Verified Security of Merkle-Damgaard

Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, Cesar Kunz, Malte Skoruppa, Santiago Zanella-Béguelin

Verifiable Security of Merkle-Damgård

Michael Backes[§‖], Gilles Barthe[†], Matthias Berg[§], Benjamin Grégoire[‡],
César Kunz[¶†], Malte Skoruppa[§] and Santiago Zanella Béguelin[*]

[§]*Saarland University, Saarbrücken, Germany* [‖]*Max Planck Institute for Software Systems, Germany*
Email: {backes, berg, skoruppa}@cs.uni-saarland.de
[¶]*Universidad Politécnica de Madrid, Spain* [†]*IMDEA Software Institute, Madrid, Spain*
Email: {Gilles.Barthe, Cesar.Kunz}@imdea.org
[‡]*INRIA Sophia Antipolis-Méditerranée, France*
Email: Benjamin.Gregoire@inria.fr
[*]*Microsoft Research*
Email: santiago@microsoft.com

***Abstract*—Cryptographic hash functions provide a basic data authentication mechanism and are used pervasively as building blocks to realize many cryptographic functionalities, including block ciphers, message authentication codes, key exchange protocols, and encryption and digital signature schemes. Since weaknesses in hash functions may imply vulnerabilities in the constructions that build upon them, ensuring their security is essential. Unfortunately, many widely used hash functions, including SHA-1 and MD5, are subject to practical attacks. The search for a secure replacement is one of the most active topics in the field of cryptography. In this paper we report on the first machine-checked and independently-verifiable proofs of collision-resistance and indifferentiability of Merkle-Damgård, a construction that underlies many existing hash functions. Our proofs are built and verified using an extension of the EasyCrypt framework, which relies on state-of-the-art verification tools such as automated theorem provers, SMT solvers, and interactive proof assistants.**

I. Introduction

Cryptographic hash functions provide a basic data authentication mechanism and are routinely used as building blocks in other cryptographic constructions. For a given input m, a cryptographic hash function H outputs a digest $H(m)$ of some small fixed length. For most tasks, it is required that finding distinct inputs with the same digest—a collision—be difficult. However, recent research has demonstrated that widely used hash functions, including SHA-1 and MD5, are vulnerable to collision attacks [28], [36], [37]. In response to these concerns, the U.S. National Institute of Standards and Technology (NIST) started in November 2007 a public competition to develop new cryptographic hash functions to augment a set of standard functions that includes the SHA-1 and SHA-2 algorithms. This competition, commonly known as the *SHA-3 competition*, motivated a growing interest in developing cryptographic hash functions and in rigorously scrutinizing their security.

Verified security [8], [10] is an emerging approach to security proofs of cryptographic systems. It adheres to the same principles as provable security, but revisits its realization from a formal verification perspective. When taking a verified security approach, proofs are mechanically verified and built with the aid of state-of-the-art verification tools, such as SMT solvers, automated theorem provers and interactive proof assistants. EasyCrypt [8] is an automated framework that aims to make verified security accessible to cryptographers with a limited background in formal methods; it has been successfully applied to verify exact security bounds of several digital signature and encryption schemes.

In this paper, we report on an extension of EasyCrypt and its application to build and verify exact security proofs of the Merkle-Damgård construction [23], [31], which underlies the design of many cryptographic hash functions. In its simplest formulation, Merkle-Damgård iterates a compression function $f : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ over the blocks of an input message padded to a block boundary. For a fixed public initialization vector IV, the digest of a padded message with blocks $x_1 \parallel \cdots \parallel x_\ell$ is computed as

$$f(x_\ell, f(x_{\ell-1}, \ldots f(x_1, \mathsf{IV}) \ldots))$$

One way of arguing that iterated constructions like Merkle-Damgård are secure is to show that they preserve security properties of the underlying compression function. The seminal works of Merkle [31] and Damgård [23] show that if messages are padded in some specific way, finding two colliding messages for the above iterated construction is at least as hard as finding two colliding inputs for the compression function f; said otherwise, that the construction preserves the collision resistance of the compression function. We present a proof of a generalization of this result in EasyCrypt. Our proof applies when the padding function is suffix-free, i.e. the padding of a message m is not a suffix of the padding of any other message m'.

An alternative method for proving the security of a hash function is to show that it behaves as a random oracle when the compression function, or some other lower-level

building block, is assumed to be ideal. The indifferentiability framework of Maurer et al. [30] provides a rigorous simulation-based definition that captures this intuition and implies a strong composability result. Glossing over technical subtleties [33], a hash function H indifferentiable from a random oracle can be plugged into a cryptosystem proven secure in the random oracle model for H without compromising the security of the cryptosystem. We present a proof in EasyCrypt of the indifferentiability of the Merkle-Damgård construction from a random oracle. Our proof, which follows the proof of Coron et al. [22], applies when the padding function is prefix-free, i.e. the padding of a message m is not a prefix of the padding of any other message m'.

Organization of the Paper: Section II overviews the foundations and verification mechanisms implemented in our extension to EasyCrypt; Section III describes the Merkle-Damgård construction and its security properties; Section IV describes a machine-checked proof that Merkle-Damgård preserves collision resistance when used with a suffix-free padding, while Section V describes a machine-checked proof of its indifferentiability from a random oracle when the padding is prefix-free; Section VI discusses the applicability of our results to generalizations of the Merkle-Damgård construction and the finalists of NIST SHA-3 competition. We conclude in Section VII.

II. A Primer on EasyCrypt

Building a cryptographic proof in EasyCrypt is a process that can be decomposed in the following steps:

- Defining a formal context, including types, constants and operators, and giving it meaning by declaring axioms and stating derived lemmas.
- Defining a number of games, each of them composed of a collection of procedures (written in the probabilistic imperative language described below) and adversaries declared as abstract procedures with access to oracles.
- Proving logical judgments that establish equivalences between games. This may be done fully automatically, with the help of hints from the user in the form of relational invariants, or interactively using basic tactics and automated strategies.
- Deriving inequalities between probabilities of events in games, either by using previously proven logical judgments or by direct computation.

In the remainder of this section, we briefly overview some key aspects of the process of building an EasyCrypt proof. Note that the work reported in this article benefited from several extensions of the tool with respect to [8]; these extensions include:

1) Support for reasoning about programs with loops. Loops were used to represent iteration in the Merkle-Damgård construction.
2) Mechanization of the Failure Event Lemma of [11], implemented in EasyCrypt as an extension to the mechanism that directly computes probability bounds. This was used to bound the success probability of the distinguisher in the proof of indifferentiability presented in Sect. V.
3) Proof engineering mechanisms to manage the size of proof obligations and the theories that external solvers use. These mechanisms were essential for the successful verification of the proofs presented in this paper.

A. Input Language

Probabilistic experiments are defined as programs in pWHILE, a strongly-typed imperative probabilistic programming language. The grammar of pWHILE commands is defined as follows:

$\mathcal{C}$	::=	skip	nop
		$\mid \mathcal{V} \leftarrow \mathcal{E}$	deterministic assignment
		$\mid \mathcal{V} \xleftarrow{\$} \mathcal{DE}$	probabilistic assignment
		$\mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C}$	conditional
		$\mid \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C}$	loop
		$\mid \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E})$	procedure call
		$\mid \mathcal{C};\ \mathcal{C}$	sequence

The only non-standard feature of the language are probabilistic assignments; an assignment $x \xleftarrow{\$} d$ evaluates the expression d in the current state to a distribution μ on values, samples a value according to μ and assigns it to variable x. The key to the flexibility of EasyCrypt is that the base language of expressions and distribution expressions can be extended by the user to suit the needs of the verification task. The rich base language includes expressions over Booleans, integers, fixed-length bitstrings, lists, finite maps, and option, product and sum types. User-defined operators can be axiomatized or defined in terms of other operators. In the following, we let $\{0,1\}^\ell$ denote the uniform distribution on bitstrings of length ℓ.

A program (equivalently, a game) in EasyCrypt is represented as a set of global variables together with a collection of procedures. Some of these procedures are concrete and given a definition as a command $c \in \mathcal{C}$, while some others may be abstract and left undefined. Quantification over adversaries in cryptographic proofs is achieved by representing them as abstract procedures parametrized by a set of oracles; these oracles must be instantiated as other procedures in the program.

Commands operate on program memories, which map local and global variables to values; we let $\mathcal{M}$ denote the set of memories. The semantics of a command $c \in \mathcal{C}$ is a function $[\![c]\!] : \mathcal{M} \to \mathcal{D}(\mathcal{M})$ from program memories to sub-distributions on program memories. Note that programs that do not terminate with probability 1 generate sub-distributions with total probability less than 1. We refer the reader to [9] for a detailed description of the semantics

of pWHILE as it has been formalized in the Coq proof assistant. In what follows, we denote by $\Pr[c, m : A]$ the probability of event A w.r.t. to the distribution $[\![c]\!]\ m$ and often omit the initial memory m when it is not relevant.

Although EasyCrypt is not tied to any particular cryptographic model, it provides good support to reason about proofs developed in the random oracle model. A random oracle $\mathcal{O} : X \to Y$ is modelled in EasyCrypt as a stateful procedure that maps values in X into uniformly and independently distributed values in Y. The state of a random oracle can be represented as a global finite map $\boldsymbol{L}$ that is initially empty. Queries are answered consistently so that identical queries are given the same answer:

$$\begin{array}{l}\textbf{Oracle } \mathcal{O}(x): \\ \quad \mathsf{if}\ x \notin \mathsf{dom}(\boldsymbol{L})\ \mathsf{then}\ \boldsymbol{L}[x] \xleftarrow{\$} Y \\ \quad \mathsf{return}\ \boldsymbol{L}[x]\end{array}$$

B. Probabilistic Relational Hoare Logic

The foundation of EasyCrypt is a probabilistic Relational Hoare Logic (pRHL), whose judgments are quadruples of the form:

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi$$

where c_1, c_2 are programs and Ψ, Φ are first-order relational formulae. Relational formulae are defined by the grammar:

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \Rightarrow \Phi \mid \forall x.\ \Phi \mid \exists x.\ \Phi$$

where e stands for a Boolean expression over logical variables and program variables tagged with either $\langle 1 \rangle$ or $\langle 2 \rangle$ to denote their interpretation in the left or right-hand side program; the only restriction is that logical variables must not occur free. The special keyword res denotes the return value of a procedure and can be used in the place of a program variable. We write $e\langle i \rangle$ for the expression e in which all program variables are tagged with $\langle i \rangle$. A relational formula is interpreted as a relation on program memories. For example, the formula $x\langle 1 \rangle + 1 \leq y\langle 2 \rangle$ is interpreted as the relation

$$R = \{(m_1, m_2) \mid m_1(x) + 1 \leq m_2(y)\}$$

The validity of a pRHL judgment is defined in terms of a lifting operator $\mathcal{L} : \mathcal{P}(A \times B) \to \mathcal{P}(\mathcal{D}(A) \times \mathcal{D}(B))$. Concretely,

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=} \forall m_1, m_2.\ m_1\ \Psi\ m_2 \Rightarrow ([\![c_1]\!]\ m_1)\ \mathcal{L}(\Phi)\ ([\![c_2]\!]\ m_2)$$

Formally, let μ_1 be a probability distribution on a set A and μ_2 a probability distribution on a set B. We define the lifting $\mu_1\, \mathcal{L}(R)\, \mu_2$ of a relation $R \subseteq A \times B$ to μ_1 and μ_2 by the clause:

$$\exists \mu : \mathcal{D}(A \times B).\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{supp}(\mu) \subseteq R$$

where $\pi_1(\mu)$ (resp. $\pi_2(\mu)$) denotes the projection of μ on its first (resp. second) component and $\mathsf{supp}(\mu)$ is the support of μ as a sub-probability measure—if μ is discrete, this is just the set of pairs with positive probability.

