Automated Security Proofs for Message Authentication Codes*

Martin Gagné, Pascal Lafourcade, and Yassine Lakhnech

Laboratoire VERIMAG
Université Grenoble 1, CNRS, FRANCE
firstname.lastname@imag

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. We present a method for automatically proving the security for block-cipher-based and hash-based MACs in the ideal cipher model. Our method proceeds in two steps, following the traditional method for constructing MACs. 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. 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 quite expressive. As a result, our logic can be used to prove functions based on block ciphers and hash functions. Second, we 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.

Using our method, we implemented a tool that can prove the security of many CBC-based MACs (DMAC, ECBC, FCBC and XCBC to name only a few), PMAC and HMAC.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Cryptography, Message authentication code, Hoare Logic

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

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 secure encryption, to obtain chosen-ciphertext 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, 8, 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 prove the security of MACs, we first break the MAC algorithms into two parts: a 'front-end', whose work is to compress long input messages 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

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


© M. Gagné, P. Lafourcade and Y. Lakhnech;
licensed under Creative Commons License NC-ND
Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that the front-ends of MACs are almost-universal hash functions. We then make a list of operations which, when composed with an almost-universal hash function, yield a secure MAC.

Our result differs significantly from our previous work 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 since the predicates required to model the security of authentication are far more complicated than those necessary to prove the security of encryption. In particular, we have to consider the simultaneous execution of a program on two input messages in order to bound collision probabilities. As a result, we propose a new semantics, and define new invariants for determining equality or inequality of values. We also present a treatment of for loops, which allows us to prove the security of protocols that can take arbitrary strings as an input. This is an important improvement over previous results that only deal with schemes that had fixed-length inputs.

Finally, we implemented our method into a tool [14] that can be used to prove the security of several well-known MACs, such as HMAC [4], DMAC [17], ECBC, FCBC and XCBC [8] and PMAC [9].

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. This work was continued by Gagné et al. [15], who showed a Hoare logic for verifying 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. However, they rely on a game-based approach and require human assistance to enter the sequence of games. In contrast, our method is fully automatic.

Outline: In Section 3, 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

Notation and Conventions

Throughout this paper, 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.

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

▶ **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,[1] $MAC(sk, m)$ takes a secret key and a message m, and outputs a *tag*, and $V(sk, m, tag)$ takes a secret key, a message and a tag, and outputs a bit: 1 for a correct tag, 0 otherwise.

▶ **Definition 2** (Unforgeability)**.** A MAC (K, MAC, V) is unforgeable under a chosen-message attack (UNF-CMA) if for every oracle polynomial-time algorithm $\mathcal{A}$ 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]. Our verification technique assumes that the MAC is constructed in this way.

▶ **Definition 3** (Almost-Universal Hash)**.** A family of functions $\mathcal{H} = \{h_i\}_{i\in\{0,1\}^\eta}$ is a family of almost-universal hash functions if for any two strings a and b, $\mathsf{Pr}_{h_i\in\mathcal{H}}[h_i(a) = h_i(b)]$ 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 because the adversary against an almost-universal hash function is non-adaptive, and must output his attempt at finding a collision in the almost-universal hash function independently from the choice of the key.

Block Cipher Security and Random Oracle Model

Many MAC constructions are based on block cipher and fixed-input hash functions, and their security are proven in the ideal cipher model and the random oracle model respectively. Due to space restrictions, we refer a reader unfamiliar with these concepts to Appendix A.

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

3.1 Grammar

We consider the language defined by the following BNF grammar.

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

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. We omit the key used every time to simplify the notation, but it is understood that a key was selected at random at the beginning of the experiment and remains the same throughout.
- $x := \mathcal{H}(y)$ denotes the application of the hash function $\mathcal{H}$ to the value of y and assigning the result to x.
- $x := y \oplus z$ 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.

[1] The secret key sk can consist of one or several strings, depending on the MAC.

- $x := \rho^i(y)$ denotes the i-fold application of the function ρ to the value of y (that is, $\rho(\ldots(\rho(y)\ldots))$, where ρ is repeated i times) and assigning the result to x.
- for $l = p$ to q do: $[\mathsf{cmd}_l]$ denotes the successive execution of $\mathsf{cmd}_p, \mathsf{cmd}_{p+1}, \ldots, \mathsf{cmd}_q$.
- $c_1; c_2$ is the sequential composition of c_1 and c_2.

The function ρ is used to compute the *tweak* in *tweakable block ciphers* ([16]). The function used to compute this tweak can vary from one protocol to the next, so we only specify that it must be a public function. 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.

We assume that, prior to executing the MAC, the message has been padded using some unambiguous padding scheme, so that all the message blocks $m_1, \ldots, m_i$ are of equal and appropriate length for the scheme, usually the input length of the block cipher. We denote by Var the full set of variables used in the program. This set always includes input and output variables, and the special variable k that contains a secret key. The set **var** $\vec{x_i}$ contains the variables of the programs that are neither input variables, output variables or the secret key.

▶ **Definition 4** (Generic Hash Function)**.** A generic hash function $Hash$ on i message blocks $m_1, \ldots, m_i$ and with output c, is represented by a tuple $(\mathcal{F_E}, \mathcal{F_H}, Hash(m_1\| \ldots \|m_i, c) :$ **var** $\vec{x_i}$; $\mathsf{cmd}_i)$, where $\mathcal{F_E}$ is a family of pseudorandom permutations (usually a block cipher), $\mathcal{F_H}$ is a family of cryptographic hash functions, and $Hash(m_1\| \ldots \|m_i, c) :$ **var** $\vec{x_i}; \mathsf{cmd}_i$ is the code of the hash function, where the commands of cmd_i are built using the grammar described above.

Using this formalism, we describe the hash functions $Hash_{CBC}$, which is used in DMAC [17] and ECBC [8]:

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

Due to space restrictions, we put the description of $Hash_{CBC'}$, $Hash_{PMAC}$ and $Hash_{HMAC}$, which are used in FCBC, XCBC [8], PMAC [9] and HMAC [4], in Appendix B.

