Generic Indifferentiability Proofs of Hash Designs

Marion Daubignard
Université Grenoble 1,
CNRS, Verimag, FRANCE

Pascal Lafourcade
Université Grenoble 1,
CNRS, Verimag, FRANCE

Yassine Lakhnech
Université Grenoble 1,
CNRS, Verimag, FRANCE

firstname.lastname@imag.fr

ABSTRACT

Security arguments were one of SHA-3 competition requirements. Indeed, most of the second round candidates provide indifferentiability proofs of their constructions. In this paper, we propose a general framework for proving indifferentiability of hash functions. Indeed, we propose a generic simulator design and a generic way to compute a bound of the indifferentiability from a random oracle of constructions using this simulator. The general bound provided by our theorem is relevant, as we show in two examples of its application, on chopMD and the sponge constructions. This work participates to the effort of formalization and unification of security proofs, to enable their verification and increase confidence in the guarantees they provide.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Public key cryptosystems; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Security, Verification

Keywords

Indifferentiability, Hash Constructions,

1. INTRODUCTION

Motivation. As a keystone of the design of secure and reliable systems, the security guarantees provided by cryptographic primitives must be precisely appraised. Of course, the strategy consisting in the systematic attack of constructions keeps helping to fulfil better security goals. Nevertheless, repeatedly unsuccessful attacks are not deemed sufficient to certify any cryptographic design anymore. Indeed, *provable security* proposes a mathematical approach to security which is nowadays widely adopted. This global tendency is confirmed by the requirements imposed by NIST on candidates for the SHA-3 competition, aiming at the selection of a new hash function to augment the Secure Hash Standard. Submissions were invited to include *"any security argument that is applicable, such as a security reduction proof"*.

As far as cryptographic hash functions are concerned, the notion of *indifferentiability of a random oracle* is a currently well-admitted security criterion. Initially introduced by Maurer et al. in [22], the concept of indifferentiability was then publicized and tailored to suit hash designs in [14]. It is based on the acknowledgement that these functions are generally built as two-tier constructions. *Inner-primitives* are designed (e.g. blockciphers), which should output bitstrings looking random to an outsider. They have fixed input and output sizes. However, a lot of cryptosystems rely on the ability to compute hash values for inputs of any size, or at least up to a very large length (e.g. 2^{64} bits). To address this, hash designers use iteration based *hash constructions* (also called *hash designs*) to enable domain extension. Indeed, they split the global input to be hashed into several chunks and use them successively to compute fixed length inputs to be fed to the inner-primitive. The most popular domain-extension technique was proposed independently by Merkle in [24] and Damgård in [15].

Intuitively, the idea behind indifferentiability of a random oracle is to compare two idealized systems based on the actual hash construction. On the one hand, the hash design when plugged on random fixed input length inner-primitives yields the first system. On the other hand, the second system consists of a random oracle playing the role of the hash function and a simulator in place of the inner-primitives. The hash design is said to be indifferentiable from a random oracle as soon as there exists a simulator such that both systems are indistinguishable. The difficulty to elaborate a good simulator lies in the fact that the simulator *does not have access* to the random oracle memory. As a result, an adversary trying to distinguish both systems can issue a query to the random oracle, with the simulator being none the wiser. It is thus delicate for a simulator to mimic the real-world consistency between inner-primitive queries.

Related work and contributions. Though some flaws in the indifferentiability security notion have been highlighted in [25], this criterion remains a standard security goal to be achieved by hash functions. Roughly, it allows to track down structural design weaknesses allowing generic attacks. Therefore, following Coron et al. in [14], a lot of effort has been devoted to developing concrete security proofs of indifferentiability. Numerous SHA-3 candidates which have been proposed have benefited of such a proof, amongst which JH in [10], Grøstl in [1], Shabal in [11], Keccak in [7]. The problem is though, as it is the case for public-key cryptography, that while hash designs increase in number and complexity, their security proofs become more and more involved and difficult to check. To address this issue, a first possibility is to aim to broaden our view

of what is required to ensure indifferentiability (e.g. [20]). In this respect, Dodis et al. [18] propose for example a new security notion, preimage-awareness, and show it to be a sufficient condition for indifferentiability.

In a slightly different approach, our ambition is to provide a framework and a generic simulator, along with a macro-theorem to compute an indifferentiability bound, instead of proving every novel construction from scratch. This work is not the first to aim to address the lack of unified provable security infrastructure in which to carry out indifferentiability proofs. It follows in the steps of Chang et al. who provide in [12] generic proofs for respectively most popular prefix-free padding designs, Bhattacharyya et al. who propose in [9] a generic simulator and a list of events to optimally bound indifferentiability of a certain number of domain extenders, Shabal designers who set formal definitions of graph-based proofs and present their proof of Shabal as a roadmap to carry out others in [11]. A similar procedure is followed in Bhattacharyya et. al in the proof of JH [10].

We propose a way to formalize the two-tier structure composed of hash designs and inner-primitives they are built on top of. It captures all designs we are aware of, e.g. the SHA-3 finalists JH [26], Grøstl [21], Keccak [8], Skein [19] and BLAKE [2], Shabal [11], the EMD transform [5], HMAC and NMAC modes [4]... Our definition generalizes that of generic domain extenders proposed by Bhattacharyya et al. in [9] since it allows post-processing and multiple inner-primitives. After that, we provide a generic simulator inspired from [9] and the numerous examples put to use in the proofs mentioned above. In addition to being usable in a more general setting, our simulator description features a detailed specification of the underlying graph. Then, we show that a series of bad events can prevent the simulator from ensuring consistency existing between real-world hash values and inner-primitives results. Not content with listing these four events, we group them together and characterize them as events on a real-world counterpart of the simulator graph. This enables us to state a theorem proposing a generic simulator and a generic bound of the indifferentiability of a construction and the simulator.

On the way to establishing the theorem, we highlight a particular feature (namely a partial-injectivity property) that the hash constructions should meet in order for the theorem bound to be significantly low. This is useful in two aspects. First, it provides insight on properties that designs should achieve to be minimally safe. Secondly, as the context of use of the theorem is large, the condition steers the attempts of theorem users to massage their hash construction in order to obtain an interesting bound.

Eventually, we show on two examples that the bounds provided by our theorem are relevant: for the chop solution, we achieve the same result as Maurer and Tessaro in [23] in case of prefix-free padding and a better bound than that of Chang and Nandi in [13] in the general case. Moreover, the application of our result on the sponge construction (underlying the Keccak design) highlights the lack of an additional term in the bound provided by Bertoni et al. in [7], as was anticipated but not justified by Bresson et al. in [11].

Outline. In the next section, we introduce our formalization of oracles, adversaries and propose a generic definition to capture hash designs. It allows us to formally define indifferentiability within our framework. In section 3, we detail the construction of a generic simulator and the graph it is based on in a slightly restricted context. We then define characteristic graphs, real-world counterparts of the simulator graph, and show how they can be used to capture events of bad simulation. After stating our theorem, examples of application appear in section 4. We generalize in section 5 our theorem to an unrestricted context.

2. DEFINITIONS AND NOTATIONS

Mappings. Given a mapping f from A to B, $a \in A$ and $b \in B$, we denote by $f.(a,b)$ the mapping g such that if a is in the domain of f then $g = f$, otherwise g is such that $g(a) = b$ and for any $x \in A$, $x \neq a$, $g(x) = f(x)$. A *memory* is a mapping from a finite set of variables to values.

Lists. Given a set A, we denote by A^* (resp. A^+) the set of finite lists with elements in A (resp. non-empty finite lists). The empty list is denoted by $[\,]$. The append to the right of an element $a \in A$ to a list $L \in A^*$ is denoted by $L : a$. The selective append of a to L, denoted by $L.a$, is defined by $L.a = L$, if $a \in L$ and $L.a = L : a$, otherwise. Given an indexed set $A = (a_i)_{i\in\mathbb{N}}$, and an index set I, $[a_i]_{i\in I}$ denotes the list of elements a_i for $i \in I$. We denote by Π_i the $i-th$ canonical projection, i.e. $\forall i = 1..n, \Pi_i((e_1,\ldots,e_n)) = e_i$. Moreover, $\mathsf{dom}(L)$ denotes the set $\{\Pi_1(a) | a \in L\}$, and $L(a)$ is the set of elements of L with first component a.

Strings. Given a bitstring w, $|w|$ denotes the length of w. For $s \leq |w|$, $\mathsf{Last}_s(w)$ and $\mathsf{First}_s(w)$ denote the suffix of w, respectively its prefix, of length s. For $1 \leq m \leq n \leq |w|$, $w[m,n]$ denotes the substring of w starting with its m-th bit and ending its n-th bit. The concatenation of two bitstrings x and y is denoted by $x||y$. A string of length 0 is denoted by λ.

Misc.. Given an integer x, $\lceil x \rceil$ denotes the ceiling of x. We use $\mathbf{1}$ to denote the unit type. Symbol $\delta_{\varphi(x)}$, where $\varphi(x)$ is a boolean condition depending on x evaluates to 1 if $\varphi(x)$ holds and 0 otherwise. Given a finite set A, the uniform distribution over A is denoted by $\mathcal{U}_A$, the set of subsets of A is written $\mathfrak{P}(A)$, and the number of elements of A is denoted as $|A|$. Partial functions are denoted by $\xrightarrow{part}$.

2.1 Formalization of Oracles and Adversaries

Following [3], we use oracle systems to describe cryptographic schemes. An oracle system is a set of stateful oracles.

DEFINITION 1. *An* oracle system $\mathbb{O}$ *is given by:*

- *a finite set* $\mathsf{N}_\mathbb{O} = \{\mathcal{O}_1, \cdots, \mathcal{O}_n\}$ *of oracles with a distinguished oracle* $o_\mathrm{F} \in \mathsf{N}_\mathbb{O}$*, the finalization oracle.*
- *for each oracle* $\mathcal{O}_i$*, a countable set* M_i *of oracle memories (states) and an implementation* $\mathsf{Imp}(\mathcal{O}_i)^{\mathcal{O}_{j_1},\cdots,\mathcal{O}_{j_i}} : \mathsf{In}(\mathcal{O}_i) \times \mathsf{M}_\mathbb{O} \longrightarrow \mathcal{D}(\mathsf{Out}(\mathcal{O}_i) \times \mathsf{M}_\mathbb{O})$*, where* $\mathsf{In}(\mathcal{O}_i)$ *and* $\mathsf{Out}(\mathcal{O}_i)$ *are finite sets, called the* query domain *and* the answer domain *of* $\mathcal{O}_i$*, respectively, and* $\mathsf{M}_\mathbb{O} = \mathsf{M}_1 \times \cdots \times \mathsf{M}_n$*. The notation* $\mathsf{Imp}(\mathcal{O}_i)^{\mathcal{O}_{j_1},\cdots,\mathcal{O}_{j_i}}$ *means that the implementation* $\mathsf{Imp}(\mathcal{O}_i)$ *of* $\mathcal{O}_i$ *has exclusive access to memories in* $\prod_{k\in[1..n]-\{j_1,\ldots,j_i\}} \mathsf{M}_k$*. However, while executing* $\mathsf{Imp}(\mathcal{O}_i)$*, the oracles* $\mathcal{O}_{j_1},\cdots,\mathcal{O}_{j_i}$ *may be called, which causes reading and writing in* $\mathsf{M}_{j_1},\cdots,\mathsf{M}_{j_i}$*.*

  Given a function $k : \{j_1,\cdots,j_i\} \to \mathbb{N}$ *and* $t \in \mathbb{N}$*, we say that* $\mathsf{Imp}(\mathcal{O}_i)^{\mathcal{O}_{j_1},\cdots,\mathcal{O}_{j_i}}$ *is* (k,t)*-bounded, if* one *of its executions makes at most* $k(j_m)$ *calls to* $\mathcal{O}_{j_m}$ *and takes at most time* t*.*
- $\bar{m}_\mathbb{O} \in \mathsf{M}_\mathbb{O}$ *is the initial memory.*
- *we require that* $\mathsf{In}(o_\mathrm{F}) = \{\mathsf{true}, \mathsf{false}\}$ *and* $\mathsf{Out}(o_\mathrm{F}) = \mathbf{1}$*.*

EXAMPLE 2.1. *Consider the hash function ChopMD introduced in [14] and inspired of [17]. It is obtained from the Merkle-Damgård construction by chopping off the last s bits of the output in order to prevent extension attacks, i.e. the ability of an adversary to find the hash of a long message out of the hash of one of its prefixes. This construction can be described as an oracle system that contains two oracles: $ChopMD_s$ and $\mathcal{F}$. The memory of $ChopMD_s$ consists of a mapping L_{chop} and that of $\mathcal{F}$ of a mapping $L_\mathcal{F}$. Their implementations are described in Figure 2.1 using an imperative notation.*

Oracle $ChopMD_s$
$\mathsf{In}(ChopMD_s) = \{0,1\}^{\leq 2^{64}},\ \mathsf{Out}(ChopMD_s) = \{0,1\}^{n-s}$
$\mathsf{Imp}(ChopMD_s)^{\mathcal{F}}(x) =$
if $x \in \mathsf{dom}(L_{chop})$ then
  return $L_{chop}(x)$
else
  $l := \lceil |x|/r \rceil$;
  $w := x||10^{l*r-|x|-1}$;
  $(w_1, \ldots, w_l) := (w[1,r], \ldots, w[r*(l-1)+1, r*l])$;
  $a^0 := 0^n$;
  for $j = 1$ to l do
    $q^j := a^{j-1}||w_j$;
    $a^j \leftarrow \mathcal{F}(q^j)$;
  endfor
  $a^f := \mathsf{First}_{n-s}(a^l)$;
  $L_{chop} := L_{chop}.(x, a^f)$;
  return a^f
endif

Oracle $\mathcal{F}$
$\mathsf{In}(\mathcal{F}) = \{0,1\}^{n+r},\ \mathsf{Out}(\mathcal{F}) = \{0,1\}^{n}$
$\mathcal{F}(q) =$
if $q \in \mathsf{dom}(L_{\mathcal{F}})$ then
  return $L_{\mathcal{F}}(q)$
else
  $a \leftarrow \mathcal{U}_{\{0,1\}^n}$;
  $L_{\mathcal{F}} := L_{\mathcal{F}}.(q, a)$;
  return a
endif

Figure 1: ChopMD Implementation

Two oracle systems $\mathbb{O}$ and $\mathbb{O}'$ are said to be *compatible* iff they have the same set of oracles, with same query and answer domains. Compatible systems can however differ in the implementations and memories. We therefore use the notation $\mathsf{Imp}_{\mathbb{O}}(\mathcal{O}_i)$ and $\mathsf{Imp}_{\mathbb{O}'}(\mathcal{O}_i)$ for implementations in $\mathbb{O}$ and $\mathbb{O}'$, to be able to distinguish them.

Consider an oracle system $\mathbb{O}$. Adversaries interact with oracle systems by making queries, and receiving answers. Such an interaction produces an *exchange* (for an oracle system $\mathbb{O}$), which is a triple $(\mathcal{O}, q, a)$ where $\mathcal{O} \in \mathsf{N}_{\mathbb{O}}$, $q \in \mathsf{In}(\mathcal{O})$ and $a \in \mathsf{Out}(\mathcal{O})$. Let Xch denote the set of exchanges. *Final exchanges* are of the form $(o_{\mathrm{F}}, -, -)$, i.e. queries to the finalization oracle. The set of final exchanges is denoted by $\mathsf{Xch}_{\mathrm{F}}$. The sets Que of queries and Ans of answers are, respectively, defined by $\mathsf{Que} = \{(\mathcal{O}, q) \mid (\mathcal{O}, q, a) \in \mathsf{Xch}\}$ and $\mathsf{Ans} = \{(\mathcal{O}, a) \mid (\mathcal{O}, q, a) \in \mathsf{Xch}\}$, while for a particular oracle $\mathcal{O}_i$, $\mathsf{Que}(\mathcal{O}_i)$ (resp. $\mathsf{Ans}(\mathcal{O}_i), \mathsf{Xch}(\mathcal{O}_i)$) only contains elements of Que (resp. $\mathsf{Ans}, \mathsf{Xch}$) starting with $\mathcal{O}_i$.

DEFINITION 2. *An $\mathbb{O}$-adversary $\mathbb{A}$ is given by a countable set $\mathsf{M}_{\mathbb{A}}$ of adversary memories, an initial memory $\bar{m}_{\mathbb{A}} \in \mathsf{M}_{\mathbb{A}}$ and functions for querying and updating:*

$$\begin{array}{rcl} \mathsf{A} & : & \mathsf{M}_{\mathbb{A}} \xrightarrow{part} \mathcal{D}(\mathsf{Que} \times \mathsf{M}_{\mathbb{A}}) \\ \mathsf{A}_{\downarrow} & : & \mathsf{Xch} \times \mathsf{M}_{\mathbb{A}} \to \mathcal{D}(\mathsf{M}_{\mathbb{A}}) \end{array}$$

Informally, the interaction between an oracle system and an adversary starts from the initial memory $(\bar{m}_{\mathbb{A}}, \bar{m}_{\mathbb{O}})$. Using A, $\mathbb{A}$ computes a query to $\mathbb{O}$ and updates its memory. Upon receiving a query, $\mathbb{O}$ updates its memory and replies to $\mathbb{A}$, which in turn updates its memory. This goes on until $\mathbb{A}$ calls the finalization oracle. We formalize this interaction as the execution of a transition system, defined below.

DEFINITION 3. *A* transition system $\mathcal{TS}$ *consists of:*

- *a (countable non-empty) set* M *of memories (states), with a distinguished initial memory* $\bar{m}$*;*
- *a set* Σ *of actions, with a distinguished subset* Σ_{F} *of finalization actions;*
- *a (partial probabilistic) transition function* $\mathsf{st} : \mathsf{M} \xrightarrow{part} \mathcal{D}(\Sigma \times \mathsf{M})$.

A partial execution sequence of $\mathcal{TS}$ is a sequence η of the form $m_0 \xrightarrow{X_1} m_1 \xrightarrow{X_2} \ldots \xrightarrow{X_k} m_k$ such that $m_0 = \bar{m}, X_i \in \Sigma$, $m_{i-1}, m_i \in \mathsf{M}$, and $\Pr[\mathsf{st}(m_{i-1}) = (X_i, m_i)] > 0$, for $i = 1, \ldots, k$. If $k = 1$, then η is a step. If $X_k \in \Sigma_{\mathrm{F}}$ or $m_k \notin \mathsf{dom}(\mathsf{st})$, then η is an execution sequence of length k. A probabilistic transition system $\mathcal{TS}$ induces a sub-distribution on executions, denoted $\mathcal{TS}$, such that the probability of a finite execution sequence η is

$$\Pr[\mathcal{TS} = \eta] = \prod_{i=1}^{k} \Pr[\mathsf{st}(m_{i-1}) = (X_i, m_i)]$$

A transition system is of height $k \in \mathbb{N}$ if all its executions have length at most k; in this case, $\mathcal{TS}$ is a distribution.

DEFINITION 4. *Let $\mathbb{O}$ be an oracle system and $\mathbb{A}$ be an $\mathbb{O}$-adversary . The composition $\mathbb{A} \mid \mathbb{O}$ is a transition system such that $\mathsf{M} = \mathsf{M}_{\mathbb{A}} \times \mathsf{M}_{\mathbb{O}}$, the initial memory is $(\bar{m}_{\mathbb{A}}, \bar{m}_{\mathbb{O}})$, the set of actions is $\Sigma = \mathsf{Xch}$, and $\Sigma_{\mathrm{F}} = \mathsf{Xch}_{\mathrm{F}}$, and*

$$\begin{array}{rcl} \mathsf{st}_{\mathbb{A}|\mathbb{O}}(m_{\mathbb{A}}, m_{\mathbb{O}}) & \stackrel{def}{=} & ((\mathcal{O}, q), m'_{\mathbb{A}}) \leftarrow \mathsf{A}(m_{\mathbb{A}}); \\ & & (a, m'_{\mathbb{O}}) \leftarrow \mathcal{O}(q, m_{\mathbb{O}}); \\ & & m''_{\mathbb{A}} \leftarrow \mathsf{A}_{\downarrow}((\mathcal{O}, q, a), m'_{\mathbb{A}}); \\ & & \mathsf{return}\ ((\mathcal{O}, q, a), (m''_{\mathbb{A}}, m'_{\mathbb{O}})) \end{array}$$

Let $k : \mathsf{N}_{\mathbb{O}} \to \mathbb{N}$. An adversary is called *k-bounded*, if for every $\mathcal{O} \in \mathsf{N}_{\mathbb{O}}$, the number of queries to $\mathcal{O}$ in every execution of $\mathbb{A} \mid \mathbb{O}$ is not greater than $k(\mathcal{O})$. An adversary is called *bounded*, if it is k-bounded for some k. Thus, k bounds the number of oracle calls that can be performed by an adversary. To meaningfully state security properties of oracle systems, we do not only need to bound the number of oracle calls but also the adversary's global running time. Therefore, we rather consider bounds of the form $(k, t) \in (\mathsf{N}_{\mathbb{O}} \to \mathbb{N}) \times \mathbb{N}$ and talk about (k, t)-bounded adversaries, whose set we denote $\mathsf{Adv}(k, t)$.

