HAL
open science

Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3

José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, Pierre-Yves Strub

To cite this version:

José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, et al.. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. CCS 2019 - 26th ACM Conference on Computer and Communications Security, Nov 2019, London, United Kingdom. pp.1607-1622, 10.1145/3319535.3363211 . hal-02404581

HAL Id: hal-02404581

https://hal.science/hal-02404581v1

Submitted on 11 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Machine-Checked Proofs for Cryptographic Standards

Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3

José Bacelar Almeida
Universidade do Minho
INESC-TEC

Cécile Baritel-Ruet
Université Côte d'Azur
Inria Sophia-Antipolis

Manuel Barbosa
Universidade do Porto
INESC-TEC

Gilles Barthe
MPI-SP
IMDEA Software Institute

François Dupressoir
University of Surrey
University of Bristol

Benjamin Grégoire
Inria Sophia-Antipolis

Vincent Laporte
Inria

Tiago Oliveira
Universidade do Porto
INESC-TEC
FCUP

Alley Stoughton
Boston University

Pierre-Yves Strub
École Polytechnique

ABSTRACT

We present a high-assurance and high-speed implementation of the SHA-3 hash function. Our implementation is written in the Jasmin programming language, and is formally verified for functional correctness, provable security and timing attack resistance in the EasyCrypt proof assistant. Our implementation is the first to achieve simultaneously the four desirable properties (efficiency, correctness, provable security, and side-channel protection) for a non-trivial cryptographic primitive.

Concretely, our mechanized proofs show that: 1) the SHA-3 hash function is indifferentiable from a random oracle, and thus is resistant against collision, first and second preimage attacks; 2) the SHA-3 hash function is correctly implemented by a vectorized x86 implementation. Furthermore, the implementation is provably protected against timing attacks in an idealized model of timing leaks. The proofs include new EasyCrypt libraries of independent interest for programmable random oracles and modular indifferentiability proofs.

KEYWORDS

high-assurance cryptography; EasyCrypt; Jasmin; SHA-3; indifferentiability

ACM Reference Format:
José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3363211

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '19, November 11–15, 2019, London, United Kingdom

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363211

1 INTRODUCTION

A stated goal of recent competitions for cryptographic standards is to gain trust from the broad cryptography community through open and transparent processes. These processes generally involve open-source reference and optimized implementations for performance evaluation, rigorous security analyses for provable security evaluation and, often, informal evaluation of security against side-channel attacks. These artefacts contribute to building trust in candidates, and ultimately in the new standard. However, the disconnect between implementations and security analyses is a major cause for concern. This paper explores how formal approaches could eliminate this disconnect and bring together implementations (most importantly, efficient implementations) and software artefacts, in particular machine-checked proofs, supporting security analyses. We put forward four desirable properties for formal approaches:

- functional correctness: efficient implementations should be proved equivalent to reference implementations and to algorithmic specifications of the standardised cryptographic construction that are both human-readable and interpreted by machines. Such specifications and implementations should be *proved* to have the same input/output behaviour (or interactive behaviour in the case of protocols);
- provable security: rigorous security proofs should be provided both for algorithms and for implementations. For the highest level of assurance, security proofs should be machine-checked and establish guarantees for the (machine-readable) algorithmic specifications. Security for both efficient and reference implementations will follow from the functional correctness proofs, using the baseline adversarial models from provable security;

- side-channel resistance: implementations should be provably secure against side-channel attacks, in relevant ideal models. For instance, it is commonly required that implementations are secure in an abstract model of timing, where implementations leak secrets if they contain secret-dependent memory accesses or control-flow instructions, a notion known as "cryptographic constant-time". Combined with provable security, it entails security in a stronger adversarial model where side-channel leakage is available to the adversary;
- efficiency: formal proofs should remain fully compatible with efficiency considerations. They should neither constrain in any way the code of the implementations (although constrained intermediate implementations could be used as proof artefacts) nor impact its performance.

Contributions. We demonstrate through a relevant use case of the feasibility of formal approaches with respect to the stated goals of functional correctness, provable security, side-channel resistance and efficiency. Our use case is the SHA-3 standard. Our choice is guided by two main considerations. Firstly, the SHA-3 standard will likely be used to protect real-world applications for many years to come. Secondly, its security proof is intricate, and involves techniques that are not routinely addressed in machine-checked security proofs.

