effect.

The function ρ is used to process the *tweak* in a common construction for *tweakable block ciphers* [16]. A fixed-input-length almost-universal function is often sufficient, but exact implementations vary from one scheme to the next, and we want to allow for the possibility of functions that have additional properties. When a scheme uses a function ρ, the properties of the function ρ required for the proof will be added to the initial conditions of the verification procedure using the predicates of Section 3.3. We do not any other assumptions about ρ other than it is a function with fixed output length.

Definition 5 (Generic Hash Function). *A generic hash function $Hash$ on message blocks $m_1, \dots, m_n$ with output c_n, is represented by a tuple $(\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, Hash(m_1\| \dots \|m_n, c_n)$: **var** $\boldsymbol{x}$; cmd), where $\mathcal{F}_\mathcal{E}$ is a family of pseudorandom permutations (usually a block cipher), $\mathcal{F}_\mathcal{H}$ is a family of cryptographic hash functions, and $Hash(m_1\| \dots \|m_n, c_n)$: **var** $\boldsymbol{x}$; cmd is the program of the hash function, where $\boldsymbol{x}$ is the set of all the variables in the program that are neither input variables m_i, output variable c_n, or the special variable k (used to hold a secret key), and the program cmd is in the language described by our grammar.*

The secret key sk of the generic hash is a combination of the value of the special variable k and the choice of the block cipher $\mathcal{E}$ in the family $\mathcal{F}_\mathcal{E}$.

We assume that, prior to executing the MAC, the message has been padded using some unambiguous padding scheme, so that all the message blocks $m_1, \dots, m_n$ are of equal and appropriate length for the scheme, usually the input length of the block cipher. We also assume that each variable in the program cmd is assigned at most once, as it is clear that any program obtained from our language can be transformed into an equivalent program with this property, and that the input variables $m_1, \dots, m_n$ never appear on the left side of any command since these variables already hold a value before the execution of the program. For simplicity of exposition, we henceforth assume that all the programs in this paper satisfy these assumptions.

We present to the right the program for $Hash_{CBC}$, the hash function that is used as a running example in this paper. We give the program for other hash functions that can be verified with our method in in Appendix A.

$Hash_{CBC}(m_1\| \dots \|m_n, c_n)$:
var $i, z_2, \dots, z_n, c_1, \dots, c_{n-1}$;
$c_1 := \mathcal{E}(m_1)$;
for $i = 2$ to n do:
  $[z_i := c_{i-1} \oplus m_i; c_i := \mathcal{E}(z_i)]$

3.2 Semantics

In our analysis, we consider the execution of a program on two inputs simultaneously. These simultaneous executions will enable us to keep track of the probability of equality and inequality of strings between the two executions, thereby allowing us to prove that the function is almost-universal.

Each command is a function that takes a configuration and outputs a configurations. A *configuration* γ is a tuple $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H})$ where S and S' are states, $\mathcal{E}$ is a block cipher, $\mathcal{H}$ is a hash function (that will be modeled as a random oracle), and $\mathcal{L}_\mathcal{E}$ and $\mathcal{L}_\mathcal{H}$ are sets of strings.

A *state* is a function $S : \mathsf{Var} \to \{0,1\}^* \cup \bot$, where Var is the full set of variables in the program, that assigns bitstrings to variables (the symbol $\bot$ is used to indicate that no value has been assigned to the variable yet). A configuration contains two states, one for each execution of the program.

The set $\mathcal{L}_\mathcal{E}$ records the values for which the functions $\mathcal{E}$ was computed. The set is common for both executions of the program. Every time a command of the type $x := \mathcal{E}(y)$ is executed in the program, we add $S(y)$ and $S'(y)$ to $\mathcal{L}_\mathcal{E}$ if they are not already present. We define $\mathcal{L}_\mathcal{H}$ for the hash function $\mathcal{H}$ similarly.

Let Γ denote the set of configurations and $\mathrm{DIST}(\Gamma)$ the set of distributions on configurations. The semantics is given below, where $S\{x \mapsto v\}$ denotes the state which assigns the value v to the variable x, and behaves like S for all other variables and $\circ$ denotes function composition. The semantic function $\mathsf{cmd} : \Gamma \to \Gamma$ of commands can be lifted in the usual way to a function $\mathsf{cmd}^* : \mathrm{DIST}(\Gamma) \to \mathrm{DIST}(\Gamma)$ by point-wise application of cmd. By abuse of notation we also denote the lifted semantics by $[\![\mathsf{cmd}]\!]$.

$$
\begin{array}{l}
[\![x := \mathcal{E}(y)]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad (S\{x \mapsto \mathcal{E}(S(y))\}, S'\{x \mapsto \mathcal{E}(S'(y))\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E} \cup \{S(y), S'(y)\}, \mathcal{L}_\mathcal{H}) \\
[\![x := \mathcal{H}(y)]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad (S\{x \mapsto \mathcal{H}(S(y))\}, S'\{x \mapsto \mathcal{H}(S'(y))\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H} \cup \{S(y), S'(y)\}) \\
[\![x := y]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = (S\{x \mapsto S(y)\}, S'\{x \mapsto S'(y)\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \\
[\![x := y \oplus z]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad (S\{x \mapsto S(y) \oplus S(z)\}, S'\{x \mapsto S'(y) \oplus S'(z)\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \\
[\![x := y||z]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad (S\{x \mapsto S(y)||S(z)\}, S'\{x \mapsto S'(y)||S'(z)\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \\
[\![x := \rho(i, y)]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad (S\{x \mapsto \rho(i, S(y))\}, S'\{x \mapsto \rho(i, S'(y))\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \\
[\![\mathsf{for}\ l = p\ \mathsf{to}\ q\ \mathsf{do}\text{: } [\mathsf{cmd}_l]]\!]\gamma = \begin{cases} [\![\mathsf{cmd}_q]\!] \circ [\![\mathsf{cmd}_{q-1}]\!] \circ \ldots \circ [\![\mathsf{cmd}_p]\!]\gamma \text{ if } p \leq q \\ \gamma \text{ otherwise} \end{cases} \\
[\![c_1; c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]
\end{array}
$$

The set of initial distributions $\mathrm{DIST}_0(\mathbb{H})$, where $\mathbb{H} = (\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, Hash(m_1\| \ldots \|m_n, c_n) : \mathbf{var}\ \boldsymbol{x}; \mathsf{cmd})$ is a generic hash, contains all the following distributions:

$$
\begin{array}{l}
\mathcal{D}_0^{(M,M')} = [\mathcal{E} \xleftarrow{\$} \mathcal{F}_\mathcal{E}(1^\eta); \mathcal{H} \xleftarrow{\$} \mathcal{F}_\mathcal{H}(1^\eta); u \xleftarrow{\$} \{0,1\}^\eta : \\
\quad (S\{k \mapsto u, m_1\| \ldots \|m_n \mapsto M\}, S'\{k \mapsto u, m_1\| \ldots \|m_n \mapsto M'\}, \mathcal{E}, \mathcal{H}, \emptyset, \emptyset)]
\end{array}
$$

where M and M' are any two n block messages and k is a variable holding a secret string needed in some MACs (among our examples, $Hash_{PMAC}$ and $Hash_{HMAC}$ need it). Note that $\mathcal{F}_\mathcal{E}$, $\mathcal{F}_\mathcal{H}$, the domain Var of the states and the length n of the input messages are defined in $\mathbb{H}$. These distributions capture the initial situation of Definition 3 where the variables m_i contain the blocks of M and M' in S and S' respectively.

The set $\mathrm{DIST}(\mathbb{H})$ is obtained by executing a program on one of the initial distributions. It contains all the distributions of the form $[\![\mathsf{cmd}]\!]X_0$, where $X_0 \in \mathrm{DIST}_0(\mathbb{H})$ and cmd is a program.

A notational convention. It is easy to see that commands never modify $\mathcal{E}$ or $\mathcal{H}$. Therefore, we can, without ambiguity, write $(\hat{S}, \hat{S}', \mathcal{L}'_\mathcal{E}, \mathcal{L}'_\mathcal{H}) \xleftarrow{\$} [\![c]\!](S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H})$ instead of $(\hat{S}, \hat{S}', \mathcal{E}, \mathcal{H}, \mathcal{L}'_\mathcal{E}, \mathcal{L}'_\mathcal{H}) \xleftarrow{\$} [\![c]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H})$.

3.3 Assertion Language

Like [15], our assertion languages deals with block ciphers, so it stands to reason that some of our predicates will be similar to theirs. However, the definition of all the predicates has to be adapted to our new semantics with two simultaneous executions. We also need additional predicates to describe equality or inequality of strings between the two executions, that will allow us to capture the definition of almost-universal hash functions. We first give an intuitive description of our predicates, then we define them all formally.

Empty**:** means that the probability that $\mathcal{L}_\mathcal{E}$ contains an element is negligible.
$\mathsf{Eq}(x, y)$**:** means that the probability that $S(x) \neq S'(y)$ is negligible.
$\mathsf{Uneq}(x, y)$**:** means that the probability that $S(x) = S'(y)$ is negligible.
$\mathsf{E}(\mathcal{E}; x; V)$**:** means that the probability that the value of x is either in $\mathcal{L}_\mathcal{E}$ or equal to that of a variable in V is negligible.
$\mathsf{H}(\mathcal{H}; x; V)$**:** means that the probability that the value of x is either in $\mathcal{L}_\mathcal{H}$ or equal to that of a variable in V is negligible.
$\mathsf{Ind}(x; V; V')$**:** means that no adversary has non-negligible probability to distinguish whether he is given results of computations performed using the value of x or a random value, when he is given the values of the variables in V and the values of the variables in V' from the parallel execution. In addition to variables in Var, the set V can contain special symbols $\ell_\mathcal{E}$ or $\ell_\mathcal{H}$. When the symbol $\ell_\mathcal{E}$ is present, it means that, in addition to the other variables in V, the distinguisher is also given the values in $\mathcal{L}_\mathcal{E}$, similarly for $\ell_\mathcal{H}$.

Our Hoare logic is based on statements from the following language.

$$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \psi$$
$$\psi ::= \mathsf{Ind}(x; W; V') \mid \mathsf{Eq}(x, y) \mid \mathsf{Uneq}(x, y) \mid \mathsf{Empty} \mid \mathsf{E}(\mathcal{E}; x; V) \mid \mathsf{H}(\mathcal{H}; x; V)$$

where $x, y \in \mathsf{Var}$ and $V, V' \subseteq \mathsf{Var}$, and $W \subseteq \mathsf{Var} \cup \{\ell_\mathcal{E}, \ell_\mathcal{H}\}$. We refer to the statements produced by this grammar as *formulas*.

We introduce a few notational shortcuts that will help in formally defining our predicates. For any set $V \subseteq \mathsf{Var}$, we denote by $S(V)$ the multiset resulting from the application of S on each variable in V. Also, for a set $W \subseteq \mathsf{Var} \cup \{\ell_\mathcal{E}\}$ with $\ell_\mathcal{E} \in W$, we use $S(W)$ as a shorthand for $S(W \setminus \{\ell_\mathcal{E}\}) \cup \mathcal{L}_\mathcal{E}$, and similarly for $\ell_\mathcal{H}$. For a set $V \subseteq \mathsf{Var} \cup \{\ell_\mathcal{E}, \ell_\mathcal{H}\}$ and an element $x \in \mathsf{Var} \cup \{\ell_\mathcal{E}, \ell_\mathcal{H}\}$, we write V, x as a shorthand for $V \cup \{x\}$ and $V - x$ as a shorthand for $V \setminus \{x\}$.