Security properties abstract away from the state of adversaries, and are modeled using traces. Informally, a trace τ is an execution sequence η from which the adversary memories have been erased.

DEFINITION 5. *Let $\mathbb{O}$ be an oracle system.*

- *A* partial trace *is a sequence τ of the form $m_0 \xrightarrow{X_1} m_1 \xrightarrow{X_2} \ldots \xrightarrow{X_k} m_k$, where $m_0, \ldots, m_k \in \mathsf{M}_{\mathbb{O}}$ and $X_1, \ldots, X_k \in \mathsf{Xch}$ such that for $i = 1, \ldots, k$ and $X_i = (\mathcal{O}_i, q_i, a_i)$, $\Pr[\mathcal{O}_i(q_i, m_{i-1}) = (a_i, m_i)] > 0$. A* trace *is a partial trace τ such that $m_0 = \bar{m}_{\mathbb{O}}$ and $X_k = (o_{\mathrm{F}}, \_, \_)$.*
- *An $\mathbb{O}$-event E is a predicate over $\mathbb{O}$-traces, whereas an extended $\mathbb{O}$-event E is a predicate over partial $\mathbb{O}$-traces.*

We say that two finite partial traces τ_1 and τ_2 are *concatenable* traces, denoted $\tau_1 :: \tau_2$, iff the last memory involved in τ_1 is the first involved in τ_2.

The probability of an (extended) event is derived directly from the definition of $\mathbb{A} \mid \mathbb{O}$: since the fact that each execution sequence $\eta \in \mathsf{Exec}(\mathbb{A}|\mathbb{O})$ induces a trace $\mathcal{T}(\eta)$ simply by erasing the adversary memory at each step, one can define for each trace τ the set

$\mathcal{T}^{-1}(\tau)$ of execution sequences that are erased to τ, and for every (extended) event E the probability:

$$\begin{array}{rcl}\Pr(\mathbb{A}|\mathbb{O}:E) & = & \Pr(\mathbb{A}|\mathbb{O}:\mathcal{T}^{-1}(E)) \\ & = & \sum_{\{\eta\in\mathsf{Exec}(\mathbb{A}|\mathbb{O})|E(\mathcal{T}(\eta))=\mathsf{true}\}}\Pr(\mathbb{A}|\mathbb{O}:\eta)\end{array}$$

In the sequel, we will consider *functional* oracles that respond with the same answer to a query asked multiple times. Formally, we say that an oracle $\mathcal{O}$ of $\mathbb{O}$ is *functional and distributed as* $\mathcal{O}^*$, if 1.) every memory $m \in \mathsf{M}_{\mathbb{O}}$ contains a variable $L_{\mathcal{O}}$ that takes values in $\mathsf{Xch}(\mathcal{O})^*$, 2.) $\bar{m}_{\mathbb{O}}.L_{\mathcal{O}}$ is the empty list and 3.) the implementation of $\mathcal{O}$ is as follows, where $\mathcal{O}^*(x)$ is a distribution on $\mathsf{Out}_{\mathcal{O}}$ (possibly depending on input x):

$\mathsf{Imp}(\mathcal{O})(x) =$ if $x \in \mathsf{dom}(L_{\mathcal{O}})$ then return $L_{\mathcal{O}}(x)$
else $y \leftarrow \mathcal{O}^*(x)$; $L_{\mathcal{O}} := L_{\mathcal{O}}.(x,y)$; return y
endif

2.2 Hash construction indifferentiability

Cryptographic hash functions are usually built out of a set of functions, called *inner-primitives*, that are applied successively to blocks that constitute the input message. While the hash function is supposed to take as input messages of any length and produce a hash of a fixed length, the inner-primitives have fixed input/output length. Therefore, hash functions are based on so-called *domain extenders* [24, 15], that specify how the input message is split into blocks and how the inner-primitives are applied to the current block and the previous inner-primitive outputs.

In [9], a formal definition for domain extenders is presented Though applicable to several known constructions, this definition suffers from the fact that it does not capture constructions that include a post-processing function. Such a function is used to compute the global hash result out of the multiple inner-primitive outputs. Another limitation is that it does not deal with the case of multiple inner-primitives. For instance, the Chop [14] and Grøstl [21] constructions are out of scope of this definition. Therefore, we propose a new definition based on the notion of *overlayer* that allows us to capture all hash functions based on domain extenders we are aware of. A hash construction can then be described as an overlayer applied to an oracle system, where this latter defines the inner-primitives. The definition of overlayers exploits the fact that in all known hash designs the order in which the inner-primitives are called is input independent. In other words, the sequence of calls of inner-primitives generated by every hash input is the prefix of a statically known finite sequence $[\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}]$ of inner-primitives.

DEFINITION 6. *Consider an oracle system $\mathbb{O}$ with oracles in $\mathsf{N}_{\mathbb{O}}$. An $\mathbb{O}$-overlayer h is a tuple $(\mathsf{In}_{\mathcal{H}}, \mathsf{Out}_{\mathcal{H}}, [\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}], \mathsf{init}, \Theta, (\mathsf{H}_j)_{j\in\{1..\mathfrak{L}\}}, \mathsf{H}_{post}, Ind_{post})$ where:*

- *$\mathsf{In}_{\mathcal{H}}$ and $\mathsf{Out}_{\mathcal{H}}$ are finite sets of bitstrings defining query and answer domain of the hash design.*
- *$[\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}]$ is the statically known sequence of oracles in $\mathsf{N}_{\mathbb{O}}$, which describes the order in which the oracles are queried.*
- *$\mathsf{init} : \mathsf{In}_{\mathcal{H}} \to [1,\mathfrak{L}]$ outputs the number of oracle calls necessary for computing the hash of x. Notice that $\mathsf{init}(x)$ determines the exact sequence of these calls. Indeed, for computing the hash of an input x, this sequence is the prefix of $[\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}]$ of length $\mathsf{init}(x)$. We require that for any x, x', $\mathcal{O}^{\mathsf{init}(x)} = \mathcal{O}^{\mathsf{init}(x')}$, which we denote by $\mathcal{O}_{\ell ast}$.*
- *$\Theta : \mathsf{In}_{\mathcal{H}} \to (\{0,1\}^{\leq r})^+$: $\Theta(x) = (\theta_1(x),\ldots,\theta_{\mathsf{init}(x)}(x))$ where $\theta_j(x)$ is the function of the input used to compute the j-th query to oracle $\mathcal{O}^j$. It usually consists in r-blocks of the padded input x. We suppose that Θ is injective.*
- *functions $\mathsf{H}_1 : \{0,1\}^{\leq r} \to \mathsf{In}(\mathcal{O}^1)$ and $\mathsf{H}_j : \{0,1\}^{\leq r} \times \mathsf{Xch} \to \mathsf{In}(\mathcal{O}^j)$ for $j \geq 2$ compute the j-th query performed by $\mathcal{H}$ using $\theta_j(x)$ and when $j \geq 2$ the previous step exchange with oracle $\mathcal{O}^{j-1}$.*
- *$\mathsf{H}_{post} : \mathsf{In}_{\mathcal{H}} \times \dot{\mathsf{Xch}}^* \to \mathsf{Out}_{\mathcal{H}}$ and $Ind_{post} : \mathsf{In}_{\mathcal{H}} \to \mathfrak{P}([1,\mathfrak{L}])$: $\mathsf{H}_{post}(x,[Q]_{k\in Ind_{post}(x)})$ is the hash of x, if $Q = (\mathcal{O}^k, q^k, a^k)_{k\in[1,\mathsf{init}(x)]}$ is the list of exchanges generated by the H_j functions for x.*

The set of $\mathbb{O}$-overlayers is denoted by $\mathbb{O}$-OverL.

DEFINITION 7. *The composition of an $\mathbb{O}$-overlayer h with $\mathbb{O}$ defines an oracle system which contains the oracles of $\mathbb{O}$ augmented with the overlayer oracle $\mathcal{H}$ given by:*

- *the memory $L_{\mathcal{H}}$ of oracle $\mathcal{H}$ is a mapping from $\mathsf{In}_{\mathcal{H}}$ to $\mathsf{Out}_{\mathcal{H}} \times \mathsf{Xch}^*$; its initial value is the empty mapping;*
- *The implementation of oracle $\mathcal{H}$ is:*

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) =$ if $x \in \mathsf{dom}(L_{\mathcal{H}})$ then
  return $L_{\mathcal{H}}(x)$
else
  $l := \mathsf{init}(x)$;
  $(x_1,\ldots,x_l) := \Theta(x)$;
  $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1)$;
  $a^1 \leftarrow \mathcal{O}^1(q^1)$;
  $Q := [(\mathcal{O}^1, q^1, a^1)]$;
  for $j = 2$ to l do
    $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}))$;
    $a^j \leftarrow \mathcal{O}^j(q^j)$;
    $Q := Q : (\mathcal{O}^j, q^j, a^j)$;
  endfor
  $a^f := \mathsf{H}_{post}(x, [Q]_{k\in Ind_{post}(x)})$;
  $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q)$;
  return a^f
endif

EXAMPLE 2.2. *The oracle system chopMD can be seen as the application of an overlayer to $\mathcal{F}$. The statically known list of oracles is the list of length $\lceil \frac{2^{64}}{r} \rceil$ whose elements are the oracle $\mathcal{F}$, $\mathsf{init}(x) = \lceil |x|/r \rceil$, $\Theta(x)$ is the function padding x into w and then cutting it into r-blocks, $\mathsf{H}_j(x,(\mathcal{F}, q^{j-1}, a^{j-1})) = a^{j-1}||x_j$, and finally $\mathsf{H}_{post}(x,(\mathcal{F},q,a)) = \mathsf{First}_{n-s}(a)$. Eventually, $Ind_{post}(x) = \{\mathsf{init}(x)\}$, since the only exchange used by H_{post} is the last one performed during an execution of $ChopMD_s$.*

EXAMPLE 2.3. *The sponge construction [6] relies on an inner primitive $\mathcal{F}$, which is a random function from $\{0,1\}^{r+c}$ into $\{0,1\}^{r+c}$, where r is the length of blocks parsed during preprocessing. The output size is parameterized by an integer we denote K. While the general design deals with any possible K, in the sequel we assume for sake of simplicity that $K = kr$, and refer the readers to [6] for more details. The sponge algorithm comprises two phases: firstly, the input is padded using Pad_{sp}, an injective, easily computable and invertible padding function outputting a bitstring $x_1||\ldots||x_p$ of length $p * r$. Then, the algorithm iteratively applies a bitwise xor operation to x_j and the first r-block of the previous answer from $\mathcal{F}$ to compute its next query. Secondly, it queries $\mathcal{F}$ k times more to get a collection of answers $(a^{p+1},\ldots,a^l)$. The final output is then obtained by concatenation of the first r bits of each a^j: $\mathsf{First}_r(a^{p+1})||\ldots||\mathsf{First}_r(a^l)$. The implementation is provided in Figure 2.2.*

The sponge hash oracle results from the composition of an overlayer to $\mathcal{F}$. As a bound $\mathfrak{L}_{sp}$, we choose as previously $\lceil \frac{2^{64}}{r} \rceil$. Function init_{sp} is precised in Figure 2.2, as long as a formula for the index p marking the last step of the first phase of the computation. We

just write l and p for these integers in what follows. On top of that we have $\mathsf{H}_{post}(x, [(q^j, a^j)]_{j\in[p+1,\ldots,l]}) = \mathsf{First}_r(a^{p+1})||\ldots||\mathsf{First}_r(a^l)$, which allows to precise $Ind_{post} = \{p+1, \ldots, l\}$. We define $\mathsf{H}_j(\alpha, (\mathcal{F}, q^{j-1}, a^{j-1}))$ by $(\alpha||0^c \oplus a^{j-1})$. Moreover, we let $\theta_j(x) = x_j$ for $j \in \{1..p\}$, and $\theta_j(x) = \lambda$ for $j \in \{p+1..l\}$ (p and l depend on input x).

$\mathsf{In}(Sponge) = \{0,1\}^{\leq 2^{64}}$, $\mathsf{Out}(Sponge) = \{0,1\}^K$
$\mathsf{Imp}(Sponge)(x) =$
if $x \in \mathsf{dom}(L_{sp})$ then
  return $L_{sp}(x)$
else
  $l := \mathsf{init}_{sp}(x)$;
  $w := \mathsf{Pad}_{sp}(x)$;
  $p := \frac{|w|}{r}$;
  $(x_1, \ldots, x_p) := (w[1,r], \ldots, w[r*(p-1)+1, r*p])$;
  for $j = 1$ to p do
    $q^j := (x_j||0^c) \oplus a^{j-1}$;
    $a^j \leftarrow \mathcal{F}(q^j)$;
    $Q := Q : (q^j, a^j)$;
  endfor
  for $j = p+1$ to l do
    $q^j := a^j$;
    $a^j \leftarrow \mathcal{F}(q^j)$;
    $Q := Q : (q^j, a^j)$;
  endfor
  $a^f := \mathsf{First}_r(a^{p+1})||\ldots||\mathsf{First}_r(a^l)$;
  $L_{sp} := L_{sp}.(x, a^f, Q)$;
  return a^f
endif

where $\mathsf{init}_{sp} = p + k$, with $p = \lceil |x|/r \rceil + \delta_{\mathsf{Last}_r(x)=0^r}$ (the design specifies that $\mathsf{Pad}_{sp}(x)$ should not end with 0^r), and where $k * r$ is the length of the output of the sponge construction.

Figure 2: Sponge Implementation

Indifferentiability of hash designs can be seen as a notion comparing an oracle system composed with a matching overlayer and an *idealization of the overlayer*. The idealization usually used is that of an independent random oracle. We denote by $\mathcal{U}_{\mathcal{H}}$ the oracle system where $\mathcal{H}$ is the functional oracle distributed as the uniform distribution on $\mathsf{Out}_{\mathcal{H}}$.

DEFINITION 8. *Consider an oracle $\mathbb{O}$ and an $\mathbb{O}$-OverL h. The system $(\mathcal{H}^{\mathbb{O}}, \mathbb{O})$ defined by the composition of h with $\mathbb{O}$ is said to be* indifferentiable from its idealization $\mathcal{U}_{\mathcal{H}}$*, if there is an oracle set $\mathbb{S}^{\mathcal{U}_{\mathcal{H}}}$ that is (k_s, t_s)-bounded and such that the oracle systems $(\mathcal{H}^{\mathbb{O}}, \mathbb{O})$ and $(\mathcal{U}_{\mathcal{H}}, \mathbb{S}^{\mathcal{U}_{\mathcal{H}}})$ are compatible and ϵ-indistinguishable, for any adversary $\mathbb{A} \in \mathsf{Adv}(k,t)$, $Indiff(\mathcal{H}, \mathbb{O}, \mathbb{S}) \leq \epsilon(k,t)$, where $Indiff(\mathcal{H}, \mathbb{O}, \mathbb{S})$ denotes*

$$|Pr[\mathbb{A}|(\mathcal{H}^{\mathbb{O}}, \mathbb{O}) : \textit{true}] - Pr[\mathbb{A}|(\mathcal{U}_{\mathcal{H}}, \mathbb{S}^{\mathcal{U}_{\mathcal{H}}}) : \textit{true}]|$$

The oracle set $\mathbb{S}$ of this definition is usually refered to as the simulator. It is not a stand-alone oracle system, since it requires access to $\mathcal{U}_{\mathcal{H}}$ to compute its outputs.

3. A GENERIC SIMULATOR FOR HASH DESIGNS BASED ON RANDOM FUNCTIONS

3.1 Particular inner-primitives

We can distinguish three categories of oracle systems $\mathbb{O}$ on top of which hash constructions are built. The first case is when $\mathbb{O}$ only contains *independent* oracles. Then, no component of state is shared between any oracles of the system and no intern querying is allowed. Therefore, oracle $\mathcal{O}^i$ can only modify its own component of state M_i, but no other component, not even through querying another oracle of the system. The other possibility is that some oracles in the system depend on each other. We separately study two kinds of dependencies: dependency via query and dependency via shared memory. We choose the first situation as a starting point to build and expose our generic approach.

In this section, we consider a given hash design using a set of independent inner-primitives. The construction is modeled by an overlayer h applied to an oracle system $\mathbb{O}$. Our goal is to provide a proof that the overlayer oracle $\mathcal{H}$ is indifferentiable from oracle $\mathcal{U}_{\mathcal{H}}$ when $\mathbb{O}$ implements independent random functions. Formally, we denote $\mathcal{U}_{\mathbb{O}}$ the oracle system compatible with $\mathbb{O}$, such that any oracle $\mathcal{O}^i$ of the system is functional and distributed as $\mathcal{U}_{\mathsf{Out}_i}$. We must then provide an implementation for a generic simulator and a way to compute a bound to the probability of distinguishing between the original system $(\mathcal{U}_{\mathbb{O}}, \mathcal{H}^{\mathcal{U}_{\mathbb{O}}})$ and its simulated counterpart $(\mathbb{S}^{\mathcal{U}_{\mathcal{H}}}, \mathcal{U}_{\mathcal{H}})$. We recall that the simulator has the same set of oracles that $\mathbb{O}$; we thus provide implementations for all oracles in $\mathbb{O}$. It could be the case that some of the oracles in $\mathbb{O}$ do not appear in the overlayer static sequence of oracles to call. In such a case, it is obvious that the oracle in the real and simulated world can be implemented in the same way. As a result, we suppose that all oracles in $\mathbb{O}$ do appear in the overlayer sequence.

3.2 Defining a simulator

When designing a simulator, one must be careful to preserve dependencies between oracles of $\mathbb{O}$ and the overlayer oracle $\mathcal{H}$. Indeed, any inconsistency potentially allows the attacker to distinguish between the real and simulated world. In particular, if an equality holds in the real world and can be efficiently checked by the adversary, then the simulator has to contrive it to hold as well. A well-known example of such a situation is the following: the adversary can compute a hash value for x on its own, say h_1, using the oracles in $\mathbb{O}$, and then verify the result by querying $\mathcal{H}$ and obtaining h_2. If the simulator does not detect such trials, it is very likely that equality between h_1 and h_2, while holding in the real world, is not verified in the simulated world.

A first idea is to provide simulator with a way to represent dependencies between $\mathbb{O}$-queries with respect to $\mathcal{H}$-queries. We are interested in depicting what we call *request chains*, that is to say, lists of exchanges with oracles in $\mathbb{O}$ corresponding to sequences of requests of use to the computation of some hash value. To do that, graphs appear to be well-suited data structures. They constitute our point of departure.