3.2 Semantics

In our analysis, we consider the execution of a program on two inputs simultaneously. These simultaneous executions will enable us in Section 4 to define predicates about the equality or inequality of intermediate values in the program.

A program takes as input a *configuration* $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L_E}, \mathcal{L_H})$ and yields a distribution on configurations. A configuration is composed of two states S and S', a block cipher $\mathcal{E}$, a hash function $\mathcal{H}$, and two lists of pairs $\mathcal{L_E}$ and $\mathcal{L_H}$. The two states S and S', which are functions that take a variable as input and return a value in $\{0,1\}^* \cup \{\perp\}$ (the symbol $\perp$ indicates that no value has been assigned to the variable yet), assign values to all the variables in each of the two simultaneous executions of the program. The lists $\mathcal{L_E}$ and $\mathcal{L_H}$ record the values for which the functions $\mathcal{E}$ and $\mathcal{H}$ were computed respectively. These lists are common to both executions of the program. We denote by $\mathcal{L_E}.dom$ and $\mathcal{L_E}.res$ the lists obtained by projecting each pair in $\mathcal{L_E}$ to its first and second element respectively. We define $\mathcal{L_H}.dom$ and $\mathcal{L_H}.res$ similarly.

Let Γ denote the set of configurations and $\textsc{Dist}(\Gamma)$ the set of distributions on configurations. The semantics is given in Table 1, where $\delta(x)$ denotes the Dirac measure, i.e. $\Pr[x] = 1$, $S\{x \mapsto v\}$ denotes the state which assigns the value v to the variable x, and behaves like S for all other variables, $\mathcal{L_E} \cdot (x, y)$ denotes the addition of element (x, y) to $\mathcal{L_E}$ and $\circ$ denotes function composition. The semantic function $\phi : \Gamma \to \textsc{Dist}(\Gamma)$ of commands can be lifted in the usual way to a function $\phi^* : \textsc{Dist}(\Gamma) \to \textsc{Dist}(\Gamma)$ by point-wise application of ϕ. 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 \left\{ \begin{array}{l}
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \text{ if } (S(y), v), (S'(y), v') \in \mathcal{L}_\mathcal{E} \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E} \cdot (S(y), v), \mathcal{L}_\mathcal{H}) \\
\qquad\qquad \text{if } (S(y), v) \notin \mathcal{L}_\mathcal{E}, (S'(y), v') \in \mathcal{L}_\mathcal{E} \text{ and } v = \mathcal{E}(S(y)) \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E} \cdot (S'(y), v'), \mathcal{L}_\mathcal{H}) \\
\qquad\qquad \text{if } (S(y), v) \in \mathcal{L}_\mathcal{E}, (S'(y), v') \notin \mathcal{L}_\mathcal{E} \text{ and } v' = \mathcal{E}(S'(y)) \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, (\mathcal{L}_\mathcal{E} \cdot (S(y), v)) \cdot (S'(y), v'), \mathcal{L}_\mathcal{H}) \\
\qquad\qquad \text{if } (S(y), v), (S'(y), v') \notin \mathcal{L}_\mathcal{E} \text{ and } v = \mathcal{E}(S(y)), v' = \mathcal{E}(S'(y))
\end{array} \right. \\
[\![x := \mathcal{H}(y)]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \\
\quad \left\{ \begin{array}{l}
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \text{ if } (S(y), v), (S'(y), v') \in \mathcal{L}_\mathcal{H} \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H} \cdot (S(y), v)) \\
\qquad\qquad \text{if } (S(y), v) \notin \mathcal{L}_\mathcal{H}, (S'(y), v') \in \mathcal{L}_\mathcal{H} \text{ and } v = \mathcal{H}(S(y)) \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H} \cdot (S'(y), v')) \\
\qquad\qquad \text{if } (S(y), v) \in \mathcal{L}_\mathcal{H}, (S'(y), v') \notin \mathcal{L}_\mathcal{H} \text{ and } v' = \mathcal{H}(S'(y)) \\
\delta(S\{x \mapsto v\}, S'\{x \mapsto v'\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, (\mathcal{L}_\mathcal{H} \cdot (S(y), v)) \cdot (S'(y), v')) \\
\qquad\qquad \text{if } (S(y), v), (S'(y), v') \notin \mathcal{L}_\mathcal{H} \text{ and } v = \mathcal{H}(S(y)), v' = \mathcal{H}(S'(y))
\end{array} \right. \\
[\![x := y \oplus z]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \delta(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}) = \delta(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(t)]\!](S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) = \delta(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}\colon [\mathsf{cmd}_l]]\!] = [\![\mathsf{cmd}_q]\!] \circ [\![\mathsf{cmd}_{q-1}]\!] \circ \ldots \circ [\![\mathsf{cmd}_p]\!] \\
[\![c_1; c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]
\end{array}
$$

Table 1 The semantics of the programming language

Here, we are only interested in the distributions that can be constructed in polynomial time by an adversary having access only to the random oracle. We denote their set by $\textsc{Dist}(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$, where $\mathcal{F}_\mathcal{E}$ is a family of block ciphers, $\mathcal{F}_\mathcal{H}$ is a family of hash functions, and is defined as the set of distributions of the form:

$$
[\mathcal{E} \xleftarrow{\$} \mathcal{F}_\mathcal{E}(1^\eta); \mathcal{H} \xleftarrow{\$} \mathcal{F}_\mathcal{H}(1^\eta); u \xleftarrow{\$} \{0,1\}^\eta; (S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} \mathcal{A}^\mathcal{H}(1^\eta) : \\
(S\{k \mapsto u\}, S'\{k \mapsto u\}, \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H})]
$$

where k is a variable holding a secret string needed in some MACs (among our examples, $Hash_{PMAC}$ and $Hash_{HMAC}$ need it) and $\mathcal{A}$ is a probabilistic polynomial-time algorithm with oracle access to the hash function, such that $\mathcal{L}_\mathcal{H}$ contains the list of queries made by $\mathcal{A}$ to the random oracle, and $\mathcal{L}_\mathcal{E}$ is empty since $\mathcal{A}$ does not have access the key of the block cipher.