We define that a distribution X satisfies φ, denoted $X \models \varphi$ as follows:

- $X \models \varphi \wedge \varphi'$ iff $X \models \varphi$ and $X \models \varphi'$
- $X \models \varphi \vee \varphi'$ iff $X \models \varphi$ or $X \models \varphi'$
- $X \models \mathsf{Empty}$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X : \mathcal{L}_\mathcal{E} \neq \emptyset]$ is negligible
- $X \models \mathsf{Eq}(x, y)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X : S(x) \neq S'(y)]$ is negligible
- $X \models \mathsf{Uneq}(x, y)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X : S(x) = S'(y)]$ is negligible

– $X \models \mathsf{E}(\mathcal{E}; x; V)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : \{S(x), S'(x)\} \cap (\mathcal{L}_\mathcal{E} \cup S(V - x) \cup S'(V - x)) \neq \emptyset]$ is negligible[1]
– $X \models \mathsf{H}(\mathcal{H}; x; V)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : \{S(x), S'(x)\} \cap (\mathcal{L}_\mathcal{H} \cup S(V - x) \cup S'(V - x)) \neq \emptyset]$ is negligible
– $X \models \mathsf{Ind}(x; V; V')$ iff the two following formulas hold:

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(x), S(V - x) \cup S'(V'))] \sim [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X; u \xleftarrow{\$} \mathcal{U} : (u, S(V - x) \cup S'(V'))]$$

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S'(x), S'(V - x) \cup S(V'))] \sim [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X; u \xleftarrow{\$} \mathcal{U} : (u, S'(V - x) \cup S(V'))]$$

We now present a few lemmas that show useful relations and properties of our predicates. In all these lemmas, it is assumed that $\mathbb{H}$ is any generic hash. The proof of these lemmas is in Appendix B.

Lemma 1. *The following relations are true for any sets V_1, V_2, V_3, V_4 and variables x, y with $x \neq y$*

1. $\mathsf{Ind}(x; V_1; V_2) \Rightarrow \mathsf{Ind}(x; V_3; V_4)$ *if* $V_3 \subseteq V_1$ *and* $V_4 \subseteq V_2$
2. $\mathsf{H}(\mathcal{H}; x; V_1) \Rightarrow \mathsf{H}(\mathcal{H}; x; V_2)$ *if* $V_2 \subseteq V_1$
3. $\mathsf{E}(\mathcal{E}; x; V_1) \Rightarrow \mathsf{E}(\mathcal{E}; x; V_2)$ *if* $V_2 \subseteq V_1$
4. $\mathsf{Ind}(x; V_1, \ell_\mathcal{H}; \emptyset) \Rightarrow \mathsf{H}(\mathcal{H}; x; V_1)$
5. $\mathsf{Ind}(x; V_1, \ell_\mathcal{E}; \emptyset) \Rightarrow \mathsf{E}(\mathcal{E}; x; V_1)$
6. $\mathsf{Ind}(x; \emptyset; \{y\}) \Rightarrow \mathsf{Uneq}(x, y) \wedge \mathsf{Uneq}(y, x)$

Note that lines 4, 5 and 6 are particularly helpful because the predicate Ind is much easier to propagate than the other predicates.

We also show that, as a consequence of our definition of $\textsc{Dist}(\mathbb{H})$, we can always infer the following predicates on the message blocks. This lemma is useful for proving the rules corresponding to commands that introduce a new message block.

Lemma 2. *Let $X \in \textsc{Dist}(\mathbb{H})$. Then for any integer i, $1 \leq i \leq n$, $X \models \mathsf{Eq}(m_i, m_i) \vee \mathsf{Uneq}(m_i, m_i)$.*

The following formalizes the intuition that if a value can be computed in polynomial time from other values available, then adding this value does not give the adversary any useful information. In general, we say that an expression e is *constructible* from values in a set V if e can be computed in polynomial time from V. But for the purpose of the following lemma, it is sufficient to define constructible expressions as only single variables x, as well as $x \oplus y$ and $x \| y$ for any variables x and y.

[1] Since the variable x is removed from the set V when taking the probability, we always have $X \models \mathsf{E}(\mathcal{E}; x; V)$ iff $X \models \mathsf{E}(\mathcal{E}; x; V, x)$. This is to remove the trivial case that $\{S(x), S'(x)\} \cap (\mathcal{L}_\mathcal{E} \cup \{S(x), S'(x)\}) = \emptyset$ never holds, and to simplify the notation. The same is also used for predicates $\mathsf{H}(\mathcal{H}; x; V)$ and $\mathsf{Ind}(x; V; V')$.

Lemma 3. *For any any $X \in \text{DIST}(\mathbb{H})$, any sets of variables V, any expression e constructible from V, and any variable x, z such that $z \notin \{x\} \cup \textsf{Var}(e)$ if $X \models \textsf{Ind}(z; V; V')$ then $[\![x := e]\!](X) \models \textsf{Ind}(z; V, x; V')$. We emphasize that here we use the notation $\textsf{Var}(e)$ (in its usual sense), that is to say, the variable z does not appear* at all *in e.*
Similarly, if $X \models \textsf{Ind}(z; V'; V)$, then $[\![x := e]\!](X) \models \textsf{Ind}(z; V'; V, x)$.

The following, which is useful for proving some of the rules dealing with the concatenation commands, shows that the value of any given variable always have the same length in each execution.

Lemma 4. *For any distribution $X \in \text{DIST}(\mathbb{H})$, any program $\textsf{cmd}$ produced by our grammar any $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![\textsf{cmd}]\!]X$ and any variable $v \in \textsf{Var}$, $|S(v)| = |S'(v)|$.*

4 Proving Almost-Universal Hash

Our main contribution is a Hoare logic for proving that a program is an almost-universal hash function. We require that the program be written in a way so that, on input $m_1\| \dots \|m_n$, the program must assign values to variables $c_1, \dots, c_n$ in such a way that the variable c_1 contains the output of the function on input m_1, the variable c_2 contains the output of the function on input $m_1\|m_2$ and so on. We model the security of an almost-universal hash function using our predicates as follows.

Proposition 1. *Let $\mathbb{H} = (\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, Hash(m_1\| \dots \|m_n, c_n) : \textbf{var}\ \boldsymbol{x}; \textsf{cmd})$ be a generic hash function on n-block messages. Then, $\mathbb{H}$ is an almost-universal hash function if, for every positive integer n, $UNIV(n)$ holds in the distribution obtained by executing the program on any distribution in $\text{DIST}_0(\mathbb{H})$, where*

$$UNIV(n) = \left(\bigwedge_{i=1}^{n-1} \textsf{Uneq}(c_n, c_i) \wedge \bigwedge_{i=1}^{n} \textsf{Eq}(m_i, m_i)\right) \vee \bigwedge_{i=1}^{n} \textsf{Uneq}(c_n, c_i)$$

The proof of this proposition is in Appendix B.

Hoare Logic Rules

We present a set of rules of the form $\{\varphi\}\textsf{cmd}\{\varphi'\}$, meaning that execution of command $\textsf{cmd}$ in any distribution in $\text{DIST}(\mathbb{H})$ that satisfies φ leads to a distribution that satisfies φ'. Using Hoare logic terminology, this means that the triple $\{\varphi\}\textsf{cmd}\{\varphi'\}$ is valid.

Since the predicates $\textsf{Eq}(m_i, m_i)$ are useful only if the whole prefix of the two messages up to the i^{th} block are equal, so that keeping track of the equality or inequality of the message blocks after the first point at which the messages are different is unnecessary. For this reason, when we design our rules, we never produce the predicates $\textsf{Uneq}(m_i, m_i)$ even when they would be correct.

We group rules together according to their corresponding commands. In all the rules, unless indicated otherwise, we assume that $t \notin \{x, y, z\}$ and $x \notin \{y, z\}$. . In addition, for all rules involving the predicate $\textsf{Ind}$, we assume that $\ell_\mathcal{E}$ and $\ell_\mathcal{H}$ can be among the elements in the set V. Since some of the rules (for example, rule (G5)) are valid only under certain slightly complex conditions, we use square brackets in the statement of some conditions to remove any ambiguity about their meaning. The proofs of soundness of our rules are given in Appendix B.

We first introduce a few general rules for consequence, sequential composition, conjunction and disjunction. Let $\phi_1, \phi_2, \phi_3, \phi_4$ be any four formulas in our logic, and let cmd, cmd_1, cmd_2 be any three commands. These rules are standard, and their proof are omitted.

(Csq) if $\phi_1 \Rightarrow \phi_2$, $\phi_3 \Rightarrow \phi_4$ and $\{\phi_2\}\mathsf{cmd}\{\phi_3\}$, then $\{\phi_1\}\mathsf{cmd}\{\phi_4\}$
(Seq) if $\{\phi_1\}\mathsf{cmd}_1\{\phi_2\}$ and $\{\phi_2\}\mathsf{cmd}_2\{\phi_3\}$, then $\{\phi_1\}\mathsf{cmd}_1; \mathsf{cmd}_2\{\phi_3\}$
(Conj) if $\{\phi_1\}\mathsf{cmd}\{\phi_2\}$ and $\{\phi_3\}\mathsf{cmd}\{\phi_4\}$, then $\{\phi_1 \wedge \phi_3\}\mathsf{cmd}\{\phi_2 \wedge \phi_4\}$
(Disj) if $\{\phi_1\}\mathsf{cmd}\{\phi_2\}$ and $\{\phi_3\}\mathsf{cmd}\{\phi_4\}$, then $\{\phi_1 \vee \phi_3\}\mathsf{cmd}\{\phi_2 \vee \phi_4\}$

Initialization:
We find that the following predicates holds in any distribution $X \in \mathrm{DIST}_0(\mathbb{H})$.

(Init) $\{\mathsf{Ind}(k; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - k) \wedge \mathsf{Eq}(k, k) \wedge \mathsf{Empty}\}$

We recall that k is a special variable holding a secret key. It is sampled at random before executing the program and is the same in both executions, so it is indistinguishable from a random value given any other value.

Generic preservation rules:
Rules (G1) to (G6) show how predicates are preserved by most of the commands when the predicates concern a variable other than that being operated on. For all these rules, we assume that t and t' can be y or z and cmd is either $x := \rho(i, y)$, $x := y$, $x := y\|z$, $x := y \oplus z$, $x := \mathcal{E}(y)$, or $x := \mathcal{H}(y)$.

(G1) $\{\mathsf{Eq}(t, t')\}$ cmd $\{\mathsf{Eq}(t, t')\}$ even if $t = y$ or $t = z$
(G2) $\{\mathsf{Uneq}(t, t')\}$ cmd $\{\mathsf{Uneq}(t, t')\}$ even if $t = y$ or $t = z$
(G3) $\{\mathsf{E}(\mathcal{E}; t; V)\}$ cmd $\{\mathsf{E}(\mathcal{E}; t; V)\}$ provided $x \notin V$ and cmd is not $x := \mathcal{E}(y)$
(G4) $\{\mathsf{H}(\mathcal{H}; t; V)\}$ cmd $\{\mathsf{H}(\mathcal{H}; t; V)\}$ provided $x \notin V$ and cmd is not $x := \mathcal{H}(y)$
(G5) $\{\mathsf{Ind}(t; V; V')\}$ cmd $\{\mathsf{Ind}(t; V; V')\}$ provided [cmd is not $x := \mathcal{E}(y)$ or $x := \mathcal{H}(y)$], [$x \notin V$ unless x is constructible from $V - t$] and [$x \notin V'$ unless x is constructible from $V' - t$]
(G6) $\{\mathsf{Empty}\}$ cmd $\{\mathsf{Empty}\}$ provided cmd is not $x := \mathcal{E}(y)$

We note that, for rules (G3) to (G6), the straightforward preservation rule does not apply when the command is either of the form $x := \mathcal{E}(y)$ or $x := \mathcal{H}(y)$, because some predicates may no longer hold if the block cipher or the random oracle is computed more than once on any given point. Therefore, the preservation of these predicates for the block cipher and hash commands will have to be handled separately in rules (B4) to (B6) and (H3) to (H5). For rule (G5), in general, we say that the value of a variable x is *constructible* from the values of variables in V if there exists a deterministic polynomial-time algorithm that can compute the value of x from the values in V. In this case, it means that the variables in the right-hand side of cmd are all in V.

Function ρ:

(P1) $\{\mathsf{Eq}(y, y)\}$ $x := \rho(i, y)$ $\{\mathsf{Eq}(x, x)\}$ for integer i

Since the details of the function ρ are not known in advance, we can infer only one rule, that ρ preserves equality, because it is a deterministic function.

Assignment:
Rules (A1) to (A8), for the assignment, are all straightforward, and follow simply from the simple fact that after the command, the value of x is equal to the value of y.