DEFINITION 9. *A* simulator graph $G = (v_{root}, V, E)$ *is given by:*

- *a root v_{root},*
- *a finite vertex set $V \subseteq \mathsf{Xch}$,*
- *a labeled edge set $E \subseteq (V \cup v_{root}) \times (\{1..\mathfrak{L}\} \times \{0,1\}^{\leq r}) \times V$ such that:*
  1. *for all $(\mathcal{O}^{j-1}, q, a), (\mathcal{O}^j, q', a') \in V$ (with $j \geq 2$), for all $(j, x_j) \in \{1..\mathfrak{L}\} \times \{0,1\}^{\leq r}$, $((\mathcal{O}^{j-1}q, a), (j, x_j), (\mathcal{O}^j, q', a')) \in E$ if and only if $q' = \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q, a))$,*
  2. *for all $(\mathcal{O}^1, q, a) \in V$, $(v_{root}, (1, x_1), (\mathcal{O}^1, q, a)) \in E$ if and only if $q = \mathsf{H}_1(x_1)$.*

The set of simulator graphs is denoted by $\mathcal{SG}$, and we define the initial simulator graph $G_{init} = (v_{root}, \{v_{root}\}, \emptyset)$. We now provide a few useful definitions related to paths in this graph.

DEFINITION 10. *A* rooted path *is a path starting with vertex v_{root}. A vertex is* rooted *whenever it belongs to a rooted path. A* meaningful path *is a rooted path such that if $[(1, x_1), \ldots, (L, x_L)]$ is the list of labels on the sequence of edges, then there exists x such that $\forall j = 1..L$, $\theta_j(x) = x_j$. Such a path is said to be* complete *when $L = \mathsf{init}(x)$. Bitstring x is then said to* label *the complete meaningful path to which it corresponds.*

Intuitively, we want to catch the attempts of the adversary to build a request chain which it could use to effictively compare a hash value built on its own with the result of a query to $\mathcal{H}$. The expression 'last query of a hash value' is a shorthand that we use to name the last query to be performed to $\mathbb{O}$ during the computation of a hash value in the real setting. First, let us acknowledge that in most hash constructions, on the one hand, it is difficult for an adversary to compute a hash value without querying for its matching last query. On the other hand, it is difficult to perform this last query without having performed before every other query of the real chain of requests used to compute the hash result. Therefore, a strategy to outsmart the adversary consists in trying to identify potential last queries when they are asked to $\mathbb{O}$.

To do so, we can use the graph: if the adversary has asked all queries but the last necessary to build the hash value for a given x, then when last query q is performed, there exists a rooted path in the simulator graph to which q can be linked as the last vertex. This would require an update of the graph by the simulator at every query this latter receives. To describe such an update process, we begin by defining the following function.

DEFINITION 11. *We let $\overline{E} : V \times V \to \mathfrak{P}((V \cup v_{root}) \times (\{1..\mathfrak{L}\} \times \{0,1\}^{\leq r}) \times V)$ be the function mapping a pair of vertice (v, v') to the (possibly empty) set of edges linking v to v'.*

DEFINITION 12. *A simulator graph update function* $\mathsf{upd}G$ *is a function which, on input an exchange $(\mathcal{O}, q, y)$ and a simulator graph $G = (v_{root}, V, E)$, outputs the simulator graph G' given by:*

- $V' = V \cup \{(\mathcal{O}, q, y)\}$;
- $E' = E \cup \{\overline{E}((\mathcal{O}, q, y), v), v \in V\} \cup \{\overline{E}(v, (\mathcal{O}, q, y)), v \in V \cup v_{root}\}$.

Assuming that there exists a bound $t_{\overline{E}}$ of the execution time of $\overline{E}$ independent of its inputs, the time required to update a graph G containing α vertice is bound by $2\alpha t_{\overline{E}}$. This is a safe assumption in the sense that the number of edges that can exist between two vertice is finite. That being said, a simulator taking a very significant amount of time to update its graph is not very interesting. We discuss this in more details after specifying the simulator.

To put to work the simulating strategy suggested above, we need an algorithm which, on input a graph and a query, identifies a rooted path rendered complete by this query when it exists, and in such a case provides the complete path and the hash input labeling it. Such an algorithm is called a *path-finder*. Intuitively, it should have a non-trivial output as soon as there exists a satisfactory path, and any non-trivial output of $\mathcal{P}$ should indeed correspond to a satisfying answer.

DEFINITION 13. *A* path-finder algorithm $\mathcal{P}$ *takes as input a potential last query $q \in \mathsf{In}_{\mathcal{O}_{\ell ast}}$ and a simulator graph G. Its output is either triple $(\mathsf{false}, \lambda, [\,])$, or triple $(\mathsf{true}, x, List)$, with $(x, List) \in \mathsf{In}_{\mathcal{H}} \times V^*$ such that:*

1. *if for any answer y to q, there exists in the updated graph $\mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y), G)$ a complete meaningful path then we have $\mathcal{P}(q, G) \neq (\mathsf{false}, \lambda, [\,])$.*
2. *if $\mathcal{P}(q, G) = (x, [(\mathcal{O}^1, q_1, y_1), \ldots, (\mathcal{O}^{l-1}, q_{l-1}, y_{l-1})])$ then for any answer y to q, there exists in the updated graph $\mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y)\ , G)$ a complete meaningful path going (in this order) through $(\mathcal{O}^j, q_j, y_j)_{j=1..(l-1)}$ and (q, y) and labeled by x.*

We assume that the execution time of the path-finder algorithm is bound by a time $t_{\mathcal{P}}(\alpha)$, where α is the number of vertice in the input graph. We can safely assume that this time grows with α. To provide a more precise estimation of the execution time, we can consider the following rough specification for the path-finder. It first proceeds to the a graph update with vertex (q, λ) and edges in $\overline{E}(v, (q, \lambda))$ for vertice v in its input graph. This yields a bound of time of $\alpha t_{\overline{E}}$ for this operation. Then, the path-finder searches for a path in the resulting graph, taking a time assessed as $\mathcal{O}(\alpha^2)$. Eventually, it has to find the value of x labeling the future complete path, which requires to invert Θ in some time t_Θ.

In the event that $\mathcal{P}$ outputs some bitstring and request chain $(x, [v_1, \ldots, v_{\mathsf{init}(x)-1}])$, it must influence the answer y provided by the simulator to match a query q. Indeed, we have pointed out that the simulator must enforce equality between the hash value for x and the post-processing applied to the full request chain. If we denote $v_{\mathsf{init}(x)} = (\mathcal{O}_{\ell ast}, q, y)$, we would like to guarantee $\mathsf{H}_{post}(x, [v_j]_{j \in Ind_{post}(x)}) = \mathcal{H}(x)$ in the simulated world. When $\mathcal{H}$ is modeled as the oracle system $\mathcal{U}_{\mathcal{H}}$, the only way to compute the value for $t = \mathcal{H}(x)$ is to query the random oracle on x. After that, we would have to provide the simulator with a sampling algorithm to find a value for y such that the equation holds.

For a given value of t_0, there is a set of values y such that $\mathsf{H}_{post}(x, [v_j]_{j \in Ind_{post}(x)}) = t_0$, which we denote $PreIm(t_0)$. Notice here that if we have specified the overlayer such that $\mathsf{Out}_{\mathcal{H}}$ is larger than the actual range of $\mathcal{H}$, some sets $PreIm(t_0)$ are empty. In the sequel we suppose that $\mathsf{Out}_{\mathcal{H}}$ *is the range of* $\mathcal{H}$, so that for all t_0, $PreIm(t_0)$ contains at least one element. To choose one value out of this set of solutions and preserve the original distribution of pairs (y, t), we define distribution $\mathsf{H}^{-1}_{post}(x, List, t_0, q)$ such that for every $y_0 \in PreIm(t_0)$, we have:

$$Pr[\mathsf{H}^{-1}_{post}(x, List, t_0, q) = y_0] = \frac{Pr[y \leftarrow \mathcal{U}_{\mathcal{O}_{\ell ast}} : y = y_0]}{Pr[\mathcal{U}_{\mathcal{H}} = t_0]}$$

Since the numerator does not depend on y_0, we thus consider the uniform distribution on $PreIm(t_0)$. We choose to denote the algorithm sampling in $PreIm(t_0)$ like the distribution. We assume that the execution time of this algorithm is upper bounded by time t_{post}, which is independent of inputs.

DEFINITION 14. *The generic simulator $\mathbb{S}$ is a set of oracles compatible with $\mathbb{O}$. These oracles have memories $(L_{\mathcal{S}}, G) \in \mathsf{Xch} \times \mathcal{SG}$, and the initial memory $\bar{m}$ of a system containing the simulator is chosen so that $\bar{m}.(L_{\mathcal{S}}, G) = ([\,], G_{init})$. Moreover $\mathcal{O}_{\ell ast}$ is implemented as follows:*

$\mathsf{Imp}_{\mathbb{S}}(\mathcal{O}_{\ell ast})^{\mathcal{H}}(q) =$ if $q \in \mathsf{dom}(L_{\mathcal{S}})$ then
  return $L_{\mathcal{S}}(q)$
elsif $\mathcal{P}(q, G) \neq \perp$ then
  $(x, List) := \mathcal{P}(q, G)$;
  $t \leftarrow \mathcal{H}(x)$;
  $y := \mathsf{H}^{-1}_{post}(x, List, t, q)$;
else $y \leftarrow \mathcal{U}_{\mathcal{O}_{\ell ast}}$;
endif
$G := \mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y), G)$;
$L_{\mathcal{S}} := L_{\mathcal{S}}.(\mathcal{O}_{\ell ast}, q, y)$;
return y

This implementation is (k_s, t_s)-bounded, where $k_s(\mathcal{O}) = \delta_{\mathcal{O}=\mathcal{H}}$, and $t_s = 2\alpha t_{\overline{E}} + t_{post} + t_{\mathcal{P}}(\alpha)$ with α a bound of the number of vertice in G. For any $\mathcal{O} \neq \mathcal{O}_{\ell ast}$ in $\mathsf{N}_{\mathbb{O}}$, the simulator implementation is:

$\mathsf{Imp}_{\mathbb{S}}(\mathcal{O})^{\mathcal{H}}(q) =$ **if** $q \in \mathsf{dom}(L_{\mathcal{S}}(\mathcal{O}))$ **then**
  return $L_{\mathcal{S}}(\mathcal{O}, q)$
else $y \leftarrow \mathcal{U}_{\mathcal{O}}$;
endif
$G := \mathsf{upd}G((\mathcal{O}, q, y), G)$;
$L_{\mathcal{S}} := L_{\mathcal{S}}.(\mathcal{O}, q, y)$;
return y

This implementation is $(0, t'_s)$-bounded, where $t'_s = 2\alpha t_{\overline{E}}$ with α a bound of the number of vertex in G.

This simulator works completely independently of the fact that an update might result in $\overline{E}$ creating a great number of edges between two given vertice of the simulator graph. However, we notice that if it is possible that the path-finder can answer two distinct hash inputs x, x' corresponding to meaningful paths, the simulator can only anticipate the adversary query for one of these inputs to $\mathcal{H}$. As a consequence, we highlight that the following partial-injectivity hypothesis rather be satisfied by applications H_j.

DEFINITION 15. *For $j \geq 2$, function $\mathsf{H}_j : \{0,1\}^{\leq r} \times \mathsf{Xch} \rightarrow \mathsf{In}(\mathcal{O}^j)$ is* partially injective *iff given (q, a) and q' there exists a unique $u \neq \lambda$ such that $\mathsf{H}_j(u, (\mathcal{O}^{j-1}, q, a)) = q'$. Similarly, $\mathsf{H}_1 : \{0,1\}^{\leq r} \rightarrow \mathsf{In}(\mathcal{O}^1)$ is partially injective iff given q', there exists a unique $u \neq \lambda$ such that $\mathsf{H}_1(u) = q'$.*

Besides, this hypothesis allows to bound quite accurately the execution time of the simulator since $\overline{E}$ can create a maximum of $2 * \mathfrak{L}$ edges between two vertice.

We stress that the partial injectivity assumption holds for both examples we have presented in the previous section.

3.3 Bounding the indifferentiability advantage

Even though the path-finder can correct some obvious inconsistencies, there are still cases in which it is not sufficient. When a last query is made to the simulator, consistency can only be enforced if, on the one hand, the path-finder can detect that it is a last query, and on the other hand, there does not exist another value to which the query is already bound. The first annoying event consists in a query made to the inner primitive which is indeed going to be a last query. Firstly, this can happen when the path-finder does not detect being in presence of a last query. This occurs in case all intermediate queries to the inner primitive have not previously been performed, and the query is not visibly-rooted. Secondly, it is also possible that our query becomes a last query after having been asked directly, during a subsequent query to $\mathcal{H}$. First possibility is captured by event ULQ, for Undetected Last Query, while the second event is called DQ-LQ for Direct Query-Last Query. The second source of failure follows from a collision between two last queries corresponding to different hash inputs, or between an intermediate and a last query. These events are denoted SLQ, for Same Last Query, and ILQ, for Intermediate and Last Query. They correspond to those found in [9], which is not surprising this we generalize this work.

We notice that the four events listed above can be visualized using the same kind of graph structure that we have defined for the simulator. As a consequence, we adapt this latter to describe the dependencies between $\mathbb{O}$-queries. Thus, this new graph is built for interactions involving *the real setting* $(\mathcal{H}^{\mathbb{O}}, \mathbb{O})$. There is an important thing that our simulator graph definition does not capture: the *visibility* of a vertex to the adversary. Indeed, if a vertex appears in the graph as a result of a direct query to oracles in $\mathbb{O}$, we should consider the vertex as visible for the adversary. On the contrary, if we consider a vertex appearing in the graph on behalf of intermediate queries, the adversary can only gather information about the queries appearing in the final computation, namely, for exchanges with an index in Ind_{post}. These vertice are deemed partially visible to the adversary, while the remaining exchanges are considered invisible. We take these elements into account in our definition for a characteristic graph.

DEFINITION 16. *A* characteristic graph *CG is defined by a tuple $(v_{root}, CV, CE, \mathcal{V})$ where :*

- *v_{root}, CV and CE are such that $(v_{root}, CV, CE) \in \mathcal{SG}$*
- *$\mathcal{V}$ is a visibility map, which associates to every vertex in CV a value in $\{Vis, PVis, Inv\}$ (standing for visible, partially visible and invisible and are ordered this way).*

The set of characteristic graphs is denoted by $\mathcal{CG}$. We distinguish a particular graph $CG_{init} = (v_{root}, [\,], \emptyset, \mathcal{V}_{init})$ with $\mathsf{dom}(\mathcal{V}_{init}) = \emptyset$ which we call the initial characteristic graph.

If we build a characteristic graph while the execution takes place, updating it at each (direct or indirect) query of an oracle in $\mathbb{O}$, we notice that three of the bad events result in some kind of collision in the graph. If a query is both intermediate and last query for two hash inputs, it belongs to two distinct meaningful paths - note that a loop on a vertex is possible. Likewise, two different hash inputs with the same last query yield a collision between meaningful paths. Moreover, event DQ-LQ results in the creation of an edge between a meaningful path and a preexisting but not necessarily rooted vertex. These three events can be merged into event $Collide$ corresponding to the issue of a query $(\mathcal{O}, q, y)$ creating a collision between a meaningful path containing $(\mathcal{O}, q, y)$ and a vertex of the updated[1] graph. Finally, when an undetected last query is performed, a meaningfully rooted vertex v from the non-visible part of the graph becomes visible. Let P be the visible part of a meaningful path from a root to vertex v, and let v' be the head of P. Of course, v' is not linked to v_{root} by an edge, otherwise v would be visibly rooted. Nevertheless, v' is rooted, so that there exists an edge from an non-visible rooted node to v', first visible node of path P. As a consequence, the probability of realizing event ULQ is bounded by that of the adversary issuing a direct query to oracles in $\mathbb{O}$ resulting in such a v', an event we name $Reveal$.

We aim to express the bound to the indifferentiability of the hash construction at hand in terms of the probability that these events occur. To enable this, we need to introduce a formal definition for the sequence of characteristic graphs involved in an execution. To be able to capture the occurence of $Collide$, we want this sequence to contain every intermediate updated graph for intermediate queries during a hash computation to be able to write our $Collide$ event. Since we already know how to compute probabilities of events on traces, we choose to define a way to transform traces into characteristic graph sequences. This in turn allows us to formalize events expressed on graphs and their probabilities.

DEFINITION 17. *A* partial graph-trace *is defined by a finite sequence $CG_0 \xrightarrow{v_1} CG_1 \xrightarrow{v_2} \ldots \xrightarrow{v_k} CG_k$ of vertice v_i and characteristic graphs $CG_i = (v_{root}, CV^i, CE^i, \mathcal{V}^i)$ such that $(v_{root}, CV^{i+1}, CE^{i+1}) = \mathsf{upd}G(v_i, (v_{root}, CV^i, CE^i))$ and for all $v \in CV^i, \mathcal{V}^{i+1}(v) \geq \mathcal{V}^i(v)$.*

Similarly to notations introduced for partial traces, partial graph-traces τ^1_G and τ^2_G are concatenable as soon as last graph of τ^1_G is the

[1] to take into account the possibility of creating a loop

first appearing in τ_G^2. It is denoted by $\tau_G^1 :: \tau_G^2$. A graph-step is a partial graph-trace of length 1. It does not necessarily correspond to a step w.r.t. traces.

We can see any finite trace as the concatenation of a finite number of steps. Hence, if we have functions to transform steps into partial graph-traces, we can use them as building blocks to define a function mapping finite traces to partial graph-traces. There are two kinds of actions which can be performed in a step, calls to oracles of $\mathbb{O}$ or calls to $\mathcal{H}$. We define two distinct partial graph-trace building functions to deal with the two possibilities. They take as arguments a characteristic graph $CG = (v_{root}, CV, CE, \mathcal{V})$, and a list of exchanges with oracles in $\mathbb{O}$ and output the partial graph-trace corresponding to the sequence of characteristic graphs created by performing these successive calls to oracles in $\mathbb{O}$.

DEFINITION 18. *Let CG_0 be a characteristic graph with visibility map $\mathcal{V}^0$, and v be a direct query to an oracle $\mathcal{O}$ in $\mathbb{O}$ and its answer. The partial graph-trace building function for $\mathcal{O}$ calls is defined as follows:*

$\mathcal{O}\mathsf{TrBuilder}(CG_0, v) = CG_0 \xrightarrow{v} CG_1$ *where the visibility map of CG_1 is given by* $\mathcal{V}^1(v') = \begin{cases} \mathcal{V}^0(v') \text{ if } v' \neq v, \\ \mathcal{V}^0(v) = Vis. \end{cases}$

DEFINITION 19. *We consider a characteristic graph CG_0, a query to $\mathcal{H}$ denoted x and the list $[v_1, \ldots, v_k]$ of exchanges with $\mathcal{F}$ this query generates.*

The partial graph-trace building function for $\mathcal{H}$ calls is given by: $\mathcal{H}\mathsf{TrBuilder}(CG_0, x, [v_1, \ldots, v_k]) = CG_0 \xrightarrow{v_1} CG_1 \xrightarrow{v_2} \ldots \xrightarrow{v_k} CG_k$ *where, for $i \in \{0, .., (k-1)\}$ visibility map $\mathcal{V}^{i+1}$ of graph CG_{i+1} is defined as:*

$$\mathcal{V}^{i+1}(v) = \begin{cases} \mathcal{V}^i(v) \text{ if } v \neq v_i, \\ max(PVis, \mathcal{V}^i(v)) \text{ if } i \in Ind_{post}, \\ max(Inv, \mathcal{V}^i(v)) \text{ otherwise.} \end{cases}$$

Using these definitions, we can proceed to the specification of the function $\mathsf{TrTransform}$ mapping finite traces to partial graph-traces. We choose to describe it using a an auxiliary function TrGr turning every one-step partial trace into a partial graph-trace given a characteristic graph as starting point. In case the step represents an exchange with $\mathcal{H}$, we need to use the list of exchanges with $\mathbb{O}$ used to compute the output of $\mathcal{H}$. We recall that $L_{\mathcal{H}}$ stores triples whose third component consist of these lists.