Figure 1 shows some selected rules that can be used to derive valid pRHL judgments. There are two kinds of rules: two-sided rules, which require that the related programs have the same syntactic form, and one-sided rules, which do not impose this requirement. One-sided rules are symmetric in nature and admit a *left* and a *right* variant. We briefly comment on some rules. The two-sided rule [Rnd] for random assignments requires the distributions from where values are sampled be uniform on some set X; to apply the rule one must exhibit a function $f : X \to X$ that may depend on the state and is 1-1 if the precondition holds. The one-sided rule [Rand$\langle 1 \rangle$] for random assignments simply requires that the post-condition is established for all possible outcomes; in effect, this rule treats random assignment as a non-deterministic assignment.

Similarly to Hoare logic, the rules for while loops require to exhibit an appropriate relational invariant Φ. The two-sided rule [While] applies when the loops execute in lockstep and thus requires proving that the guards are equivalent. The one-sided rule [While$\langle 1 \rangle$] further requires exhibiting a decreasing variant v and a lower bound m. The premises ensure that the loop is absolutely terminating, which is crucial for the soundness of the rule.

The relational Hoare logic also allows capturing the well-known cryptographic argument "*x is uniformly distributed and independent of the adversary's view*", which is certainly one of the most difficult to formalize. We formalize this argument in EasyCrypt by proving that re-sampling x preserves the semantics of the program. Suppose we want to prove that in a program c, a variable x used in an oracle $\mathcal{O}$ is uniformly distributed and independent of the view of an adversary $\mathcal{A}^{\mathcal{O}}$. Let $\mathcal{O}'$ be the same as $\mathcal{O}$ except that it re-samples x when needed. We identify a condition used that holds whenever $\mathcal{A}$ obtained some information about x (and thus, re-sampling would not preserve the semantics). We then prove that the conditional statement $c' \stackrel{\text{def}}{=} \mathsf{if}\ \neg\mathsf{used}\ \mathsf{then}\ x \xleftarrow{\$} X$ can *swap* with calls to $\mathcal{O}$ and $\mathcal{O}'$, i.e.

$$\vdash c'; y \leftarrow \mathcal{O}(\vec{e}) \sim y \leftarrow \mathcal{O}'(\vec{e}); c' : \Phi \Longrightarrow \Phi$$

where Φ implies equality over all global variables. From this, we can conclude that c' can also swap with calls to $\mathcal{A}^{\mathcal{O}}$ and $\mathcal{A}^{\mathcal{O}'}$, and hence that the semantics of the program c is preserved when $\mathcal{O}$ is replaced by $\mathcal{O}'$. The advantage of using such kind of reasoning is that it is generally much easier to reason about a game where x is sampled *lazily*, since its distribution is locally known.

We conclude with some observations on the mechanization of reasoning in pRHL. We implement in EasyCrypt several variants of two-sided and one-sided rules of pRHL in the form of tactics that can be applied in a goal-oriented fashion to prove the validity of judgments. For instance,

$$\frac{\vdash c_1 \sim c_2 : \Phi \Longrightarrow \Phi' \quad \vdash c_1' \sim c_2' : \Phi' \Longrightarrow \Phi''}{\vdash c_1; c_1' \sim c_2; c_2' : \Phi \Longrightarrow \Phi''}[\text{Seq}]$$

$$\vdash x \leftarrow e \sim \mathsf{skip} : \Phi\,\{e\langle 1\rangle / x\langle 1\rangle\} \Longrightarrow \Phi\ [\text{Asn}\langle 1\rangle] \qquad \vdash \mathsf{skip} \sim x \leftarrow e : \Phi\,\{e\langle 2\rangle / x\langle 2\rangle\} \Longrightarrow \Phi\ [\text{Asn}\langle 2\rangle]$$

$$\frac{\Psi \Rightarrow \mathsf{bijective}(f) \quad \Psi \Rightarrow \forall v \in X.\ \Phi\,\{v, f(v)/x\langle 1\rangle, y\langle 2\rangle\}}{\vdash x \xleftarrow{\$} X \sim y \xleftarrow{\$} X : \Psi \Longrightarrow \Phi}[\text{Rnd}] \qquad \frac{\Psi \Rightarrow \forall v \in \mathsf{supp}(d).\ \Phi\,\{v/x\langle 1\rangle\}}{\vdash x \xleftarrow{\$} d \sim \mathsf{skip} : \Psi \Longrightarrow \Phi}[\text{Rnd}\langle 1\rangle]$$

$$\frac{\vdash c_1 \sim c_2 : \Psi \wedge e\langle 1\rangle \Longrightarrow \Phi \quad \vdash c_1' \sim c_2 : \Psi \wedge \neg e\langle 1\rangle \Longrightarrow \Phi}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim c_2 : \Psi \Longrightarrow \Phi}[\text{Cond}\langle 1\rangle]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \wedge b_1\langle 1\rangle \Longrightarrow \Phi \qquad \Phi \Rightarrow b_1\langle 1\rangle = b_2\langle 2\rangle}{\vdash \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2 : \Phi \Longrightarrow \Phi \wedge \neg b_1\langle 1\rangle}[\text{While}]$$

$$\frac{\vdash c_1 \sim \mathsf{skip} : \Phi \wedge (b_1 \wedge v = n)\langle 1\rangle \Longrightarrow \Phi \wedge v\langle 1\rangle < n \qquad \Phi \wedge v\langle 1\rangle \leq m \Rightarrow \neg b\langle 1\rangle}{\vdash \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{skip} : \Phi \Longrightarrow \Phi \wedge \neg b_1\langle 1\rangle}[\text{While}\langle 1\rangle]$$

$$\frac{\Psi \Rightarrow \Psi' \quad \vdash c_1 \sim c_2 : \Psi' \Longrightarrow \Phi' \quad \Phi' \Rightarrow \Phi}{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi}[\text{Sub}] \qquad \frac{\vdash c_1 \sim c_2 : \Psi \wedge \Psi' \Longrightarrow \Phi \quad \vdash c_1 \sim c_2 : \Psi \wedge \neg\Psi' \Longrightarrow \Phi}{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi}[\text{Case}]$$

Figure 1. Selected pRHL rules

instead of implementing rule [Rnd⟨1⟩], we combine it with the [Seq] rule to obtain the following more easily applicable rule:

$$\frac{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \forall v \in \mathsf{supp}(d).\ \Phi\,\{v/x\langle 1\rangle\}}{\vdash c_1;\ x \xleftarrow{\$} d \sim c_2 : \Psi \Longrightarrow \Phi}$$

The application of a tactic may generate additional verification subgoals, and logical side conditions that are checked using SMT solvers, automated theorem provers and, as a last recourse, interactive proof assistants. Depending on their nature, application of the tactics can be fully automated or require user input. For instance, applying the tactics that mechanize the rules for while loops, requires the user to provide an adequate invariant. In the case of the two-sided rule, a new subgoal is generated to prove the correctness of the user-provided invariant, whereas the equivalence of the loop guards is checked automatically as a logical side-condition.

In addition to tactics that mechanize basic rules of pRHL, EasyCrypt implements automated strategies that combine the application of a weakest precondition transformer wp with heuristics to apply basic tactics. The wp transformer operates on deterministic loop-free programs. These strategies can often be used to deal automatically with large fragments of proofs, letting the user focus in the parts that require ingenuity.

C. Reasoning about Probabilities

Since cryptographic results are stated as inequalities on probabilities rather than pRHL judgments, it is important to derive probability claims from pRHL judgments. This can be done mechanically by applying rules in the style of

$$\frac{m_1\,\Psi\,m_2 \quad \vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi \quad \Phi \Rightarrow A\langle 1\rangle \Rightarrow B\langle 2\rangle}{\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]}$$

Game-based proofs often argue that two programs c_1 and c_2 behave identically unless a failure event F is triggered. This is used to conclude that the difference in probability of any event between the two programs is bounded by the probability of F in one of them. Although a syntactic characterization of this lemma is often used (in which failure is represented by a Boolean flag), it can be conveniently expressed and implemented in EasyCrypt in a more general form using pRHL.

Lemma 1 (Fundamental Lemma)**.** *Let c_1 and c_2 be two terminating commands and A, B, F events such that*

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow F\langle 1\rangle \Leftrightarrow F\langle 2\rangle \wedge (\neg F\langle 1\rangle \Rightarrow A\langle 1\rangle \Leftrightarrow B\langle 2\rangle)$$

Then, if the initial memories of both games satisfy Ψ,

$$|\Pr[c_1 : A] - \Pr[c_2 : B]| \leq \Pr[\mathsf{c_1} : F] = \Pr[\mathsf{c_2} : F]$$

In most applications of the above lemma, the failure event F can only be triggered in oracle queries made by an adversary. When the adversary can only make a known bounded number of queries, the following lemma, which we implemented in EasyCrypt, provides a means to bound the probability of failure. (We describe its hypotheses informally, but note that most of them can be captured by pRHL judgments.)

Lemma 2 (Failure event lemma)**.** *Consider a program $c_1; c_2$, an integer expression i, an event F, and $u \in \mathbb{R}$. Assume the following:*

- *Free variables in F and i are only modified by c_1 or oracles in some set O;*
- *After executing c_1, F does not hold and $0 \leq i$;*
- *Oracles $\mathcal{O} \in O$ do not decrease i and strictly increase i when F is triggered;*
- *For every oracle $\mathcal{O}$ in O, $\neg F \Rightarrow \Pr[\mathcal{O} : F] \leq u$*

Then, $\Pr[c_1; c_2 : F \wedge i \leq q] \leq q \cdot u$.