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

We give an intuitive description of the six invariants used in our logic:
Empty: means that the probability that $\mathcal{L}_\mathcal{E}$ contains an element is negligible.
$\mathsf{Equal}(\mathbf{x}, \mathbf{y})$: means that the probability that $S(x) \neq S'(y)$ is negligible.
$\mathsf{Unequal}(\mathbf{x}, \mathbf{y})$: means that the probability that $S(x) = S'(y)$ is negligible.
$\mathsf{E}(\mathcal{E}; \mathbf{x}; \mathbf{V})$: means that the probability that the value of x is either in $\mathcal{L}_\mathcal{E}.\mathsf{dom}$ or in V is negligible.
$\mathsf{H}(\mathcal{H}; \mathbf{x}; \mathbf{V})$: means that the probability that the value of x is either in $\mathcal{L}_\mathcal{H}.\mathsf{dom}$ or in V is negligible.

$\mathsf{Indis}(\mathbf{x}; \mathbf{V}; \mathbf{V'})$: 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 $\mathcal{L}_{\mathcal{E}}.dom$, similarly for $\ell_{\mathcal{H}}$.

In the following, 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. We also use $S(V, \ell_{\mathcal{E}})$ as a shorthand for $S(V) \cup \mathcal{L}_{\mathcal{E}}.dom$, and similarly for $S(V, \ell_{\mathcal{H}})$ and $S(V, \ell_{\mathcal{E}}, \ell_{\mathcal{H}})$. For a set V and a variable, we write V, x as a shorthand for $V \cup \{x\}$ and $V - x$ as a shorthand for $V \setminus \{x\}$ and we use $\mathsf{Indis}(x)$ as a shorthand for $\mathsf{Indis}(x; \mathsf{Var}, \ell_{\mathcal{E}}, \ell_{\mathcal{H}}; \mathsf{Var})$.

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

$\varphi ::= \mathsf{true} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \psi$

$\psi ::= \mathsf{Indis}(x; V; V') \mid \mathsf{Equal}(x, y) \mid \mathsf{Unequal}(x, y) \mid \mathsf{Empty} \mid \mathsf{E}(\mathcal{E}; x; V) \mid \mathsf{H}(\mathcal{H}; x; V)$

where $x, y \in \mathsf{Var}$ and $V, V' \subseteq \mathsf{Var}$, except for $\mathsf{Indis}(x; V; V')$ where $V \subseteq \mathsf{Var} \cup \{\ell_{\mathcal{E}}, \ell_{\mathcal{H}}\}$.

More formally, we define that a distribution X satisfies ψ, denoted $X \models \psi$ as follows:

- $X \models \mathsf{true}$, $X \models \varphi \wedge \varphi'$ 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 $\mathsf{Pr}[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : \mathcal{L}_{\mathcal{E}} \neq \emptyset]$ is negligible
- $X \models \mathsf{Equal}(x, y)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : S(x) \neq S'(y)]$ is negligible
- $X \models \mathsf{Unequal}(x, y)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : S(x) = S'(y)]$ is negligible
- $X \models \mathsf{E}(\mathcal{E}; x; V)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : \{S(x), S'(x)\} \cap (\mathcal{L}_{\mathcal{E}}.\mathsf{dom} \cup S(V - x) \cup S'(V - x)) \neq \emptyset]$ is negligible[2]
- $X \models \mathsf{H}(\mathcal{H}; x; V)$ iff $\mathsf{Pr}[(S, S', \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}) \xleftarrow{\$} X : \{S(x), S'(x)\} \cap (\mathcal{L}_{\mathcal{H}}.\mathsf{dom} \cup S(V - x) \cup S'(V - x)) \neq \emptyset]$ is negligible
- $X \models \mathsf{Indis}(x; V; V')$ iff the two following formulas hold:

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

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

▶ **Lemma 5.** *The following relations are true for any sets V_1, V_2, V_3 and variables x, y with $x \neq y$*

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

The proof of this lemma is in Appendix C. Note that results 4, 5 and 6 are particularly helpful because the invariant Indis is much easier to propagate than the other invariants.

[2] 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 $\mathsf{Pr}[\{S(x), S'(x)\} \cap (\mathcal{L}_{\mathcal{E}}.\mathsf{dom} \cup S(\{x\}) \cup S'(\{x\})) \neq \emptyset]$ never holds, and to simplify the notation. The same is also used for invariants $\mathsf{H}(\mathcal{H}; x; V)$ and $\mathsf{Indis}(x; V; V')$.

4 Proving Almost-Universal Hash

The main contribution of this paper is the procedure to prove that a program is an almost-universal hash function. To do this, 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. Under this assumption, we determine the condition under which a program computes an almost-universal hash function family.

▶ Proposition 6. A generic hash $(\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, P(m_1 \| \dots \| m_l, c) : \textbf{var}\ \vec{x_l}; \mathsf{cmd}_l)$ is an almost-universal hash function on string of length at most l blocks if, at the end of execution, the following invariants hold:

$$UNIV(l) = \left(\mathsf{Unequal}(c_l, c_l) \vee \bigwedge_{j=1}^{i} \mathsf{Equal}(m_j, m_j)\right) \wedge \bigwedge_{i=1}^{l-1} \mathsf{Unequal}(c_l, c_i)$$

The intuition for this is that if m consists of l message blocks and m' consists of $n \leq l$ message blocks, then the fact that the probability that the hashes of m_1 and m_2 are equal is negligible, when the probability is taken over the choice of the key, is captured exactly by the invariant $\mathsf{Unequal}(c_l, c_n)$. The proof of this theorem is in Appendix C.

We prove that this predicate holds at the end of execution by propagating and generating invariants through the program using rules of our Hoare Logic.

Hoare Logic Rules

We present a set of rules of the form $\{\varphi\}\mathsf{cmd}\{\varphi'\}$, meaning that execution of command cmd in any distribution that satisfies φ leads to a distribution that satisfies φ'. Using Hoare logic terminology, this means that the triple $\{\varphi\}\mathsf{cmd}\{\varphi'\}$ is valid. We group rules together according to their corresponding commands. The proofs of soundness of our rules are given in Appendix C.