(A1) $\{true\}\ x := m_i\ \{(\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x)) \vee\ \mathsf{Uneq}(x, x)\}$
(A2) $\{\mathsf{Eq}(y, y)\}\ x := y\ \{\mathsf{Eq}(x, x)\}$
(A3) $\{\mathsf{Uneq}(y, y)\}\ x := y\ \{\mathsf{Uneq}(x, x)\}$
(A4) $\{\mathsf{Ind}(y; V; V')\}\ x := y\ \{\mathsf{Ind}(x; V; V')\}$ if $x \notin V'$ unless $y \in V'$ and $y \notin V$
(A5) $\{\mathsf{E}(\mathcal{E}; y; V)\}\ x := y\ \{\mathsf{E}(\mathcal{E}; x; V) \wedge \mathsf{E}(\mathcal{E}; y; V)\}$ if $y \notin V$
(A6) $\{\mathsf{H}(\mathcal{H}; y; V)\}\ x := y\ \{\mathsf{H}(\mathcal{H}; x; V) \wedge \mathsf{H}(\mathcal{H}; y; V)\}$ if $y \notin V$
(A7) $\{\mathsf{E}(\mathcal{E}; t; V, y)\}\ x := y\ \{\mathsf{E}(\mathcal{E}; t; V, x, y)\}$
(A8) $\{\mathsf{H}(\mathcal{H}; t; V, y)\}\ x := y\ \{\mathsf{H}(\mathcal{H}; t; V, x, y)\}$

Concatenation:
Rules (C1) to (C6) propagate the predicates for the concatenation command.

(C1) $\{\mathsf{Eq}(y, y)\}\ x := y\|m_i\ \{(\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x)) \vee\ \mathsf{Uneq}(x, x)\}$
(C2) $\{\mathsf{Eq}(y, y) \wedge \mathsf{Eq}(z, z)\}\ x := y\|z\ \{\mathsf{Eq}(x, x)\}$
(C3) $\{\mathsf{Uneq}(y, y)\}\ x := y\|z\ \{\mathsf{Uneq}(x, x)\}$
(C4) $\{\mathsf{Ind}(y; V, y, z; V') \wedge \mathsf{Ind}(z; V, y, z; V')\}\ x := y\|z\ \{\mathsf{Ind}(x; V, x; V')\}$ provided $[y \neq z]$, $[x, y, z \notin V]$ and $[x \notin V'$ unless $y, z \in V']$
(C5) $\{\mathsf{Ind}(y; V, \ell_{\mathcal{E}}; V)\}\ x := y\|z\ \{\mathsf{E}(\mathcal{E}; x; V)\}$
(C6) $\{\mathsf{Ind}(y; V, \ell_{\mathcal{H}}; V)\}\ x := y\|z\ \{\mathsf{H}(\mathcal{H}; x; V)\}$

The most important rule for the concatenation is (C4), which states that the concatenation of two random strings results in a random string. Note that it is important for this rule that $y \neq z$, otherwise the string x consists of a string twice repeated, which can be distinguished easily from a random value. The condition $x \notin V'$ unless $y, z \in V'$ is similar to rule (G5), and follows from the constructibility of x from y and z. Rules (C5) and (C6) state that if a string is indistinguishable from a random value given all the values in the set of queries to the block cipher (or the hash function), then clearly it cannot be a prefix of one of the strings $\mathcal{L}_{\mathcal{E}}$. For rules (C1), (C3), (C5) and (C6), the roles of y and z, or y and m_i in the case of (C1), can be exchanged.

Xor operator:
Rules (X1) to (X4) describe the effect of the Xor operation.

(X1) $\{\mathsf{Eq}(y, y)\}\ x := y \oplus m_i\ \{(\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x)) \vee\ \mathsf{Uneq}(x, x)\}$
(X2) $\{\mathsf{Ind}(y; V, y, z; V')\}\ x := y \oplus z\ \{\mathsf{Ind}(x; V, x, z; V')\}$ provided $[y \neq z]$, $[y \notin V]$ and $[x \notin V'$ unless $y, z \in V']$
(X3) $\{\mathsf{Eq}(y, y) \wedge \mathsf{Eq}(z, z)\}\ x := y \oplus z\ \{\mathsf{Eq}(x, x)\}$
(X4) $\{\mathsf{Eq}(y, y) \wedge \mathsf{Uneq}(z, z)\}\ x := y \oplus z\ \{\mathsf{Uneq}(x, x)\}$

Rules (X2) is reminiscent of a one-time-pad encryption: if a value z is xor-ed with a random-looking value y, than the result is similarly random-looking provided the value of y is not given. Again, the condition $x \notin V'$ unless $y, z \in V'$ is similar to rule (G5), and follows from the constructibility of x from y and z. The other rules are propagation of the Eq and Uneq predicates. Due to the commutativity of the xor, the role of y and z, or y and m_i in the case of (X1), can be exchanged in all the rules above.

Block cipher:
Since block ciphers are modeled as random functions, that is, functions picked at random among all functions from $\{0, 1\}^\eta$ to $\{0, 1\}^\eta$, the output of the function for a point on

which the block cipher has never been computed is indistinguishable from a random value.

(B1) $\{\mathsf{Empty}\}\ x := \mathcal{E}(m_i)\ \{(\mathsf{Uneq}(x,x) \wedge \mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{E}, \ell_\mathcal{H}; \mathsf{Var})) \vee$
$(\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x,x) \wedge \mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{E}, \ell_\mathcal{H}; \mathsf{Var} - x))\}$

(B2) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Uneq}(y,y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{E}, \ell_\mathcal{H}; \mathsf{Var})\}$

(B3) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Eq}(y,y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{E}, \ell_\mathcal{H}; \mathsf{Var} - x) \wedge \mathsf{Eq}(x,x)\}$

(B4) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Ind}(t; V; V')\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(t; V, x; V', x)\}$ even if $t = y$, provided $\ell_\mathcal{E} \notin V$

(B5) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Ind}(t; V, \ell_\mathcal{E}, y; V', y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(t; V, \ell_\mathcal{E}, x, y; V', x, y)\}$

(B6) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{E}(\mathcal{E}; t; V, y)\}\ x := \mathcal{E}(y)\ \{\mathsf{E}(\mathcal{E}; t; V, y)\}$

This is expressed in rules (B1) to (B3), and also used in the proof of many other rules. Note that, when executing $x := \mathcal{E}(y)$ on a new value, if the values of y from the two executions are equal, then of course the values of x will be equal afterwards. However, if the values of y are not the same in the two executions, then the values of x will be indistinguishable from two *independent* random values afterwards.

Since the querying of a block cipher twice at any point is undesirable, we always require the predicate E as a precondition. We also have rules similar to (B2) to (B6), with the predicate $\mathsf{E}(\mathcal{E}; y; \emptyset)$ replaced by the predicate Empty, since both imply that the value of y is not in $\mathcal{L}_\mathcal{E}$.

Hash Function:

We note that the distinguishing adversary, described in Section 2, does not have access to the random oracle. This is sufficient for our purpose since our goal is only to prove inequality of strings, not their indistinguishability from random strings. As a result, the rules for the hash function are essentially the same as those for the block cipher.

(H1) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Uneq}(y,y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{E}, \ell_\mathcal{H}; \mathsf{Var})\}$

(H2) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Eq}(y,y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(x; \mathsf{Var}, \ell_\mathcal{H}; \mathsf{Var} - x) \wedge \mathsf{Eq}(x,x)\}$

(H3) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Ind}(t; V; V')\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(t; V, x; V', x)\}$ even if $t = y$, provided $\ell_\mathcal{H} \notin V$

(H4) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Ind}(t; V, \ell_\mathcal{H}, y; V', y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(t; V, \ell_\mathcal{H}, x, y; V', x, y)\}$

(H5) $\{\mathsf{H}(\mathcal{H}; t; V, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{H}(\mathcal{H}; t; V, y)\}$

For loop:

(F1) $\{\psi(p-1)\}$ $\mathsf{for}\ l = p\ \mathsf{to}\ q\ \mathsf{do}{:}\ [\mathsf{cmd}_l]$ $\{\psi(q)\}$ provided
$\{\psi(l-1)\}\ \mathsf{cmd}_l\ \{\psi(l)\}$ for $p \leq l \leq q$

The rule for the For loop simply states that if an indexed formula $\psi(i)$ is preserved through one iteration of the loop, then it is preserved through the entire loop. We discuss methods for finding such a formula in Section 5.

Combining our logic with Proposition 1, we obtain the following theorem.

Theorem 1. *Let* $(\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, Hash(m_1 \| \ldots \| m_n, c_n)$: **var** $\boldsymbol{x}; \mathit{cmd})$ *describe the program to compute a hash function* $Hash$ *on an* n *block message. Then,* $Hash$ *is an almost-universal hash function if, for every positive integer* n, $\{init\}$ cmd $\{UNIV(n)\}$.

The theorem is the consequence of Proposition 1 and of the soundness of our Hoare logic. We then say that a sequence of formulas $[\phi_0, \ldots, \phi_n]$ is a proof that a program $[\mathsf{cmd}_1, \ldots, \mathsf{cmd}_n]$ computes an almost-universal hash function if $\phi_0 = true$, $\phi_n \Rightarrow UNIV(n)$ and for all i, $1 \leq n$, $\{\phi_{i-1}\}$ cmd_i $\{\phi_i\}$ holds.

5 Implementation

We chose to go forward through the program, instead of the more common approach of going backward from the end, after implementing both methods. Going backward through the program can require exploring multiple combinations of choices that all need to be explores when many rules can lead to the necessary predicate. The presence of the logical-or connector in our logic often resulted in an exponential number of possibilities at each step. As a result, our prototype for the forward method was able to find proofs much faster than an implementation of the backwards method.

We start at the beginning of the program and, at each command, apply every possible rule. Once done, we test if the predicate $UNIV(n)$ holds at the end of the program. One downside of this forward approach is that the application of every possible rule can be very time consuming because the formulas tend to grow after each command, which leads to more and more rules being applied at every step. For this reason, we need a way to filter out unneeded predicates, so that execution time remains reasonable.

5.1 Predicate Filter

We say that ϕ is a *predicate on* x if ϕ is either $\mathsf{Eq}(x,y)$, $\mathsf{Uneq}(x,y)$, $\mathsf{E}(\mathcal{E};x;V)$, $\mathsf{H}(\mathcal{H};x;V)$ or $\mathsf{Ind}(x;V_1,V_2)$ (for some $y \in \mathsf{Var}$ and $V_1,V_2 \subseteq \mathsf{Var}$). We say that a predicate ϕ on variable x is *obsolete for program* p if x does not appear anywhere in p and if $\neg(\phi \Rightarrow \mathsf{Uneq}(c_n,c_i))$ and $\neg(\phi \Rightarrow \mathsf{Eq}(m_i,m_i))$ for any i, $1 \leq i \leq n$.[2] The following theorem shows that once a predicate is obsolete, it can be discarded.

Theorem 2. *If there exists a proof* $[\phi_0,\ldots,\phi_n]$ *that a program* $p = [\mathsf{cmd}_1,\ldots,\mathsf{cmd}_n]$ *computes an almost-universal hash function, then there also exists a proof* $[\phi'_0,\ldots,\phi'_n]$ *that p computes an almost-universal hash function where for each i,* $\phi_i \Rightarrow \phi'_i$ *and each* ϕ'_i *does not contain any obsolete predicates for* $[\mathsf{cmd}_{i+1},\ldots,\mathsf{cmd}_n]$.

The theorem is a consequence of the fact that, in our logic, the rules for creating a predicate on x following the execution of command $x := e$ only have as preconditions predicates on the variables in e. As a result, we can always filter out obsolete predicates after processing each command.

Also, we note that the only commands that can make a predicate $\mathsf{Eq}(m_i,m_i)$ appear are those of the form $x := e$ in which m_i appears in e. As a result, if we find that, for some integer l, the predicate $\mathsf{Eq}(m_l,m_l)$ is not present in one of the conjunctions of the current formula (after transforming the formula in disjunctive normal form) and that the variable m_l is no longer present in the rest of the program, then there is no longer any chance that it will satisfy the conjunction with $\bigwedge_{j=1}^{n} \mathsf{Eq}(m_j,m_j)$ from $UNIV(n)$. Therefore, we can also safely filter out all other predicates of the form $\mathsf{Eq}(m_i,m_i)$ from that conjunction.

We also add a *heuristic filter* to speed up the execution of our method. We make the hypothesis that the predicate $\mathsf{Ind}(c_n;V;\{c_1,\ldots,c_{n-1}\})$ will be present at the end of the program, which is the case for all our examples, so that we can filter out $\mathsf{Ind}(c_i;V;V')$

[2] Here, p will usually be the rest of the program after the program point at which the predicate ϕ holds.

if $i < n$ and c_i is no longer present in the remainder of the program. In addition to speeding up the program, filtering out these predicates greatly simplifies the construction of loop invariants discussed in the next section. If we fail to produce a proof while using the heuristic filter, we simply attempt again to find a proof without it.