Finally, EasyCrypt implements a simple mechanism to directly compute bounds for the probability of an event in a program. This mechanism can establish, for instance, that the probability that a value uniformly chosen from a set X equals an expression that does not depend on it is exactly $1/|X|$, or that the probability that the same uniformly sampled value belongs to a list of n values that does not depend on it is at most $n/|X|$.

III. The Merkle-Damgård Construction

Merkle-Damgård is a method for building a variable input-length (VIL) hash function from a fixed input-length (FIL) compression function. In its simplest form, the digest of a message is computed by first padding it to a block boundary and then iterating a compression function f over the resulting blocks starting from an initial chaining value IV. A compression function f maps a pair of bitstrings of length k and n (equivalently, a bitstring of length $k+n$) to a bitstring of length n:

$$f : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

A padding function pad converts an arbitrary length message into a list of bitstrings of block size (k is the block-size):

$$\mathsf{pad} : \{0,1\}^* \to (\{0,1\}^k)^*$$

Definition 3 (Merkle-Damgård)**.** *Let f be a compression function and* pad *a padding function as above, and let* $\mathsf{IV} \in \{0,1\}^n$ *be a public value, known as the initialization vector. The hash function* MD *is defined as follows:*

$$\begin{array}{lcl} \mathsf{MD} & : & \{0,1\}^* \to \{0,1\}^n \\ \mathsf{MD}(m) & \stackrel{def}{=} & f^*(\mathsf{pad}(m), \mathsf{IV}) \end{array}$$

where $f^* : (\{0,1\}^k)^* \times \{0,1\}^n \to \{0,1\}^n$ is recursively defined by the equations

$$f^*(\mathsf{nil}, y) \stackrel{def}{=} y \qquad f^*(x::xs, y) \stackrel{def}{=} f^*(xs, f(x,y))$$

The security properties of the compression function preserved by the Merkle-Damgård construction greatly depend on an adequate choice of padding to thwart certain types of attacks. In the remainder, we consider prefix- and suffix-free padding functions.

Definition 4 (Prefix- and suffix-free padding)**.** *A padding function* pad *is* prefix-free *(resp.* suffix-free*) iff for any distinct messages m, m', there is no xs such that* $\mathsf{pad}(m') = \mathsf{pad}(m) \parallel xs$ *(resp.* $\mathsf{pad}(m') = xs \parallel \mathsf{pad}(m)$*).*

Security properties of hash functions are stated as claims about the difficulty of an attacker in achieving certain goals. Collision resistance states that it is hard to find distinct a, b such that $H(a) = H(b)$. Pre-image resistance states that given a digest h, it is hard to find a such that $H(a) = h$. Second preimage resistance states that given a, it is hard to find $b \neq a$ such that $H(a) = H(b)$. Finally, resistance to length-extension attacks states that it is hard to compute $H(a \parallel b)$ from $H(a)$. The precise formulation of these notions and their relationship is addressed in detail in [34].

An established method for proving the security of domain extenders, like MD above, is to show that they are property-preserving: for instance, the seminal works of Merkle [31] and Damgård [23] show that if the compression function f is collision resistant, then the hash function MD with some specific padding function is also collision resistant. Property preservation also applies for other notions; a representative panorama of property preservation for collision resistance, preimage and second preimage resistance appears in [4]. In Section IV we use EasyCrypt to reduce the collision resistance of suffix-free MD to the collision resistance of the underlying compression function.

An alternative method for proving the security of domain extenders is to show that they preserve ideal functionalities, i.e. that when applied to ideal functionalities they yield an ideal functionality. The notion of indifferentiability of Maurer et al. [30] provides an appropriate framework.

Definition 5 (Indifferentiability)**.** *A procedure $\mathcal{C}$ with oracle access to an ideal primitive $\mathcal{G}$ is $(t_{\mathcal{S}}, q, \epsilon)$-indifferentiable from $\mathcal{F}$ if there exists a simulator $\mathcal{S}$ with oracle access to $\mathcal{F}$ and executing within time $t_{\mathcal{S}}$, such that for any distinguisher $\mathcal{D}$ that makes at most q oracle queries, the following inequality holds*

$$\left|\Pr\left[b \leftarrow \mathcal{D}^{\mathcal{C},\mathcal{G}}() : b\right] - \Pr\left[b \leftarrow \mathcal{D}^{\mathcal{F},\mathcal{S}}() : b\right]\right| \leq \epsilon$$

Intuitively, the distinguisher is either given access to $\mathcal{C}^{\mathcal{G}}$ and $\mathcal{G}$, or it is given access to $\mathcal{F}$ and $\mathcal{S}^{\mathcal{F}}$ (see Figure 2). The probability that it succeeds in distinguishing the two scenarios must be small.

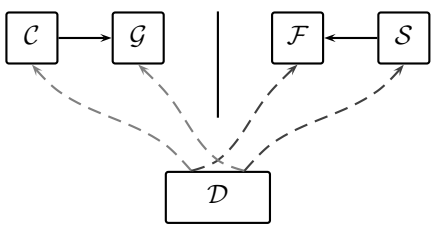

Figure 2. Indifferentiability of $\mathcal{C}$ from an ideal functionality $\mathcal{F}$

In the application considered in this paper, $\mathcal{C}$ represents the Merkle-Damgård construction, $\mathcal{G}$ represents the compres-

sion function and $\mathcal{F}$ represents an idealized hash function. Thus, the role of $\mathcal{S}$ is to simulate the behavior of the compression function, i.e. it should behave towards $\mathcal{F}$ like $\mathcal{G}$ behaves towards the Merkle-Damgård construction. In Section V, we use EasyCrypt to define a simulator $\mathcal{S}$ that proves indifferentiability of MD from a VIL random oracle when the compression function $\mathcal{G}$ is modeled as a FIL random oracle—random oracles [13] are functions that map values in the input domain into uniformly and independently distributed values in the output domain; see Section II for a precise definition.

We conclude this section with two observations about proofs of indifferentiability and property preservation. First, indifferentiability from a random oracle provides weaker guarantees than initially anticipated—see [20] and [33] respectively for discussions on the random oracle model and on the notion of indifferentiability—but nevertheless remains a useful heuristics in the design of hash functions. Second, the two methods are complementary. On the one hand, indifferentiability from a VIL random oracle entails resistance against collision, preimage, second preimage, and length-extension attacks. Thus, preservation of ideal functionalities apparently yields stronger guarantees than property preservation. On the other hand, however, property preservation is typically established under weaker hypotheses and exact security bounds derived from indifferentiability proofs generally deliver looser bounds than direct proofs based on property preservation.

IV. Collision Resistance

We show that finding collisions for MD with a suffix-free padding is at least as hard as finding collisions for f. A collision for the compression function f is a pair of inputs xy_1, xy_2 satisfying the predicate

$$\mathsf{coll}(xy_1, xy_2) \stackrel{\text{def}}{=} xy_1 \neq xy_2 \wedge f(xy_1) = f(xy_2)$$

Theorem 6. *Let* MD *be a Merkle-Damgård hash function with compression function f and a suffix-free padding* pad*. For any algorithm $\mathcal{A}$ finding collisions for* MD *of at most length p, there exists an algorithm $\mathcal{B}$ that finds collisions for f with the same probability and with an overhead of $O(p \cdot t_f)$, where t_f is a bound on the time needed for one evaluation of f.*

Consider the experiment $\mathsf{CR}^{\mathsf{MD}}$ below, in which an adversary $\mathcal{A}$ performs a collision attack against MD:

Game $\mathsf{CR}^{\mathsf{MD}}$:
$(m_1, m_2) \leftarrow \mathcal{A}()$;
$h_1 \leftarrow \mathsf{F}(\mathsf{m}_1)$;
$h_2 \leftarrow \mathsf{F}(\mathsf{m}_2)$;
return $(m_1 \neq m_2 \wedge h_1 = h_2)$

Oracle $\mathsf{F}(m)$:
$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
while $xs \neq \mathsf{nil}$ do
  $y \leftarrow f(\mathsf{hd}(xs), y)$;
  $xs \leftarrow \mathsf{tl}(xs)$;
return y

We prove in EasyCrypt that the algorithm $\mathcal{B}$ shown in Fig. 3 finds collisions for f in the experiment CR^f with at least the same probability as $\mathcal{A}$ finds collisions for MD in $\mathsf{CR}^{\mathsf{MD}}$, i.e.

$$\Pr\left[\mathsf{CR}^{\mathsf{MD}} : \mathsf{res}\right] \leq \Pr\left[\mathsf{CR}^{f} : \mathsf{res}\right] \tag{1}$$

(Recall that res is a keyword that stands for the value returned by the main procedure of the games.) Algorithm $\mathcal{B}$ obtains from $\mathcal{A}$ a pair of messages m_1, m_2, pads them, and iterates the compression function over the first blocks of the longer padded message until the remaining suffix is the same length as the other padded message. It then performs the remaining iterations needed to compute $\mathsf{MD}(m_1)$ and $\mathsf{MD}(m_2)$ in parallel. If m_1, m_2 forms a collision for MD, a collision for f must occur during one of these iterations. Algorithm $\mathcal{B}$ stops as soon as it detects one such collision, returning the colliding inputs as a result.