Since the invariants $\mathsf{Equal}(m_i)$ 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 that point is unnecessary. For this reason, when we design our rules, we never produce the invariants $\mathsf{Unequal}(m_i, m_i)$ even when they would be correct.

In all the rules below, unless indicated otherwise, we assume that $t \notin \{x, y, z\}$ and $x \notin \{y, z\}$. In addition, for all rules involving the invariant Indis, we assume that $\ell_\mathcal{E}$ and $\ell_\mathcal{H}$ can be among the elements in the set V.

Initialization:

The following predicate holds at the beginning of execution of the program. The string k is part of the secret key sk of the MAC.

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

Generic preservation rules:

The following rules show how invariants are preserved by most of the commands when the invariants 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) $\{\mathsf{Equal}(t, t')\}\ \mathsf{cmd}\ \{\mathsf{Equal}(t, t')\}$

(G2) $\{\mathsf{Unequal}(t, t')\}\ \mathsf{cmd}\ \{\mathsf{Unequal}(t, t')\}$

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

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

(G5) $\{\mathsf{Indis}(t; V; V')\}\ \mathsf{cmd}\ \{\mathsf{Indis}(t; V; V')\}$ 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 $V' - t$

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

Function ρ:
Since the details of the function ρ are not known in advance, we cannot infer many rules, other than the following, which is a simple consequence of the fact that ρ is a function.

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

Assignment:
Most of the rules for the assignment follow simply from the fact that the value of x is equal to the value of y.

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

Concatenation:
The most important rule for the concatenation is (C4), which states that the concatenation of two random strings results in a random string. Rules (C5) and (C6) state that if a string is indistinguishable from a random value given all the values in the list of queries to the block cipher (or the hash function), then clearly it cannot be a prefix of one of the strings $\mathcal{L}_{\mathcal{E}}$.

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

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 (X2) is reminiscent of a one-time-pad encryption of z with a random-looking value y. The other rules are propagation of the Equal and Unequal predicates.

(X1) $\{\mathsf{Equal}(y,y)\}\ x := y \oplus m_i\ \{(\mathsf{Equal}(m_i,m_i) \wedge \mathsf{Equal}(x,x)) \vee \mathsf{Unequal}(x,x)\}$
(X2) $\{\mathsf{Indis}(y;V,y,z;V')\}\ x := y \oplus z\ \{\mathsf{Indis}(x;V,x,z;V')\}$ provided $y \neq z$, $y \notin V$ and $x \notin V'$ unless $y,z \in V'$
(X3) $\{\mathsf{Equal}(y,y) \wedge \mathsf{Equal}(z,z)\}\ x := y \oplus z\ \{\mathsf{Equal}(x,x)\}$
(X4) $\{\mathsf{Equal}(y,y) \wedge \mathsf{Unequal}(z,z)\}\ x := y \oplus z\ \{\mathsf{Unequal}(x,x)\}$

Due to the commutativity of the xor, the role of y and z can be exchanged in all the rules above.

Block cipher:
Since block ciphers are modeled as ideal ciphers, 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. This is expressed in rules (B1) to (B3), and also used in the proof of many other rules. Since the querying of a block cipher twice at any point is undesirable, we always require the invariant E as a precondition.

(B1) $\{\mathsf{Empty}\}\ x := \mathcal{E}(m_i)\ \{(\mathsf{Equal}(m_i,m_i) \wedge \mathsf{Equal}(x,x) \wedge \mathsf{Indis}(x;\mathsf{Var},\ell_{\mathcal{E}},\ell_{\mathcal{H}};\mathsf{Var}-x)) \vee (\mathsf{Unequal}(x,x) \wedge \mathsf{Indis}(x))\}$
(B2) $\{\mathsf{E}(\mathcal{E};y;\emptyset) \wedge \mathsf{Unequal}(y,y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Indis}(x)\}$

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

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

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

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

We also have rules similar to (B2) to (B5), with the invariant $\mathsf{E}(\mathcal{E}; y; \emptyset)$ replaced by the invariant 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 an unusual decision, but 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) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Unequal}(y, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Indis}(x)\}$

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

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

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

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

For loop:

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

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

Finally, we introduce a few general rules for consequence, sequential composition, conjunction and disjunction. Let $\phi_1, \phi_2, \phi_3, \phi_4$ be any four predicates in our logic, and let cmd, cmd_1, cmd_2 be any three commands.

(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 $\{\phi_1 \vee \phi_3\}\mathsf{cmd}\{\phi_2 \vee \phi_4\}$

▶ **Theorem 7.** *A generic hash* $Hash(m_1 \| \ldots \| m_i, c)$: **var** $\vec{x_i}$; cmd_i *computes an almost-universal hash function if* $\{init\}\mathsf{cmd}_i\{UNIV(i)\}$.

The theorem is the consequence of Proposition 6 and of the soundness of our Hoare logic. We then say that a sequence of predicates $[\phi_0, \ldots, \phi_n]$ 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, $1 \leq n$, $\{\phi_{i-1}\}\ \mathsf{cmd}_i\ \{\phi_i\}$ holds.

5 Implementation

To use our method, we start at the beginning of the program, at each command apply every possible rule and, once done, test if the invariant $UNIV(n)$ holds at the end of the program. One possible downside of this approach is that the application of every possible rule could be very time consuming. For this reason, we need a way to filter out unneeded invariants, so that execution time remains reasonable.

Invariant Filter

We say that ϕ is an *invariant on* x if ϕ is either $\mathsf{Equal}(x, y)$, $\mathsf{Unequal}(x, y)$, $\mathsf{E}(\mathcal{E}; x; V)$ $\mathsf{H}(\mathcal{H}; x; V)$ or $\mathsf{Indis}(x; V_1, V_2)$. We say that an invariant ϕ on variable x is *obsolete for program* p if x does not appear anywhere in p and if $\neg(\phi \Rightarrow \mathsf{Unequal}(c_n, c_i))$ and $\neg(\phi \Rightarrow \mathsf{Equal}(m_i, m_i))$ for any i, $1 \leq i \leq n$. The following theorem shows that once an invariant is obsolete, it can be discarded.