DEFINITION 20. *Function* TrGr *transforms a step into a partial graph-trace starting with characteristic graph CG as follows:*

$$\mathsf{TrGr}(m \xrightarrow{X} m', CG) = \begin{cases} \mathcal{O}\mathsf{TrBuilder}(CG, (\mathcal{O}, q, y)), \text{if } X = (\mathcal{O}, q, y). \\ \mathcal{H}\mathsf{TrBuilder}(CG, x, \Pi_3(m'.L_{\mathcal{H}})), \text{if } X = (\mathcal{H}, x, a). \end{cases}$$

DEFINITION 21. *The* trace transformer $\mathsf{TrTransform}$ *is an application that maps a trace of interaction with $(\mathcal{F}, \mathcal{H})$ given by $m_0 \xrightarrow{X_1} m_1 \xrightarrow{X_2} \ldots m_{k-1} \xrightarrow{X_k} m_k$ to a* graph-trace *defined as $\mathsf{TrGr}(m_0 \xrightarrow{X_1} m_1, CG_{init}) :: \ldots :: \mathsf{TrGr}(m_{k-1} \xrightarrow{X_k} m_k, CG)$, where CG denotes the last graph obtained by iteratively transforming steps into partial graph traces until reaching memory m_{k-1} of the trace.*

We call *graph-traces* the images of traces by $\mathsf{TrTransform}$. A *graph-event* is a predicate E_G over a graph-trace. To compute its probability, we have to give the proper weight to graph-traces which satisfy E_G. This is done by taking into account all the executions of all the traces which map to a given graph-trace.

DEFINITION 22. *Let E_G be a graph-event. Its probability is given by:*

$$\Pr(\mathbb{A}|\mathbb{O} : E_G) = \sum_{\{\eta \in ExecSet\}} \Pr(\mathbb{A}|\mathbb{O} : \eta)$$

where $ExecSet = \{\mathsf{Exec}(\mathbb{A}|\mathbb{O}) \mid E_G(\mathsf{TrTransform}(\mathcal{T}(\eta))) = \mathit{true}\}$

DEFINITION 23. *Let $CG = (v_{root}, CV, CE, \mathcal{V})$ and $CG' = (v_{root}, CV', CE', \mathcal{V}')$ be two characteristic graphs such that the vertex $CG \xrightarrow{(\mathcal{O},q,y)} CG'$ is a graph-step.*

- *Collide is true at graph-step $CG \xrightarrow{(\mathcal{O},q,y)} CG'$ iff $(\mathcal{O}, q, y) \notin CV$, there exists a meaningful path going through $(\mathcal{O}, q, y)$ in CG', and a collision between a rooted path containing $(\mathcal{O}, q, y)$ and a preexisting vertex is created, i.e. there exists $(\mathcal{O}', q', y') \in CV'$ such that $((\mathcal{O}, q, y), \_, (\mathcal{O}', q', y')) \in CE'$.*
- *Reveal is true at graph-step $CG \xrightarrow{(\mathcal{O},q,y)} CG'$ if $(\mathcal{O}, q, y)$ is the first visible vertex of a meaningful non-visibly rooted path in CG': $\mathcal{V}(\mathcal{O}, q, y) = Vis$ and $\exists (\mathcal{O}', q', y') \in CV$ such that a meaningful path goes through $(\mathcal{O}', q', y')$, the edge $((\mathcal{O}', q', y'), \_, (\mathcal{O}, q, y))$ is in CE, $\mathcal{V}((\mathcal{O}', q', y')) \neq Vis$.*

THEOREM 1. *Let h be an overlayer using as inner-primitives the oracle system idealized as $\mathcal{U}_{\mathbb{O}}$. The composition of the overlayer and $\mathcal{U}_{\mathbb{O}}$ yields oracle system $(\mathcal{H}^{\mathcal{U}_{\mathbb{O}}}, \mathcal{U}_{\mathbb{O}})$. Bounds of the execution time of $\overline{E}$, C_{post}^{-1} and path-finder $\mathcal{P}$ are denoted by $t_{\overline{E}}$, t_{post} and $t_{\mathcal{P}}$. Using the characteristic graph CG and the path-finder algorithm described previously, we can define a simulator $\mathbb{S}$ as in section 3.2. Then, for all adversary $\mathbb{A} \in \mathsf{Adv}(k, t)$,*

- *$\mathbb{S}$ is (k_s, t_s)-bounded with $k_s(\mathcal{H}) = 1$, and $t_s = 2(k'+1)t_{\overline{E}} + t_{post} + t_{\mathcal{P}}(k'+1)$, where $k' = \sum_{\mathcal{O} \in \mathbb{O}} k(\mathcal{O})$.*
- *$|Pr[\mathbb{A}|(\mathcal{H}^{\mathcal{U}_{\mathbb{O}}}, \mathcal{U}_{\mathbb{O}}) : \mathit{true}] - Pr[\mathbb{A}|(\mathcal{U}_{\mathcal{H}}, \mathbb{S}^{\mathcal{U}_{\mathcal{H}}}) : \mathit{true}]| \leq Pr[\mathbb{A}|(\mathcal{H}^{\mathcal{U}_{\mathbb{O}}}, \mathcal{U}_{\mathbb{O}}) : \mathsf{F}_{Collide \vee Reveal}]$*

where $t_{\overline{E}}$, t_{post} and $t_{\mathcal{P}}$ respectively bound of the execution time of $\overline{E}$, C_{post}^{-1} and path-finder $\mathcal{P}$ (this last time depends on the number of vertice in its input graph, here bound by $k'+1$).

A formal proof of the theorem carried out in an extended version of the logic CIL of [3] is provided in [16].

4. EXAMPLES OF APPLICATIONS

4.1 Computation of the Bound for $ChopMD_s$

We consider the hash construction $ChopMD_s$ defined in section 2.1, and a simulator $\mathbb{S}$ implemented as described in section 3.2. We first compute the probability that $\mathsf{F}_{Collide}$ happens when i-th query (q, y) is (directly or indirectly) addressed to $\mathcal{F}$. We assume that the adversary issues a meaningful query, and assess the probability that (q, y) is linked to a vertex $(q', y') \in CV \cup \{(q, y)\}$ by an edge $((q, y), \_, (q', y'))$. For all pair (q', y') in CV, an edge is added between (q, y) and (q', y') iff $y = \mathsf{First}_n(q')$. It means that, to achieve our event, y has to coincide with the prefix of some q' for a vertex $(q', y') \in CV \cup \{(q, y)\}$. This yields i candidates for q'. Besides, y is drawn uniformly at random in $\{0, 1\}^n$. Consequently, the probability that an edge resulting in $\mathsf{F}_{Collide}$ is created at query i is bounded by $\frac{i}{2^n}$.

We then evaluate the probability that the adversary reveals a non-visible query whereas $Collide$ has never happened. If $Reveal$ happens with direct query (q, y), then there exists a non-visible vertex (q', y') linked by an edge to (q, y) by definition of the event. Hence, $y' = \mathsf{First}_n(q)$. Moreover, there is exactly one (q', y')

linked to (q, y) since $Collide$ does not happen. Besides, (q', y') is at worst partially visible, so that the adversary knows at most $n - s$ first bits of y' when issuing query q. The probability that $y' = \mathsf{First}_n(q)$ is thus bounded by $\frac{1}{2^s}$. As a consequence, if a total number of $k(\mathcal{F})$ direct queries are performed to $\mathcal{F}$, the probability of $\mathsf{F}_{Reveal} \wedge \mathsf{G}_{\neg Collide}$ is bounded by $\frac{k(\mathcal{F})}{2^s}$ for our events.

THEOREM 2. *We consider the $ChopMD_s$ construction and simulator $\mathbb{S}$ implemented as in section 3.2. For an adversary $\mathbb{A} \in \mathsf{Adv}(k, t)$,*

$$Indiff(ChopMD_s, \mathcal{F}, \mathbb{S}) \leq \frac{(k_{tot})(k_{tot}+1)}{2^{n+1}} + \frac{k(\mathcal{F})}{2^s}$$

where $k_{tot} = k(\mathcal{F}) + \mathfrak{L} * k(\mathcal{H})$.

Results concerning indifferentiability of the Chop construction already appear in various works. In [14], Coron et. al. determine a bound for this construction considering a random permutation in place of $\mathcal{F}$. In spite of this, it seems relevant to notice that their proof results in a bound of $\mathcal{O}(\frac{(\mathfrak{L}*k_{tot})^2}{2^s})$.

Later, Maurer and Tessaro show in [23] that using a prefix-free padding function allows to conclude to a bound of $\mathcal{O}(\frac{(\mathfrak{L}*k_{tot})^2}{2^n})$. This result is particularly interesting, since it beats the usual birthday bound: indeed, $n - s$ bits are output by the hash function, and n is the output-length of the inner primitive. We notice assuming prefix-free padding, we obtain the same bound. Since no *meaningful* path can be obtained as an extension of a meaningful path, $Reveal$ can only happen when $y' = \mathsf{First}_n(q)$ belongs to an *invisible* vertex. As a consequence, the adversary has to guess all n bits of y' and its probability of success is bounded by $\frac{1}{2^n}$. Our second term is turned into $\frac{k(\mathcal{F})}{2^n}$.

Eventually, in [13], Chang and Nandi provide a very refined computation for $ChopMD_s$ without assuming prefix-free padding. It leads to a bound of

$$\frac{k_{tot}^2}{2^{n+1}} + \frac{(3(n-s)+1) * k(\mathcal{F}) + (n-s) * k(\mathcal{H})}{2^s} + \frac{k(\mathcal{F}) + k(\mathcal{H})}{2^{n-s-1}}$$

or $\mathcal{O}(\frac{3(n-s)(k(\mathcal{F})+k(\mathcal{H}))}{2^s})$. This improves the result given by [14], quadratic in the number of queries. The result we obtain here is to our knowledge the best current bound for the $ChopMD_s$ design.

4.2 Computation of the bound for $Sponge$

We are interested in the sponge design detailed in section 2.2. We start by bounding the probability that $Collide$ is realized at the i-th fresh (direct or indirect) query (q, y) to $\mathcal{F}$. Assuming (q, y) belongs to a meaningful path, we assess the probability of creation of an edge between (q, y) and vertex (q', y'). In a sponge-based characteristic graph, an edge links (q, y) to (q', y') iff $\mathsf{Last}_c(q') = \mathsf{Last}_c(y)$. At step i, it yields a probability of $\frac{i}{2^c}$ that this equality is satisfied for $(q', y') \in CV \cup \{(q, y)\}$. It follows that the probability of $\mathsf{F}_{Collide}$ is bound by $\frac{k_{tot}(k_{tot}+1)}{2^{c+1}}$ for an interaction during which a total number of k_{tot} (direct and indirect) queries are issued to $\mathcal{F}$.

We then assess the probability of $\mathsf{F}_{Reveal} \wedge \mathsf{G}_{\neg Collide}$. Assume the event is realized at the issue of direct query (q, y). In the case of the sponge design, when a vertex (q', y') is partially visible, we must assume that the adversary knows the first r bits of y'. Revealing a non-visible vertex consists in querying $\mathcal{F}$ on a value q of which last c bits coincide with a (at most) partially visible y'. The fact that $Collide$ has never happened allows to assert that there is only one satisfying y' for query q. Potential partial visibility of y' accounts for the fact that only the last c bits of y' are random. As a result, the probability of our event at step i is bounded $\frac{1}{2^c}$, and globally bounded by $\frac{k(\mathcal{F})}{2^c}$ for the complete interaction with the adversary.

THEOREM 3. *We consider the $Sponge$ construction and simulator $\mathbb{S}$ implemented as in section 3.2. For an adversary $\mathbb{A} \in \mathsf{Adv}(k, t)$,*

$$Indiff(Sponge, \mathcal{F}, \mathbb{S}) \leq \frac{k_{tot}(k_{tot}+1)}{2^{c+1}} + \frac{k(\mathcal{F})}{2^c}$$

where $k_{tot} = k(\mathcal{F}) + \mathfrak{L}_{sp} * k(\mathcal{H})$.

In [7], Bertoni et. al. present a clever proof of the indifferentiability the sponge construction concluding to a bound of $\frac{k_{tot}(k_{tot}+1)}{2^{c+1}}$. We obtain a greater bound, containing a term which is omitted in their final bound computation, as was first suggested in [11]. The missing term corresponds to the probability that $Reveal$ happens, which, even though the authors propose a simulator different from ours, should not be overlooked in their computation. Nevertheless, it does not alter the merits of their proof which mainly lie in the graph construction and simulator they propose.

5. GENERALIZATION DEPENDENT INNER-PRIMITIVES HASH DESIGNS

In section 3, the scope of the result we provide is restricted by our hypothesis that the inner-primitives contitute an oracle system with independent oracles. In this section, we outline the theoretical extensions needed to deal with other kinds of systems. We begin with a special case of oracles sharing a memory, namely when a pair of oracles implements a permutation and its inverse. Indeed, at the heart of a great number of designs lie dedicated blockciphers. However, we choose to limit ourselves to this instance of shared memory, since we are not aware of examples which exhibit another type of such dependency. Then, we move on to the case of oracles whose implementations can call other oracles of the system. For clarity of the presentation, we assume that in this latter situation, no memory component is shared between any oracles. It is nonetheless completely affordable to combine both generalizations to carry out a proof for a hash design based on inner-primitives presenting both aspects. In the sequel, we study a given construction modeled by an overlayer h applied to an oracle system $\mathbb{O}$.

5.1 Shared memory : dealing with permutations and their inverses

Let $\mathcal{O}$ and $\mathcal{O}^{-1}$ be a pair of oracles of $\mathbb{O}$ implementing a permutation and its inverse between $\mathsf{In}_{\mathcal{O}}$ and $\mathsf{Out}_{\mathcal{O}}$[2]. We choose to model $\mathcal{O}$ and $\mathcal{O}^{-1}$ by random permutations sharing a list variable $L_{\mathcal{O},\mathcal{O}^{-1}}$. We say that $\mathcal{O}$ and $\mathcal{O}^{-1}$ are *coupled* oracles. In the system $\mathcal{U}_{\mathbb{O}}$, $\mathcal{O}$ is implemented by:

$\mathsf{Imp}(\mathcal{O})(q) =$ if $q \in L_{\mathcal{O},\mathcal{O}^{-1}}$ then return y
else $y \leftarrow \mathcal{U}_{\mathsf{Out}_{\mathcal{O}} - \Pi_2(L_{\mathcal{F},\mathcal{F}^{-1}})}$;
$L_{\mathcal{O},\mathcal{O}^{-1}} := L_{\mathcal{O},\mathcal{O}^{-1}}.(q, y)$; return y
endif

The implementation of $\mathcal{O}^{-1}$ is similar. In the independent oracle scenario, the simulator graph features one vertex type $(\mathcal{O}^j, \_, \_)$ per oracle $\mathcal{O}^j$ of the system. In case of coupled oracles, we choose to represent both exchanges with $\mathcal{O}$ and exchanges with $\mathcal{O}^{-1}$ by the same type of vertex $((\mathcal{O}, \mathcal{O}^{-1}), q, y)$. On top of that, the edge set E remains defined as previously, as well as the function $\overline{E}$ computing edge sets and the update function. The path-finder is provided according to specifications of section 3.2.

[2]It is possible that the permutation only maps a subset of $\mathsf{In}_{\mathcal{O}}$ onto $\mathsf{Out}_{\mathcal{O}}$. In such a case, a similar reasoning can be carried out by splitting the oracle arguments into a key and an input.

That said, the simulator design depends on whether one of the coupled oracles is the last oracle called by a hash computation. If not, the simulator implements the coupled oracles using its shared list variable $L_{\mathcal{S}}$ to emulate the shared memory of $\mathcal{O}$ and $\mathcal{O}^{-1}$ by updating queries (q, y) to $\mathcal{O}$ and $\mathcal{O}^{-1}$ in the form $((\mathcal{O}, \mathcal{O}^{-1}), q, y)$. We just provide below the simulator implementation of $\mathcal{O}$, that of $\mathcal{O}^{-1}$ is strictly the same.

$\mathsf{Imp}_{\mathbb{S}}(\mathcal{O})(q) =$ if $q \in L_{\mathcal{S}}$ then return y
else $a \leftarrow \mathcal{U}_{\mathsf{Out}_{\mathcal{O}} - \Pi_2(L_{\mathcal{S}}(\mathcal{O}, \mathcal{O}^{-1}))}$;
$G := \mathsf{upd}G(((\mathcal{O}, \mathcal{O}^{-1}), q, y), G)$;
$L_{\mathcal{S}} := L_{\mathcal{S}}.((\mathcal{O}, \mathcal{O}^{-1}), q, y)$; return y
endif

In case one of the coupled oracles is the last oracle called, we assume without loss of generality that $\mathcal{O}_{\ell ast} = \mathcal{O}$. In such a case, we need to define the sampling algorithm H^{-1}_{post}, enabling the simulator to answer coherently with $\mathcal{U}_{\mathcal{H}}$ when it detects a last query. As previously, let $PreIm(t_0)$ denote the set of values y such that $\mathsf{H}_{post}(x, [v_j]_{j \in Ind_{post}(x)}) = t_0$, for given x and list $[v_j]_{j \in Ind_{post}(x)}$ such that $v_{\mathsf{init}(x)} = (q, y)$. We recall that we mentioned the case when $PreIm(t_0)$ is empty in section 3, and excluded it by assuming that $\mathsf{Out}_{\mathcal{H}}$ *was the range of* $\mathcal{H}$. In the present case, the simulator has, on top of the consistency constraint, to preserve the one-to-one property of the mapping yield by our coupled oracles. As a consequence, the sampling algorithm should not draw uniformly in $PreIm(t_0)$ but in $PreIm(t_0) - L_{\mathcal{S}}(\mathcal{O}, \mathcal{O}^{-1})$. This raises the issue that values available in $PreIm(t_0)$ might all already be bound to other arguments $PreIm(t_0) - L_{\mathcal{S}}(\mathcal{O}, \mathcal{O}^{-1})$ might be empty. In such a situation, the sampling algorithm is defined to output λ. As a consequence, it is easy to deduce the implementation of $\mathcal{O}$ by using the path-finder and H^{-1}_{post}.

As for the indifferentiability bound, on top of the $Collide$ and $Reveal$ occurence, we need to bound the probability that λ is output by the simulated $\mathcal{O}$. Given a lower bound β of the cardinality of $PreIm(t_0)$ indenpendent of of t_0, we can have two results: 1.) if the adversary asks $max(k(\mathcal{O}), k(\mathcal{O}^{-1})) \leq \beta$ queries, the probability that the simulator outputs λ is null, 2.) for the general case, this probability is bounded by that of generating a β-collision on a set of size $|\mathsf{In}_{\mathcal{O}}|$.

5.2 Interdependent implementations

Let us now deal with the case of an oracle system $\mathbb{O}$ modeling inner-primitives for which no memory is shared between oracles, but such that the implementation of $\mathcal{O}$ in $\mathbb{O}$ queries a potentially non-empty set $\mathsf{ImpCalls}(\mathcal{O})$ of other oracles in $\mathbb{O}$. Reciprocally, we let $\mathsf{CalledBy}(\mathcal{O})$ the set of oracles whose implementation call that of $\mathcal{O}$. We also assume that the oracles of $\mathbb{O}$ are all functional. Considering such a situation can be made relevant by two possible reasons. First, decomposing a hash design into an overlayer and an underlying oracle system can result in an uninteresting bound (e.g. if H_j do not verify the partial injectivity property of section 3). Secondly, such a decomposition might not be possible according to our definition of overlayer. Indeed, H_j are meant to depend on the *previous exchange only*. For example, this is not the case for the hash construction Grøstl [21].

We proceed with the adaptation of the definition of simulator graph to this situation. In section 3, all exchanges appear as vertice of the graph. Here, we choose to represent only exchanges of oracles appearing in the static sequence of the overlayer, i.e. $V \subseteq \mathsf{Xch}_G = \{(\mathcal{O}, q, y) | \mathcal{O} \in [\mathcal{O}^1, \ldots, \mathcal{O}^{\mathfrak{L}}]\}$. The update algorithm specification requires a new function on top of the edge-computing function $\overline{E}$. The idea is that for a given $\mathcal{O}$, there are other oracle implementations depending on $\mathcal{O}$, namely $\mathcal{O}' \in \mathsf{CalledBy}(\mathcal{O})$. Thus, an exchange $(\mathcal{O}, q, y)$ possibly completes a suite of $[(\mathcal{O}_i, q_i, a_i)]_i$ corresponding to the external call sequence $\mathcal{O}'$ would perform to compute an output y' for some query q'. In such a case and if $\mathcal{O}' \in \mathsf{Xch}_G$, $(\mathcal{O}', q', y')$ must be added to the simulator graph vertex set. To formalize this, we propose the following definition.