Game CR^f :
$(xy_1, xy_2) \leftarrow \mathcal{B}()$;
return $\mathsf{coll}(xy_1, xy_2)$

Adversary $\mathcal{B}()$:
$(m_1, m_2) \leftarrow \mathcal{A}()$;
$xs_1 \leftarrow \mathsf{pad}(m_1)$; $y_1 \leftarrow \mathsf{IV}$;
$xs_2 \leftarrow \mathsf{pad}(m_2)$; $y_2 \leftarrow \mathsf{IV}$;
while $|xs_1| > |xs_2|$ do
  $y_1 \leftarrow f(\mathsf{hd}(xs_1), y_1)$; $xs_1 \leftarrow \mathsf{tl}(xs_1)$;
while $|xs_1| < |xs_2|$ do
  $y_2 \leftarrow f(\mathsf{hd}(xs_2), y_2)$; $xs_2 \leftarrow \mathsf{tl}(xs_2)$;
while $\neg\mathsf{coll}((\mathsf{hd}(xs_1), y_1), (\mathsf{hd}(xs_2), y_2)) \wedge xs_1 \neq \mathsf{nil}$ do
  $y_1 \leftarrow f(\mathsf{hd}(xs_1), y_1)$; $xs_1 \leftarrow \mathsf{tl}(xs_1)$;
  $y_2 \leftarrow f(\mathsf{hd}(xs_2), y_2)$; $xs_2 \leftarrow \mathsf{tl}(xs_2)$;
return $((\mathsf{hd}(xs_1), y_1), (\mathsf{hd}(xs_2), y_2))$

Figure 3. A collision-finder $\mathcal{B}$ for the compression function f

In order to show (1) it suffices to prove the relational judgment:

$$\vdash \mathsf{CR}^{\mathsf{MD}} \sim \mathsf{CR}^{f} : \mathsf{true} \Longrightarrow \mathsf{res}\langle 1\rangle \Rightarrow \mathsf{res}\langle 2\rangle \tag{2}$$

Proving this judgment involves non-trivial relational reasoning because equivalent computations in the related games are not performed in lockstep. We begin by inlining the call to $\mathcal{B}$ in CR^f and showing that the relational post-condition

$$(m_1, m_2)\langle 1\rangle = (m_1, m_2)\langle 2\rangle \wedge (h_1 = \mathsf{MD}(m_1) \wedge h_2 = \mathsf{MD}(m_2))\langle 1\rangle$$

holds after the call to $\mathcal{A}$ in both programs and the two calls to F in $\mathsf{CR}^{\mathsf{MD}}$. To show this, we prove that oracle F correctly implements function MD using the one-sided rule for loops—the needed invariant is simply $f^*(xs, y) = \mathsf{MD}(m)$. At this point, note that if $m_1 = m_2$, judgment (2) holds trivially (we only have to check that $\mathcal{B}$ terminates). We are left with the case $m_1 \neq m_2$. Assume w.l.o.g. that $|\mathsf{pad}(m_2)| \leq |\mathsf{pad}(m_1)|$, in which case $\mathcal{B}$ never enters its second loop and the following invariant holds for the first:

$$f^*(xs_1, y_1) = \mathsf{MD}(m_1) \wedge f^*(xs_2, y_2) = \mathsf{MD}(m_2) \wedge m_1 \neq m_2 \wedge |xs_2| \leq |xs_1| \wedge xs_2 = \mathsf{pad}(m_2) \wedge \exists xs'.\ xs' \parallel xs_1 = \mathsf{pad}(m_1) \tag{3}$$

We prove that if the messages m_1, m_2 output by $\mathcal{A}$ collide, the last loop necessarily exits because a collision is found. This can be shown by means of the following loop invariant:

$$f^*(xs_1, y_1) = \mathsf{MD}(m_1) \wedge f^*(xs_2, y_2) = \mathsf{MD}(m_2) \wedge$$
$$|xs_2| = |xs_1| \wedge$$
$$(xs_1 = xs_2 \Rightarrow y_1 \neq y_2)$$

Note that (3) and the negation of the guard of the first loop imply that the above invariant holds initially. In particular, the last implication holds because if xs_1 and xs_2 were equal, there would exist a prefix xs' such that $xs' \parallel \mathsf{pad}(m_2) = \mathsf{pad}(m_1)$, contradicting the fact that pad is suffix-free. Finally, observe that the last loop can exit either because a collision for f is found or because $xs_1 = \mathsf{nil}$. In this latter case, it must be the case that $xs_2 = \mathsf{nil}$ and therefore $y_1 = \mathsf{MD}(m_1) = \mathsf{MD}(m_2) = y_2$. However, from the last implication in the invariant we also have $y_1 \neq y_2$, which leads to a contradiction that renders this case trivial.

V. Indifferentiability

We prove the indifferentiability of the MD construction from a random oracle in $\{0,1\}^* \to \{0,1\}^n$ when its compression function f is modeled as a random oracle in $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ and its padding function is prefix-free. Our proof is based on [22].

Theorem 7 (Indifferentiability of MD)**.** *The Merkle-Damgård construction* MD *with an ideal compression function f, prefix-free padding* pad*, and initialization vector* IV *is $(t_{\mathcal{S}}, q_{\mathcal{D}}, \epsilon)$-indifferentiable from a variable input-length random oracle $F : \{0,1\}^* \to \{0,1\}^n$, where*

$$\epsilon = \frac{3\ell^2\ q_D^2}{2^n} \qquad t_{\mathcal{S}} = O(\ell\ q_D^2)$$

and ℓ is an upper bound on the block-length of $\mathsf{pad}(m)$ for any message m appearing in a query of the distinguisher.

In what we call the real scenario, a distinguisher $\mathcal{D}$ has access to an oracle F_q implementing the function MD and to a random oracle $f_q : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ that models the compression function. In contrast, in the ideal scenario, $\mathcal{D}$ has access to a random oracle $F_q : \{0,1\}^* \to \{0,1\}^n$ and f_q is simulated. See Fig. 4 for a formulation of these two scenarios as games. To prevent $\mathcal{D}$ from making more than q oracle queries, we enforce a bound $q = \ell\ q_{\mathcal{D}}$ on the counter $\mathbf{q}_f$, that counts the number of evaluations of the compression function in game $\mathsf{G}_{\mathrm{real}}$. Note that this is more permissive than the proof of Coron et al. [22], since it allows the distinguisher to trade queries to F_q for queries to f_q. Indeed, if $\mathcal{D}$ makes n_f queries to f_q and n_F queries to F_q, we require

$$\mathbf{q}_f \leq n_f + \ell\ n_F \leq \ell\ (n_f + n_F) \leq \ell\ q_{\mathcal{D}} = q$$

We show that the simulator f_q in $\mathsf{G}_{\mathrm{ideal}}$ behaves consistently with a random oracle. Whenever the distinguisher makes a query (x, y) to oracle f_q, the simulator looks among all previous queries for a sequence that could be the chain of inputs to the compression function used to compute the hash of some message m, for which x is the last block of $\mathsf{pad}(m)$. We call such a sequence a *complete chain*, and we define it formally below. When such a sequence is found, the simulator queries F for the hash of m and forwards the answer to the distinguisher. Otherwise, the simulator answers with a uniformly distributed random value. Figure 5 shows how this simulator would react to a sequence of queries

$$y_2 \leftarrow f_q(x_1, \mathsf{IV});\ y_3 \leftarrow f_q(x_2, y_2);\ y_4 \leftarrow f_q(x_3, y_3)$$

where $x_1 \parallel x_2 \parallel x_3 = \mathsf{pad}(m)$. The first two queries will be answered with random values, while the third completes a chain and is answered by forwarding $\mathsf{pad}^{-1}(x_1 \parallel x_2 \parallel x_3)$ to F; this maintains the consistency with the real scenario.

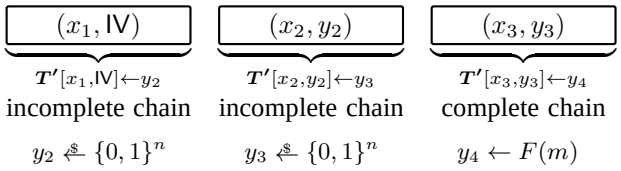

Figure 5. An example illustrating how the simulator works

Definition 8 (Complete chain)**.** *A complete chain in a map $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ is a sequence $(x_1, y_1) \ldots (x_i, y_i)$ such that $y_1 = \mathsf{IV}$ and*

1) $\forall j = 1 \ldots i-1.\ (x_j, y_j) \in \mathsf{dom}(T) \wedge T[x_j, y_j] = y_{j+1}$
2) *$x_1 \parallel \ldots \parallel x_i$ is in the domain of pad^{-1}*

The function $\mathsf{findseq}((x, y), \boldsymbol{T'})$ used by the simulator searches in $\boldsymbol{T'}$ for a complete chain of the form $(x_1, y_1) \ldots (x_i, y_i)(x, y)$ and returns $x_1 \| \ldots \| x_i$, or $\perp$ if no such chain is found.

To help SMT solvers and automated provers check logical side-conditions arising in our proofs, we needed to derive several auxiliary lemmas: e.g., if a finite map T is injective and does not map any entry to the value IV, every complete chain is determined by its last element—that is, for any given (x, y), the value of $\mathsf{findseq}((x, y), \boldsymbol{T'})$ is uniquely determined. All of these lemmas have been mechanically verified based solely on the axiomatization and definitions of elementary operations. In many cases, EasyCrypt is able to verify the validity of these lemmas automatically. The more involved lemmas have been manually verified in the Coq proof assistant.

The proof proceeds by stepwise transforming the game $\mathsf{G}_{\mathrm{real}}$ into the game $\mathsf{G}_{\mathrm{ideal}}$, upper-bounding the probability that the outcome of consecutive games differ. By summing up over these probabilities, we obtain a concrete bound for the advantage of the distinguisher in telling apart the initial and final games. Specifically, we prove:

$$|\Pr[\mathsf{G}_{\mathrm{real}} : b] - \Pr[\mathsf{G}_{\mathrm{ideal}} : b]| \leq \frac{3q^2}{2^n} \qquad (4)$$

Game $\mathsf{G}_{\mathrm{real}}$:	**Oracle** $F_q(m)$:	**Oracle** $f(x,y)$:	**Oracle** $f_q(x,y)$:

$\mathbf{q}_f \leftarrow 0$;
$\boldsymbol{T} \leftarrow \emptyset$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b
	$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
	if $\mathbf{q}_f + \lvert xs\rvert \leq q$ then
	$\mathbf{q}_f \leftarrow \mathbf{q}_f + \lvert xs\rvert$;
	while $xs \neq \mathsf{nil}$ do
	$y \leftarrow f(\mathsf{hd}(xs), y)$;
	$xs \leftarrow \mathsf{tl}(xs)$
	return y
		if $(x,y) \notin \mathsf{dom}(\boldsymbol{T})$ then
		$z \xleftarrow{\$} \{0,1\}^n$;
		$\boldsymbol{T}[x,y] \leftarrow z$
		return $\boldsymbol{T}[x,y]$
			if $\mathbf{q}_f + 1 \leq q$ then
			$\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
			$z \leftarrow f(x,y)$;
			else $z \leftarrow \mathsf{IV}$
			return z

Game $\mathsf{G}_{\mathrm{ideal}}$:	**Oracle** $F_q(m)$:	**Oracle** $F(m)$:	**Oracle** $f_q(x,y)$:

$\mathbf{q}_f \leftarrow 0$;
$\boldsymbol{R}, \boldsymbol{T'} \leftarrow \emptyset$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b
	$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
	if $\mathbf{q}_f + \lvert xs\rvert \leq q$ then
	$\mathbf{q}_f \leftarrow \mathbf{q}_f + \lvert xs\rvert$;
	$z \leftarrow F(m)$
	else $z \leftarrow \mathsf{IV}$
	return z
		if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
		$z \xleftarrow{\$} \{0,1\}^n$;
		$\boldsymbol{R}[m] \leftarrow z$
		return $\boldsymbol{R}[m]$
			if $\mathbf{q}_f + 1 \leq q$ then
			if $(x,y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
			$xs \leftarrow \mathsf{findseq}((x,y), \boldsymbol{T'})$
			if $xs \neq \perp$ then
			$\boldsymbol{T'}[x,y] \leftarrow F(\mathsf{pad}^{-1}(xs) \Vert [x]))$
			else
			$\boldsymbol{T'}[x,y] \xleftarrow{\$} \{0,1\}^n$
			$z \leftarrow \boldsymbol{T'}[x,y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
			else $z \leftarrow \mathsf{IV}$
			return z