▶ **Theorem 8.** *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 invariants 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 an invariant on x following the execution of command $x := e$ only have as preconditions invariants on the variables in e. As a result, we can always filter out obsolete invariants after processing each commands.

Also, we note that the only commands that can make an invariant $\mathsf{Equal}(m_i, m_i)$ 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 invariant $\mathsf{Equal}(m_l, m_l)$ is not present in one of the conjunction of the current predicate (after transforming the predicate in DNF 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 $\bigwedge_{j=1}^{n} \mathsf{Equal}(m_j, m_j)$ from $UNIV(n)$. Therefore, we can also safely filter out all other invariants of the form $\mathsf{Equal}(m_i, m_i)$ from that conjunction.

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

Finding Loop Predicates

The programs describing the almost-universal hash function usually contains for loops. It is therefore necessary to have an automatic procedure to detect the predicate $\psi(i)$ that allows us to apply rule (F1). We now show a heuristic that can be used to construct such a predicate, 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 "for $l = p$ to q do: $[\mathsf{cmd}_l]$", we express the precondition in the form $\varphi(p-1)$. The classical method for finding a stable predicate consists in processing the instructions c_l contained in the loop to find the predicate $\psi(l)$ such that $\{\varphi(l-1)\}\ c_l\ \{\phi(l)\}$. If $\varphi(l) \Rightarrow \psi(l)$, then we have found an predicate such that $\{\varphi(l-1)\}\ c_l\ \{\varphi(l)\}$ and we can apply rule $(F1)$. Otherwise, we repeat this process with $\varphi'(l) = \varphi(l) \wedge \psi(l)$ until we find a stable predicate.

Unfortunately, for certain loops, one could repeat the process infinitely and never obtain a stable predicate. If, after a certain number n of iterations of the process above, we did not find a stable invariant[3], we decide that the classical method has failed and so we need a new heuristic to construct the stable predicate. The heuristic we describe here is inspired from widening methods in abstract interpretation. We start over with invariant $\varphi(l-1)$, and process the code of the loop once to find invariant $\psi_1(l)$ such that $\{\varphi(l-1)\}\ c_l\ \{\psi_1(l)\}$. Then, we repeat this starting with invariant $\varphi(l-1) \wedge \psi_1(l-1)$ to find invariant $\psi_2(l)$ such that $\{\varphi(l-1) \wedge \psi_1(l-1)\}\ c_l\ \{\psi_2(l)\}$. By analyzing the predicates $\varphi(l)$, $\varphi(l) \wedge \psi_1(l)$ and $\varphi(l) \wedge \psi_1(l) \wedge \psi_2(l)$, we identify the predicate $\gamma(l)$ such that $\gamma(l)$ appears in $\varphi(l)$, $\gamma(l-1)$ appears in $\psi_1(l)$ and $\gamma(l-2)$ appears in $\psi_2(l)$. We then use $\varphi'(l) = \varphi(l) \wedge \bigwedge_{j=p-1}^{j=l-1} \gamma(j)$ as our new starting predicate.

▶ **Example 9.** We now apply this method to $Hash_{CBC}$. After processing command $c_1 := \mathcal{E}(m_1)$, we obtain the predicate $\varphi(1) = (\mathsf{Equal}(m_1, m_1) \wedge \mathsf{Equal}(c_1, c_1) \wedge \mathsf{Indis}(c_1; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_1)) \vee \mathsf{Indis}(c_1)$. Parameterizing this in terms of l, we obtain

[3] The choice of the number of times the process is repeated is completely arbitrary, we choose to try only two iterations since it is sufficient for all our examples.

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

After processing the code of the loop on $\varphi(l-1)$, we obtain the following:

$$\psi_1(l) = (\mathsf{Equal}(m_{l-1}, m_{l-1}) \wedge \mathsf{Equal}(m_l, m_l) \wedge \mathsf{Equal}(c_l, c_l) \wedge \mathsf{Indis}(c_l; \mathsf{Var}, \ell_{\mathcal{E}}; \mathsf{Var} - c_l)) \vee \mathsf{Indis}(c_l)$$

and repeat the same process with $\varphi(l-1) \wedge \psi_1(l-1)$ to obtain

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

We deduce $\gamma(l) = \mathsf{Equal}(m_l, m_l)$ and use

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

as our next attempt at finding a stable predicate. We find that $\varphi'(l)$ is a stable predicate 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 $\varphi'(n) \Rightarrow UNIV(n)$, thereby proving that $Hash_{CBC}$ computes an almost-universal hash function.

We programmed a tool in OCaml implementing our method for proving that the front end of MACs are almost-universal hash functions. The program requires about 1000 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 tool 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.

▶ Proposition 10. Let $\mathcal{F}_{\mathcal{E}}$ be a family of block ciphers, $\mathcal{H} = \{h_i\}_{i \in \{0,1\}^\eta}$ and $\mathcal{H}' = \{h_i\}_{i \in \{0,1\}^\eta}$ be families of almost-universal hash function and $\mathcal{G}$ be a random oracle. If $h \xleftarrow{\$} \mathcal{H}$, $h_{\mathcal{E}} \xleftarrow{\$} \mathcal{H}'$, $\mathcal{E} \xleftarrow{\$} \mathcal{F}_{\mathcal{E}}$, $\mathcal{G}$ 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 $sk = (i, \mathcal{E})$.
- $MAC_2(m) = \mathcal{G}(l \| h_i(m))$ is a secure MAC with key $sk = (i, k)$.
- $MAC_3(m) = \begin{cases} \mathcal{E}_1(h_i(m')) \text{ where } m' = pad(m) \text{ if } m\text{'s length is not a multiple of } \eta \\ \mathcal{E}_2(h_i(m)) \text{ if } m\text{'s length is a multiple of } \eta \end{cases}$
  is a secure MAC with key $sk = (i, \mathcal{E}_1, \mathcal{E}_2)$.
- $MAC_4(m) = \begin{cases} \mathcal{E}(h_{\mathcal{E}}(m') \oplus k_1) \text{ where } m' = pad(m) \text{ if } m\text{'s length is not a multiple of } \eta \\ \mathcal{E}(h_{\mathcal{E}}(m) \oplus k_2) \text{ if } m\text{'s length is a multiple of } \eta \end{cases}$
  is a secure MAC with key $sk = (\mathcal{E}, k_1, k_2)$

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

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 *ACM-CCS '10*, pages 375–386, 2010.