5.2 Finding Loop Invariants

The programs describing the almost-universal hash function usually contains for loops. It is therefore necessary to have an automatic procedure to detect the formula $\psi(i)$ that allows us to apply rule (F1). We now show a heuristic that can be used to construct such an invariant, and illustrate how it works by applying them to $Hash_{CBC}$, described in Section 3.1. One could easily verify that it also works on $Hash_{CBC'}$, $Hash_{HMAC}$ and $Hash_{PMAC}$.

Once we hit a command "$\mathsf{for}\ l = p\ \mathsf{to}\ q\ \mathsf{do}$: [$\mathsf{cmd}_l$]", we express the formula that holds before the loop is executed in the form $\varphi(p-1)$. The classical method for finding a stable invariant consists in processing the instructions cmd_l contained in the loop to find the formula $\psi(l)$ such that $\{\varphi(l-1)\}\ \mathsf{cmd}_l\ \{\psi(l)\}$. If $\psi(l) \Rightarrow \varphi(l)$, then we have found a formula such that $\{\varphi(l-1)\}\ \mathsf{cmd}_l\ \{\varphi(l)\}$ and we can apply rule $(F1)$.

Unfortunately, for most loops, this simple process either does not yield a stable invariant, or gives a stable invariant too weak to produce a proof. We need a heuristic to construct stronger stable invariants. The heuristic we describe here is inspired from widening methods in abstract interpretation. We start with formula $\varphi(l-1)$, and process the program of the loop once to find formula $\psi_1(l)$ such that $\{\varphi(l-1)\}\ \mathsf{cmd}_l\ \{\psi_1(l)\}$. Then, we repeat this starting with formula $\psi_1(l-1)$ to find formula $\psi_2(l)$ such that $\{\psi_1(l-1)\}\ \mathsf{cmd}_l\ \{\psi_2(l)\}$. The idea is then to inspect formulas $\varphi(l)$, $\psi_1(l)$ and $\psi_2(l)$ for patterns that can be extrapolated. For example, we can try to identify a predicate $\gamma(l)$ such that: (i) $\gamma(l)$ appears in $\varphi(l)$, (ii) $\gamma(l-1) \wedge \gamma(l)$ appears in $\psi_1(l)$, (iii) $\gamma(l-2) \wedge \gamma(l-1) \wedge \gamma(l)$ appears in $\psi_2(l)$. We then use a new starting formula $\varphi'(l)$ which is just like $\varphi(l)$, except that the occurrence of $\gamma(l)$ in $\varphi(l)$ is replaced by $\bigwedge_{j=p-1}^{j=l} \gamma(j)$ in $\varphi'(l)$. Note that, by construction, $\varphi(p-1)$ is equal to $\varphi'(p-1)$, so we know that $\varphi'(p-1)$ is satisfied at the beginning of the loop.[3]

Example: We now apply this method to $Hash_{CBC}$. After processing command $c_1 := \mathcal{E}(m_1)$, we obtain the formula $\varphi(1) = (\mathsf{Ind}(c_1; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_1) \wedge \mathsf{Eq}(m_1, m_1) \wedge \mathsf{Eq}(c_1, c_1)) \vee \mathsf{Ind}(c_1)$. Parameterizing this in terms of l, we obtain

$$\varphi(l) = (\mathsf{Eq}(m_l, m_l) \wedge \mathsf{Eq}(c_l, c_l) \wedge \mathsf{Ind}(c_l; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_l)) \vee \mathsf{Ind}(c_l)$$

We recall that the two instructions in the loop of $Hash_{CBC}$ are the following: $z_i := c_{i-1} \oplus m_i;\quad c_i := \mathcal{E}(z_i)$. After processing the program of the loop on $\varphi(l-1)$, we obtain the following.

$$\begin{aligned}\psi_1(l) = &(\mathsf{Eq}(m_{l-1}, m_{l-1}) \wedge \mathsf{Eq}(m_l, m_l) \wedge \mathsf{Eq}(c_l, c_l) \wedge \mathsf{Ind}(c_l; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_l))\\ &\vee \mathsf{Ind}(c_l)\end{aligned}$$

[3] We can similarly try to find patterns that appear only after the first iteration of the loop, that is, $\gamma(l)$ appears in $\psi_1(l)$ and $\gamma(l-1) \wedge \gamma(l)$ appears in $\psi_2(l)$, in which case $\bigwedge_{j=p}^{j=l} \gamma(j)$ is added in $\varphi'(l)$.

We get this by applying rules (G1), (X1) and (X2) for the first command and rules (G1), (B2) and (B3) for the second command. Note that $\psi_1(l) \Rightarrow \varphi(l)$, so we could use $\varphi(l)$ to apply rule (F1), but this would not yield a proof of $Hash_{CBC}$. We repeat the same process with $\psi_1(l-1)$ to obtain

$$\psi_2(l) = (\mathsf{Eq}(m_{l-2}, m_{l-2}) \wedge \mathsf{Eq}(m_{l-1}, m_{l-1}) \wedge \mathsf{Eq}(m_l, m_l) \wedge \mathsf{Eq}(c_l, c_l) \wedge \mathsf{Ind}(c_l; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_l)) \vee \mathsf{Ind}(c_l).$$

This requires applying the same rules as before, but rule (G1) more often applied for each command. We find $\gamma(l) = \mathsf{Eq}(m_l, m_l)$ and use

$$\varphi'(l) = \left(\left(\bigwedge_{i=1}^{l} \mathsf{Eq}(m_i, m_i)\right) \wedge \mathsf{Eq}(c_l, c_l) \wedge \mathsf{Ind}(c_l; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_l)\right) \vee \mathsf{Ind}(c_l)$$

as our next attempt at finding a stable invariant. We find that $\varphi'(l)$ is a stable invariant for the loop. So we apply the rule (F1) to obtain that $\varphi'(n)$ holds at the end of the program, and we easily find that $\varphi'(n) \Rightarrow UNIV(n)$ for all positive integer n, thereby proving that $Hash_{CBC}$ computes an almost-universal hash function.

5.3 Prototype

We programmed an OCaml prototype of our method for proving that the front end of MACs are almost-universal hash functions. The program requires about 2000 lines of code, and can successfully produce proofs of security for all the examples discussed in this paper in less than one second on a personal workstation. Our prototype is available on [14].

6 Proving MAC Security

As mentioned in Section 2, we prove the security of MACs in two steps: first we show that the 'compressing' part of the MAC is an almost-universal hash function family, and then we show that the last section of the MAC, when applied to an almost-universal hash function, results in a secure MAC. The following shows how a secure MAC can be constructed from an almost-universal hash function. The proof can be found in [4, 8, 9], so we do not repeat them here.

Theorem 3. *Let $\mathcal{H} = \{h_i\}_{i \in \{0,1\}^\eta}$ and $\mathcal{H}' = \{h_i\}_{i \in \{0,1\}^\eta}$ be families of almost-universal hash function, $\mathcal{F}_{\mathcal{E}}$ be a family of block ciphers and $\mathcal{G}$ be a random oracle. If $h \xleftarrow{\$} \mathcal{H}$, $h_{\mathcal{E}} \xleftarrow{\$} \mathcal{H}'$, $\mathcal{E}, \mathcal{E}_1, \mathcal{E}_2 \xleftarrow{\$} \mathcal{F}_{\mathcal{E}}$, $\mathcal{G}$ is sampled at random from all functions with the appropriate domain and range and $k, k_1, k_2 \xleftarrow{\$} \{0,1\}^\eta$, then the following hold:*

- $MAC_1(m) = \mathcal{E}(h_i(m))$ *is a secure MAC with key* $sk = (i, k_{\mathcal{E}})$.[4]
- $MAC_2(m) = \mathcal{G}(k \| h_i(m))$ *is a secure MAC with key* $sk = (i, k)$.

[4] Here, $k_{\mathcal{E}}$ denotes the secret key associated with block cipher $\mathcal{E}$.

– $MAC_3(m) =$
$$\begin{cases} \mathcal{E}_1(h_i(m')) \textit{ where } m' = pad(m) \textit{ if } m\textit{'s length is not a multiple of } \eta \\ \mathcal{E}_2(h_i(m)) \textit{ if } m\textit{'s length is a multiple of } \eta \end{cases}$$
is a secure MAC with key $sk = (i, k_{\mathcal{E}_1}, k_{\mathcal{E}_2})$.

– $MAC_4(m) =$
$$\begin{cases} \mathcal{E}(h_{\mathcal{E}}(m') \oplus k_1) \textit{ where } m' = pad(m) \textit{ if } m\textit{'s length is not a multiple of } \eta \\ \mathcal{E}(h_{\mathcal{E}}(m) \oplus k_2) \textit{ if } m\textit{'s length is a multiple of } \eta \end{cases}$$
is a secure MAC with key $sk = (k_{\mathcal{E}}, k_1, k_2)$

Combining $Hash_{CBC}$ with MAC_1 and MAC_3 yields the message authentication codes DMAC and ECBC respectively, using $Hash_{CBC'}$ with MAC_3 and MAC_4 yields FCBC and XCBC, combining $Hash_{PMAC}$ and MAC_4 yields a four key construction of PMAC and using $Hash_{HMAC}$ with MAC_2 yields HMAC.

7 Conclusion

We presented a Hoare logic that can be used to automatically prove the security of constructions for almost-universal hash functions based on block ciphers and compression functions modeled as random oracles. We can then obtain a secure MAC by combining with a few operations, such as those presented in Section 6. Our method can be used to prove the security of DMAC, ECBC, FCBC, XCBC, a two-key variant of HMAC and a four-key variant of PMAC. Since we do not have a global view of the algorithm, we cannot prove the one key variants of HMAC or PMAC, nor can we prove CMAC or OMAC, which are one-key variants of XCBC. It is however relatively simple to derive the security of these one-key schemes by hand once the security of the multiple key variants has been proven. It remains an open problem to integrate this step into the logic.

It should be possible to extend our logic to prove exact reduction bounds for the security of the ϵ-universal hash function. This could be done by keeping track of exact security for each predicate to obtain a bound on the final invariant. We are also working on integrating our tool for verifying the security of MACs with the tool for verifying the security of encryption modes of operation of [15], to get a general tool for producing security proofs of symmetric modes of operation.

References

1. G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 375–386. ACM, 2010.
2. G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.
3. G. Barthe, B. Grégoire, Y. Lakhnech, and S. Z. Béguelin. Beyond provable security verifiable ind-cca security of oaep. In *CT-RSA*, Lecture Notes in Computer Science, pages 180–196. Springer, 2011.
4. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96, Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

5. M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, pages 341–358, 1994.
6. M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73, New York, USA, November 1993. ACM, ACM.
7. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. Umac: Fast and secure message authentication. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 216–233, 1999.
8. J. Black and P. Rogaway. Cbc macs for arbitrary-length messages: The three-key constructions. In *Advances in Cryptology CRYPTO 00, Lecture Notes in Computer Science*, pages 197–215. Springer-Verlag, 2000.
9. J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - EUROCRYPT 2002. Lecture Notes in Computer Science*, pages 384–397. Springer-Verlag, 2002.
10. B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.
11. R. Corin and J. den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2006.
12. J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lahknech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, (CCS'08)*, Alexandria, USA, Oct. 2008.
13. C. Fournet, M. Kohlweiss, and P. Strub. Modular code-based cryptographic verification. In Y.Chen, G. Danezis, and V. Shmatikov, editors, *ACM-CCS'11*, pages 341–350. ACM, 2011.
14. M. Gagné, P. Lafourcade, and Y. Lakhnech. OCaml implementation of our method. Computer Science Department, Saarland University, Germany, February 2013. Available at `http://www.infsec.cs.uni-saarland.de/~gagne/macChecker/macChecker.html`.
15. M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini. Automated proofs for encryption modes. In *13th Annual Asian Computing Science Conference Focusing on Information Security and Privacy: Theory and Practice (ASIAN'09)*, volume 5913 of *LNCS*, pages 39–53, 2009.
16. M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.
17. E. Petrank and C. Rackoff. Cbc mac for real-time data sources. *JOURNAL OF CRYPTOLOGY*, 13:315–338, 1997.
18. M. Wegman and J. L. Carter. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1919.
19. M. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

A Programs for Hash Functions

We present below the codes for all the other hash functions corresponding to MACs that we have tested with our method.