Figure 4. The games $\mathsf{G}_{\mathrm{real}}$ and $\mathsf{G}_{\mathrm{ideal}}$

Game $\mathsf{G}_{\mathrm{real}'}$:	**Oracle** $F_q(m)$:	**Oracle** $f(x,y)$:	**Oracle** $f_q(x,y)$:

$\mathbf{q}_f \leftarrow 0$;
$\boldsymbol{T}, \boldsymbol{T'} \leftarrow \emptyset$;
$\boldsymbol{Y} \leftarrow \mathsf{nil}$;
$\boldsymbol{Z} \leftarrow \mathsf{IV}::\mathsf{nil}$;
$\mathbf{bad}_1 \leftarrow \mathsf{false}$;
$\mathbf{bad}_2 \leftarrow \mathsf{false}$;
$\mathbf{bad}_3 \leftarrow \mathsf{false}$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b
	$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
	if $\mathbf{q}_f + \lvert xs\rvert \leq q$ then
	$\mathbf{q}_f \leftarrow \mathbf{q}_f + \lvert xs\rvert$;
	while $\lvert xs\rvert > 1$ do
	$y \leftarrow f_{\mathbf{bad}}(\mathsf{hd}(xs), y)$;
	$xs \leftarrow \mathsf{tl}(xs)$
	$y \leftarrow f_{\mathbf{bad}}(\mathsf{hd}(xs), y)$
	return y
		if $(x,y) \notin \mathsf{dom}(\boldsymbol{T})$ then
		$z \xleftarrow{\$} \{0,1\}^n$;
		$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
		$\boldsymbol{T}[x,y] \leftarrow z$
		return $\boldsymbol{T}[x,y]$
		Oracle $f_{\mathbf{bad}}(x,y)$:
		if $(x,y) \notin \mathsf{dom}(\boldsymbol{T})$ then
		$z \xleftarrow{\$} \{0,1\}^n$;
		$\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
		$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
		$\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
		$\boldsymbol{T}[x,y] \leftarrow z$
		return $\boldsymbol{T}[x,y]$
			if $\mathbf{q}_f + 1 \leq q$ then
			if $(x,y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
			$xs \leftarrow \mathsf{findseq}((x,y), \boldsymbol{T'})$
			if $xs \neq \perp$ then
			$\boldsymbol{T'}[x,y] \leftarrow f_{\mathbf{bad}}(x,y)$
			else
			if $\mathsf{set\_bad3}(y, \boldsymbol{T'}, \boldsymbol{T})$ then
			$\mathbf{bad}_3 \leftarrow \mathsf{true}$;
			$\boldsymbol{T'}[x,y] \leftarrow f(x,y)$
			else
			$\boldsymbol{T'}[x,y] \leftarrow f_{\mathbf{bad}}(x,y)$
			$z \leftarrow \boldsymbol{T'}[x,y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
			else $z \leftarrow \mathsf{IV}$
			return z

Figure 6. The game $\mathsf{G}_{\mathrm{real}'}$

We begin by considering the game $\mathsf{G}_{\mathrm{real}'}$ defined in Fig. 6. We introduce events $\mathbf{bad}_1$, $\mathbf{bad}_2$, and $\mathbf{bad}_3$ that will be needed later. First, we introduce a copy of oracle f, which we call $f_{\mathbf{bad}}$. Both use the same map $\boldsymbol{T}$ to store previously answered queries, the difference is that $f_{\mathbf{bad}}$ may trigger events $\mathbf{bad}_1$ and $\mathbf{bad}_2$. We also introduce the lists $\boldsymbol{Y}$ and $\boldsymbol{Z}$ that allow us to appropriately detect when these events occur. In addition, we modify the simulator f_q to maintain a map $\boldsymbol{T'}$ of queries known to the distinguisher. Observe that $\boldsymbol{T'} \subseteq \boldsymbol{T}$, because queries to F_q result in entries being added only to $\boldsymbol{T}$, whereas queries to f_q result in the same entries being added to both $\boldsymbol{T}$ and $\boldsymbol{T'}$. Additionally, the simulator f_q behaves in two different ways depending on whether $\mathsf{findseq}((x,y), \boldsymbol{T'}) \neq \perp$. If this condition holds, there is a complete chain in map $\boldsymbol{T'}$ ending in (x,y). In this case, in game $\mathsf{G}_{\mathrm{ideal}}$ the simulator should call oracle F to maintain consistency with the random oracle; otherwise the simulator could just sample a fresh random value. In this game, oracle f_q returns the same answer in both cases, but sets $\mathbf{bad}_{\{1,2,3\}}$ accordingly. Lastly, we also unroll the last iteration of the loop in F_q.

Note that instrumenting the game with the additional map $\boldsymbol{T'}$ and the failure events $\mathbf{bad}_{\{1,2,3\}}$ does not change the observable behavior. Therefore,

$$\Pr\left[\mathsf{G}_{\mathrm{real}} : b\right] = \Pr\left[\mathsf{G}_{\mathrm{real}'} : b\right]$$

In game $\mathsf{G}_{\mathrm{realRO}}$, defined in Fig. 7, we introduce a random oracle $\mathrm{RO} : \{0,1\}^* \to \{0,1\}^n$ and replace every call $f_{\mathbf{bad}}(x,y)$ in game $\mathsf{G}_{\mathrm{real}'}$ where (x,y) ends a complete chain in $\boldsymbol{T}$ with a call to $\mathrm{RO}(m,y)$ where m is the

unpadded message of the chain. I.e., in oracle f_q we call RO if findseq is successful and in oracle F_q we call RO instead of the last call to $f_{\mathbf{bad}}$. We also introduce the map $\boldsymbol{I} : \mathbb{N} \to \{0,1\}^n \times \mathbb{B}$ which enumerates all sampled chaining values and includes a *tainted* flag to keep track of values known to the distinguisher. We introduce an indirection in map $\boldsymbol{T}$ and $\boldsymbol{T'}$ through the use of map $\boldsymbol{I}$. This allows us to keep track of the order in which queries were made and to know which answers we could re-sample without introducing inconsistencies in the view of the distinguisher.

The failure events that were introduced in the last step capture certain dependencies on previous queries that the distinguisher may exploit to tell apart games $\mathsf{G}_{\mathrm{real}'}$ and $\mathsf{G}_{\mathrm{realRO}}$. We prove that games $\mathsf{G}_{\mathrm{real}'}$ and $\mathsf{G}_{\mathrm{realRO}}$ behave the same provided these failure events do not occur.

1) $\mathbf{bad}_1$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that is either IV or has already been sampled for a distinct query before. The role of this event is twofold: on the one hand, if IV is sampled as a random value, then there could exist a complete chain in $\boldsymbol{T}$ that is a suffix of another complete chain in $\boldsymbol{T}$ as illustrated in the first example of Figure 8 (here $\boldsymbol{T}[x_2, y_2] = \mathsf{IV}$). The problem is that oracle F_q in the game $\mathsf{G}_{\mathrm{real}}$ will generate the same values for the two messages corresponding to those two chains, while F_q in the game $\mathsf{G}_{\mathrm{ideal}}$ most likely will not. On the other hand, if a sampled value has been sampled for another query before, then there could exist two complete chains in $\boldsymbol{T}$ that collide at some point and are identical from that point on as illustrated in the second example of Figure 8. Again the two corresponding messages would yield the same answer in $\mathsf{G}_{\mathrm{real}}$ but most likely not in $\mathsf{G}_{\mathrm{ideal}}$ on queries to F_q. By requiring that event $\mathbf{bad}_1$ does not occur, we guarantee that in game $\mathsf{G}_{\mathrm{real}'}$ the map $\boldsymbol{T}$ is injective and does not map any value to IV.
2) $\mathbf{bad}_2$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that has already been used as a chaining value in a previous query. This means that this query may be part of a chain of which the distinguisher has already queried later points in the chain, which should not be possible. The event also captures that no fixed-points (i.e. entries of the form $\boldsymbol{T}[x, y] = y$) should be sampled.
3) $\mathbf{bad}_3$ is triggered whenever a chaining value y in a query has already been sampled as a random value and is in the range of $\boldsymbol{T}$ for some previous query (x', y'), but (x', y') does not appear in the domain of $\boldsymbol{T'}$ and (x', y') is not the last element of a complete chain in $\boldsymbol{T}$. Intuitively, this means that y was never returned by f_q or F_q and hence the distinguisher managed to guess a random value.

In order to relate games $\mathsf{G}_{\mathrm{real}'}$ and $\mathsf{G}_{\mathrm{realRO}}$ in case that

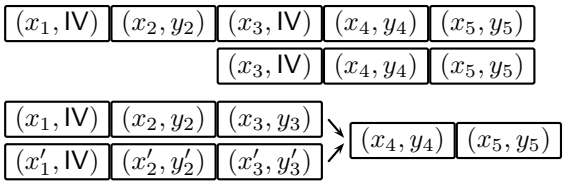

Figure 8. Two examples illustrating the necessity of event $\mathbf{bad}_1$

$\mathsf{findseq}((x, y), \boldsymbol{T'})$ in f_q succeeds in both games, we need to show that the call $f_{\mathbf{bad}}(x, y)$ in $\mathsf{G}_{\mathrm{real}'}$ and the call $\mathrm{RO}(m, y)$ in $\mathsf{G}_{\mathrm{realRO}}$ behave similarly. For this we show that the following invariant is preserved in both games: for all complete chains c in the map $\boldsymbol{T}$ of game $\mathsf{G}_{\mathrm{real}'}$ with $\mathsf{last}(c) \in \mathsf{dom}(T)$, it holds that c's associated message is in $\mathsf{dom}(\boldsymbol{R})$ of game $\mathsf{G}_{\mathrm{realRO}}$ and, vice versa, every message in $\mathsf{dom}(\boldsymbol{R})$ of game $\mathsf{G}_{\mathrm{realRO}}$ has a corresponding complete chain c in the map $\boldsymbol{T}$ of game $\mathsf{G}_{\mathrm{real}'}$ with $\mathsf{last}(c) \in \mathsf{dom}(T)$. This invariant allows EasyCrypt to prove this case by inferring that $(x, y) \in \mathsf{dom}(\boldsymbol{T})$ in game $\mathsf{G}_{\mathrm{real}'}$ if and only if $m \in \mathsf{dom}(\boldsymbol{R})$ in game $\mathsf{G}_{\mathrm{realRO}}$.