2 G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO '11*, pages 71–90, 2011.

3 G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin. Beyond provable security verifiable ind-cca security of OAEP. In *CT-RSA*, LNCS, pages 180–196. Springer, 2011.

4 M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO '96, LNCS*, pages 1–15. Springer-Verlag, 1996.

5 M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In *CRYPTO '94*, pages 341–358, 1994.

6 M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM-CCS '93*, pages 62–73, 1993.

7 J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO '99*, pages 216–233, 1999.

8 J. Black and P. Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. In *CRYPTO '00, LNCS*, pages 197–215, 2000.

9 J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *EUROCRYPT 2002. LNCS*, pages 384–397. Springer-Verlag, 2002.

10 B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *LNCS*, pages 537–554. Springer, 2006.

11 R. Corin and J. den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs. In *ICALP '06*, pages 252–263, 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 *ACM-CCS'08*, 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. Laboratoire VERIMAG, Université Joseph Fourier, France, April 2012. Available at `http://www-verimag.imag.fr/~gagne/macChecker.html`.

15 M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini. Automated proofs for encryption modes. In *ASIAN'09*, volume 5913 of *LNCS*, pages 39–53, 2009.

16 M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In Moti Yung, editor, *CRYPTO '02*, volume 2442 of *LNCS*, 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 Block Ciphers and Random Oracle Model

Block Cipher Security

A block cipher is a family of permutations $\mathcal{E} : \{0,1\}^k \times \{0,1\}^\eta \to \{0,1\}^\eta$ indexed with a key $K \in \{0,1\}^k$. 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 11** (Pseudo-Random Functions)**.** Let $P : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ 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.

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

where Φ_n is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$.

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 analyze hash functions in cryptographic protocols [6].

B Other Hash Examples

We present here a few other examples of almost-universal hash functions used in common MAC algorithms

The function $Hash_{CBC'}$, used in FCBC and XCBC is described as follows:

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

The function $Hash_{HMAC}$, used in HMAC is as follows:

$Hash_{HMAC}(m_1\| \ldots \|m_n, c_n)$: **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: $[z_i := c_{i-1}\|m_i; c_i := \mathcal{H}(z_i)]$

Finally, the function $Hash_{PMAC}$, used in PMAC is as follows:

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

C Proofs

Before presenting the proofs for all the claims in our paper, we present a few results that will be used repeatedly in our proofs.

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.

▶ **Lemma 12.** *For any $X, X' \in \text{DIST}(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$, any set of variables V, any expression e constructible from V, and any variable x, if $X \sim_V X'$ then $[\![x := e]\!](X) \sim_{V,x} [\![x := e]\!](X')$.*

Proof. We assume $X \sim_V X'$. Suppose that $[\![x := e]\!](X) \not\sim_{V,x} [\![x := e]\!](X')$. This means there exists a polynomial-time adversary $\mathcal{A}$ that, on input $S(V, x)$ drawn either from $[\![x := e]\!](X)$ or $[\![x := e]\!](X')$, guesses the right initial distribution with non-negligible probability. We let $\mathcal{B}$ be the adversary against $X \sim_V X'$ which simply computes x from values in $S(V)$ – which can be done in polynomial time since e is constructible from values in V – and runs $\mathcal{A}(V, x)$. It is clear that the advantage of $\mathcal{B}$ is exactly that of $\mathcal{A}$, which would imply that it is not negligible, although we assumed $X \sim_V X'$. ◀

▶ **Corollary 13.** *For any $X \in \text{DIST}(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$, any sets of variables V, any expression e constructible from V, and any variable x, z such that $z \notin \{x\} \cup \mathsf{Var}(e)$ if $X \models \mathsf{Indis}(z; V; V')$ then $[\![x := e]\!](X) \models \mathsf{Indis}(z; V, x; V')$. We emphasize that here we use the notation $\mathsf{Var}(e)$ (in its usual sense), that is to say, the variable z does not appear* at all *in e.*
Similarly, if $X \models \mathsf{Indis}(z; V'; V)$, then $[\![x := e]\!](X) \models \mathsf{Indis}(z; V'; V, x)$.

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

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

Since $z \notin \{x\} \cup \mathsf{Var}(e)$ using the same technique as in Lemma 12, we easily obtain

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

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

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

The following will be useful when dealing with the concatenation command.

▶ **Lemma 14.** *For any distribution $X \in \text{DIST}(\Gamma, \mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H})$, any program cmd produced by our grammar any $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![\mathsf{cmd}]\!]X$ and any variable $v \in \mathsf{Var}$, $|S(v)| = |S'(v)|$.*

Proof. This is a trivial consequence of the fact that the message blocks always have equal length in both executions. All values computed from there will therefore also have equal length. ◀

C.1 Proof of Proposition 6

▶ Proposition 6. A generic hash $(\mathcal{F}_\mathcal{E}, \mathcal{F}_\mathcal{H}, P(m_1 \| \ldots \| m_l, c) : \textbf{var } \vec{x_l}; \mathsf{cmd}_l)$ is an almost-universal hash function on string of length at most l blocks if, at the end of execution, the following invariants hold:

$$UNIV(l) = \Big(\mathsf{Unequal}(c_l, c_l) \vee \textstyle\bigwedge_{j=1}^{i} \mathsf{Equal}(m_j, m_j)\Big) \wedge \textstyle\bigwedge_{i=1}^{l-1} \mathsf{Unequal}(c_l, c_i)$$

Proof. Say M_1 is a l block message, and M_2 is an $k \leq l$ block message. Then, 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', 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 predicate $UNIV(k)$ holds, then we have that either $\mathsf{Unequal}(c_k, c_l)$, which shows that the probability that the hashes are equal is negligible, or $k = l$ and all the message blocks are equal which imply that $M_1 = M_2$. ◀