Concretely, our implementation is written in Jasmin [1, 5], a framework that targets high-assurance and high-speed implementations using "assembly in the head" (a mixture of high-level and low-level, platform-specific, programming) and a formally verified, predictable, compiler which empowers programmers to write highly efficient fine-tuned code. The generated (verified) x86-64 assembly code matches in performance the best available implementations for this primitive, including for example a current OpenSSL version. Machine-checked proofs of equivalence and provable security are developed in EasyCrypt[1] [11], a proof assistant for cryptographic proofs, using the embedding developed by Almeida et al. [5]. More precisely, as illustrated by Figure 1, we establish:

- functional correctness: the highly efficient implementations are proved functionally equivalent to a readable Jasmin reference implementation of the SHA-3 standard;
- provable security: we prove that the Sponge construction is indifferentiable from a random oracle when the underlying permutation is modelled as a random object—from this result we derive concrete bounds for the standard notions of collision-resistance, and resistance against first- and second-preimage attacks in the random permutation model;
- side-channel resistance: we prove that the implementation only leaks the length of public data, in the abstract model of timing used to reason about "cryptographic constant-time". This property is useful when hash function is integrated into higher-level primitives, say key derivation functions, where hashed inputs are secret.

Our results are established at different levels. Our provable security analysis is based on an EasyCrypt model of the sponge construction, which matches the (bit-oriented) specification in the SHA-3 standard. At this level we adopt the standard approach for cryptographic

[1]https://www.easycrypt.info

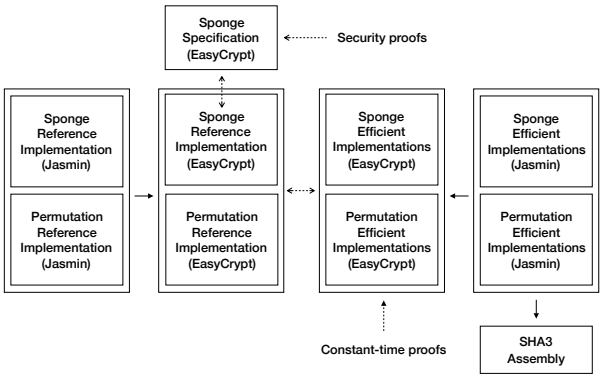

Figure 1: Our results. Full lines represent extraction to EasyCrypt and compilation to assembly by the Jasmin compiler. Dashed lines represent equivalence and security proofs, formalized in EasyCrypt.

proofs of indifferentiability and treat the underlying permutation as an ideal object. In contrast, constant-time security is therefore established as close to the computational platform as possible: our analysis of potential timing side-channels is carried out over highly optimized (byte- and word-oriented) Jasmin implementations of SHA-3.

We then use automatic extraction and equivalence proofs in EasyCrypt to bridge these two levels of results. First, our optimized Jasmin implementations are proved equivalent to a readable reference implementation of the standard, which includes the SHA-3 permutation. Finally, we also prove that the model of the Sponge that we proved theoretically secure is functionally equivalent to the Jasmin reference implementation of this construction, when instantiated with the same permutation. This establishes a link between theoretical security and implementation security.

We note that the proofs of the different properties vary in difficulty. The proof of side-channel resistance is not hard, the proof of functional correctness and indifferentiability are more involved. The latter builds on contributions of independent interest:

- A methodology for proving indifferentiability modularly, which could also be applied in pen-and-paper proofs;
- A new and generic formalization of programmable random oracles that more precisely captures partial adversary knowledge in eager sampling arguments.

Comparison with [1, 5]. Jasmin [1] is an assembly-like language, which gives the developer full control over low-level implementation details, but also provides support for functional verification, and a certified compiler. Almeida et al. [5] extend its toolset to support the verification of local optimizations, by extracting optimized and non-optimized code to EasyCrypt, where they are proved functionally equivalent. The extraction mechanism is also used to prove that the code does not address memory or branch based on secret inputs. In this paper, we further exploit this extraction mechanism, and bring together Jasmin and EasyCrypt to demonstrate

Table 1: Summary of related work

Framework	Functional Correctness	Provable Security	Side-channel resistance	Efficiency
FCF + VST [35]	✓	✓(specific)	✗	✗
EasyCrypt + Frama-C + CompCert [2, 3]	✓	✓	✓	✗
Fiat Cryptography [24]	✓(source-level)	✗	✗(limited)	✓(vs GMP)
HACL* [29, 36]	✓(source-level)	✗	✓(source-level)	✓(vs OpenSSL)
Cryptol + SAW [34]	✓(source-level)	✗	✗	✓
Jasmin [1, 5]	✓	✗	✓	✓
This paper	✓	✓	✓	✓

that a single toolchain that addresses all four desirable properties—including in contexts where provable security is non-trivial—is possible today.