Proving that the aforementioned invariant is preserved in the games requires several other invariants. Most of them merely relate the representation of maps in both games; we omit these technical details. The essential invariant is that the distinguisher queries f_q for points in a chain only if it has already queried the preceding part of the chain. This is important as it implies that each chain will be completed by a query for its last element, in which case findseq will detect this query and the corresponding message will be added to $\boldsymbol{R}$. In game $\mathsf{G}_{\mathrm{real}'}$, the predicate set_bad3 enforces this ordering by triggering event $\mathbf{bad}_3$. The probability of this event is negligible, because it means that y was never output by f_q or F_q and hence is not known to the distinguisher. In game $\mathsf{G}_{\mathrm{realRO}}$, we use the map $\boldsymbol{I}$ to iterate over all chaining values in order to check for the ordering mentioned above.

In oracle F_q of game $\mathsf{G}_{\mathrm{realRO}}$, the computation of the Merkle-Damgård construction is split into three stages due to the different usage of the maps $\boldsymbol{T'}$, $\boldsymbol{T'_i}$, and $\boldsymbol{T}$. The first loop computes the construction for values that were already queried by the distinguisher and are therefore in $\mathsf{dom}(\boldsymbol{T'})$. The restriction that the distinguisher may only query chains in order implies that such values occur only in the prefix of a chain. The second loop handles values that were already used before by oracle F_q, and the third loop samples fresh chaining values. Relating the final call to $f_{\mathbf{bad}}$ in game $\mathsf{G}_{\mathrm{real}'}$ and the final call to RO in game $\mathsf{G}_{\mathrm{realRO}}$ is similar to this case in oracle f_q. We prove that the advantage in differentiating between games $\mathsf{G}_{\mathrm{real}'}$ and $\mathsf{G}_{\mathrm{realRO}}$ is upper bounded by the probability of any of $\mathbf{bad}_1, \mathbf{bad}_2, \mathbf{bad}_3$ occurring in game $\mathsf{G}_{\mathrm{realRO}}$.

$$|\Pr[\mathsf{G}_{\mathrm{real}'} : b] - \Pr[\mathsf{G}_{\mathrm{realRO}} : b]| \leq \Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_1 \vee \mathbf{bad}_2 \vee \mathbf{bad}_3]$$

Game $\mathsf{G}_{\mathrm{realRO}}$:
$\mathbf{q}_f \leftarrow 0$;
$\mathbf{q}'_f \leftarrow 1$;
$\boldsymbol{T}, \boldsymbol{T'}, \boldsymbol{T'_i}, \boldsymbol{R}, \boldsymbol{I} \leftarrow \emptyset$;
$\boldsymbol{I}[0] \leftarrow (\mathsf{IV}, \mathsf{false})$;
$\boldsymbol{Y} \leftarrow \mathsf{nil}$;
$\boldsymbol{Z} \leftarrow \mathsf{IV}::\mathsf{nil}$;
$\mathbf{bad}_1 \leftarrow \mathsf{false}$;
$\mathbf{bad}_2 \leftarrow \mathsf{false}$;
$\mathbf{bad}_3 \leftarrow \mathsf{false}$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b

Oracle $F_q(m)$:
$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
$i \leftarrow 0$;
if $\mathbf{q}_f + |xs| \leq q$ then
  $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
  while $|xs| > 1 \wedge$
    $(\mathsf{hd}(xs), y) \in \mathsf{dom}(\boldsymbol{T'})$ do
    $i \leftarrow \boldsymbol{T'_i}[\mathsf{hd}(xs), y]$;
    $y \leftarrow \boldsymbol{T'}[\mathsf{hd}(xs), y]$;
    $xs \leftarrow \mathsf{tl}(xs)$;
  while $|xs| > 1 \wedge$
    $(\mathsf{hd}(xs), i) \in \mathsf{dom}(\boldsymbol{T})$ do
    $i \leftarrow \boldsymbol{T}[\mathsf{hd}(xs), i]$;
    $y \leftarrow \mathsf{fst}(\boldsymbol{I}[i])$;
    $xs \leftarrow \mathsf{tl}(xs)$;
  while $|xs| > 1$ do
    $z \xleftarrow{\$} \{0,1\}^n$;
    $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
    $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
    $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
    $\boldsymbol{T}[\mathsf{hd}(xs), i] \leftarrow \mathbf{q}'_f$;
    $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{true})$;
    $i \leftarrow \mathbf{q}'_f$;
    $y \leftarrow z$;
    $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
    $xs \leftarrow \mathsf{tl}(xs)$
  $y \leftarrow \mathsf{fst}(RO(m, y))$
return y

Oracle $RO(m, y)$:
if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
  $z \xleftarrow{\$} \{0,1\}^n$;
  $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
  $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
  $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
  $\boldsymbol{R}[m] \leftarrow (z, \mathbf{q}'_f)$
  $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$
  $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
return $\boldsymbol{R}[m]$

Oracle $f_q(x, y)$:
if $\mathbf{q}_f + 1 \leq q$ then
  if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
    $xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$
    if $xs \neq \perp$ then
      $m \leftarrow \mathsf{pad}^{-1}(xs)) \parallel [x])$;
      $(z, j) \leftarrow RO(m, y)$;
      $\boldsymbol{T'}[x, y] \leftarrow z$; $\boldsymbol{T'_i}[x, y] \leftarrow j$;
    else
      $found, found\_bad3 \leftarrow \mathsf{false}$;
      $j, k' \leftarrow 0$;
      while $k' < \mathbf{q}'_f$ do
        if $\mathsf{snd}(\boldsymbol{I}[k'])$ then
          $found\_bad3 \leftarrow (\mathsf{fst}(\boldsymbol{I}[k']) = y)$;
        else if $\neg found \wedge \mathsf{fst}(\boldsymbol{I}[k']) = y \wedge$
          $(x, k') \in \mathsf{dom}(\boldsymbol{T}) \wedge$
          $\mathsf{snd}(\boldsymbol{I}[\boldsymbol{T}[x, k']])$ then
          $found \leftarrow \mathsf{true}$; $j \leftarrow \boldsymbol{T}[x, k']$;
        $k' \leftarrow k' + 1$;
      if $found$ then
        $z \leftarrow \mathsf{fst}(\boldsymbol{I}[j])$; $\boldsymbol{I}[j] \leftarrow (z, \mathsf{false})$;
        $\boldsymbol{T'}[x, y] \leftarrow z$; $\boldsymbol{T'_i}[x, y] \leftarrow j$;
      else
        if $found\_bad3$ then
          $\mathbf{bad}_3 \leftarrow \mathsf{true}$;
          $z \xleftarrow{\$} \{0,1\}^n$;
          $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$;
          $\boldsymbol{T'}[x, y] \leftarrow z$;
          $\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
          $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
        else
          $z \xleftarrow{\$} \{0,1\}^n$;
          $\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
          $\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
          $\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
          $\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathsf{false})$;
          $\boldsymbol{T'}[x, y] \leftarrow z$;
          $\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
          $\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
  $z \leftarrow \boldsymbol{T'}[x, y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
else $z \leftarrow \mathsf{IV}$
return z

Figure 7. The game $\mathsf{G}_{\mathrm{realRO}}$

To finish the proof, we have to relate $\Pr[\mathsf{G}_{\mathrm{realRO}} : b]$ with $\Pr[\mathsf{G}_{\mathrm{ideal}} : b]$ and bound the probability of the failure events in game $\mathsf{G}_{\mathrm{realRO}}$. We first focus on the probability of $\mathbf{bad}_1$ and $\mathbf{bad}_2$. Event $\mathbf{bad}_1$ (resp. $\mathbf{bad}_2$) is set when a freshly sampled value z is in the list $\boldsymbol{Z}$ (resp. $\boldsymbol{Y}$); since the size of both lists is bounded by q, this occurs with probability at most $q\ 2^{-n}$, for each of the possible q queries.

Note that oracles F_q, RO, and f_q in game $\mathsf{G}_{\mathrm{realRO}}$ use the same code to detect the failure events $\mathbf{bad}_1$ and $\mathbf{bad}_2$ when sampling a fresh value z. We can wrap this code in a new oracle that meets the conditions of Lemma 2: we take $u = q\ 2^{-n}$ and $i = |\boldsymbol{Z}|$ (resp. $|\boldsymbol{Y}|$). We get

$$\Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_1] \leq \frac{q^2}{2^n} \quad \Pr[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_2] \leq \frac{q^2}{2^n}$$

We are left to bound the probability of $\mathbf{bad}_3$ and relate the game $\Pr[\mathsf{G}_{\mathrm{realRO}} : b]$ with $\Pr[\mathsf{G}_{\mathrm{ideal}} : b]$. Note that in game $\mathsf{G}_{\mathrm{realRO}}$ chaining values are sampled eagerly, i.e. for a query m, oracle F_q samples chaining values z that are independent of the distinguisher's view (their associated flag is set to true). These values might later on become known to the distinguisher if it recomputes the Merkle-Damgård construction for m using oracle f_q (we identify this case setting $found = \mathsf{true}$). We want to transform the game so that chaining values are sampled lazily (as in game $\mathsf{G}_{\mathrm{ideal}}$).