C.2 Proof of Lemma 5

▶ **Lemma 5.** *The following relations are true for any sets V_1, V_2, V_3 and variables x, y with $x \neq y$*

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

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 $X \models \mathsf{Indis}(x, V)$, then the probability that the value of x is equal to the value of any variable in V (or any values in $\mathcal{L}_{\mathcal{E}}.\mathsf{dom}$, $\mathcal{L}_{\mathcal{H}}.\mathsf{dom}$ 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 $S(V)$. ◀

C.3 Initialization

▶ Proposition 15 (Rule (init)).
$\mathsf{INIT}\ \{\mathsf{Indis}(k; \mathsf{Var}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}; \mathsf{Var} - k) \wedge \mathsf{Empty}\}$

Proof. We note that the initialization command can only appear at the beginning of a program. Let X be an initial distribution, as described in the definition of security of ϵ-universal hash function. We have that $[\![\mathsf{INIT}]\!]X = X$ because the initialization command has no impact on the distribution. So we have to prove that $X \models \mathsf{Empty}$ and $X \models \mathsf{Indis}(k; \mathsf{Var}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{H}}; \mathsf{Var} - k)$. The former is obvious since the adversary has no access to $\mathcal{E}$ in the attack. The latter is also clear because k is sampled randomly and independently after the adversary terminates. ◀

C.4 Generic Preservation

▶ Proposition 16 (Rule (G1)).
$\{\mathsf{Equal}(t)\}\ \mathsf{cmd}\ \{\mathsf{Equal}(t)\}$ 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 17 (Rule (G2)).
$\{\mathsf{Unequal}(t)\}\ \mathsf{cmd}\ \{\mathsf{Unequal}(t)\}$ 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 18 (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, $\mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X : S(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V) \vee S'(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S'(V)] = \mathsf{Pr}[(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} [\![x := \mathcal{E}(y)]\!]X : S(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V) \vee S'(t) \in \mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S'(V)]$ because, the values in the sets $S(V)$, $S'(V)$ and $Elist.dom$ are unchanged by the command. ◀

▶ Proposition 19 (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 20 (Rule (G5)).
$\{\mathsf{Indis}(t; V; V')\}$ cmd $\{\mathsf{Indis}(t; V; V')\}$ 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 $V' - t$

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 Corollary 13. It follows that $(X \models \mathsf{Indis}(t; V)) \Rightarrow ([\![\mathsf{cmd}]\!]X \models \mathsf{Indis}(t; V))$. ◀

▶ Proposition 21 (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 $\mathcal{L}_\mathcal{E}$. ◀

C.5 Function ρ

▶ Proposition 22 (Rule (P1)).
$\{\mathsf{Equal}(y)\}\ x := \rho(y)\ \{\mathsf{Equal}(x)\}$

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

C.6 Assignment

▶ Proposition 23 (Rule (A1)).
$\{true\}\ x := m_i\ \{\mathsf{Equal}(m_i, m_i) \wedge \mathsf{Equal}(x, x) \vee \mathsf{Unequal}(x, x)\}$

Proof. Clearly, if $(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X$ and $S(m_i) = S'(m_i)$, then we have $[\![x := y]\!]X \models \mathsf{Equal}(m_i, m_i) \wedge \mathsf{Equal}(x, x)$. Otherwise, $[\![x := y]\!]X \models \mathsf{Unequal}(x, x)$. ◀

▶ Proposition 24 (Rules (A2) to (A9)). The following rules hold.
- (A2) $\{\mathsf{Equal}(y, y)\}\ x := y\ \{\mathsf{Equal}(x, x)\}$
- (A3) $\{\mathsf{Unequal}(y, y)\}\ x := y\ \{\mathsf{Unequal}(x, x)\}$
- (A4) $\{\mathsf{Indis}(y; V; V')\}\ x := y\ \{\mathsf{Indis}(x; V; V')\}$ 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. ◀

C.7 Concatenation

▶ Proposition 25 (Rule (C1)).
$\{\mathsf{Equal}(y, y)\}\ x := y\|m_i\ \{(\mathsf{Equal}(m_i, m_i) \wedge \mathsf{Equal}(x, x)) \vee \mathsf{Unequal}(x, x)\}$

Proof. Clearly, if $X \models \mathsf{Equal}(y, y)$, $(S, S', \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X$ and $S(m_i) = S'(m_i)$, then we have $[\![x := y\|m_i]\!]X \models \mathsf{Equal}(m_i, m_i) \wedge \mathsf{Equal}(x, x)$. Otherwise, $[\![x := y\|z]\!]X \models \mathsf{Unequal}(x, x)$. ◀

▶ Proposition 26 (Rule (C2)).
$\{\mathsf{Equal}(y, y) \wedge \mathsf{Equal}(z, z)\}\ x := y\|z\ \{\mathsf{Equal}(x, x)\}$

Proof. Trivial. ◀

▶ Proposition 27 (Rule (C3)).
$\{\mathsf{Unequal}(y, y)\}\ x := y\|z\ \{\mathsf{Unequal}(x, x)\}$

Proof. Trivial consequence of the fact that for any distribution X and $(S, S', \mathcal{E}, \mathcal{H}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}) \xleftarrow{\$} X$, with overwhelming probability, $S(y) \neq S'(y)$, and, from Lemma 14, $|S(y)| = |S'(y)|$ implies that $S(y)\|S(z) \neq S'(y)\|S'(z)$. ◀

▶ Proposition 28 (Rule (C4)).
$\{\mathsf{Indis}(y; V, y, z; V') \wedge \mathsf{Indis}(z; V, y, z; V')\}\ x := y\|z\ \{\mathsf{Indis}(x; V, x; V')\}$ 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 $X \models \mathsf{Indis}(y; V, y, z) \wedge \mathsf{Indis}(z; V, y, z)$ with $x, y, z \notin V$ and $x \notin V'$. We have that

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

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, $[u_1, u_2 \xleftarrow{\$} \mathcal{U} : u_1\|u_2] = [u \xleftarrow{\$} \mathcal{U} : u]$. 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:

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

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 $X \models \mathsf{Indis}(y; V, y, z)$. 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 $[\![x := y \| z]\!] X \models \mathsf{Indis}(x; V, x; V')$.