$Hash_{CBC'}(m_1\|\ldots\|m_n, c_n)$:
var $i, z_2, \ldots, z_n, c_1, \ldots, c_{n-1}$;
$c_1 := m_1$;
for $i = 2$ to n do:
  $[z_i := \mathcal{E}(c_{i-1})$;
  $c_i := z_i \oplus m_i]$

$Hash_{HMAC}(m_1\|\ldots\|m_n, c_n)$:
var $i, z_1, \ldots, z_n, c_1, \ldots, c_{n-1}$;
$z_1 := k\|m_1$;
$c_1 = \mathcal{H}(z_1)$;
for $i = 2$ to n do:
  $[z_i := c_{i-1}\|m_i$;
  $c_i := \mathcal{H}(z_i)]$

$Hash_{PMAC}(m_1\|\ldots\|m_n, c_n)$:
var $i, w_1, x_1, y_1, \ldots, w_n, x_n, y_n,$
  $z_1, \ldots, z_n, c_1, \ldots, c_{n-1}$
$c_1 := m_1$;
$w_1 := \rho(1, k)$;
$x_1 := w_1 \oplus m_1$;
$z_1 = \mathcal{E}(x_1)$;
for $i = 2$ to n do:
  $[c_i := z_{i-1} \oplus m_i$;
  $w_i := \rho(i, k)$;
  $x_i := w_i \oplus m_i$;
  $y_i := \mathcal{E}(x_i)$;
  $z_i := z_{i-1} \oplus y_i]$

B Proofs

Proof of Lemma 1

Proof. These are all fairly straightforward.

1. If an algorithm could distinguish $(S(x), S(V_3) \cup S'(V_4))$ from $(u, S(V_3) \cup S'(V_4))$, a similar algorithm would be able to distinguish $(S(x), S(V_1) \cup S'(V_2))$ from $(u, S(V_1) \cup S'(V_2))$ by simply disregarding the values in $S(V_1) \setminus S(V_3)$ and $S'(V_2) \setminus S'(V_4)$.
2. and 3) are trivial: $x \notin T \Rightarrow x \notin T'$ for $T' \subset T$.
4. to 6) follow from the simple observation that if $X \models \mathsf{Ind}(x, V)$, then the probability that the value of x is equal to the value of any variable in V (or any values in $\mathcal{L}_{\mathcal{E}}$.dom, $\mathcal{L}_{\mathcal{H}}$.dom or in the simultaneous execution, if $\mathcal{L}_{\mathcal{E}}$ or $\mathcal{L}_{\mathcal{H}}$ is in V) is negligible, otherwise an adversary could distinguish the value of x from a random value by comparing it to all the values in $S(V)$.

Proof of Lemma 2

Proof. Since $X \in \textsc{Dist}(\Gamma, \mathcal{F}_{\mathcal{E}}, \mathcal{F}_{\mathcal{H}})$, then, by definition, $X = [\![\mathsf{cmd}]\!]\mathcal{D}_0^{(M,M')}$ for some program cmd and n-block messages M, M'. We note that, by design, for every configuration $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}})$ that has non-zero probability in $\mathcal{D}_0^{(M,M')}$, $S(m_i)$ is equal to the i^{th} block of M and $S'(m_i)$ is equal to the i^{th} block of M'. Therefore, it is clear that either the i^{th} blocks of M and M' are equal, in which case $\mathcal{D}_0^{(M,M')} \models \mathsf{Eq}(m_i, m_i)$, or they are not equal, and we have $\mathcal{D}_0^{(M,M')} \models \mathsf{Uneq}(m_i, m_i)$. The result then follows from our assumption that the message variables are never assigned new values.

Proof of Lemma 3

Proof. Since $X \models \mathsf{Ind}(z; V; V')$, we have the two following equations:

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(z), S(V - z) \cup S'(V'))] \sim \\ [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X; u \xleftarrow{\$} \mathcal{U} : (u, S(V - z) \cup S'(V'))]$$

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S'(z), S'(V - z) \cup S(V'))] \sim \\ [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X; u \xleftarrow{\$} \mathcal{U} : (u, S'(V - z) \cup S(V'))]$$

Suppose there exists an algorithm $\mathcal{A}$ which can distinguish distribution $[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X : (S(z), S(V - z, x) \cup S'(V'))]$ from $[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X; u \xleftarrow{\$} \mathcal{U} : (u, S(V - z, x) \cup S'(V'))]$ with non-negligible probability. Then we construct an algorithm $\mathcal{B}$ as follows:
On input $(S(z), S(V - z) \cup S'(V'))$, $\mathcal{B}$ uses the values in $S(V - z)$ to compute the value v of e and runs algorithm $\mathcal{A}$ on $(S(z), S(V - z) \cup \{v\} \cup S'(V'))$. It should be clear that $\mathcal{B}$ is successful in distinguishing $[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(z), S(V - z) \cup S'(V'))]$ from $[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X; u \xleftarrow{\$} \mathcal{U} : (u, S(V - z) \cup S'(V'))]$ with the same probability as $\mathcal{A}$, which contradicts $X \models \mathsf{Ind}(z; V; V')$. Therefore,

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X : (S(z), S(V - z, x) \cup S'(V'))] \sim \\ [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X; u \xleftarrow{\$} \mathcal{U} : (u, S(V - z, x) \cup S'(V'))]$$

Similarly,

$$[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X : (S'(z), S'(V - z, x) \cup S(V'))] \sim \\ [(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := e]\!]X; u \xleftarrow{\$} \mathcal{U} : (u, S'(V - z, x) \cup S(V'))]$$

which means $[\![x := e]\!]X \models \mathsf{Ind}(z; V, x; V')$.

The proof that $X \models \mathsf{Ind}(z; V'; V)$ implies $[\![x := e]\!](X) \models \mathsf{Ind}(z; V'; V, x)$ is done in exactly the same way.

Proof of Lemma 4

Proof. This is a trivial consequence of the fact that the message variables always have equal length in both executions, and the message variables are the only ones that are assigned a value in $\textsc{Dist}_0(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$. All values computed from there will therefore also have equal length.

Proof of Proposition 1

Proof. (sketch) Say M_1 is a k-block message, and M_2 is an l-block message with $1 \leq l \leq k$. If $l < k$, let M_2' be obtained from M_2 by padding with any string up to k blocks, otherwise $M_2' = M_2$. We want to show that, either $M_1 = M_2$, or the probability that M_1 and M_2 hash to the same value is negligible. Thanks to our constraint on the construction of the program, with M_1 placed as the message in S and M_2 placed in S', which happens in $\mathcal{D}_0^{(M_1, M_2')}$, we will have that c_k contains the hash of M_1 in the first execution and c_l contains the hash of M_2 in the second execution. If the invariant $UNIV(k)$ holds after executing the program on $\mathcal{D}_0^{(M_1,M_2')}$, then, when $k \neq l$, then we have that $\mathsf{Uneq}(c_k, c_l)$ holds, and when $k = l$, then either $\mathsf{Uneq}(c_k, c_k)$ holds, or $\bigwedge_{i=1}^{n} \mathsf{Eq}(m_i, m_i)$ does, which shows that the probability that the hashes are equal is negligible or $M_1 = M_2$, as required.

We recall all the rules in Table 1, and prove all the rules, grouping them according to the corresponding commands.

B.1 Initialization

Proposition 2 (Rule (init)). *If* $X \in \mathrm{DIST}_0(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$, *then* $X \models \{\mathsf{Ind}(k; \mathit{Var}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}; \mathit{Var} - k) \wedge \mathsf{Empty}\}$

Proof. Let $X \in \mathrm{DIST}_0(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$. We have to prove that $X \models \mathsf{Ind}(k; \mathsf{Var}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}; \mathsf{Var} - k)$ and $X \models \mathsf{Empty}$. The former is obvious from the definition of $\mathrm{DIST}_0(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$. The latter is also clear because k is sampled randomly in the definition of $\mathrm{DIST}_0(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$.

B.2 Generic Preservation

Proposition 3 (Rule (G1)).
$\{\mathsf{Eq}(t)\}$ cmd $\{\mathsf{Eq}(t)\}$ *even if* $t = y$ *or* $t = z$

Proof. Trivial since $t \neq x$ and only the value of x can be changed by the command.

Proposition 4 (Rule (G2)).
$\{\mathsf{Uneq}(t)\}$ cmd $\{\mathsf{Uneq}(t)\}$ *even if* $t = y$ *or* $t = z$

Proof. Trivial since $t \neq x$ and only the value of x can be changed by the command.

Proposition 5 (Rule (G3)).
$\{\mathsf{E}(\mathcal{E}; t; V)\}$ cmd $\{\mathsf{E}(\mathcal{E}; t; V)\}$ *provided* $x \notin V$ *and* cmd *is not* $x := \mathcal{E}(y)$

Proof. Clearly, $\Pr[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \overset{\$}{\leftarrow} X : S(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V) \vee S'(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S'(V)] = \Pr[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \overset{\$}{\leftarrow} [\![x := \mathcal{E}(y)]\!]X : S(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V) \vee S'(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S'(V)]$ because, the values in the sets $S(V)$, $S'(V)$ and $Elist.dom$ are unchanged by the command.

Proposition 6 (Rule (G4)).
$\{\mathsf{H}(\mathcal{H}; t; V)\}$ cmd $\{\mathsf{H}(\mathcal{H}; t; V)\}$ *provided* $x \notin V$ *and* cmd *is not* $x := \mathcal{H}(y)$

Proof. Similar to the proof of Rule (G3).

Proposition 7 (Rule (G5)).
$\{\mathsf{Ind}(t; V; V')\}$ cmd $\{\mathsf{Ind}(t; V; V')\}$ *provided* cmd *is not* $x := \mathcal{E}(y)$ *or* $x := \mathcal{H}(y)$, *and* $x \notin V$ *unless* x *is constructible from* $V - t$ *and* $x \notin V'$ *unless* x *is constructible from* $V' - t$

Generic Preservation

(G1) $\{\mathsf{Eq}(t,t')\}$ cmd $\{\mathsf{Eq}(t,t')\}$ even if $t=y$ or $t=z$

(G2) $\{\mathsf{Uneq}(t,t')\}$ cmd $\{\mathsf{Uneq}(t,t')\}$ even if $t=y$ or $t=z$

(G3) $\{\mathsf{E}(\mathcal{E};t;V)\}$ cmd $\{\mathsf{E}(\mathcal{E};t;V)\}$ provided $x\notin V$ and cmd is not $x:=\mathcal{E}(y)$

(G4) $\{\mathsf{H}(\mathcal{H};t;V)\}$ cmd $\{\mathsf{H}(\mathcal{H};t;V)\}$ provided $x\notin V$ and cmd is not $x:=\mathcal{H}(y)$

(G5) $\{\mathsf{Ind}(t;V;V')\}$ cmd $\{\mathsf{Ind}(t;V;V')\}$ provided [cmd is not $x:=\mathcal{E}(y)$ or $x:=\mathcal{H}(y)$], [$x\notin V$ unless x is constructible from $V-t$] and [$x\notin V'$ unless x is constructible from $V'-t$]

(G6) $\{\mathsf{Empty}\}$ cmd $\{\mathsf{Empty}\}$ provided cmd is not $x:=\mathcal{E}(y)$

Function ρ:

(P1) $\{\mathsf{Eq}(y,y)\}$ $x:=\rho^i(y)$ $\{\mathsf{Eq}(x,x)\}$ for positive integer i

Assignment:

(A1) $\{true\}$ $x:=m_i$ $\{(\mathsf{Eq}(m_i,m_i)\wedge\mathsf{Eq}(x,x))\vee\mathsf{Uneq}(x,x)\}$

(A2) $\{\mathsf{Eq}(y,y)\}$ $x:=y$ $\{\mathsf{Eq}(x,x)\}$

(A3) $\{\mathsf{Uneq}(y,y)\}$ $x:=y$ $\{\mathsf{Uneq}(x,x)\}$

(A4) $\{\mathsf{Ind}(y;V;V')\}$ $x:=y$ $\{\mathsf{Ind}(x;V;V')\}$ if $x\notin V'$ unless $y\in V'$ and $y\notin V$