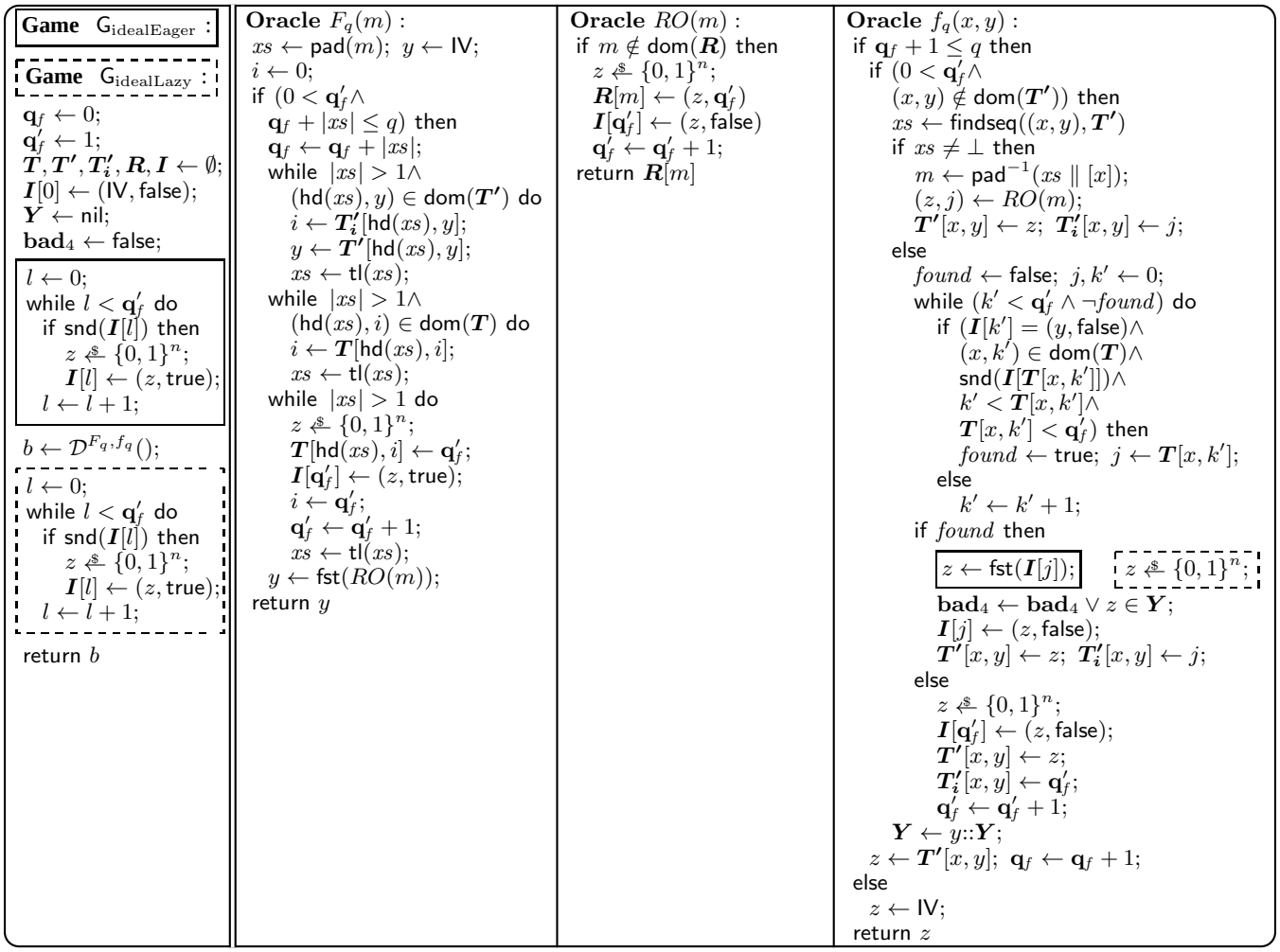

Figure 9. The games $\mathsf{G}_{\mathrm{idealEager}}$ and $\mathsf{G}_{\mathrm{idealLazy}}$

The same kind of argument can be used for $\mathbf{bad}_3$. This event is set whenever the distinguisher makes a query (x, y) to f_q with y coinciding with a value uniformly and independently distributed w.r.t. its view.

We modify game $\mathsf{G}_{\mathrm{realRO}}$ in order to prepare for the transition from eager to lazily sampled chaining values: the body of game $\mathsf{G}_{\mathrm{idealEager}}$ (see Figure 9) contains a loop which re-samples all chaining values that are unknown to the adversary, i.e., the values for which the second component in map $\boldsymbol{I}$ is set to true. Furthermore, game $\mathsf{G}_{\mathrm{idealEager}}$ drops the failure events $\mathbf{bad}_{\{1,2,3\}}$, but introduces a new failure event $\mathbf{bad}_4$. We show that if $\mathbf{bad}_3$ is triggered in game $\mathsf{G}_{\mathrm{realRO}}$, then in $\mathsf{G}_{\mathrm{idealEager}}$ $\mathbf{bad}_4$ is set to true or there exists an i such that $\boldsymbol{I}[i] = (v, \mathsf{true})$ and $v \in \boldsymbol{Y}$. We get

$$\begin{aligned} \Pr\left[\mathsf{G}_{\mathrm{realRO}} : b\right] &= \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : b\right] \\ \Pr\left[\mathsf{G}_{\mathrm{realRO}} : \mathbf{bad}_3\right] &\leq \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] \end{aligned}$$

where $\mathrm{I}_\exists = \exists i.\ 0 \leq i \leq \mathbf{q}'_f \wedge \mathsf{snd}(\boldsymbol{I}[i]) \wedge \mathsf{fst}(\boldsymbol{I}[i]) \in \boldsymbol{Y}$.

In game $\mathsf{G}_{\mathrm{idealLazy}}$ (see Figure 9), the loop we introduced in the last game is swapped with the call to the distinguisher and oracle f_q samples the chaining values lazily (the branch $found$ re-samples the value of z). In order to prove the equivalence with the previous game, we need to show that the loop that resamples the values unknown to the adversary *swaps* with calls to oracles F_q and f_q in games $\mathsf{G}_{\mathrm{idealEager}}$ and $\mathsf{G}_{\mathrm{idealLazy}}$. We obtain

$$\begin{aligned} \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : b\right] &= \Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : b\right] \\ \Pr\left[\mathsf{G}_{\mathrm{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] &= \Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] \end{aligned}$$

It is easy to see that games $\mathsf{G}_{\mathrm{idealLazy}}$ and $\mathsf{G}_{\mathrm{ideal}}$ are equivalent w.r.t. b; the global variable $\mathbf{q}_f$ and the maps $\boldsymbol{R}$ and $\boldsymbol{T'}$ are equivalent in both games. The other variables in game $\mathsf{G}_{\mathrm{idealLazy}}$ and its loops do not influence the behavior of its oracles. We show that

$$\Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : b\right] = \Pr\left[\mathsf{G}_{\mathrm{ideal}} : b\right].$$

We still have to bound the probability of $\mathbf{bad}_4 \vee \mathrm{I}_\exists$ in game $\mathsf{G}_{\mathrm{idealLazy}}$. To do this, we simply modify the while loop in the code of the game by replacing the instruction $z \xleftarrow{\$} \{0,1\}^n$ with

$$z \xleftarrow{\$} \{0,1\}^n; \mathbf{bad}_4 \leftarrow \mathbf{bad}_4 \vee z \in \boldsymbol{Y}$$

This leads to a game $\mathsf{G}_{\mathrm{idealLazy}'}$, for which we show

$$\Pr\left[\mathsf{G}_{\mathrm{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] \leq \Pr\left[\mathsf{G}_{\mathrm{idealLazy}'} : \mathbf{bad}_4\right]$$

We finally use the same technique as for $\mathbf{bad}_1$ to bound the probability of $\mathbf{bad}_4$ in game $\mathsf{G}_{\mathrm{idealLazy}'}$, and obtain

$$\Pr\left[\mathsf{G}_{\mathrm{idealLazy}'} : \mathbf{bad}_4\right] \leq \frac{q^2}{2^n}$$

Putting the (in-)equalities proved above together we prove (4), which completes the proof of Theorem 7.

VI. Security Proofs of Generalized Merkle-Damgård

To avoid inheriting structural weaknesses in the original Merkle-Damgård construction, existing hash functions employ instead slight variants of it. One well-known variant is the wide-pipe design, which uses an internal state larger than the final output [22], [27]. Many variants are subsumed by the following Generalized Merkle-Damgård construction.

Definition 9 (Generalized Merkle-Damgård)**.** *Let* $\mathsf{IV} \in \{0,1\}^n$ *be a public initialization vector and* f, g *be two compression functions of type*

$$f, g : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

Consider a function $\mathsf{pad} : \{0,1\}^* \to (\{0,1\}^k)^* \times \{0,1\}^k$ *that converts an arbitrary length message into a non-empty list of blocks of length* k*, singling out the last block. The hash function* GMD *is defined as follows:*

$$\begin{array}{lcl} \mathsf{GMD} & : & \{0,1\}^* \to \{0,1\}^\ell \\ \mathsf{GMD}(m) & \stackrel{def}{=} & \mathbf{let}\ (x,y) = \mathsf{pad}(m)\ \mathbf{in}\ [g(y, f^*(x, \mathsf{IV}))]^\ell \end{array}$$

where f^* *is defined as in Def. 3 and* $[x]^\ell$ *chops off the* $n-\ell$ *least significant bits from* x*, i.e. discards all but the leading* ℓ *bits.*

The NIST SHA-3 competition started in November 2007 with the objective of selecting new cryptographic hash functions to augment the set specified by the U.S. Federal Information Processing Standard (FIPS) 180-3, which includes the SHA-1 and SHA-2 algorithms. After receiving 64 entries, NIST selected 51 candidates for the first round, further narrowed down the list to just 14 candidates for the second round, and announced 5 finalists in December 2010: BLAKE [6], Grøstl [25], JH [38], Keccak [14], and Skein [24]. A public comment period has started after this announcement and the winner is expected to be selected before the end of 2012.

The security of all SHA-3 finalists, and of many second round candidates, has been thoroughly scrutinized. Two survey articles summarize known results [2], [3]. While the algorithmic descriptions of the finalists and their exact security bounds fit in one page (see [3]), the corresponding security proofs are technically involved and need to be cautiously adapted to account for the specificities of each function. As a consequence, it is difficult to assess the validity of security claims for individual candidates and machine checking their proofs is an appealing perspective. In the remainder of this section we discuss the applicability of the proofs presented in Sections IV and V to SHA-3 finalists.

The five SHA-3 finalists are based on the iterated hash function design that underlies the Merkle-Damgård construction, but incorporate some variations such as round-dependent tweaks, counters, final transformations, and chopping. We observe that, in a more or less contrived way, all the finalists can be considered as variants of the Generalized Merkle-Damgård (Definition 9). The compression functions of the finalists are either block-cipher based (BLAKE, Skein) or permutation-based (Grøstl, JH, Keccak). Moreover, all finalists use suffix-free padding rules, while the padding rules of BLAKE and Skein are additionally prefix-free [3].

Our formalization models compression functions as functions of two arguments: a message block and a chaining value. This represents a deviation with respect to the compression functions of BLAKE and Skein. The compression function of BLAKE additionally takes a counter and a random salt value, whereas the compression function of Skein builds on a *tweakable* block cipher and takes as additional input a round-specific tweak. The additional arguments of the compression functions of BLAKE and Skein could be formalized as an integral part of the padding rule; the padding function can compute the appropriate round-specific values and append them to the message blocks. This alternative description would have the advantage of matching the model that we use in our results about the MD hash function. However, all finalists except BLAKE use chopping or a final transformation, which are formalized neither in our proof of collision resistance nor in our proof of indifferentiability. This rules out a direct application of our results, with the exception of BLAKE, for which Theorem 6 does apply. We leave it for future work to formalize this instantiation in EasyCrypt.