The case when $y, z \in V'$ is similar, the result follows from the argument above and Corollary 13. ◀

▶ Proposition 29 (Rules (C5) and (C6)).

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

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

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 $X \models \mathsf{Indis}(y; V, \mathcal{L}_\mathcal{E}; \emptyset)$, $\mathcal{A}$'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 $\mathcal{L}_\mathcal{E}$.

◀

C.8 Xor

▶ Proposition 30 (Rule (X1)).
$\{\mathsf{Equal}(y,y)\}\ x := y \oplus m_i\ \{(\mathsf{Equal}(x,x) \wedge \mathsf{Equal}(m_i,m_i)) \vee \mathsf{Unequal}(x,x)\}$

Proof. Clearly, if $X \models \mathsf{Equal}(y,y)$, $(S,S',\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H}) \xleftarrow{\$} X$ and $S(m_i) = S'(m_i)$, then we have $[\![x := y \oplus m_i]\!]X \models \mathsf{Equal}(m_i,m_i) \wedge \mathsf{Equal}(x,x)$. Otherwise, $[\![x := y \oplus m_i]\!]X \models \mathsf{Unequal}(x,x)$. ◀

▶ Proposition 31 (Rule (X2)).
$\{\mathsf{Indis}(y;V,y,z;V')\}\ x := y \oplus z\ \{\mathsf{Indis}(x;V,x,z;V')\}$ 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 $X \models \mathsf{Indis}(y;V,y,z)$ with $y \neq z$, $y \notin V$ and $x \notin V'$. We have that

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

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:

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

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

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

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 $X \models \mathsf{Indis}(y;V,y,z;V')$.

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

The case when $y,z \in V'$ is similar, the result follows from the argument above and Corollary 13. ◀

▶ Proposition 32 (Rule (X3)).
$\{\mathsf{Equal}(y,y) \wedge \mathsf{Equal}(z,z)\}\ x := y \oplus z\ \{\mathsf{Equal}(x,x)\}$

Proof. Trivial. ◀

▶ Proposition 33 (Rule (X4)).
$\{\mathsf{Equal}(y,y) \wedge \mathsf{Unequal}(z,z)\}\ x := y \oplus z\ \{\mathsf{Unequal}(x,x)\}$

Proof. Trivial. ◀

C.9 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 34 (Rules (B1), (B2) and (B3)).

(B1) $\{\mathsf{Empty}\}\ x := \mathcal{E}(m_i)\ \{(\mathsf{Equal}(m_i,m_i) \wedge \mathsf{Equal}(x,x) \wedge \mathsf{Indis}(x;\mathsf{Var},\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H};\mathsf{Var}-x))$
$\vee (\mathsf{Unequal}(x,x) \wedge \mathsf{Indis}(x))\}$

(B2) $\{\mathsf{E}(\mathcal{E};y;\emptyset) \wedge \mathsf{Unequal}(y,y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Indis}(x)\}$

(B3) $\{\mathsf{E}(\mathcal{E};y;\emptyset) \wedge \mathsf{Equal}(y,y)\}\ x := \mathcal{E}(y)\ \{\mathsf{Indis}(x;\mathsf{Var},\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H};\mathsf{Var}-x) \wedge \mathsf{Equal}(x,x)\}$

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. We have two cases to consider:
- if $S(m_i) \neq S'(m_i)$, i.e. $X \models \mathsf{Unequal}(m_i,m_i)$, 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 $[\![x := \mathcal{E}(y)]\!]X \models \mathsf{Indis}(x)$ is immediate. It should be clear that, in this case, $\mathsf{Unequal}(m_i,m_i)$ is preserved by $x := \mathcal{E}(m_i)$.
- if $S(m_i) = S'(m_i)$, that is $X \models \mathsf{Equal}(m_i,m_i)$, then clearly $[\![x := \mathcal{E}(m_i)]\!]X \models \mathsf{Equal}(m_i,m_i) \wedge \mathsf{Equal}(x,x)$ 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 $[\![x := \mathcal{E}(y)]\!]X \models \mathsf{Indis}(x;\mathsf{Var},\mathcal{L}_\mathcal{E},\mathcal{L}_\mathcal{H};\mathsf{Var}-x)$ is also clear.

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

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

◀

▶ Proposition 35 (Rule (B4)).
$\{\mathsf{E}(\mathcal{E};y;\emptyset) \wedge \mathsf{Indis}(t;V;V')\}\ x := \mathcal{E}(y)\ \{\mathsf{Indis}(t;V,x;V',x)\}$ 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 $X \models \mathsf{Indis}(t;V;V)$. The same can be argued with the roles of S and S' reversed. ◀

▶ Proposition 36 (Rules (B5)).

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

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 $\mathsf{Indis}(t; V, \mathcal{L}_\mathcal{E}, x, y; V', x, y)$ can be proven in the same way as in the proof of Rule (B4). ◀

▶ Proposition 37 (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 $\mathcal{L}_\mathcal{E}.\mathsf{dom} \cup S(V, y) \cup S'(V, y)$. ◀

C.10 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 38 (Rules (H1) to (H5)).
(H1) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Unequal}(y, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Indis}(x)\}$
(H2) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Equal}(y, y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Indis}(x; \mathsf{Var}, \mathcal{L}_\mathcal{E}, \mathcal{L}_\mathcal{H}, \mathsf{Var} - x) \wedge \mathsf{Equal}(x, x)\}$
(H3) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Indis}(t; V; V')\}\ x := \mathcal{H}(y)\ \{\mathsf{Indis}(t; V, x; V', x)\}$ provided $\mathcal{L}_\mathcal{H} \notin V$, even if $t = y$
(H4) $\{\mathsf{H}(\mathcal{H}; y; \emptyset) \wedge \mathsf{Indis}(t; V, \mathcal{L}_\mathcal{H}, y; V', y)\}\ x := \mathcal{H}(y)\ \{\mathsf{Indis}(t; V, \mathcal{L}_\mathcal{H}, x, y; V', x, y)\}$
(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). ◀

C.11 For Loop

▶ Proposition 39 (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. ◀