Formally verified proofs of hash functions. Backes et al [8] prove indifferentiability of the Merkle-Damgård construction using EasyCrypt. However, their proof does not connect to verified implementations. Daubignard, Fouque and Lakhnech [22] prove indifferentiability of several hash designs, including the Sponge construction, using the Computational Indistinguishability Logic (CIL) from Barthe et al. [10]. Although CIL has been mechanized in Coq [17], Daubignard, Fouque and Lakhnech's proofs [22] are carried out with pen-and-paper. They obtain bounds similar to ours, albeit slightly less tight.

Appel [7] proves functional correctness of SHA-256, and leverages the CompCert verified compiler [25] to carry guarantees to low-level implementations. However, this work does not consider side-channels or provable security.

Other proofs part of larger projects are discussed below.

Other related work. There is a growing body of work on high-assurance and high-performance cryptographic implementations. We briefly review it against the four desirable properties desirable of such formal approaches as applied to the production of reference implementations for cryptographic standards (see Table 1).

Beringer et al. [13] leverage the Coq-based Foundational Cryptography Framework (FCF) [27] for machine-checked cryptographic proofs and the Coq-based Verified Software Toolchain (VST) [6] for the verification of C programs to formally relate a version of OpenSSL's implementation of HMAC to a machine-readable specification that is proved secure following [12]. Ye et al. [35] extend this result to the OpenSSL implementation of HMAC-DRBG. In both cases, functional correctness and provable security results are obtained on existing C code. The use of a complete toolchain fully-integrated in Coq certainly provides advantages by significantly reducing the trusted computing base, but comes at somewhat of a cost. However, in order to preserve those results through compilation, the CompCert certified compiler [25] must be used, which incurs a significant performance cost. In addition, no guarantees are given—even at source level—regarding side-channels.

Almeida et al. [2] use EasyCrypt to prove security (of RSA-OAEP) on C-like programs with notions of timing leaks. They modify CompCert to strictly enforce preservation of timing behaviour in compiled programs. The same authors later extend and clarify the methodology [3] to separate the concerns of provable security, functional correctness and side-channel security, and use EasyCrypt, Frama-C [19] and CompCert to prove the INT-PTXT security of a compiled executable implementation of TLS 1.2's notorious MAC-then-Encode-then-CBC-Encrypt (TLS-MEE-CBC) against timing-aware attackers. The combination of tools used significantly increases the Trusted Computing Base for the framework, and requires care when transitioning from one tool to the other in the toolchain. However, EasyCrypt provides more flexibility than FCF in proving cryptographic security, and the ability to develop specialized static analyses for timing leaks provides additional guarantees—albeit at a slightly lower level of assurance than that obtained by Ye et al. [35], for example. Since semantics preservation down to compiled code requires the use of CompCert, this approach suffers the same performance issues as Ye et al.'s.

Erbsen et al. [24] propose Fiat Cryptography, a Coq-based framework for developing proofs of functional correctness for implementations of mathematical operations commonly used in cryptography. Their framework produces C code which performs much better than existing libraries, but only guarantees correctness at source level. Further, they do not consider provable security, and cover timing channels by compiling only to straight-line code that uses constant-time arithmetic operations, which limits the kind of primitives their framework can consider.

Zinzindohoué et al. [36] leverage the techniques developed by Protzenko et al. [29] to prove safety and functional correctness of assembly-like programs written in the F* programming language [33]. Their focus is on functional correctness and efficiency, and they obtain partial guarantees on side-channel security through the use of abstract types. Their verified code compiles to C, which is where high-assurance guarantees are given. The code they produce rivals OpenSSL in performance, and has been deployed in Mozilla's NSS library [15]. EverCrypt [28] is a more complete cryptographic library that combines source-level and target-level implementations. As noted by the authors, formal verification in EverCrypt is for now primarily confined to functional correctness and side-channel resistance. Moreover, for source code implementations, there is a trade-off between using verified compilers that carry guarantees to assembly code or off-the-shelf compilers that deliver efficient assembly.