DEFINITION 24. *A* vertex update function $\mathsf{CreateVertex}$ *is a function taking as arguments an exchange, an oracle, and a series of lists and outputting a subset of* Xch_G. *It is assumed to be such that: for all exchange* $(\mathcal{O}, q, y)$, *for all* $\mathcal{O}' \in \mathsf{CalledBy}(\mathcal{O})$, *for all exchange* $(N', q', y') \in \mathsf{Xch}_G$ *and lists* $[L_{\mathcal{O}''}]_{\mathcal{O}'' \in \mathsf{ImpCalls}(\mathcal{O}')}$, *then* $(\mathcal{O}', q', y') \in \mathsf{CreateVertex}((\mathcal{O}, q, y), \mathcal{O}', [L_{\mathcal{O}''}]_{\mathcal{O}'' \in \mathsf{ImpCalls}(\mathcal{O}')})$ *if and only if there exists a sequence of exchanges* $[(\mathcal{O}_i, q_i, a_i)]_i$ *in* $[L_{\mathcal{O}''}]_{\mathcal{O}'' \in \mathsf{ImpCalls}(\mathcal{O}')} \cup \{(\mathcal{O}, q, y)\}$ *such that if a call* q' *to* $\mathcal{O}'$ *is executed in a memory state* m *where* $L_{\mathcal{O}''}$ *are prefixes of* $m.L_{\mathcal{O}''}$ *and* $L_{\mathcal{O}} : (q, y)$ *is a prefix of* $m.L_{\mathcal{O}}$ *then call* $\mathcal{O}'(q')$ *provokes exchanges* $[(\mathcal{O}_i, q_i, a_i)]_i$ *and outputs* y'.

Using such a function, we can write a formal description of the update function of the simulator graph.

DEFINITION 25. *A generalized simulator graph update function* $\mathsf{upd}G$ *is a function which, on input an exchange* $(\mathcal{O}, q, y)$, *a simulator graph* $G = (v_{root}, V, E)$ *and the shared simulator state* $L_{\mathcal{S}}$, *outputs the simulator graph* G' *given by:*

- *a vertex set* V' *constructed out of* V *such that:*
  - *if* $\mathcal{O}$ *appears in* $[\mathcal{O}^1, \ldots, \mathcal{O}^{\mathfrak{L}}]$, *then* $(\mathcal{O}, q, y)$ *is added to* V,
  - *for all* $\mathcal{O}' \in \mathsf{CalledBy}(\mathcal{O}) \cap [\mathcal{O}^1, \ldots, \mathcal{O}^{\mathfrak{L}}]$, *the vertex set* $\mathsf{CreateVertex}((\mathcal{O}, q, y), \mathcal{O}', [L_{\mathcal{O}''}]_{\mathcal{O}'' \in \mathsf{ImpCalls}(\mathcal{O}')})$ *is added to* V.
- $E' = E \cup \{\overline{E}(v, v'), v \in V \cup v_{root}, v' \in V' - V\} \cup \{\overline{E}(v', v), v \in V, v' \in V' - V\}$.

With these new definitions, it is possible to build a satisfying simulator following the same specification as in section 3 and to obtain an identic version of our main theorem.

6. CONCLUSION

In this paper, we have introduced a generic framework for proving indifferentiability of hash designs. Indeed, we have proposed a generic definition to capture hash constructions, namely overlayers. We have specified a generic simulator along with an underlying graph structure enabling the detection of potential last queries. On top of that, we have highlighted a way to define another graph, the characteristic graph, so as to evaluate a bound to the indifferentiability of a random oracle of hash constructions with respect to our generic simulator. This work paves the way to a formalization of indifferentiability proofs allowing the generation of machine-checkable proofs, and participates to the global effort towards automatic verification of security proofs.

7. REFERENCES

[1] E. Andreeva, B. Mennink, and B. Preneel. On the indifferentiability of the grøstl hash function. In J. A. Garay and R. D. Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, volume 6280 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2010.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. Sha-3 proposal blake. Submission to NIST (Round 3), 2010.

[3] 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)*, Chicago, USA, oct 2010. ACM.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the 37th Symposium on Foundations of Computer Science, IEEE*, pages 514–523. IEEE, 1996.

[5] M. Bellare and T. Ristenpart. Multi-property-preserving hash domain extension and the emd transform. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin / Heidelberg, 2006.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge functions. Ecrypt Hash Workshop, may 2007.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the indifferentiability of the sponge construction. In N. P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The keccak sha-3 submission. Submission to NIST (Round 3), 2011.

[9] R. Bhattacharyya, A. Mandal, and M. Nandi. Indifferentiability characterization of hash functions and optimal bounds of popular domain extensions. In B. Roy and N. Sendrier, editors, *Progress in Cryptology - INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 199–218. Springer Berlin / Heidelberg, 2009.

[10] R. Bhattacharyya, A. Mandal, and M. Nandi. Security analysis of the mode of jh hash function. In S. Hong and T. Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 168–191. Springer, 2010.

[11] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau. Shabal, a submission to nist's cryptographic hash algorithm competition. Submission to NIST, 2008.

[12] D. Chang, S. Lee, M. Nandi, and M. Yung. Indifferentiable security analysis of popular hash functions with prefix-free padding. In X. Lai and K. Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2006.

[13] D. Chang and M. Nandi. Improved indifferentiability security analysis of chopmd hash function. In K. Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2008.

[14] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-damgård revisited: How to construct a hash function. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[15] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer Berlin / Heidelberg, 1990.

[16] M. Daubignard, P. Lafourcade, and Y. Lakhnech. Generic indifferentiability proofs of hash designs. Technical report, CNRS, VERIMAG, Grenoble University, 23 May 2011. http://www-verimag.imag.fr/∼daubigna/TR-CCS2011.pdf.

[17] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness extraction and key derivation using the cbc, cascade and hmac modes. In M. K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 494–510. Springer, 2004.

[18] Y. Dodis, T. Ristenpart, and T. Shrimpton. Salvaging merkle-damgård for practical applications. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 371–388. Springer Berlin / Heidelberg, 2009.

[19] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The skein hash function family. Submission to NIST (Round 3), 2010.

[20] E. Fleischmann, M. Gorski, and S. Lucks. Some observations on indifferentiability. In R. Steinfeld and P. Hawkes, editors, *Information Security and Privacy - 15th Australasian Conference, ACISP 2010, Sydney, Australia, July 5-7, 2010. Proceedings*, volume 6168 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2010.

[21] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a sha-3 candidate. Submission to NIST (Round 3), 2011.

[22] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *Theory of Cryptography*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin / Heidelberg, 2004.

[23] U. M. Maurer and S. Tessaro. Domain extension of public random functions: Beyond the birthday barrier. In A. Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 2007.

[24] R. C. Merkle. One way hash functions and des. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989.

[25] T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In K. G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19,*

2011. Proceedings, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2011.

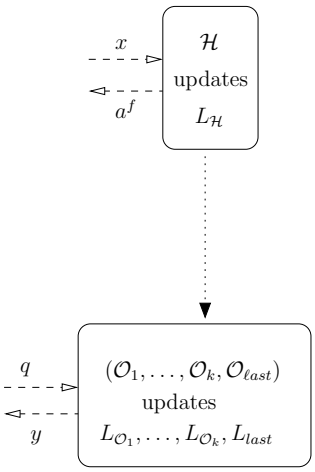
[26] H. Wu. The hash function jh. Submission to NIST (round 3), 2011.

A Proof of the Theorem

In this section, we propose a formal proof of theorem 1. The proof proceeds by successive modifications of the behavior of oracle systems, through progressive change of their implementations. This method is akin to game-based proofs, but puts to use elements of the proof system developped in [BDKL10], which semantics is very close to that presented in this paper. In this framework, the probabilistic transition systems formed by oracle systems interacting with adversaries are compared thanks to an extension of the notion of probabilistic bisimulation. Imperfect simulation is captured by the more refined concept of *bisimulation up to* some bad event.

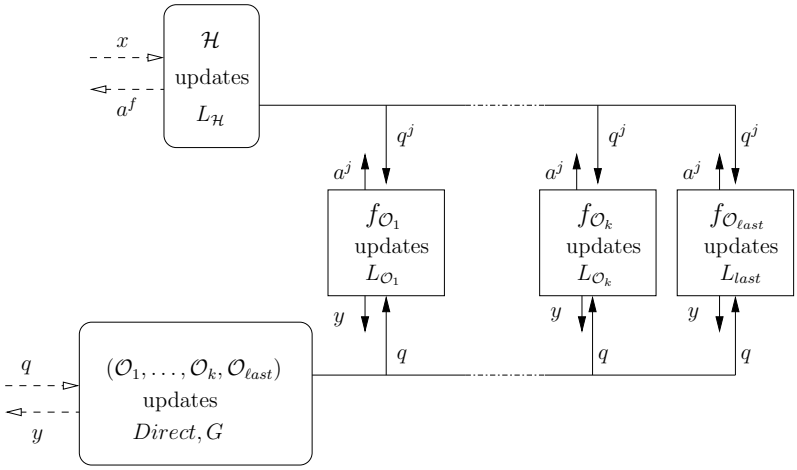
We choose to present the proof into two parts. Firstly, we provide in the next subsection an informal description of the transformations carried out throughout the proof. Each step consists in exhibiting a new oracle system, highlighting its differences with the previous one and if necessary outlining the events when the simulation is imperfect. Each system is illustrated by a corresponding figure. Along the sketch of the proof, we emphasize that it naturally falls into three parts. Secondly, in subsection A.2, we provide the theoretical framework to define and reason with the concept of *bisimulation up to*. We then give the complete implementations of oracle systems at each step, and the relations between them.

A.1 Informal Proof Overview



Initial system. The initial system corresponds to the original setting. The adversary interacts with the oracle system $(\mathcal{H}, \mathbb{O})$, where $\mathbb{O}$ consists of a finite number oracles assumed to be independent (i.e. their implementations do not share a common memory). Oracle $\mathcal{H}$ uses the oracles in $\mathbb{O}$ to compute its answer for adversaries, but these latter calls are not visible to the adversary. We arbitrarily name oracles in $\mathbb{O}$ $(\mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell ast})$. Note that this does not imply that $\mathcal{O}_1$ is the first oracle called by $\mathcal{H}$, $\mathcal{O}_2$ the second one... There is no reason for names to coincide with the fixed ordered sequence $[\mathcal{O}^1, \ldots, \mathcal{O}^{\mathfrak{L}}]$ of oracles that $\mathcal{H}$ calls successively. We choose to represent all oracles in $\mathbb{O}$ as a single box. Calls from $\mathcal{H}$ to $\mathbb{O}$ are reprensented by the dotted arrow. Adversary calls to $\mathcal{H}$ and their answers are generically denoted x and a^f, they

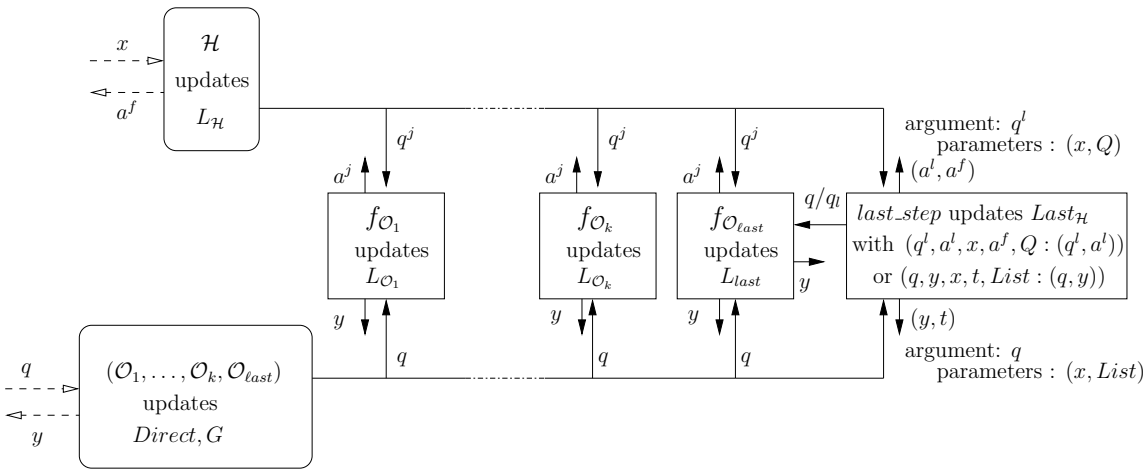
are pictured by dashed arrows. Similarly, adversary calls to $\mathcal{O}_i$ or $\mathcal{O}_{\ell last}$ and their answers are generically denoted q and y; they are also pictured by dashed arrows.



Second system. In the first step of our proof, we want to differentiate the adversary's direct queries to oracles in $\mathbb{O}$ from that of $\mathcal{H}$. To do so, let us define functions $f_{\mathcal{O}_i}$ and $f_{\mathcal{O}_{\ell last}}$ corresponding to the original implementations of oracles of $\mathbb{O}$. We modify the implementations by inlining the code of $f_{\mathcal{O}_i}$ and $f_{\mathcal{O}_{\ell last}}$ in that of $\mathcal{H}$. Then, we add a global list *Direct* which stores tuples $(\mathcal{O}_{\ldots}, q, y)$ of exchanges with an oracle $\mathcal{O}_i$ or $\mathcal{O}_{\ell last}$. Lastly, each oracle of $(\mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell last})$ updates a graph G with every exchange.

Once these modifications are performed, $\mathcal{H}$ does not call oracles $\mathcal{O}_i$'s and $\mathcal{O}_{\ell last}$ anymore; it executes their original code itself. The coherence between $\mathcal{H}$ and $(\mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell last})$ is ensured by their common dependence in lists $L_{\mathcal{O}_i}$ and L_{last} which store query and answers to $\mathcal{O}_i$'s and $\mathcal{O}_{\ell last}$. As a consequence, *Direct* only contains *direct adversary* calls to $(\mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell last})$.

On the figure, functions $f_{\mathcal{O}_i}$ and $f_{\mathcal{O}_{\ell last}}$ are depicted by squares; they are executed on arguments q, q^j or q^l and outputs a value y, which is represented by plain arrows.

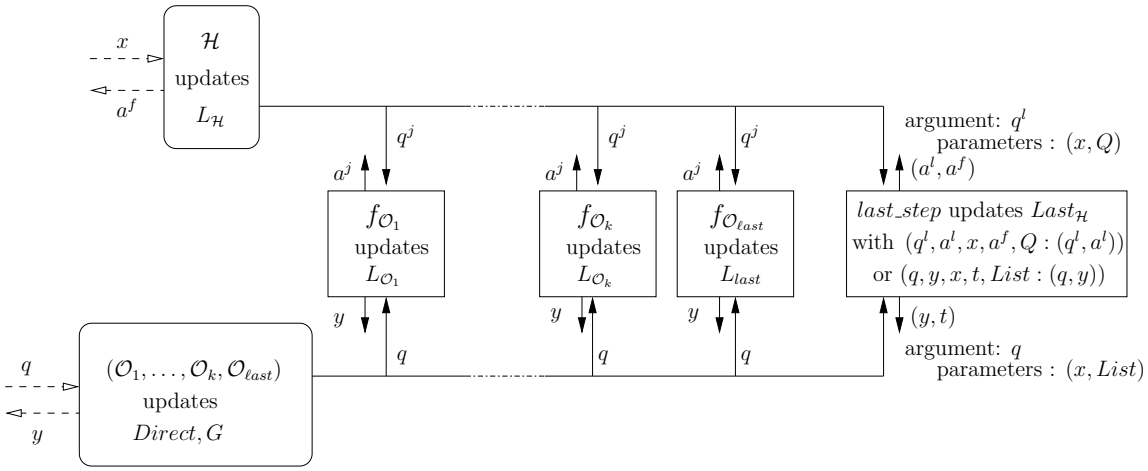


Third system. Oracle $\mathcal{O}_{\ell last}$ has a particular status; we know that in the original system, to compute an answer for input x, the last query q^l of $\mathcal{H}$ always goes to $\mathcal{O}_{\ell last}$. Let us define a function *last_step*, consisting in the whole last block of computations of $\mathcal{H}$: execution of function $f_{\mathcal{O}_{\ell last}}$, plus computation of the answer a^f. Evaluation of a^f uses the global $\mathcal{H}$-input x and list of queries Q. Thus, we define function *last_step* taking as inputs q^l,x and Q and outputting a pair (a^l, a^f) such that $a^l = f_{\mathcal{O}_{\ell last}}(q^l)$ and $a^f = \mathcal{H}(x)$. At each execution, function *last_step* updates a list $Last_{\mathcal{H}}$ with $(q^l, a^l, x, a^f, Q : (q^l, a^l))$. We remark that if a tuple $(q, y, \_, \_, \_)$ belongs to $Last_{\mathcal{H}}$, then it belongs to L_{last} too. Nevertheless, it is not reciprocal, and we cannot consider that $f_{\mathcal{O}_{\ell last}}$ is only executed through *last_step*. Indeed, $\mathcal{O}_{\ell last}$ can appear anywhere in the sequence of calls performed by $\mathcal{H}$, not just only lastly.

When building the simulator, we explain the need for a path-finder algorithm $\mathcal{P}$ to detect potential last queries when they are asked to $\mathcal{O}_{\ell last}$. In this third system, we change the control flow of $\mathcal{O}_{\ell last}$ to integrate this test. If, on given q and G, $\mathcal{P}$ outputs a tuple $(\mathsf{true}, x, List)$, we have the oracle execute *last_step* on q, x and $List$. This yields the computation of a value $y = f_{\mathcal{O}_{\ell last}}(q)$ which $\mathcal{O}_{\ell last}$ forwards as an answer to the adversary, but also the computation of a value t, corresponding to what $\mathcal{H}$ would answer on input x and list of queries $List : (q, y)$. Otherwise, if $\mathcal{P}$ outputs $(\mathsf{false}, \lambda, [\,])$, $\mathcal{O}_{\ell last}$ executes $f_{\mathcal{O}_{\ell last}}$.

Eventually, we make a difference between the three inputs $(q, x, List)$ (or (q^l, x, a^f)) of *last_step*: we say that the first one (q or q^l) is an *argument*, and that the others ($(x, List)$ or (x, Q)) are *parameters*. This is to emphasize that the function realizes a mapping (or *mathematical function*) of its input q, but not necessarily for the others. Indeed, $f_{\mathcal{O}_{\ell last}}$ implements a mathematical function, but nothing prevents two $\mathcal{H}$-inputs x to have an identical last query q.

This system is the last of the first part of the proof, somewhat consisting in the unfolding of the system.

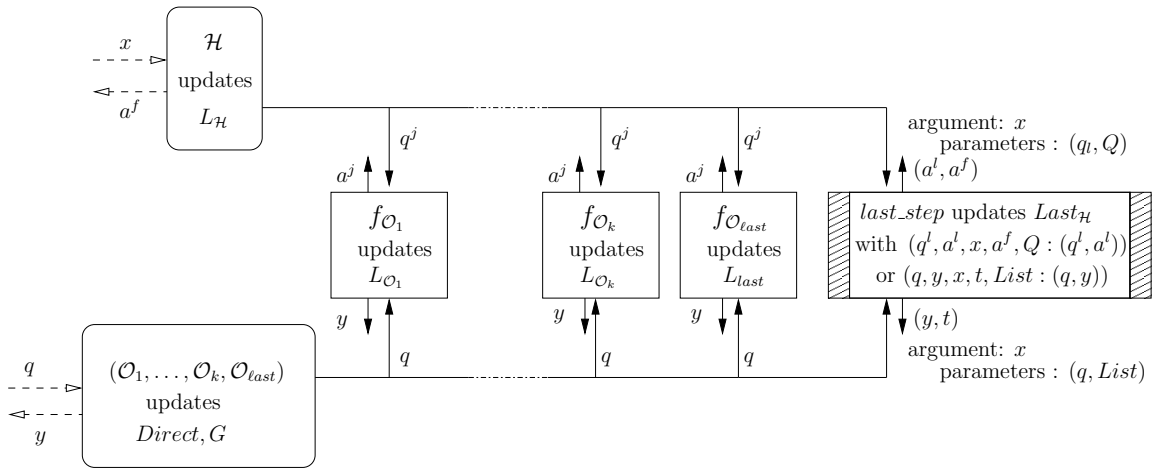


Fourth system. This fourth system is the first of the second part of the proof, in which we actually modify the behaviors of the functions. The difference between this system and the previous one lies in the independence of *last_step* from $f_{\mathcal{O}_{\ell last}}$. Indeed, at this step, *last_step* checks belonging to $\mathsf{dom}(Last_{\mathcal{H}})$ and otherwise performs a drawing. List $Last_{\mathcal{H}}$ stores the results of its executions. While it still implements a mathematical function of its first argument

onto its second argument, there is no reason that the answers y/a^l provided by *last_step* and $f_{\mathcal{O}_{\ell last}}$ for a same value of q/q^l coincide. In case they do not, the third and fourth system are not in a bisimulation relation.