(A5) $\{\mathsf{E}(\mathcal{E};y;V)\}$ $x:=y$ $\{\mathsf{E}(\mathcal{E};x;V)\wedge\mathsf{E}(\mathcal{E};y;V)\}$ if $y\notin V$

(A6) $\{\mathsf{H}(\mathcal{H};y;V)\}$ $x:=y$ $\{\mathsf{H}(\mathcal{H};x;V)\wedge\mathsf{H}(\mathcal{H};y;V)\}$ if $y\notin V$

(A7) $\{\mathsf{E}(\mathcal{E};t;V,y)\}$ $x:=y$ $\{\mathsf{E}(\mathcal{E};t;V,x,y)\}$

(A8) $\{\mathsf{H}(\mathcal{H};t;V,y)\}$ $x:=y$ $\{\mathsf{H}(\mathcal{H};t;V,x,y)\}$

Concatenation:

(C1) $\{\mathsf{Eq}(y,y)\}$ $x:=y\|m_i$ $\{(\mathsf{Eq}(m_i,m_i)\wedge\mathsf{Eq}(x,x))\vee\mathsf{Uneq}(x,x)\}$

(C2) $\{\mathsf{Eq}(y,y)\wedge\mathsf{Eq}(z,z)\}$ $x:=y\|z$ $\{\mathsf{Eq}(x,x)\}$

(C3) $\{\mathsf{Uneq}(y,y)\}$ $x:=y\|z$ $\{\mathsf{Uneq}(x,x)\}$

(C4) $\{\mathsf{Ind}(y;V,y,z;V')\wedge\mathsf{Ind}(z;V,y,z;V')\}$ $x:=y\|z$ $\{\mathsf{Ind}(x;V,x;V')\}$ provided [$y\neq z$], [$x,y,z\notin V$] and [$x\notin V'$ unless $y,z\in V'$]

(C5) $\{\mathsf{Ind}(y;V,\ell_{\mathcal{E}};V)\}$ $x:=y\|z$ $\{\mathsf{E}(\mathcal{E};x;V)\}$

(C6) $\{\mathsf{Ind}(y;V,\ell_{\mathcal{H}};V)\}$ $x:=y\|z$ $\{\mathsf{H}(\mathcal{H};x;V)\}$

Xor operator:

(X1) $\{\mathsf{Eq}(y,y)\}$ $x:=y\oplus m_i$ $\{(\mathsf{Eq}(m_i,m_i)\wedge\mathsf{Eq}(x,x))\vee\mathsf{Uneq}(x,x)\}$

(X2) $\{\mathsf{Ind}(y;V,y,z;V')\}$ $x:=y\oplus z$ $\{\mathsf{Ind}(x;V,x,z;V')\}$ provided [$y\neq z$], [$y\notin V$] and [$x\notin V'$ unless $y,z\in V'$]

(X3) $\{\mathsf{Eq}(y,y)\wedge\mathsf{Eq}(z,z)\}$ $x:=y\oplus z$ $\{\mathsf{Eq}(x,x)\}$

(X4) $\{\mathsf{Eq}(y,y)\wedge\mathsf{Uneq}(z,z)\}$ $x:=y\oplus z$ $\{\mathsf{Uneq}(x,x)\}$

Block cipher:

(B1) $\{\mathsf{Empty}\}$ $x:=\mathcal{E}(m_i)$ $\{(\mathsf{Eq}(m_i,m_i)\wedge\mathsf{Eq}(x,x)\wedge\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var}-x))\vee$ $(\mathsf{Uneq}(x,x)\wedge\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var}))\}$

(B2) $\{\mathsf{E}(\mathcal{E};y;\emptyset)\wedge\mathsf{Uneq}(y,y)\}$ $x:=\mathcal{E}(y)$ $\{\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var})\}$

(B3) $\{\mathsf{E}(\mathcal{E};y;\emptyset)\wedge\mathsf{Eq}(y,y)\}$ $x:=\mathcal{E}(y)$ $\{\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var}-x)\wedge\mathsf{Eq}(x,x)\}$

(B4) $\{\mathsf{E}(\mathcal{E};y;\emptyset)\wedge\mathsf{Ind}(t;V;V')\}$ $x:=\mathcal{E}(y)$ $\{\mathsf{Ind}(t;V,x;V',x)\}$ even if $t=y$, provided $\ell_{\mathcal{E}}\notin V$

(B5) $\{\mathsf{E}(\mathcal{E};y;\emptyset)\wedge\mathsf{Ind}(t;V,\ell_{\mathcal{E}},y;V',y)\}$ $x:=\mathcal{E}(y)$ $\{\mathsf{Ind}(t;V,\ell_{\mathcal{E}},x,y;V',x,y)\}$

(B6) $\{\mathsf{E}(\mathcal{E};y;\emptyset)\wedge\mathsf{E}(\mathcal{E};t;V,y)\}$ $x:=\mathcal{E}(y)$ $\{\mathsf{E}(\mathcal{E};t;V,y)\}$

Hash Function:

(H1) $\{\mathsf{H}(\mathcal{H};y;\emptyset)\wedge\mathsf{Uneq}(y,y)\}$ $x:=\mathcal{H}(y)$ $\{\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var})\}$

(H2) $\{\mathsf{H}(\mathcal{H};y;\emptyset)\wedge\mathsf{Eq}(y,y)\}$ $x:=\mathcal{H}(y)$ $\{\mathsf{Ind}(x;\mathsf{Var},\ell_{\mathcal{H}};\mathsf{Var}-x)\wedge\mathsf{Eq}(x,x)\}$

(H3) $\{\mathsf{H}(\mathcal{H};y;\emptyset)\wedge\mathsf{Ind}(t;V;V')\}$ $x:=\mathcal{H}(y)$ $\{\mathsf{Ind}(t;V,x;V',x)\}$ even if $t=y$, provided $\ell_{\mathcal{H}}\notin V$

(H4) $\{\mathsf{H}(\mathcal{H};y;\emptyset)\wedge\mathsf{Ind}(t;V,\ell_{\mathcal{H}},y;V',y)\}$ $x:=\mathcal{H}(y)$ $\{\mathsf{Ind}(t;V,\ell_{\mathcal{H}},x,y;V',x,y)\}$

(H5) $\{\mathsf{H}(\mathcal{H};t;V,y)\}$ $x:=\mathcal{H}(y)$ $\{\mathsf{H}(\mathcal{H};t;V,y)\}$

For loop:

(F1) $\{\psi(p-1)\}$ for $l=p$ to q do: [cmd_l] $\{\psi(q)\}$ provided $\{\psi(l-1)\}$ cmd_l $\{\psi(l)\}$ for $p\leq l\leq q$

Table 1. Rules of our Hoare Logic

Proof. It should be clear that, since $\mathcal{L}_{\mathcal{E}}$ and $\mathcal{L}_{\mathcal{H}}$ are unchanged by the command, the following hold since the values of the variables in $V - x$ are unchanged by the command:

$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : (S(t), S(V - x) \cup S'(V' - x))] =$$
$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} [\![\mathsf{cmd}]\!] X : (S(t), S(V - x) \cup S'(V' - x))]$$
$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : (S'(t), S'(V - x) \cup S'(V' - x))] =$$
$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} [\![\mathsf{cmd}]\!] X : (S'(t), S'(V - x) \cup S'(V' - x))].$$

We can add back x to V (resp. V') when x is constructible from $V - t$ (resp. $V' - t$) using Lemma 3. It follows that $(X \models \mathsf{Ind}(t; V; V')) \Rightarrow ([\![\mathsf{cmd}]\!] X \models \mathsf{Ind}(t; V; V'))$.

Proposition 8 (Rule (G6)).
$\{\mathsf{Empty}\}$ *cmd* $\{\mathsf{Empty}\}$ *provided* *cmd* *is not* $x := \mathcal{E}(y)$

Proof. This is obvious since the command does not modify $\mathcal{L}_{\mathcal{E}}$.

B.3 Function ρ

Proposition 9 (Rule (P1)).
$\{\mathsf{Eq}(y)\}\ x := \rho(y)\ \{\mathsf{Eq}(x)\}$

Proof. This is a trivial consequence of the fact that ρ is a (deterministic) function.

B.4 Assignment

Proposition 10 (Rule (A1)).
$\{true\}\ x := m_i\ \{\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x) \vee \mathsf{Uneq}(x, x)\}$

Proof. This follows immediately from Lemma 2 and the fact that after the execution of the command, the value of x is the same as the value of m_i.

Proposition 11 (Rules (A2) to (A9)). *The following rules hold.*

- *(A2)* $\{\mathsf{Eq}(y, y)\}\ x := y\ \{\mathsf{Eq}(x, x)\}$
- *(A3)* $\{\mathsf{Uneq}(y, y)\}\ x := y\ \{\mathsf{Uneq}(x, x)\}$
- *(A4)* $\{\mathsf{Ind}(y; V; V')\}\ x := y\ \{\mathsf{Ind}(x; V; V')\}$ *provided* $y \notin V \cup V'$
- *(A5)* $\{\mathsf{E}(\mathcal{E}; y; V)\}\ x := y\ \{\mathsf{E}(\mathcal{E}; x; V)\}$ *provided* $y \notin V$
- *(A6)* $\{\mathsf{H}(\mathcal{H}; y; V)\}\ x := y\ \{\mathsf{H}(\mathcal{H}; x; V)\}$ *provided* $y \notin V$
- *(A7)* $\{\mathsf{E}(\mathcal{E}; t; V, y)\}\ x := y\ \{\mathsf{E}(\mathcal{E}; t; V, x, y)\}$
- *(A8)* $\{\mathsf{H}(\mathcal{H}; t; V, y)\}\ x := y\ \{\mathsf{H}(\mathcal{H}; t; V, x, y)\}$

Proof. The proofs of all those rules are trivial consequences of the fact that if X is any distribution, then, in $[\![x := y]\!] X$, the variables x and y will always be assigned the same value.

B.5 Concatenation

Proposition 12 (Rule (C1)).
$\{\mathsf{Eq}(y,y)\}\ x := y\|m_i\ \{(\mathsf{Eq}(m_i,m_i) \wedge \mathsf{Eq}(x,x)) \vee \mathsf{Uneq}(x,x)\}$

Proof. This is a clear consequence of Lemma 2.

Proposition 13 (Rule (C2)).
$\{\mathsf{Eq}(y,y) \wedge \mathsf{Eq}(z,z)\}\ x := y\|z\ \{\mathsf{Eq}(x,x)\}$

Proof. Trivial.

Proposition 14 (Rule (C3)).
$\{\mathsf{Uneq}(y,y)\}\ x := y\|z\ \{\mathsf{Uneq}(x,x)\}$

Proof. Trivial consequence of the fact that for any distribution X and $(S,S',\mathcal{E},\mathcal{H},\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X$, with overwhelming probability, $S(y) \neq S'(y)$, and, from Lemma 4, $|S(y)| = |S'(y)|$ implies that $S(y)\|S(z) \neq S'(y)\|S'(z)$.