NIST requirements for the SHA-3 competition include collision resistance, preimage resistance and second preimage resistance. All the candidates selected as finalists satisfy these properties and (in most cases) even achieve optimal bounds for them when the underlying block-ciphers or permutations used to build their compression functions are assumed to be ideal [3]. Although the original NIST requirements did not include the property of indifferentiability from

a random oracle, this notion has also been considered in the literature and is achieved by all five finalists [1], [5], [12], [15], [16], [21]. These indifferentiability proofs hold in an idealized model for some of the building blocks of the hash function: the ideal-cipher model for block-cipher based hash functions, or the ideal-permutation model for permutation-based hash functions. Indifferentiability seems to be an excellent target for security proofs because it ensures that the high-level design of the hash function has no structural weaknesses, but also because it implies bounds for all of the classical properties enumerated above. Unfortunately, the assumption that some underlying primitive is ideal is at best unrealistic and at worst plainly wrong. Proofs of indifferentiability should be taken only as an indication for the security and as a palliative for the lack of security proofs in the standard model.

Compared to our result of Theorem 7, which assumes that the compression function is ideal, the indifferentiability of all the finalists has been proved in an ideal model for lower building blocks. We point out that assuming ideality of a lower building block is weaker than assuming ideality of the entire compression function and thus these results are stronger. Indeed, assuming ideality of the compression function seems to be inappropriate for all the finalists:

- The compression functions of JH and Keccak are trivially non-random, as collisions and preimages can be found in only one query to the underlying permutation [3], [18];
- Finding fixed-points for the compression function of Grøstl is trivial [25];
- The compression function of BLAKE has been recently shown to exhibit non-random behavior [1], [21];
- Non-randomness has been shown for reduced-round versions of Threefish, the underlying block-cipher of Skein [26].

The only two finalists that use a prefix-free padding rule, and for which our proof of indifferentiability can apply, are BLAKE and Skein. However, our proof of indifferentiability of prefix-free Merkle-Damgård relies on the assumption that the underlying compression function behaves like an ideal primitive. Thus, it cannot be applied to BLAKE, as this assumption has been invalidated. As for Skein, the assumption that its compression function is ideal is seriously weakened by the attacks on Threefish mentioned above.

Although Theorem 7 cannot be directly applied to any of the SHA-3 finalists, it constitutes a non-trivial result about the Merkle-Damgård construction and a good starting point for formalizing more complex proofs. Indeed, indifferentiability proofs based on weaker assumptions and general enough to apply to SHA-3 finalists are no significantly different from the proof we have formalized and use essentially the same techniques. We see no impediment to formalizing them in EasyCrypt.

VII. Conclusion

Despite their widespread use, the formal verification of hash functions has received little attention. To our best knowledge, Toma and Borrione [35] were the first to use theorem provers to formally verify properties of SHA-1, but their focus is on functional properties, rather than security properties. The first machine-checked proof of security for a hash design appears in [7], where the authors use the CertiCrypt framework to verify that the construction from Brier et al. [19] yields a hash function indifferentiable from a random oracle into ordinary elliptic curves. More recently, Daubignard et al. [**?**] develop a method to permute dependencies between oracles in a game, and apply their method to prove indifferentiability of hash functions from random oracles. Their method is not implemented, although the underlying framework has been machine-checked [**?**].

The prevailing method for building hash functions is to iterate a compression function on a pre-processed input message. In this paper, we have considered the Merkle-Damgård construction, which pioneered this design, and proved that the resulting hash function preserves collision resistance and is indifferentiable from a random oracle. Our results demonstrate that state-of-the-art verification tools can be used for proving the security of hash designs, and not only for cryptanalysis [32]. We will further this line of research by exploring the formalization of more general security proofs that apply to a wider range of hash functions, including finalists of the SHA-3 competition.

Acknowledgements

The authors want to thank Martín Abadi and the anonymous CSF reviewers for insightful feedback on the paper.

References

[1] E. Andreeva, A. Luykx, and B. Mennink, "Provable security of BLAKE with non-ideal compression function," Cryptology ePrint Archive, Report 2011/620, 2011.

[2] E. Andreeva, B. Mennink, and B. Preneel, "Security reductions of the second round SHA-3 candidates," in *13th International Conference on Information Securiryt, ISC 2010*, ser. Lecture Notes in Computer Science. Heidelberg: Springer, 2011.

[3] E. Andreeva, B. Mennink, B. Preneel, and M. Škrobot, "Security analysis and comparison of the SHA-3 finalists BLAKE, Grøstl, JH, Keccak, and Skein," in *NIST 3rd SHA-3 Conference*, 2012.

[4] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton, "Seven-property-preserving iterated hashing: ROX," in *Advances in Cryptology – ASIACRYPT 2007*, ser. Lecture Notes in Computer Science, no. 4833. Heidelberg: Springer, 2007, pp. 130–146.

[5] E. Andreeva, B. Mennink, and B. Preneel, "On the indifferentiability of the Grøstl hash function," Cryptology ePrint Archive, Report 2010/298, 2010.

[6] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE," December 2010.

[7] G. Barthe, B. Grégoire, S. Heraud, F. Olmedo, and S. Zanella Béguelin, "Verified indifferentiable hashing into elliptic curves," in *1st Conference on Principles of Security and Trust – POST 2012*, ser. Lecture Notes in Computer Science. Heidelberg: Springer, 2012.

[8] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, vol. 6841. Heidelberg: Springer, 2011, pp. 71–90.

[9] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin, "Beyond provable security. Verifiable IND-CCA security of OAEP," in *Topics in Cryptology – CT-RSA 2011*, ser. Lecture Notes in Computer Science, vol. 6558. Heidelberg: Springer, 2011, pp. 180–196.

[10] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2009*. New York: ACM, 2009, pp. 90–101.

[11] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Programming language techniques for cryptographic proofs," in *1st International conference on Interactive Theorem Proving, ITP 2010*, ser. Lecture Notes in Computer Science, vol. 6172. Heidelberg: Springer, 2010, pp. 115–130.

[12] M. Bellare, T. Kohno, S. Lucks, N. Ferguson, B. Schneier, D. Whiting, J. Callas, and J. Walker, "Provable security support for the Skein hash family," April 2009.

[13] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *1st ACM conference on Computer and Communications Security, CCS 1993*. New York: ACM, 1993, pp. 62–73.

[14] G. Bertoni, J. Daemen, M. Peeters, Assche, and G. Van, "The KECCAK reference," January 2011.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the indifferentiability of the sponge construction," in *Advances in Cryptology – EUROCRYPT 2008*, ser. Lecture Notes in Computer Science, vol. 4965. Heidelberg: Springer, 2008, pp. 181–197.

[16] R. Bhattacharyya, A. Mandal, and M. Nandi, "Security analysis of the mode of JH hash function," in *17th International Workshop on Fast Software Encryption, FSE 2010*, ser. Lecture Notes in Computer Science, vol. 6147. Springer, 2010, pp. 168–191.

[17] E. Biham and O. Dunkelman, "A framework for iterative hash functions – HAIFA," Cryptology ePrint Archive, Report 2007/278, 2007.

[18] J. Black, M. Cochran, and T. Shrimpton, "On the impossibility of highly-efficient blockcipher-based hash functions," *Journal of Cryptology*, vol. 22, pp. 311–329, 2009.

[19] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi, "Efficient indifferentiable hashing into ordinary elliptic curves," in *Advances in Cryptology – CRYPTO 2010*, ser. Lecture Notes in Computer Science, vol. 6223. Springer, 2010, pp. 237–254.

[20] R. Canetti, O. Goldreich, and S. Halevi, "The random oracle methodology, revisited," *J. ACM*, vol. 51, no. 4, pp. 557–594, 2004.

[21] D. Chang, M. Nandi, and M. Yung, "Indifferentiability of the hash algorithm BLAKE," Cryptology ePrint Archive, Report 2011/623, 2011.

[22] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function," in *Advances in Cryptology – CRYPTO 2005*, ser. Lecture Notes in Computer Science, vol. 3621. Heidelberg: Springer, 2005, pp. 430–448.

[23] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology – CRYPTO 1989*, ser. Lecture Notes in Computer Science, vol. 435. Heidelberg: Springer, 1990, pp. 416–427.

[24] N. Ferguson, S. Lucks, B. Schneier, D. Whithing, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein hash function family," November 2008.

[25] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," March 2011.

[26] D. Khovratovich, I. Nikolić, and C. Rechberger, "Rotational rebound attacks on reduced Skein," in *Advances in Cryptology – ASIACRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6477. Heidelberg: Springer, 2010, pp. 1–19.

[27] S. Lucks, "A failure-friendly design principle for hash functions," in *Advances in Cryptology – ASIACRYPT 2005*, ser. Lecture Notes in Computer Science, vol. 3788. Heidelberg: Springer, 2005, pp. 474–494.

[28] S. Manuel, "Classification and generation of disturbance vectors for collision attacks against SHA-1," *Designs, Codes and Cryptography*, vol. 59, pp. 247–263, 2011.

[29] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *Journal of Automated Reasoning*, vol. 24, no. 1, pp. 165–203, 2000.

[30] U. Maurer, R. Renner, and C. Holenstein, "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology," in *1st Theory of Cryptography Conference, TCC 2004*, ser. Lecture Notes in Computer Science, vol. 2951. Heidelberg: Springer, 2004, pp. 21–39.

[31] R. Merkle, "One way hash functions and DES," in *Advances in Cryptology – CRYPTO 1989*, ser. Lecture Notes in Computer Science, vol. 435. Heidelberg: Springer, 1990, pp. 428–446.

[32] I. Mironov and L. Zhang, "Applications of SAT solvers to cryptanalysis of hash functions," in *Theory and Applications of Satisfiability Testing - SAT 2006*, ser. Lecture Notes in Computer Science, vol. 4121. Heidelberg: Springer, 2006, pp. 102–115.

[33] T. Ristenpart, H. Shacham, and T. Shrimpton, "Careful with composition: Limitations of the indifferentiability framework," in *Advances in Cryptology – EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, vol. 6632. Heidelberg: Springer, 2011, pp. 487–506.

[34] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *11th International Workshop on Fast Software Encryption, FSE 2004*, ser. Lecture Notes in Computer Science, no. 3017. Heidelberg: Springer, 2004, pp. 371–388.

[35] D. Toma and D. Borrione, "Formal verification of a SHA-1 circuit core using ACL2," in *18th international conference on Theorem Proving in Higher Order Logics, TPHOLs 2005*, ser. Lecture Notes in Computer Science, vol. 3603. Heidelberg: Springer, 2005, pp. 326–341.

[36] X. Wang, Y. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *Advances in Cryptology — CRYPTO 2005*, ser. Lecture Notes in Computer Science, vol. 3621. Heidelberg: Springer, 2005, pp. 17–36.

[37] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Advances in Cryptology — EUROCRYPT 2005*, ser. Lecture Notes in Computer Science, vol. 3494. Heidelberg: Springer, 2005, pp. 561–561.

[38] H. Wu, "The hash function JH," January 2011.