As a result, we have to characterize this misbehavior and formalize it as predicates whose probability of eventually happening we then bound. Let us informally specify the predicates we detail in the next section. The problematic situation here arises as soon as some value q belongs to both $\mathsf{dom}(L_{last})$ and $\mathsf{dom}(Last_{\mathcal{H}})$. This can happen in two ways: either an element of $\mathsf{dom}(Last_{\mathcal{H}})$ is added to $\mathsf{dom}(L_{last})$ or the other way round.

- In the first case, if it happens during a call to $\mathcal{O}_{\ell last}$, then it means that the oracle has branched to execute $f_{\mathcal{O}_{\ell last}}$ after $\mathcal{P}$ has output $(\mathsf{false}, \lambda, [\,])$. This is captured by predicate ULQ (for Undetected Last Query). If it happens during a call to $\mathcal{H}$, then it means that $f_{\mathcal{O}_{\ell last}}$ is called during an intermediate (and not last) computation. This is captured by predicate ILQ, for Intermediate and Last Query.
- In the second case, if it occurs during a call to $\mathcal{O}_{\ell last}$, it means that it is the first time the query is asked directly, so that it has been an intermediate query before. Then again, ILQ captures this case. Eventually, if it occurs during a call to $\mathcal{H}$, then either ILQ is realized, or a previous direct query to $\mathcal{O}_{\ell last}$ is given as argument to *last_step*. This latter case is captured by predicate $DQ - LQ$ for Direct Query and Last Query.

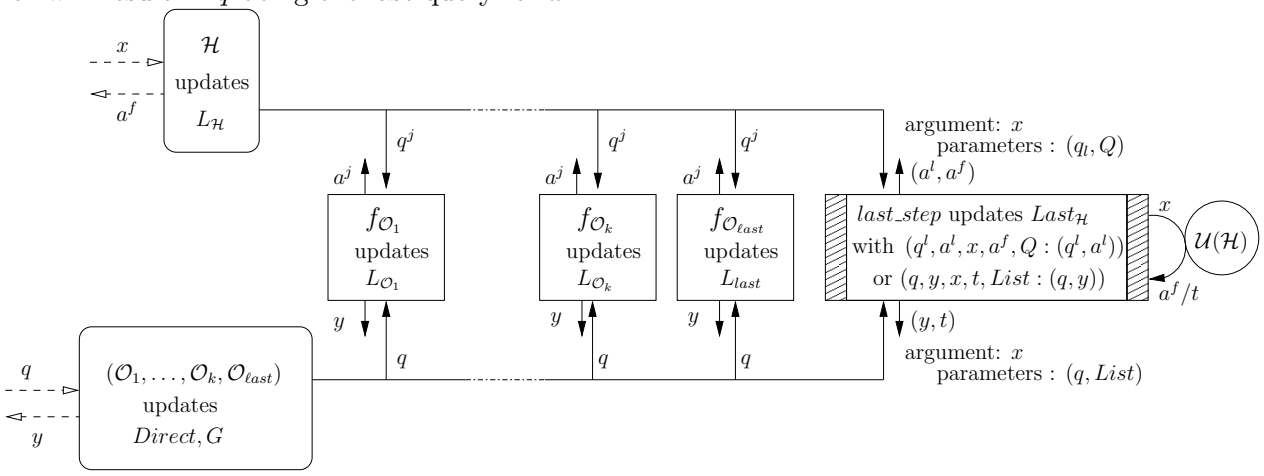


Fifth system. When defining the third system, we have introduced function *last_step* which stores the results of its executions in $Last_{\mathcal{H}}$. We have stressed that this function implements a mathematical function of the first component of $Last_{\mathcal{H}}$ (i.e. q/q^l) onto its second component (y/a^l). This follows from the form of its implementation: it checks in $Last_{\mathcal{H}}$ whether the argument q/q^l has already been assigned a value for y/a^l, in which case it takes it up as an answer, and otherwise draws a novel answer. In both cases, computation of t/a^f using the parameters x and list of queries ensues. The difference between the status of x and the list and q/q^l was emphasized by the use of the word 'argument' to designate q/q^l, while x and the list were refered to as 'parameters'.

In the fifth system, we exchange the roles played by q/q^l and x: x becomes the argument, and q becomes a parameter. Indeed, we modify the code of function *last_step* as follows.

When provided with a triple (q^l, x, Q) (or $(q, x, List)$), it checks whether there already is a value for a^l (or y) *associated to* x in $Last_{\mathcal{H}}$, keeps it as an answer if there is, and draws a new one otherwise. As a result, *last_step* now implements a mathematical function *from its third component onto its second*.

For fourth and fifth systems to behave similarly, there must be a one-to-one mapping between first components q/q^l of $Last_{\mathcal{H}}$ and third ones. Indeed, if there are two distinct values, say q and q', associated by $Last_{\mathcal{H}}$ to the same value of x, then we cannot guarantee that the fifth system provides the same answer as the fourth. As first components of $Last_{\mathcal{H}}$ represent (real or potential) last queries, we call this event SLQ for Same Last Query. To be thorough, we have to mention the possibility that two values for x correspond to one given q. This cannot happen in an execution of either systems as a consequence of the second property we impose on the path-finder. Indeed, if on input q it outputs some x, then execution $\mathcal{H}$ on x has or will result in q being the last query for x.

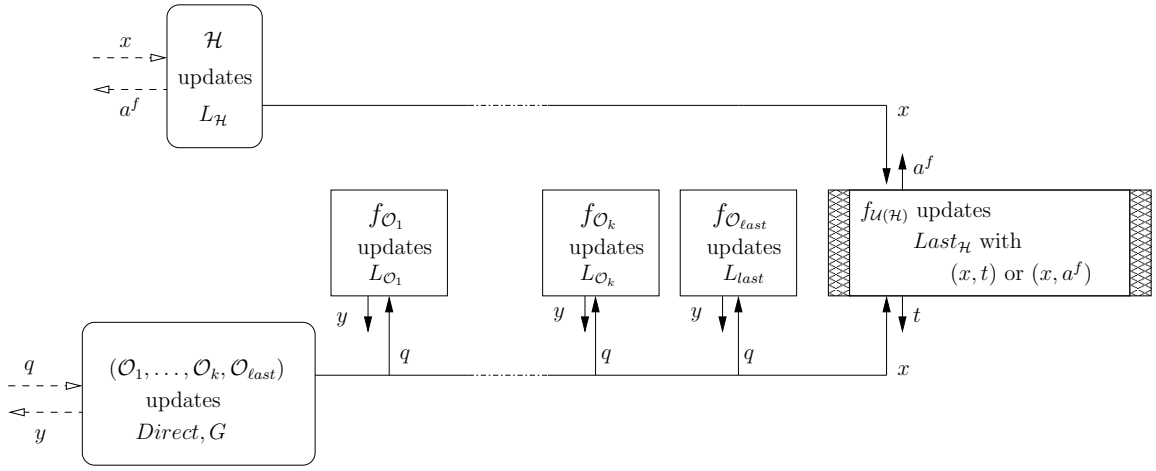


Sixth system. In this step, we again modify *last_step*. This function either originally takes up a stored value for y/a^l or draws a new one, and then computes a value for a^f or t using H_{post} and the parameters of the function. The idea of this step is to switch the order of the computations. In the sixth system, *firstly*, a value for a^f or t is either taken up as stored or drawn according to $\mathcal{U}(\mathcal{H})$, and *only then* a satisfying value for a^l/y is sampled using H^{-1}_{post}. We recall that H^{-1}_{post} is defined to preserve the original distribution of pairs (y, t) (resp. (a^l, a^f)). Using the fifth system implementation, we have seen that function *last_step* maps its argument x to the second component of $Last_{\mathcal{H}}$, y; that being said, there can potentially be distinct fourth component (t or a^f) associated to a given x. However, using the sixth system implementation, there can only be one a^f/t value for a given x, while there can be distinct values of y. Indeed, we recall that if H_{post} is not injective when everything but y is fixed, there are more than one possibility for y. Nevertheless, this does not lead to a simulation problem with respect to the adversary. Indeed, even though two different y's are drawn for a given x (one when executing $\mathcal{H}$, one when executing $\mathcal{O}_{\ell last}$), only the value of y computed while executing $\mathcal{O}_{\ell last}$ is actually used. We refer the reader to the formal proof below for more details about how we deal with this technicality.

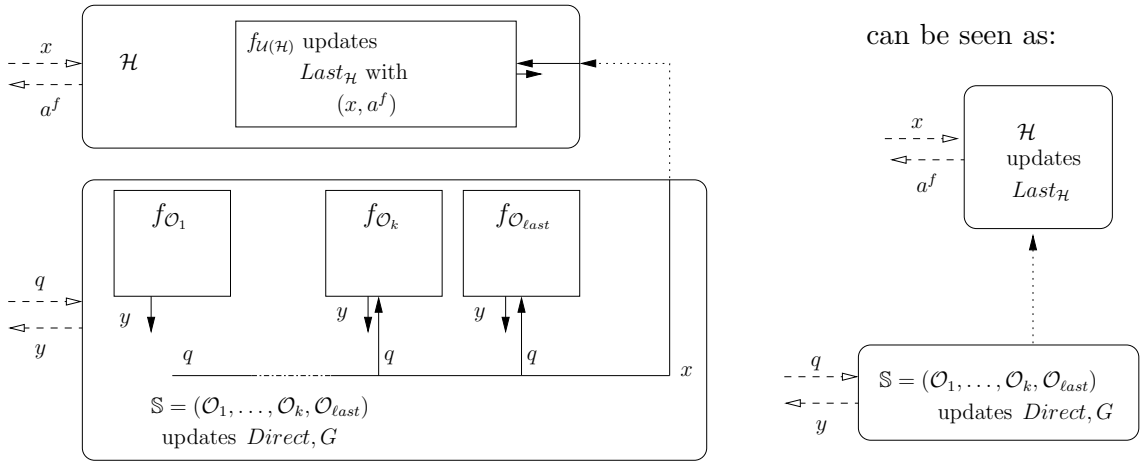
Therefore, we are left with one problematic case in which the programs potentially do not behave similarly: when there are two t's for an x. We are actually able to show, using the fact

that non-empty strings x answered by the path-finder algorithm are correct values for x, that this can never happen.

This is the last system of the second part of the proof.



Seventh system. There starts the last part of the proof, which roughly consists in folding back the system now that we are done modifying functions' behaviors. In the sixth system, $\mathcal{H}$ computes $((q^1,a^1),\ldots,(q^l,a^l))$ but these values do not influence the final output. In the seventh system, we suppress all these useless computations. Function *last_step* is not relevant anymore: only an execution of $\mathcal{O}_{last}$ requires a computation of a value y using H^{-1}_{post}. The code shared between $\mathcal{H}$ and $\mathcal{O}_{last}$ consists in the test of belonging to $Last_{\mathcal{H}}$ and the draw of a value otherwise. We denote this as function $f_{\mathcal{U}(\mathcal{H})}$. We project list $Last_{\mathcal{H}}$ to its sole necessary components, which are the third and fourth (x and t/a^f).



Final system. Eventually, we can consider $f_{\mathcal{U}(\mathcal{H})}$ as the implementation of $\mathcal{H}$ itself, and then $\mathcal{O}_{last}$ needs to issue a call to $\mathcal{H}$ when it detects a last query. The redundant list $L_{\mathcal{H}}$, identical to $Last_{\mathcal{H}}$, can be suppressed, as can be $L_{\mathcal{O}_i}$ and L_{last} which are contained in *Direct* and only

used in the simulator. Zooming out of the lefthandside figure, we obtain the righthandside figure, our final goal.

A.2 Technical Details of the Proof

A.2.1 Theoretical Elements Used in the Proof

To relate two compatible oracle systems, we use an extension of the standard notion of bisimulation. We recall that the interaction of oracle systems with adversaries are modeled as probabilistic transition systems. The notion we introduce, *bisimulation up to*, captures the bisimilarity of two probabilistic transition systems until one of their transitions fails to meet a specific condition.

We consider two compatible systems $\mathbb{O}$ and $\mathbb{O}'$. Given an oracle $\mathcal{O}$ of the systems, we define the *alloy implementation* $\widehat{\mathsf{Imp}(\mathcal{O})}$ of $\mathcal{O}$, which behaves as the implementation of $\mathcal{O}$ in $\mathbb{O}$ on input a state of $\mathsf{M}_{\mathbb{O}}$, and as the implementation of $\mathcal{O}$ in $\mathbb{O}$ on input a state of $\mathsf{M}_{\mathbb{O}'}$. Formally, we let $\hat{\mathsf{M}}$ be the *alloy state space* $\mathsf{M}_{\mathbb{O}} + \mathsf{M}_{\mathbb{O}'}$. For every $\mathcal{O} \in \mathsf{N}_{\mathbb{O}}$, we let $\widehat{\mathsf{Imp}(\mathcal{O})}$ be the disjoint sum of $\mathsf{Imp}_{\mathbb{O}}(\mathcal{O})$ and $\mathsf{Imp}_{\mathbb{O}'}(\mathcal{O})$, i.e.

$$\widehat{\mathsf{Imp}(\mathcal{O})} : \mathsf{In}(\mathcal{O}) \times \hat{\mathsf{M}} \to \mathcal{D}(\mathsf{Out}(\mathcal{O}) \times \hat{\mathsf{M}})$$

We are only interested in transitions having a positive probability to happen: we write $m_1 \xrightarrow{(\mathcal{O},q,a)}_{>0} m_2$ iff $\Pr[\widehat{\mathsf{Imp}(\mathcal{O})}(q, m_1) = (a, m_2)] > 0$.

The idea behind our notion of bisimulation is to find a relation between states of $\mathsf{M}_{\mathbb{O}}$ and $\mathsf{M}_{\mathbb{O}'}$ such that states in a same group offer, when performing a given transition step, the same probability of reaching any another group. In the sequel, the relation is formalized as an equivalence relation on the alloy state space $\hat{\mathsf{M}}$. As our systems are not unconditionnally in bisimulation, we further introduce a predicate on triples of an exchange and a pair of memories. The memories represent states before performing the exchange and after the exchange has been performed. We can then lighten the requirements on the bisimulation relation using this condition. We now provide our formal definition for bisimulation up to.

Definition 27 *Let $\varphi \subseteq \mathsf{Xch} \times \hat{\mathsf{M}} \times \hat{\mathsf{M}}$ be a predicate and let $R \subseteq \hat{\mathsf{M}} \times \hat{\mathsf{M}}$ be an equivalence relation. $\mathbb{O}$ and $\mathbb{O}'$ are bisimilar up to φ, written $\mathbb{O} \equiv_{R,\varphi} \mathbb{O}'$, iff $\bar{m}\, R\, \bar{m}'$, and for all $m_1 \xrightarrow{(\mathcal{O},q,a)}_{>0} m_2$ and $m_3 \xrightarrow{(\mathcal{O},q,a)}_{>0} m_4$ such that $m_1 R m_3$:*

- stability: *if $m_2 R m_4$ then*

  $$\varphi((\mathcal{O}, q, a), m_1, m_2) \quad \Leftrightarrow \quad \varphi((\mathcal{O}, q, a), m_3, m_4)$$

  This captures that the condition has the same truth value on complete classes.

- compatibility: *if $\varphi((\mathcal{O}, q, a), m_1, m_2)$, then*

  $$\Pr[\widehat{\mathsf{Imp}(\mathcal{O})}(q, m_1) \in (a, C)] = \Pr[\widehat{\mathsf{Imp}(\mathcal{O})}(q, m_3) \in (a, C)]$$

  where C is the equivalence class of m_2 under R.

  This captures that the probability to reach a given class is the same for every state in the source class.

Bisimulations are closely related to observational equivalence and relational Hoare logic, and allow to justify proofs by simulations. Besides, bisimulations up to subsume the Fundamental Lemma of [**?**].

In article [BDKL10], we have proven that:

Proposition 4 *For all compatible oracle systems $\mathbb{O}$ and $\mathbb{O}'$, every relation R and predicate φ s.t. $\mathbb{O} \equiv_{R,\varphi} \mathbb{O}'$ and adversary $\mathbb{A}$:*

- $\Pr(\mathbb{A} \mid \mathbb{O} : \mathit{true} \wedge \mathsf{G}_\varphi) = \Pr(\mathbb{A} \mid \mathbb{O}' : \mathit{true} \wedge \mathsf{G}_\varphi)$
- $\Pr(\mathbb{A} \mid \mathbb{O} : \mathsf{F}_{\neg\varphi}) = \Pr(\mathbb{A} \mid \mathbb{O}' : \mathsf{F}_{\neg\varphi})$

This proposition accounts for the following reasoning rule:

Proposition 5 *For all compatible oracle systems $\mathbb{O}$ and $\mathbb{O}'$, every relation R and predicate φ s.t. $\mathbb{O} \equiv_{R,\varphi} \mathbb{O}'$ and adversary $\mathbb{A}$:*

$$|\Pr(\mathbb{A} \mid \mathbb{O} : \mathit{true}) - \Pr(\mathbb{A} \mid \mathbb{O}' : \mathit{true})| \leq \Pr(\mathbb{A} \mid \mathbb{O} : \mathsf{F}_{\neg\varphi})$$

Outline. In the sequel, we denote $\mathbb{H}^1, \ldots, \mathbb{H}^8$ the oracle systems which we introduce by successive modifications of the implementations. Moreover, for each $i \in [1..7]$, we define the state space of $\mathbb{H}^i$, denoted by M_i, and an equivalence relation $\mathcal{R}^i_{i+1}$ and a predicate φ^i_{i+1} such that $\mathbb{H}^i \equiv_{\mathcal{R}^i_{i+1}, \varphi^i_{i+1}} \mathbb{H}^{i+1}$. Eventually, each of the three parts of the proof is concluded by a lemma linking its first and last system, and we conclude using these three lemmas in the end. When the modifications just concern a few lines of implementations, lines who differ in the next system end with $\hookrightarrow$, while lines who differ from the previous system begin by $\hookleftarrow$. We recall that given a list L, $L(q)$ is the set of elements of first component q. When i is clear from the context, we can use $L(q)$ for elements of i-th component q. We use $\Pi_{i,i+1}(L)$ to denote the list of pairs consisting in the projection on the i-th and $(i+1)$-th components of each element of L.

A.2.2 Part 1: unfolding of the system

As highlighted in the informal presentation of the proof, this latter falls into three parts. We first proceed to the unfolding of the system.