Proposition 15 (Rule (C4)).
$\{\mathsf{Ind}(y;V,y,z;V') \wedge \mathsf{Ind}(z;V,y,z;V')\}\ x := y\|z\ \{\mathsf{Ind}(x;V,x;V')\}$ *provided* $x,y,z \notin V$ *and* $x \notin V'$ *unless* $y,z \in V'$ *and* $y \neq z$

Proof. We first consider the case where X be a distribution such that $X \models \mathsf{Ind}(y;V,y,z) \wedge \mathsf{Ind}(z;V,y,z)$ with $x,y,z \notin V$ and $x \notin V'$. We have that

$$
\begin{aligned}
&[(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} [\![x := y\|z]\!]X : (S(x), S((V,x)-x) \cup S'(V'))] \\
&\qquad = [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} [\![x := y\|z]\!]X : (S(x), S(V) \cup S'(V'))] \\
&\qquad = [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X : (S(y)\|S(z), S(V) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X, u_1 \stackrel{\$}{\leftarrow} \mathcal{U} : (u_1\|S(z), S(V) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X, u_1 \stackrel{\$}{\leftarrow} \mathcal{U}, u_2 \stackrel{\$}{\leftarrow} \mathcal{U} : (u_1\|u_2, S(V) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X, u \stackrel{\$}{\leftarrow} \mathcal{U}\mathcal{U} : (u, S(V) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} [\![x := y\|z]\!]X, u \stackrel{\$}{\leftarrow} \mathcal{U}\mathcal{U} : (u, S((V,x)-x) \cup S'(V'))]
\end{aligned}
$$

The first two equality are consequences of the fact that $x \notin V \cup V'$ and of the semantics of $x := y\|z$. The second to last line is true because, for strings u, u_1, u_2 of appropriate sizes, $[u_1, u_2 \stackrel{\$}{\leftarrow} \mathcal{U} : u_1\|u_2] = [u \stackrel{\$}{\leftarrow} \mathcal{U} : u]$. The last line follows from the fact that $x \notin V \cup V'$. So we only have left to justify the two lines in which $S(y)$ and $S(z)$ are replaced with uniform random values u_1 and u_2 respectively. Suppose there exists an adversary $\mathcal{A}$ that can break the following:

$$
\begin{aligned}
&[(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X : (S(y)\|S(z), S(V) \cup S'(V'))] \sim \\
&\qquad\qquad [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \stackrel{\$}{\leftarrow} X, u_1 \stackrel{\$}{\leftarrow} \mathcal{U} : (u_1\|S(z), S(V) \cup S'(V'))]
\end{aligned}
$$

Then we can construct an algorithm $\mathcal{B}$ that attacks the following:

$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : (S(y), S(V, z) \cup S'(V'))] \sim \\ [(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U} : (u, S(V, z) \cup S'(V'))].$$

On input (b, B), $\mathcal{B}$ runs algorithm $\mathcal{A}$ on input $(b \| a, B - a)$ where a is the value of the variable z in A. When $\mathcal{A}$ terminates, algorithm $\mathcal{B}$ outputs the same result as $\mathcal{A}$. It should be clear that $\mathcal{B}$ is successful into distinguishing its two distributions precisely when $\mathcal{A}$ does. So if $\mathcal{A}$ succeeds in distinguishing between its two distributions with non-negligible probability, so can $\mathcal{B}$, which violates our assumption that $X \models \mathsf{Ind}(y; V, y, z)$. We can show similarly that the following also holds:

$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X, u_1 \xleftarrow{\$} \mathcal{U} : (u_1 \| S(z), S(V) \cup S'(V'))] \sim \\ [(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X, u_1 \xleftarrow{\$} \mathcal{U}, u_2 \xleftarrow{\$} \mathcal{U} : (u_1 \| u_2, S(V) \cup S'(V'))].$$

The same argument can be applied with the roles of S and S' reversed, which completes the proof that $[\![x := y \| z]\!] X \models \mathsf{Ind}(x; V, x; V')$.

The case when $y, z \in V'$ is similar, the result follows from the argument above and Lemma 3.

Proposition 16 (Rules (C5) and (C6)).

(C5) $\{\mathsf{Ind}(y; V, \mathcal{L}_{\mathcal{E}}; \emptyset)\}\ x := y \| z\ \{\mathsf{E}(\mathcal{E}; x; V)\}$
(C6) $\{\mathsf{Ind}(y; V, \mathcal{L}_{\mathcal{H}}; \emptyset)\}\ x := y \| z\ \{\mathsf{H}(\mathcal{H}; x; V)\}$

Proof.

(C5) Let $\mathcal{A}$ be the algorithm which, on input (a, A), outputs 1 if and only if a is a prefix of one of the strings in A. We examine $\mathcal{A}$ advantage in breaking the following:

$$[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X; (S(y), S(V, \mathcal{L}_{\mathcal{E}})] \sim [(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U}; (u, S(V, \mathcal{L}_{\mathcal{E}}))].$$

Since $X \models \mathsf{Ind}(y; V, \mathcal{L}_{\mathcal{E}}; \emptyset)$, $\mathcal{A}$'s advantage in distinguishing the two distributions above must be negligible. Noting that the probability that A outputs 1 when given an input from the second distribution must be negligible (because u is sampled from a domain of size exponential in the security parameter), then we must that that the probability that A outputs 1 when given an output from the first distribution is negligible as well. That is, for $(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X$, the probability that $S(y)$ is a prefix of any string in $S(V, \mathcal{L}_{\mathcal{E}})$ is negligible. Thus, the probability that $S(y) \| S(z) = S(x) \in S(V, \mathcal{L}_{\mathcal{E}})$ is negligible. Similarly, we can find that the probability that $S'(y) \| S'(z) = S'(x) \in S'(V, \mathcal{L}_{\mathcal{E}})$ is negligible as well, which shows that $[\![x := S(y) \| S(z)]\!] X \models \mathsf{E}(\mathcal{E}; x; V)$.

(C6) The proof is similar to the proof of Rule (C5), but with $\mathcal{L}_{\mathcal{H}}$ instead of $\mathcal{L}_{\mathcal{E}}$.

B.6 Xor

Proposition 17 (Rule (X1)).
$\{\mathsf{Eq}(y,y)\}\ x := y \oplus m_i\ \{(\mathsf{Eq}(x,x) \wedge \mathsf{Eq}(m_i,m_i)) \vee \mathsf{Uneq}(x,x)\}$

Proof. This easily follows from Lemma 2.

Proposition 18 (Rule (X2)).
$\{\mathsf{Ind}(y;V,y,z;V')\}\ x := y \oplus z\ \{\mathsf{Ind}(x;V,x,z;V')\}$ *provided* $y \neq z$, $y \notin V$ *and* $x \notin V'$ *unless* $y, z \in V'$

Proof. This proof is similar to the proof of Rule (C4). Let X be a distribution such that $X \models \mathsf{Ind}(y;V,y,z)$ with $y \neq z$, $y \notin V$ and $x \notin V'$. We have that

$$
\begin{aligned}
&[(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := y \oplus z]\!]X : (S(x), S((V,x,z)-x) \cup S'(V'))] \\
&\qquad = [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := y \oplus z]\!]X : (S(x), S(V,z) \cup S'(V'))] \\
&\qquad = [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(y) \oplus S(z), S(V,z) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U} : (u \oplus S(z), S(V,z) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U} : (u, S(V,z) \cup S'(V'))] \\
&\qquad \sim [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := y \oplus z]\!]X, u \xleftarrow{\$} \mathcal{U} : (u, S((V,x,z)-x) \cup S'(V'))]
\end{aligned}
$$

All those lines are justified similarly to the proof of Rule (C4), except for the two lines in which $S(y)$ is replaced with a uniform random values u, and the line in which $u \oplus S(z)$ is replaced with u. The latter is easily justified by the fact that, for any random value independent from $S(z)$, the two distributions $[u \xleftarrow{\$} \mathcal{U}; u \oplus S(z)]$ and $[u \xleftarrow{\$} \mathcal{U}; u]$ are identical (under the condition that $y \neq z$).

As for the former, suppose there exists an adversary $\mathcal{A}$ that can break the following:

$$
\begin{aligned}
&[(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(y) \oplus S(z), S(V,z) \cup S'(V'))] \sim \\
&\qquad\qquad [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U} : (u \oplus S(z), S(V,z) \cup S'(V'))]
\end{aligned}
$$

Then we can construct an algorithm $\mathcal{B}$ that attacks the following:

$$
\begin{aligned}
&[(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : (S(y), S(V,z) \cup S'(V'))] \sim \\
&\qquad\qquad [(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X, u \xleftarrow{\$} \mathcal{U} : (u, S(V,z) \cup S'(V'))].
\end{aligned}
$$

On input (b, B), $\mathcal{B}$ runs algorithm $\mathcal{A}$ on input $(b \oplus a, B)$ where a is the value of the variable z in A. When $\mathcal{A}$ terminates, algorithm $\mathcal{B}$ outputs the same result as $\mathcal{A}$. It should be clear that $\mathcal{B}$ is successful into distinguishing its two distributions precisely when $\mathcal{A}$ does. So if $\mathcal{A}$ succeeds in distinguishing between its two distributions with non-negligible probability, so can $\mathcal{B}$, which violates our assumption that $X \models \mathsf{Ind}(y;V,y,z;V')$.

The same argument can be applied with the roles of S and S' reversed, which completes the proof that $[\![x := y \oplus z]\!]X \models \mathsf{Ind}(x; V, x, z; V')$.

The case when $y, z \in V'$ is similar, the result follows from the argument above and Lemma 3.

Proposition 19 (Rule (X3)).
$\{\mathsf{Eq}(y, y) \wedge \mathsf{Eq}(z, z)\}\ x := y \oplus z\ \{\mathsf{Eq}(x, x)\}$

Proof. Trivial.

Proposition 20 (Rule (X4)).
$\{\mathsf{Eq}(y, y) \wedge \mathsf{Uneq}(z, z)\}\ x := y \oplus z\ \{\mathsf{Uneq}(x, x)\}$

Proof. Trivial.

B.7 Block Cipher

For many of the proofs of rules involving the evaluation of the block cipher, we use the fact that, in the ideal cipher model, the block cipher is modeled as a perfectly random function. As a result, if the block cipher has not yet been evaluated at a given point, then the value of the block cipher at that point is indistinguishable from an independent random value. This is due to the fact that the distinguishing adversary does not have any access to $\mathcal{E}$.

Proposition 21 (Rules (B1), (B2) and (B3)).

(B1) $\{\mathsf{Empty}\}\ x := \mathcal{E}(m_i)\ \{(\mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x) \wedge$
$\mathsf{Ind}(x; \mathit{Var}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}; \mathit{Var} - x)) \vee (\mathsf{Uneq}(x, x) \wedge \mathsf{Ind}(x))\}$
(B2) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Uneq}(y, y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(x)\}$
(B3) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Eq}(y, y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(x; \mathit{Var}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}; \mathit{Var} - x) \wedge \mathsf{Eq}(x, x)\}$

Proof.

(B1) Since $X \models \mathsf{Empty}$, we know that, with overwhelming probability, $\mathcal{E}(S(m_i))$ and $\mathcal{E}(S'(m_i))$ have never been computed before. Following Lemma 2, we either have $X \models \mathsf{Eq}(m_i, m_i)$ or $X \models \mathsf{Uneq}(m_i, m_i)$. We consider each case separately:
- if $S(m_i) \neq S'(m_i)$, i.e. $X \models \mathsf{Uneq}(m_i, m_i)$, and since neither is in $\mathcal{L}_{\mathcal{E}}.\mathsf{dom}$, then both $\mathcal{E}(S(m_i))$ and $\mathcal{E}(S'(m_i))$ look random and independent from all other values (just as if they had both been sampled randomly and independently), so $[\![x := \mathcal{E}(y)]\!]X \models \mathsf{Ind}(x)$ is immediate. It should be clear that, in this case, $\mathsf{Uneq}(m_i, m_i)$ is preserved by $x := \mathcal{E}(m_i)$.
- if $S(m_i) = S'(m_i)$, that is $X \models \mathsf{Eq}(m_i, m_i)$, then clearly $[\![x := \mathcal{E}(m_i)]\!]X \models \mathsf{Eq}(m_i, m_i) \wedge \mathsf{Eq}(x, x)$ since $\mathcal{E}$ is a function. As before, $S(m_i), S'(m_i) \notin \mathcal{L}_{\mathcal{E}}.\mathsf{dom}$, so $\mathcal{E}(S(m_i)$ is indistinguishable from a random and independent value even given all other values in the system, values except for $\mathcal{E}(S'(m_i))$, to which it is equal. So $[\![x := \mathcal{E}(y)]\!]X \models \mathsf{Ind}(x; \mathsf{Var}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}; \mathsf{Var} - x)$ is also clear.

(B2) Since $\mathsf{Uneq}(y, y)$ is given here, this is exactly the first case of the proof of Rule (B1).

(B3) Since $\mathsf{Eq}(y, y)$ is given here, this is exactly the second case of the proof of Rule (B1).

Proposition 22 (Rule (B4)).
$\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Ind}(t; V; V')\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(t; V, x; V', x)\}$ *provided* $\mathcal{L}_\mathcal{E} \notin V$*, even if* $t = y$

Proof. Since $X \models \mathsf{E}(\mathcal{E}; y; \emptyset)$, for any $(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := \mathcal{E}(y)]\!]X$, any adversary $\mathcal{A}$ that successfully distinguishes t from a random value given $S(V, x) \cup S'(V', x)$ could be simulated by an algorithm which, given only $S(V) \cup S'(V')$, samples a uniform random u and runs $\mathcal{A}(t, S(V) \cup S'(V') \cup \{u\})$ (this is for the case in which $S(y) = S'(y)$, we would need two random values if $S(y) \neq S'(y)$ but the argument is the same), which would contradict $X \models \mathsf{Ind}(t; V; V)$. The same can be argued with the roles of S and S' reversed.

Proposition 23 (Rules (B5)).

(B5) $\{\mathsf{E}(\mathcal{E}; y; \emptyset) \wedge \mathsf{Ind}(t; V, \mathcal{L}_\mathcal{E}, y; V', y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Ind}(t; V, \mathcal{L}_\mathcal{E}, x, y; V', x, y)\}$

Proof. This is a simple consequence of the fact that, while the values of y (through both S and S') get added to $\mathcal{L}_\mathcal{E}.dom$, this does not change anything to the sets $S(V, \mathcal{L}_\mathcal{E}, y) \cup S'(V', y)$ and $S'(V, \mathcal{L}_\mathcal{E}, y) \cup S(V', y)$ since the values of y were already included in both. The addition of x in $\mathsf{Ind}(t; V, \mathcal{L}_\mathcal{E}, x, y; V', x, y)$ can be proven in the same way as in the proof of Rule (B4).

Proposition 24 (Rule (B6)).
$\{\mathsf{E}(\mathcal{E}; t; V, y)\}\ x := \mathcal{E}(y)\ \{\mathsf{E}(\mathcal{E}; t; V, y)\}$

Proof. Clearly, $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : \{S(x), S'(x)\} \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V, y) \cup S'(V, y)] = \mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := \mathcal{E}(y)]\!]X : S(x) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V, y) \vee S'(x) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S'(V, y)]$ because, since $S(y), S'(y) \in S(V, y) \cup S'(V, y)$, adding $S(y), S'(y)$ to $\mathcal{L}_\mathcal{E}.dom$ will not change the set $\mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V, y) \cup S'(V, y)$.

B.8 Hash Function

All the proofs for hash function computation are essentially the same as the proofs for block cipher evaluation. This is due to our choice of using an adversary that does not have access to the random oracle when trying to distinguish distributions (see Section 3).

Proposition 25 (Rules (H1) to (H5)).

(H1) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Uneq}(y, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(x)\}$
(H2) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Eq}(y, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(x; \mathit{Var}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}, \mathit{Var} - x) \wedge \mathsf{Eq}(x, x)\}$
(H3) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Ind}(t; V; V')\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(t; V, x; V', x)\}$ *provided* $\mathcal{L}_\mathcal{H} \notin V$*, even if* $t = y$
(H4) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Ind}(t; V, \mathcal{L}_\mathcal{H}, y; V', y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Ind}(t; V, \mathcal{L}_\mathcal{H}, x, y; V', x, y)\}$
(H5) $\{\mathsf{H}(\mathcal{H}; t; V, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{H}(\mathcal{H}; t; V, y)\}$

Proof. All the proofs for hash function computation are essentially the same as the proofs for block cipher evaluation. This is due to our choice of using an adversary that does not have access to the random oracle when trying to distinguish distributions (see Section 3).

B.9 For Loop

Proposition 26 (Rule (F1)).
$\{\psi(i-1)\}$ *for* $x = i$ *to* j *do:* c_x $\{\psi(j)\}$ *provided* $\{\psi(k-1)\}$ c_k $\{\psi(k)\}$ *for* $i \leq k \leq j$

Proof. This is a simple induction on x.

C Example with 2 Block CBC

We show below the application of our logic on a program describing $Hash_{CBC}$ for a two block message, with the loop unrolled. We can see that the invariant at the end implies $UNIV(2)$: in the first two clauses, $\mathsf{Uneq}(c_2, c_1) \wedge \mathsf{Uneq}(c_2, c_2)$ is implied by the predicate $\mathsf{Ind}(c_2; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var})$, and in the third, we have equality of all the message blocks and $\mathsf{Ind}(c_2; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_2)$ which implies $\mathsf{Uneq}(c_2, c_1)$. For simplicity, we only present the invariants that are necessary to the analysis.

	(Init)	$\{\mathsf{Empty}\}$
$c_1 := \mathcal{E}(m_1);$	(B1)	$\{(\mathsf{Uneq}(c_1, c_1) \wedge \mathsf{Ind}(c_1; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var}))\vee$
		$(\mathsf{Eq}(m_1, m_1) \wedge \mathsf{Eq}(c_1, c_1) \wedge \mathsf{Ind}(c_1; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_1))\}$
$z_2 := c_1 \oplus m_2;$	(G5)(X2)	$\{(\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var}) \wedge \mathsf{Ind}(z_2; \mathsf{Var} - c_1, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var}))\vee$
		$(\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2)\wedge$
		$\mathsf{Ind}(z_2; \mathsf{Var} - c_1, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2)\wedge$
	(G1)(X1)	$\mathsf{Eq}(m_1, m_1) \wedge \mathsf{Uneq}(z_2, z_2))\vee$
		$(\mathsf{Eq}(m_1, m_1) \wedge \mathsf{Eq}(m_2, m_2) \wedge \mathsf{Eq}(z_2, z_2)\wedge$
		$\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2) \wedge$
		$\mathsf{Ind}(z_2; \mathsf{Var} - c_1, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2))\}$
$c_2 := \mathcal{E}(z_2)$	(B2)(B4)	$\{(\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{H}}; \mathsf{Var}) \wedge \mathsf{Ind}(c_2; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var}))\vee$
		$(\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2) \wedge$
	(G1)	$\mathsf{Eq}(m_1, m_1) \wedge \mathsf{Ind}(c_2; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var}))\vee$
	(B3)	$(\mathsf{Eq}(m_1, m_1) \wedge \mathsf{Eq}(m_2, m_2) \wedge \mathsf{Ind}(c_2; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var} - c_2)\wedge$
		$\mathsf{Ind}(c_1; \mathsf{Var} - z_2, \ell_{\mathcal{H}}; \mathsf{Var} - c_1 - z_2))\}$

D Prototype

For now, this prototype is meant only as a proof of concept for our method. We describe the steps required to use the prototype to verify $Hash_{CBC}$. We refer to Appendix A for the program of this function.

Variable Declaration
First, we must declare the variables of the program: $c_1, m_1, i, n, z_i, c_{i-1}, m_i$ and c_i. Our prototype uses 3 types of variables: OrdinaryVar, IndexedVar and ParamVar. `OrdinaryVar`

are those variables that are described only with a string, and do not have an index. For example, in the program, i and n are OrdinaryVar. `IndexedVar` are variable that have an integer index. For example, c_1 and m_1 are IndexedVar. The constructor for an IndexedVar takes a variable and an integer as input. `ParamVar` are variables that have a parameter, a variable and an offset, as an index. For example, c_{i-1}, c_i and m_i are ParamVar. The constructor for a ParamVar takes a string, a variable and an integer as input.

Here are a few examples of variable declarations:

```
let i = OrdinaryVar "i";;
let c1 = IndexedVar("c",1);;
let ci = ParamVar("c",i,0);;
let ciMinusOne = ParamVar("c",i,-1);;
```

Description of the Program

Our programming language supports 5 operations: block cipher, hash, exclusive or, concatenation and the ρ function. Operations correspond to the right-hand side of most commands. These operations have the following constructors: `Block(bc,op)` takes a block cipher `bc` and an operand `op` as input. In our case, only one block cipher is used and it is always the string "bc". `Hash(h,op)` takes a hash function `h` and an operand `op` as an input. Again, in our case, only one hash function is used and it is always the string "h". `Xor(op1,op2)` takes 2 operands and is the exclusive or operation. `Conc(op1,op2)` takes 2 operands and is the concatenation operation. `Rho(int,op)` takes an integer and an operand and is the i-fold application of the ρ function.

The instructions of the program are either `For` commands, describing a for-loop, or a `SimpleCmd`, which describes any other command. A SimpleCmd consists of a target variable and an operation. For example, the command $x := y \oplus z$ is described by `SimpleCmd(x,Xor(y, z))`.

A program is simply a list of commands. For example, the program describing commands contained in the For loop of $Hash_{CBC}$ would be described as follows:

```
let lcom1=SimpleCmd(zi,Xor(ciMinusOne,mi));;
let lcom2=SimpleCmd(ci,Block(BC "bc",zi));;
let forLoopPrg=[lcom1;lcom2];;
```

A For loop is described by two bounds, a variable, the loop variable, and a program, the body of the loop. Bounds are either an IntBound, described only by an integer, or a ParamBound, that take a variable and an offset. For example, the For loop in $Hash_{CBC}$ can be described by `For(IntBound 2,ParamBound(n,0),i,[lcom1;lcom2])`.

Using all this, we can describe the full code of $Hash_{CBC}$ as follows:

```
let com1 = SimpleCmd(c1, Block(BC "bc",m1));;
let lcom1=SimpleCmd(zi,Xor(ciMinusOne,mi));;
let lcom2=SimpleCmd(ci,Block(BC "bc",zi));;
let forLoopPrg = [lcom1;lcom2];;
let com2 = For(IntBound 2,ParamBound(n,0),i,forLoopPrg);;
let hash_cbc = [com1;com2];;
```

Testing a program

The function that attempts to prove the security of an almost-universal hash function by applying all our Hoare rules is `prove_prg`. This function takes as input an initial predicate, a desired postcondition and a program. The predicates are expressed in disjunctive normal form, the inner lists are conjunctions of all the invariants they contain, and the outer lists are disjunctions of all the conjunctions within. The initial predicate is generally the (init) predicate given previously in this paper, and is described as follows:

```
let init = [[ Empty;Indis(k,var,VarMinus([ Var(k) ]),true,true);
              Equal(k,k) ]];;
```

In this predicate, the notation `VarMinus([ Var(k) ])` denotes the set that contains all the variables except for `k`. The predicate we want at the end of the program is $UNIV(n)$, and it is described as follows:

```
let post = [[ Unequal(cn,cn);
              BigAnd(IntBound 1,ParamBound(n,-1),Unequal(cn,c_bav))) ];
            [ BigAnd(IntBound 1,ParamBound(n,0),Equal(m_bav,m_bav));
              BigAnd(IntBound 1,ParamBound(n,-1),Unequal(cn,c_bav)) ]];;
```

In this description, `BigAnd(b1,b2,invr)` is a predicate that takes two bounds and a predicate and represents $\bigwedge_{bigAndVar=b1}^{b2} invr$, and `c_bav` and `m_bav` are ParamVar that have the OrdinaryVar "bigAndVar" as an index. We note that our prototype assumes that the variable "bigAndVar" is always the index variable in a predicate `BigAnd`, so it is important that this particular variable is used as the index.

Using the code we have given here, we can obtain the following output for $Hash_{CBC}$ by simply executing `prove_prg init post hash_cbc;;`.

```
[[[Empty;
   Indis(OrdinaryVar "k",VarMinus [],
         VarMinus [Var(OrdinaryVar "k")],true,true);
   Equal(OrdinaryVar "k",OrdinaryVar "k")]];
 [[Indis(IndexedVar("c",1),VarMinus [],VarMinus [],true,true)];
  [Indis(IndexedVar("c",1),VarMinus [],
         VarMinus [Var(IndexedVar("c",1))],true,true);
   Equal(IndexedVar("c",1),IndexedVar("c",1));
   BigAnd(IntBound 1,IntBound 1,
          Equal(ParamVar("m",OrdinaryVar "bigAndVar",0),
                ParamVar("m",OrdinaryVar "bigAndVar",0)))]];
 [[Indis(ParamVar("c",OrdinaryVar "n",0),VarMinus[],
         VarMinus [Var(ParamVar("c",OrdinaryVar "n",0))],true,true);
   BigAnd(IntBound 1,ParamBound(OrdinaryVar "n",0),
          Equal(ParamVar("m",OrdinaryVar "bigAndVar",0),
                ParamVar("m",OrdinaryVar "bigAndVar",0)))];
  [Indis(ParamVar("c",OrdinaryVar "n",0),VarMinus [],
         VarMinus [],true,true)]]]
```

This is a trace of all invariants generated automatically for proving the security of $Hash_{CBC}$ with our prototype. More generally, this function outputs an empty list if it is unsuccessful in finding a proof of the scheme, and outputs a list of DNF predicates in which the first is `init`, the i^{th} predicate is the predicate obtained after executing the i^{th} command, and the last predicate implies `post`.