Here are the initial set of implementations (defining system $\mathbb{H}^1$). On the left, we provide the initial code for $\mathcal{H}$ and on the right, implementations of oracles of $\mathbb{O} = (\mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell ast})$. The state space of $\mathbb{H}^1$ is the product of lists $L_{\mathcal{H}}$, $L_{\mathcal{O}_i}$ and L_{last}, which realize a mapping from their first onto their second argument.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(L_{\mathcal{H}})\ \mathsf{then}$
 $\mathsf{return}\ L_{\mathcal{H}}(x)$
else
 $l := \mathsf{init}(x);$
 $(x_1, \ldots, x_l) := \Theta(x);$
 $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1);$
 $a^1 \leftarrow \mathcal{O}^1(q^1);$
 $Q := [(\mathcal{O}^1, q^1, a^1)];$
 $\mathsf{for}\ j = 2\ \mathsf{to}\ l\ \mathsf{do}$
  $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}));$
  $a^j \leftarrow \mathcal{O}^j(q^j);$
  $Q := Q : (\mathcal{O}^j, q^j, a^j);$
 endfor
 $a^f := \mathsf{H}_{post}(x, [Q]_{k \in Ind_{post}(x)});$
 $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q);$
 $\mathsf{return}\ a^f$
endif

$\mathsf{Imp}(\mathcal{O}_i)(q) =$ $\mathsf{if}\ q \in \mathsf{dom}(L_{\mathcal{O}^i})\ \mathsf{then return}\ L_{\mathcal{O}^i}(q)$
$\mathsf{else}\ y \leftarrow \mathcal{U}(\mathcal{O}_i);$
 $L_{\mathcal{O}^i} := L_{\mathcal{O}^i}.(q, y);$
 $\mathsf{return}\ y$
endif

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) =$ $\mathsf{if}\ q \in \mathsf{dom}(L_{last})\ \mathsf{then return}\ L_{last}(q)$
$\mathsf{else}\ y \leftarrow \mathcal{U}(\mathcal{O}_{\ell last});$
 $L_{last} := L_{last}.(q, y);$
 $\mathsf{return}\ y$
endif

Implementations of oracles in the second system $\mathbb{H}^2$ are as follows. M_2 is an extension of M_1. Additional components are the list $Direct$ and simulator graph G.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{Q}}}(x) =$ if $x \in \mathsf{dom}(L_{\mathcal{H}})$ then
 return $L_{\mathcal{H}}(x)$
else
 $l := \mathsf{init}(x)$;
 $(x_1, \ldots, x_l) := \Theta(x)$;
 $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1)$;
 if $q^1 \in \mathsf{dom}(L_{\mathcal{O}^1})$ then
 $a^1 := L_{\mathcal{O}^1}(q^1)$;
 else $a^1 \leftarrow \mathcal{U}(\mathcal{O}^1)$;
 $L_{\mathcal{O}^1} := L_{\mathcal{O}^1}.(q^1, a^1)$;
 endif
 } $f_{\mathcal{O}^1}$
 $Q := [(\mathcal{O}^1, q^1, a^1)]$;
 for $j = 2$ to $l - 1$ do
  $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}))$;
  if $q^j \in \mathsf{dom}(L_{\mathcal{O}^j})$ then
  $a^j := L_{\mathcal{O}^j}(q^j)$;
  else $a^j \leftarrow \mathcal{U}(\mathcal{O}^j)$;
  $L_{\mathcal{O}^j} := L_{\mathcal{O}^j}.(q^j, a^j)$;
  endif
  } $f_{\mathcal{O}^j}$
  $Q := Q : (\mathcal{O}^j, q^j, a^j)$;
 endfor
 $(\mathcal{O}_{\ell ast}, q^l) := \mathsf{H}_l(x_l, (\mathcal{O}^{l-1}, q^{l-1}, a^{l-1}))$;
 if $q^l \in \mathsf{dom}(L_{last})$ then
 $a^l := L_{last}(q^l)$;
 else $a^l \leftarrow \mathcal{U}(\mathcal{O}_{\ell ast})$;
 $L_{last} := L_{last}.(q^l, a^l)$;
 endif
 } $f_{\mathcal{O}_{\ell ast}}$
 $Q := Q : (\mathcal{O}_{\ell ast}, q^l, a^l)$;
 $a^f := \mathsf{H}_{post}(x, [Q]_{k \in Ind_{post}(x)})$;
 $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q)$;
 return a^f
endif

$\mathsf{Imp}(\mathcal{O}_i)(q) =$ if $q \in \mathsf{dom}(Direct(\mathcal{O}_i))$ then
 return $Direct(\mathcal{O}_i)(q)$
else
 if $q \in \mathsf{dom}(L_{\mathcal{O}_i})$ then
 $y := L_{\mathcal{O}_i}(q)$;
 else $y \leftarrow \mathcal{U}(\mathcal{O}_i)$;
 $L_{\mathcal{O}_i} := L_{\mathcal{O}_i}.(q, y)$;
 endif
 } $f_{\mathcal{O}_i}$
 $Direct := Direct.(\mathcal{O}_i, q, y)$;
 $G := \mathsf{upd}G((\mathcal{O}_i, q, y), G)$;
 return y
endif

$\mathsf{Imp}(\mathcal{O}_{\ell ast})(q) =$ if $q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell ast}))$ then
 return $Direct(\mathcal{O}_{\ell ast})(q)$
else
 if $q \in \mathsf{dom}(L_{last})$ then
 $y := L_{last}(q)$;
 else $y \leftarrow \mathcal{U}(\mathcal{O}_i)$;
 $L_{last} := L_{last}.(q, y)$;
 endif
 } $f_{\mathcal{O}_{\ell ast}}$
 $Direct := Direct.(\mathcal{O}_{\ell ast}, q, y)$;
 $G := \mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y), G)$;
 return y
endif

We let $\mathcal{R}_2^1$ be the equivalence relation between states of M_1 and M_2 defined by the equality of states on their common components (i.e. everthing but $Direct$ and G). The bisimulation between the pair of systems is perfect; we thus choose $\varphi_2^1 = \mathsf{true}$.

Set of implementations of the third system.

The state space M_3 consists of an extension of M_2 with an additional list, $Last_{\mathcal{H}}$, containing 5-component tuples, and realizing a mapping of the first component onto the second one.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(L_{\mathcal{H}})\ \mathsf{then}$
 $\mathsf{return}\ L_{\mathcal{H}}(x)$
else
 $l := \mathsf{init}(x);$
 $(x_1, \ldots, x_l) := \Theta(x);$
 $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1);$
 $a^1 \leftarrow f_{\mathcal{O}^1}(q^1);$
 $Q := [(\mathcal{O}^1, q^1, a^1)];$
 $\mathsf{for}\ j = 2\ \mathsf{to}\ l-1\ \mathsf{do}$
  $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}));$
  $a^j \leftarrow f_{\mathcal{O}^j}(q^j);$
  $Q := Q : (\mathcal{O}^j, q^j, a^j);$
 endfor
 $(\mathcal{O}_{\ell last}, q^l) := \mathsf{H}_l(x_l, (\mathcal{O}^{l-1}, q^{l-1}, a^{l-1}));$
 $\left.\begin{array}{l} \mathsf{if}\ q^l \in \mathsf{dom}(L_{last})\ \mathsf{then} \quad \hookrightarrow \\ a^l := L_{last}(q^l); \ \hookrightarrow \\ \mathsf{else}\ a^l \leftarrow \mathcal{U}(\mathcal{O}_{\ell last}); \\ L_{last} := L_{last}.(q^l, a^l); \ \hookrightarrow \\ \mathsf{endif} \\ a^f := \mathsf{H}_{post}(x, [Q]_{k \in Ind_{post}(x)}); \\ Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q^l, a^l, x, a^f, Q : (\mathcal{O}_{\ell last}, q^l, a^l)); \end{array}\right\} last\_step(q^l, (x, Q))$
 $Q := Q : (\mathcal{O}_{\ell last}, q^l, a^l);$
 $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q);$
 $\mathsf{return}\ a^f$
endif

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) =$ if $q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))$ then
return $Direct(\mathcal{O}_{\ell last})(q)$
else if $\mathcal{P}(q, G) = (\mathsf{true}, x, List)$ then
$\left.\begin{array}{l} \text{if } q \in \mathsf{dom}(L_{last}) \text{ then } \hookrightarrow \\ y := L_{last}(q); \hookrightarrow \\ \text{else } y \leftarrow \mathcal{U}(\mathcal{O}_{\ell last}); \\ L_{last} := L_{last}.(q, y); \hookrightarrow \\ \text{endif} \\ t := \mathsf{H}_{post}(x, [List : (q, y)]_{k \in Ind_{post}(x)}); \\ Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q, y, x, t, List : (q, y)); \end{array}\right\} last\_step(q, (x, List))$
$Direct := Direct(\mathcal{O}_{\ell last}, q, y);$
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G);$
return y
else
$y \leftarrow f_{\mathcal{O}_{\ell last}}(q);$
$Direct := Direct(\mathcal{O}_{\ell last}, q, y);$
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G);$
return y
endif
endif

The rest of the $\mathcal{O}_i$'s stay the same.

We let $\mathcal{R}_3^2$ be the equivalence relation between states of M_2 and M_3 defined by the equality of states on their common components (i.e. everthing but $Last_{\mathcal{H}}$). The bisimulation between the second and third systems is perfect; we thus choose $\varphi_3^2 = \mathsf{true}$.

To conclude this first part, we can state the following lemma, which follows from two immediate applications of proposition 5.

Lemma 1 *Systems $\mathbb{H}^1$ and $\mathbb{H}^3$ are indistinguishable, i.e. for all adversary $\mathbb{A}$,*

$$\Pr(\mathbb{A} \mid \mathbb{H}^1 : \mathit{true}) = \Pr(\mathbb{A} \mid \mathbb{H}^3 : \mathit{true}).$$

A.2.3 Part 2: making *last_step* independent

This second part comprises three transformations. As none of them changes $\mathcal{O}_i$'s we omit to mention them in this part. We could apply proposition OR for every transformation. However, if we perform successive steps and get events to bound at each step, then we have a series of events to bound in the matching series of distinct implementation sets. It is far more practical to gather all the problematic events as a conjunction in a given set of implementations and to bound the probability of the conjunction to be satisfied. As a result, to obtain a better overall bound, trying to exhibit a global relation and property relating set 3 to set 6 seems to be a better strategy.

To do so, we proceed as follows. We first present the third, fourth and fifth set of implementations, and provide a bisimulation-up-to relation ($\mathcal{R}^i_{i+1}$, φ^i_{i+1}) between them. Then, we propose a relation $\mathcal{R}^3_6$ and show that $\mathbb{H}^3$ and $\mathbb{H}^6$ are in bisimulation with respect to relation $\mathcal{R}^3_6$, up to $\varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6$.

In the fourth set of implementations, the tests that q belongs to L_{last} are replaced by tests of belonging to $Last_{\mathcal{H}}$, and last queries are not stored in L_{last} anymore.

The set of states M_4 is the same as M_3.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(L_{\mathcal{H}})\ \mathsf{then}$
 $\mathsf{return}\ L_{\mathcal{H}}(x)$
else
 $l := \mathsf{init}(x);$
 $(x_1, \ldots, x_l) := \Theta(x);$
 $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1);$
 $a^1 \leftarrow f_{\mathcal{O}^1}(q^1);$
 $Q := [(\mathcal{O}^1, q^1, a^1)];$
 $\mathsf{for}\ j = 2\ \mathsf{to}\ l-1\ \mathsf{do}$
  $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}));$
  $a^j \leftarrow f_{\mathcal{O}^j}(q^j);$
  $Q := Q : (\mathcal{O}^j, q^j, a^j);$
 endfor
 $(\mathcal{O}_{\ell last}, q^l) := \mathsf{H}_l(x_l, (\mathcal{O}^{l-1}, q^{l-1}, a^{l-1}));$
 $\hookleftarrow$ $\mathsf{if}\ q^l \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then}$ $\hookrightarrow$
 $\hookleftarrow$ $a^l := \Pi_2(Last_{\mathcal{H}}(q^l));$
  $\mathsf{else}\ a^l \leftarrow \mathcal{U}(\mathcal{O}_{\ell last});$
 $\hookleftarrow$ ~~$L_{last} := L_{last}.(q^l, a^l);$~~
  endif
  $a^f := \mathsf{H}_{post}(x, [Q]_{k \in Ind_{post}(x)});$
  $Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q^l, a^l, x, a^f, Q : (\mathcal{O}_{\ell last}, q^l, a^l));$
 $Q := Q : (\mathcal{O}_{\ell last}, q^l, a^l);$
 $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q);$
 $\mathsf{return}\ a^f$
endif

} $last\_step(q^l, (x, Q))$

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) =$ if $q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))$ then
return $Direct(\mathcal{O}_{\ell last})(q)$
else if $\mathcal{P}(q, G) = (\mathsf{true}, x, List)$ then
$\hookleftarrow$ if $q \in \mathsf{dom}(Last_{\mathcal{H}})$ then $\hookrightarrow$
$\hookleftarrow$ $y := \Pi_2(Last_{\mathcal{H}}(q))$;
else $y \leftarrow \mathcal{U}(\mathcal{O}_{\ell last})$;
$\hookleftarrow$ ~~$L_{last} := L_{last}.(q, y)$;~~
endif
$t := \mathsf{H}_{post}(x, [List : (q, y)]_{k \in Ind_{post}(x)})$;
$Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q, y, x, t, List : (q, y))$;
} $last\_step(q, (x, List))$
$Direct := Direct(\mathcal{O}_{\ell last}, q, y)$;
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G)$;
return y
else
$y \leftarrow f_{\mathcal{O}_{\ell last}}(q)$;
$Direct := Direct(\mathcal{O}_{\ell last}, q, y)$;
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G)$;
return y
endif
endif

The rest of the $\mathcal{O}_i$'s stay the same.

The idea of this step is to break the dependency between functions *last_step* and $f_{\mathcal{O}_{\ell last}}$. In system $\mathbb{H}^3$, all executions of *last_step* are performed via checking into L_{last} and answering the stored value when there is one, whereas in $\mathbb{H}^4$, executions of *last_step* look into and fill in $Last_{\mathcal{H}}$. As a consequence, (potential) last queries appear in both lists in $\mathbb{H}^3$, while only in $Last_{\mathcal{H}}$ in $\mathbb{H}^4$. We can anticipate that if an element belongs to both lists in the fourth set of implementations, there is very little chance that these latter map it to the same value.

Therefore, we define the following equivalence relation $\mathcal{R}_4^3$. Let us denote by $m_3 \in \mathsf{M}_3$ (resp. $m_4 \in \mathsf{M}_4$) a state of system $\mathbb{H}^3$ (resp. $\mathbb{H}^4$). We say that $m_3 \ \mathcal{R}_4^3 \ m_4$ if they coincide on every component but L_{last}. We additionally require that $m_3.L_{last} = m_4.L_{last} \sqcup \Pi_{1,2}(m_4.Last_{\mathcal{H}})$.

To guarantee that given two related states, the performance of an exchange yields the same behavior of our systems, we must ensure that we eliminate cases which would result in the creation of a common element to $\mathsf{dom}(Last_{\mathcal{H}})$ and $\mathsf{dom}(L_{last})$ in system $\mathbb{H}^3$. This can happen either by adding to $\mathsf{dom}(Last_{\mathcal{H}})$ an element already belonging to $\mathsf{dom}(L_{last})$, or the converse. We can also consider whether the exchange is an $\mathcal{H}$-request or an $\mathcal{O}_{\ell last}$-request. Here is the exhaustive list of possible cases:

1. During an exchange $(\mathcal{O}_{\ell last}, q, y)$, a query of $Last_{\mathcal{H}}$ which does not belong to $Direct(\mathcal{O}_{\ell last})$ is not detected by $\mathcal{P}$.

2. During an exchange $(\mathcal{O}_{\ell last}, q, y)$, a potential last query asked to $\mathcal{O}_{\ell last}$ for the first time has been computed before, i.e. $q \in \mathsf{dom}(L_{last}) - \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))$.

3. During an exchange $(\mathcal{H}, x, a^f)$, for a non-terminal index i, q^i belongs to $Last_{\mathcal{H}}$. If m' is the state after the execution, and the last addition to $m'.Last_{\mathcal{H}}$ is $(q', \_, x', \_, Q')$, then there exists in $m'.Last_{\mathcal{H}}$ a tuple $(q, \_, x, \_, Q)$ such that $q \in \mathsf{Head}(Q')$ (here $q = q^i$).

4. During an exchange $(\mathcal{H}, x, a^f)$, q^l already is in L_{last}. Two cases can arise, according to whether $q^l \in Direct(\mathcal{O}_{\ell ast})$. If not, then there exists an element $(q', \_, x', \_, Q') \in Last_{\mathcal{H}}$ such that $q \in \mathsf{Head}(Q')$.

We summarize this analysis by the following predicates:
$ILQ(\_, m, m')$:
$\exists (q, \_, x, \_, Q), (q', \_, x', \_, Q') \in m'.Last_{\mathcal{H}}$ *s.t.* $q \in \mathsf{Head}(Q')$.
$ULQ((\mathcal{O}_{\ell ast}, q, y), m, m')$:
$q \in \mathsf{dom}(m.Last_{\mathcal{H}}) - \mathsf{dom}(m.Direct(\mathcal{O}_{\ell ast})) \wedge \mathcal{P}(q, G) = \bot$
$DQ - LQ((\mathcal{H}, x, a^f)m, m')$:
$\mathsf{Last}(m'.L_{\mathcal{H}}) = (x, a^f, Q : (q^l, a^l))$ *s.t.* $q^l \in m.Direct(\mathcal{O}_{\ell ast}) - m.Last_{\mathcal{H}}$

These three predicates capture every possibility mentioned in the above list, as illustrated in the following table.

	add to L_{last} an element of $Last_{\mathcal{H}}$	add to $Last_{\mathcal{H}}$ an element of L_{last}
by calling $\mathcal{O}_{\ell ast}$	(1.)ULQ	(2.)ILQ
by calling $\mathcal{H}$	(3.)ILQ	(4.) put in L_{last} by $\begin{cases} \mathcal{H} \rightarrow ILQ \\ \mathcal{O}_{\ell ast} \rightarrow DQ - LQ \end{cases}$

The fifth system is obtained out of the fourth system by exchanging the roles of q and x in function *last_step*. The set of states M_5 is identical to M_4 except that lists $Last_{\mathcal{H}}$ are assumed to realize a mapping from their third component onto their first and second component ($x \mapsto (q, y)$).

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) =$ if $x \in \mathsf{dom}(L_{\mathcal{H}})$ then
  return $L_{\mathcal{H}}(x)$
else
  $l := \mathsf{init}(x)$;
  $(x_1, \ldots, x_l) := \Theta(x)$;
  $(\mathcal{O}^1, q^1) := \mathsf{H}_1(x_1)$;
  $a^1 \leftarrow f_{\mathcal{O}^1}(q^1)$;
  $Q := [(\mathcal{O}^1, q^1, a^1)]$;
  for $j = 2$ to $l - 1$ do
    $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}))$;
    $a^j \leftarrow f_{\mathcal{O}^j}(q^j)$;
    $Q := Q : (\mathcal{O}^j, q^j, a^j)$;
  endfor
  $(\mathcal{O}_{\ell last}, q^l) := \mathsf{H}_l(x_l, (\mathcal{O}^{l-1}, q^{l-1}, a^{l-1}))$;
  $\hookleftarrow$ if $x \in \mathsf{dom}(Last_{\mathcal{H}})$ then
  $\hookleftarrow$ $a^l := \Pi_2(Last_{\mathcal{H}}(x))$; $\hookrightarrow$
      else $a^l \leftarrow \mathcal{U}(\mathcal{O}_{\ell last})$; $\hookrightarrow$
      endif
      $a^f := \mathsf{H}_{post}(x, [Q]_{k \in Ind_{post}(x)})$; $\hookrightarrow$
      $Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q^l, a^l, x, a^f, Q : (\mathcal{O}_{\ell last}, q^l, a^l))$;
  } $last\_step(x, (q^l, Q))$
  $Q := Q : (\mathcal{O}_{\ell last}, q^l, a^l)$;
  $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q)$;
  return a^f
endif

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) =$ if $q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))$ then
return $Direct(\mathcal{O}_{\ell last})(q)$
else if $\mathcal{P}(q, G) = (\mathsf{true}, x, List)$ then
$\hookleftarrow$ if $x \in \mathsf{dom}(Last_{\mathcal{H}})$ then
$\hookleftarrow$ $y := \Pi_2(Last_{\mathcal{H}}(x))$; $\hookrightarrow$
else $y \leftarrow \mathcal{U}(\mathcal{O}_{\ell last})$; $\hookrightarrow$
endif
$t := \mathsf{H}_{post}(x, [List : (q, y)]_{k \in Ind_{post}(x)})$; $\hookrightarrow$
$Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q, y, x, t, List : (q, y))$;
$\Big\}$ $last\_step(x, (q, List))$
$Direct := Direct(\mathcal{O}_{\ell last}, q, y)$;
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G)$;
return y
else
$y \leftarrow f_{\mathcal{O}_{\ell last}}(q)$;
$Direct := Direct(\mathcal{O}_{\ell last}, q, y)$;
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G)$;
return y
endif
endif

The intuition behind this step is to switch the test of belonging to $Last_{\mathcal{H}}$ from q to x. Our implementations are well-defined as soon as list $Last_{\mathcal{H}}$ maps its third component 'x' to a single second component y. This is not the case as soon as two different values of q are linked to the same x in $Last_{\mathcal{H}}$. This corresponds to a predicate we name SLQ for same last query. We notice that the contrary, namely, that two distinct values for x match the same q, is not possible. Indeed, applications Θ and H_j are deterministic functions, and the property of $\mathcal{P}$ implies that if x is an output of $\mathcal{P}$ then all the queries and answers necessary to the computation of $\mathcal{H}(x)$ have already been performed and are thus fixed values. For the same reasons, there is a unique list of questions Q corresponding to x once all drawings have been performed.

As an equivalence relation $\mathcal{R}^4_5$, we choose the equality on all components of the state.

The formalisation of SLQ is as follows:

$SLQ(m, m')$:

$\exists (q, \_, x, \_, Q), (q', \_, x', \_, Q') \in m'.Last_{\mathcal{H}}$ $s.t.$ $q = q' \wedge x \neq x'$

Sixth set of implementations:

The set of states M_6 is identical to M_5 except that lists $Last_{\mathcal{H}}$ here map their third element onto their fourth ($x \mapsto t$). Additionally, there can only be one value q and one list Q associated to x via $Last_{\mathcal{H}}$.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(L_{\mathcal{H}})\ \mathsf{then}$
 $\mathsf{return}\ L_{\mathcal{H}}(x)$
else
 $l := \mathsf{init}(x);$
 $(x_1, \ldots, x_l) := \Theta(x);$
 $a^1 \leftarrow f_{\mathcal{O}^1} q^1;$
 $Q := [(\mathcal{O}^1, q^1, a^1)];$
 $\mathsf{for}\ j = 2\ \mathsf{to}\ l-1\ \mathsf{do}$
  $(\mathcal{O}^j, q^j) := \mathsf{H}_j(x_j, (\mathcal{O}^{j-1}, q^{j-1}, a^{j-1}));$
  $a^j \leftarrow f_{\mathcal{O}^j} q^j;$
  $Q := Q : (\mathcal{O}^j, q^j, a^j);$
 endfor
 $(\mathcal{O}_{\ell last}, q^l) := \mathsf{H}_l(x_l, (\mathcal{O}^{l-1}, q^{l-1}, a^{l-1}));$
 $f_{\mathcal{U}(\mathcal{H})}(x)$ $\left\{\begin{array}{ll} & \mathsf{if}\ x \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then} \\ \hookleftarrow & a^f := \Pi_4(Last_{\mathcal{H}}(x)); \\ \hookleftarrow & \mathsf{else}\ a^f \leftarrow \mathcal{U}(\mathcal{H}); \\ & \mathsf{endif} \end{array}\right\} last\_step(x, (q^l, Q))$
 $\hookleftarrow\ \ a^l \leftarrow \mathsf{H}^{-1}_{post}(x, Q, a^f, q^l);$
 $Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q^l, a^l, x, a^f, Q : (\mathcal{O}_{\ell last}, q^l, a^l));$
 $Q := Q : (\mathcal{O}_{\ell last}, q^l, a^l);$
 $L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f, Q);$
 $\mathsf{return}\ a^f$
endif

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) =$ $\mathsf{if}\ q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))\ \mathsf{then}$
 $\mathsf{return}\ Direct(\mathcal{O}_{\ell last})(q)$
$\mathsf{else\ if}\ \mathcal{P}(q, G) = (\mathsf{true}, x, List)\ \mathsf{then}$
 $f_{\mathcal{U}(\mathcal{H})}(x)$ $\left\{\begin{array}{ll} & \mathsf{if}\ x \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then} \\ \hookleftarrow & t := \Pi_4(Last_{\mathcal{H}}(x)); \\ \hookleftarrow & \mathsf{else}\ t \leftarrow \mathcal{U}(\mathcal{H}); \\ & \mathsf{endif} \end{array}\right\} last\_step(x, (q, List))$
 $\hookleftarrow\ \ y \leftarrow \mathsf{H}^{-1}_{post}(x, List, t, q);$
 $Last_{\mathcal{H}} := Last_{\mathcal{H}}.(q, y, x, t, List : (q, y));$
  $Direct := Direct(\mathcal{O}_{\ell last}, q, y);$
  $G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G);$
  $\mathsf{return}\ y$
 else
  $y \leftarrow f_{\mathcal{O}_{\ell last}}(q);$
  $Direct := Direct(\mathcal{O}_{\ell last}, q, y);$
  $G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G);$
  $\mathsf{return}\ y$
 endif
endif

The intuition behind this step is that we switch the order of computation of y and t. In $\mathbb{H}^5$, y is computed first, and then used to compute t. In $\mathbb{H}^6$, t is computed first, and then used to compute y, by means of the sampling algorithm H^{-1}_{post} defined in section 2.2. This difference in the order of computation modifies the properties of the states generated. System $\mathbb{H}^5$ deals with states with $Last_{\mathcal{H}}$ mapping x's to y's, but can potentially store two different values of t for a given x. On the contrary, $\mathbb{H}^6$ operates on states with $Last_{\mathcal{H}}$ mapping x's to t's, but has good probability to store two different values of y for a given x.

To deal with the possibility of two y's for an x, we choose an equivalence relation gathering in a same class states of M_6 with two different values for y given an x. We first define $\mathcal{R}^6$ between two states of M_6. We use the function $\mathsf{Forget2nd}$ which 'forgets second components', and maps lists to sets as follows: $\mathsf{Forget2nd}(L : (q, y, x, t, Q)) = \mathsf{Forget2nd}(L) \cup \{(q, x, t, Q)\}$ and $[\,]$ is associated to $\emptyset$. Let $m_6, \tilde{m_6}$ be states of M_6. Informally, we require from m_6 and $\tilde{m_6}$ that they coincide on everything but maybe y's, while imposing on y's to be coherent with the rest of the tuple. More precisely, $\tilde{m_6}\ \mathcal{R}^6\ m_6$ iff they coincide on every component but list $Last_{\mathcal{H}}$, $\mathsf{Forget2nd}(m_6.Last_{\mathcal{H}}) = \mathsf{Forget2nd}(\tilde{m_6}.Last_{\mathcal{H}})$, and $\forall (q, y, x, t, Q : (q, y)) \in m_6.Last_{\mathcal{H}}, \mathsf{H}_{post}(x, [(Q; (q, y))]_{k \in Ind_{post}(x)}) = t$.

The global definition for $m_5 \in \mathsf{M}_5$ and $m_6 \in \mathsf{M}_6$ is then that $m_5\ \mathcal{R}^5_6\ m_6$ iff:

- either they are equal,
- or there exists $\tilde{m_6} \in \mathsf{M}_6$ such that $m_5 = \tilde{m_6}$ and $\tilde{m_6}\ \mathcal{R}^6\ m_6$.

The condition $(\neg TwoT)$ enforces that $Last_{\mathcal{H}}$ realizes a mapping from its third component x on its fourth t. It can be formalized as follows.

$TwoT(\_, m, m') :$
$m'.Last_{\mathcal{H}} = m.Last_{\mathcal{H}} : [(q, y, x, t, Q)] \wedge \exists (q', y', x', t', Q')\ s.t.\ (x' = x \wedge t \neq t')$

Thanks to the properties of the path-finder, this predicate is equivalent to false; indeed, there is no such tuple in list $Last_{\mathcal{H}}$ of a state of M_6, and if such a tuple (q', y', x', t', Q') exists in a list $Last_{\mathcal{H}}$ of a state of M_5, then $q = q'$, $y = y'$ and $Q = Q'$. As a result of the application of H_{post} on equal inputs, we necessarily have $t = t'$.

Relating set 3 to set 6. We now propose a way to link the third system to the sixth via a bisimulation relation up to $\varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6$ We let $\mathcal{R}^3_6$ be the following equivalence relation. Let $m_3 \in \mathsf{M}_3, m_6 \in \mathsf{M}_6$, we say that $m_3\ \mathcal{R}^3_6\ m_6$ iff:

- every component of state coincide but L_{last} and $Last_{\mathcal{H}}$.
- list $Last_{\mathcal{H}}$ realizes a mapping from x to q and there is only one list Q per x.
- either (*) $m_3.Last_{\mathcal{H}} = m_6.Last_{\mathcal{H}}$ and $m_3.L_{last} = m_6.L_{last} \sqcup \Pi_{1,2}(m_6.Last_{\mathcal{H}})$
- or (**) $\exists \tilde{m_6}$ such that $\tilde{m_6}\ \mathcal{R}^6\ m_6$ and $\tilde{m_6}$ verifies the above equalities: $m_3.Last_{\mathcal{H}} = \tilde{m_6}.Last_{\mathcal{H}}$ and $m_3.L_{last} = \tilde{m_6}.L_{last} \sqcup \Pi_{1,2}(\tilde{m_6}.Last_{\mathcal{H}})$

Stability holds because no predicate deals with values of y, and those are the only thing which can differ between states m_3 and m_6.

We then have to justify compatibility. Let $m_3, m'_3 \in \mathsf{M}_3$ such that $m_3 \xrightarrow{(\mathcal{O},q,a)}_{>0} m'_3$ for $\mathcal{O}$ an oracle in $(\mathcal{H}, \mathcal{O}_1, \ldots, \mathcal{O}_k, \mathcal{O}_{\ell last})$. Let C denote the equivalence class of m'_3 under relation $\mathcal{R}^3_6$;

it contains one state of M_3, m'_3, and several states of M_6. Let $m_6 \in \mathsf{M}_6$ be a state in relation with m_3: $m_3\ \mathcal{R}^3_6\ m_6$. We show that $pr[\mathsf{Imp}_{\mathbb{H}^3}(\mathcal{O})(q, m_3) = (a, m'_3)] = pr[\mathsf{Imp}_{\mathbb{H}^6}(\mathcal{O})(q, m_6) = (a, C \cap \mathsf{M}_6)]$, provided $(\varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6)((\mathcal{O}, q, a), m_3, m'_3)$ holds. The compatibility follows from this equality.

We know that $m_3\ \mathcal{R}^3_6\ m_6$. We let $\tilde{m_6}$ be the state of M_6 related to m_6 such that there is one y for an x in list $\tilde{m_6}.Last_{\mathcal{H}}$. Then we define m_5 (resp.m_4) denote the state of M_5 (resp.M_4) equal with $\tilde{m_6}$. Similarly, we let $\tilde{m_6}'$ denote the particular state of $C \cap \mathsf{M}_6$ containing only one y per x in its list $Last_{\mathcal{H}}$, m'_5 (resp.m'_4) denotes the state of M_5 (resp.M_4) equal with $\tilde{m_6}'$. Thus, we can state that $C \cap \mathsf{M}_6 = Class(\ \mathcal{R}^5_6\ , m'_5)$.

Moreover, by stability, for all states $m'_6 \in C$, $(\varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6)((\mathcal{O}, q, a), m_6, m'_6)$ holds. Still by stability, $(\varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6)((\mathcal{O}, q, a), \tilde{m_6}, \tilde{m_6}')$ does holds to. As a result of the states definition, we can deduce that $\varphi^4_5(m_5, m'_5)$ and $\varphi^3_4(m_4, m'_4)$ hold too.

We can then derive the following equalities:

$$
\begin{array}{rcl}
pr[\mathsf{Imp}_{\mathbb{H}^6}(\mathcal{O})(q, m_6) = (a, C \cap \mathsf{M}_6)] & = & pr[\mathsf{Imp}_{\mathbb{H}^6}(\mathcal{O})(q, m_6) = (a, Class(\ \mathcal{R}^5_6\ , m'_5))] \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^5}(\mathcal{O})(q, m_5) = (a, Class(\ \mathcal{R}^5_6\ , m'_5))] \\
& & \text{since } \varphi^5_6((\mathcal{O}, q, a), m_6, m'_6) \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^5}(\mathcal{O})(q, m_5) = (a, m'_5)] \\
& & \text{since } Class(\ \mathcal{R}^5_6\ , m'_5) = m'_5 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^5}(\mathcal{O})(q, m_5) = (a, Class(\ \mathcal{R}^4_5\ , m'_5))] \\
& & \text{since } Class(\ \mathcal{R}^4_5\ , m'_5) = m'_5 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^4}(\mathcal{O})(q, m_4) = (a, Class(\ \mathcal{R}^4_5\ , m'_5))] \\
& & \text{since } \varphi^4_5((\mathcal{O}, q, a), m_5, m'_5) \text{ and } m_4\ \mathcal{R}^4_5\ m_5 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^4}(\mathcal{O})(q, m_4) = (a, m'_4)] \\
& & \text{since } Class(\ \mathcal{R}^4_5\ , m'_5) = m'_5 = m'_4 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^4}(\mathcal{O})(q, m_4) = (a, Class(\ \mathcal{R}^3_4\ , m'_4))] \\
& & \text{since } Class(\ \mathcal{R}^3_4\ , m'_4) = m'_4 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^3}(\mathcal{O})(q, m_3) = (a, Class(\ \mathcal{R}^3_4\ , m_4))] \\
& & \text{since } \varphi^3_4((\mathcal{O}, q, a), m_4, m'_4) \text{ and } m_3\ \mathcal{R}^3_4\ m_4 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^3}(\mathcal{O})(q, m_3) = (a, Class(\ \mathcal{R}^3_6\ , m'_3))] \\
& & \text{noticing that } Class(\ \mathcal{R}^3_4\ , m'_4) \cap \mathsf{M}_3 = Class(\ \mathcal{R}^3_6\ , m'_6) \cap \mathsf{M}_3 = m'_3 \\
& = & pr[\mathsf{Imp}_{\mathbb{H}^3}(\mathcal{O})(q, m_3) = (a, m'_3)]
\end{array}
$$

Lemma 2 *The conclusion of this part is*

$$\mathbb{H}^3 \equiv_{\mathcal{R}^3_6\ ,\Phi} \mathbb{H}^6,$$

where $\Phi = \varphi^3_4 \wedge \varphi^4_5 \wedge \varphi^5_6 = \neg(DQ - LQ) \wedge \neg(ILQ) \wedge \neg(ULQ) \wedge \neg(SLQ)$.

A.2.4 Part 3: folding back the system

In the seventh set of implementations, the state space M_7 is reduced to the lists $Last_{\mathcal{H}}$ and $L_{\mathcal{H}}$ which are now storing pairs, lists $Direct$, $L_{\mathcal{O}_i}$ and L_{last}, and graph G.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(L_{\mathcal{H}})\ \mathsf{then}$
$\quad \mathsf{return}\ L_{\mathcal{H}}(x)$
else
$\quad \left.\begin{array}{l} \mathsf{if}\ x \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then} \\ a^f := Last_{\mathcal{H}}(x); \\ \mathsf{else}\ a^f \leftarrow \mathcal{U}(\mathcal{H}); \\ \mathsf{endif} \\ Last_{\mathcal{H}} := Last_{\mathcal{H}}.(x, a^f); \end{array}\right\} f_{\mathcal{U}(\mathcal{H})}(x)$
$\quad L_{\mathcal{H}} := L_{\mathcal{H}}.(x, a^f);$
$\quad \mathsf{return}\ a^f$
endif

$\mathsf{Imp}(\mathcal{O}_{\ell ast})(q) = \ \mathsf{if}\ q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell ast}))\ \mathsf{then}$
$\quad \mathsf{return}\ Direct(\mathcal{O}_{\ell ast})(q)$
else
$\quad \mathsf{if}\ \mathcal{P}(q, G) = (\mathsf{true}, x, List)\ \mathsf{then}$
$\quad\quad \left.\begin{array}{l} \mathsf{if}\ x \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then} \\ t := Last_{\mathcal{H}}(x)); \\ \mathsf{else}\ t \leftarrow \mathcal{U}(\mathcal{H}); \\ \mathsf{endif} \\ Last_{\mathcal{H}} := Last_{\mathcal{H}}.(x, t); \end{array}\right\} f_{\mathcal{U}(\mathcal{H})}(x)$
$\quad\quad y \leftarrow \mathsf{H}^{-1}_{post}(x, List, t, q);$
$\quad\quad Direct := Direct(\mathcal{O}_{\ell ast}, q, y);$
$\quad\quad G := \mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y), G);$
$\quad\quad \mathsf{return}\ y$
$\quad \mathsf{else}$
$\quad\quad \left.\begin{array}{l} \mathsf{if}\ q \in \mathsf{dom}(L_{last})\ \mathsf{then} \\ y := L_{last}(q); \\ \mathsf{else}\ y \leftarrow \mathcal{U}(\mathcal{O}_{\ell ast}); \\ L_{last} := L_{last}.(q, y); \\ \mathsf{endif} \end{array}\right\} \text{call to } f_{\mathcal{O}_{\ell ast}}(q)$
$\quad\quad Direct := Direct(\mathcal{O}_{\ell ast}, q, y);$
$\quad\quad G := \mathsf{upd}G((\mathcal{O}_{\ell ast}, q, y), G);$
$\quad\quad \mathsf{return}\ y$
$\quad \mathsf{endif}$

The relation between states of M_6 and M_7 is equality on common components G, $Direct$, L_{last} and $L_{\mathcal{O}_i}$, and in addition to that we require that $\Pi_{3,4}(m_6.Last_{\mathcal{H}}) = m_7.Last_{\mathcal{H}}$ and $\Pi_{1,2}(m_6.L_{\mathcal{H}}) = m_7.L_{\mathcal{H}}$. Since we just remove useless computations, we can choose $\varphi^6_7 = \mathsf{true}$.

Final set of implementations:

The final state space consists in list of pairs $Last_{\mathcal{H}}$ and list $Direct$, and graph G.

$\mathsf{Imp}(\mathcal{H})^{\mathcal{O}^1,\ldots,\mathcal{O}^{\mathfrak{L}}}(x) = \mathsf{if}\ x \in \mathsf{dom}(Last_{\mathcal{H}})\ \mathsf{then}$
$a^f := Last_{\mathcal{H}}(x);$
$\mathsf{else}\ a^f \leftarrow \mathcal{U}(\mathcal{H});$
$Last_{\mathcal{H}} := Last_{\mathcal{H}}.(x, a^f);$
$\mathsf{return}\ a^f$
endif

$\mathsf{Imp}(\mathcal{O}_{\ell last})(q) = \mathsf{if}\ q \in \mathsf{dom}(Direct(\mathcal{O}_{\ell last}))\ \mathsf{then}$
$\mathsf{return}\ Direct(\mathcal{O}_{\ell last})(q)$
else
$\mathsf{if}\ \mathcal{P}(q, G) = (\mathsf{true}, x, List)\ \mathsf{then}$
$t \leftarrow \mathcal{H}(x);$
$y \leftarrow \mathsf{H}^{-1}_{post}(x, List, t, q);$
else
$y \leftarrow \mathcal{U}(\mathcal{O}_{\ell last});$
endif
$Direct := Direct(\mathcal{O}_{\ell last}, q, y);$
$G := \mathsf{upd}G((\mathcal{O}_{\ell last}, q, y), G);$
$\mathsf{return}\ y$
endif

$\mathsf{Imp}(\mathcal{O}_i)(q) = \mathsf{if}\ q \in \mathsf{dom}(Direct(\mathcal{O}_i))\ \mathsf{then}$
$\mathsf{return}\ Direct(\mathcal{O}_i)(q)$
else
$y \leftarrow \mathcal{U}(\mathcal{O}_i);$
$Direct := Direct(\mathcal{O}_i, q, y);$
$G := \mathsf{upd}G((\mathcal{O}_i, q, y), G);$
$\mathsf{return}\ y$
endif

The relation between this final system and the previous one is equality on common components.