Tomb [34] describes the use of the Cryptol and SAW tools to prove functional correctness of "real-world cryptographic implementations". The approach focuses on establishing functional correctness at source level in a variety of languages, and his focus on existing implementations does guarantee satisfactory performance. However, the approach is not compatible with any known tools

```
SPONGE_c[f, pad, r](m, ℓ)
1 :  m_0 || ... || m_{m-1} ← m || pad(r, |m|);
2 :  / absorption phase
3 :  s_a || s_c ← 0^r || 0^c;
4 :  for i = 0 ... m − 1 do
5 :     s_a || s_c ← f((s_a ⊕ m_i) || s_c);
6 :  / squeezing phase
7 :  Z ← ε; done ← false ;
8 :  while ¬done
9 :     Z ← Z || s_a;
10 :    if |Z| < ℓ
11 :       s_a || s_c ← f(s_a || s_c);
12 :    else
13 :       done ← true ;
14 :  return Z_{|ℓ};
```

Figure 2: Pseudocode for the SPONGE construction [23]

for the production of machine-checked proofs of cryptographic or side-channel security.

Proof and Implementation artefacts

All proof and implementation artefacts are available from https://gitlab.com/easycrypt/sha3. The `README.md` file contained therein further points to the relevant checking extraction and compilation tools and gives light instructions on how to use them to check the proofs and compile the code.

2 TECHNICAL OVERVIEW

The SHA-3 standard [23] defines a family of 4 hash functions and 2 extendable-output functions (XOFs). All functions follow rely on a generic construction, called the SPONGE, that is based on a fixed (unkeyed) permutation. The standard therefore also defines modularly a permutation algorithm—KECCAK-p[1600, 24]—which operates over a 1600-bit-wide state and is defined as an approved function usable in other standards. In the following, we use KECCAK-p as shorthand for this permutation.[2]

In this section we first describe the SPONGE construction and the SHA-3 functions. Then we explain how the SPONGE construction offers very strong security properties, when the underlying permutation is modelled as a purely random object, and why this gives strong heuristic evidence for the security of the SHA-3 functions in real world use. Finally we discuss implementations, their performance and security.

2.1 The SPONGE Construction

Pseudocode for the SPONGE construction is shown in Figure 2. It is parametrised by: i. the permutation f, ii. the padding algorithm pad, and iii. the *rate* (or block size) r. We write c for the construction's *capacity*, defined as the permutation's bitwidth (1600 in the standard) minus r. The construction's internal state $\mathsf{s}_a \parallel \mathsf{s}_c$ has two parts: s_a (r bits) and s_c (c bits). On input of a bitstring m, the SPONGE construction pads it to a multiple of the block size and breaks it into blocks (Line 1). The padding scheme must be injective and length-regular (both properties are necessary in a padding scheme used in secure cryptographic hashing) and must also guarantee that no padded input ends with an all-zero block (which is a necessary condition for the security of the SPONGE). The padded input is then absorbed block-by-block into the SPONGE's internal state (initialized to all `0` bits on Line 3) by interleaving the addition of blocks into the state with applications of the permutation (Lines 4-5). Once all input blocks have been absorbed, the permutation is used, again, to extract the output by blocks of size r (Lines 7-13), truncating the final block to the requested ℓ bits (Line 14).[3]

In our description and formal treatment, we abstract the permutation's bitwidth (set to 1600 by the standard) to some positive integer b, refining it only in the final steps of the proof. Thus $r + c = b$. s_c is the part of the internal state that is not exposed to or controlled by the adversary. For a fixed state width, the capacity serves as the main security parameter for the SPONGE construction, and the rate as its main performance parameter. Therefore, as illustrated in Figure 2, we often use c and r to specify a particular SPONGE construction, rather than b and r.

2.2 SHA-3, hash functions and XOFs

The SHA-3 standard defines SHA3-224, SHA3-256, SHA3-384 and SHA3-512—collectively referred to as SHA3-x in the following, as approved hash functions that accept arbitrary bitstrings as input, and deterministically produce a fixed-length digest (of length x, for SHA3-x). For a fixed output length x, these functions instantiate the SPONGE construction with KECCAK-p, fix $r = 1600 - 2 \cdot x$ and use the *multi-rate padding* scheme $\mathsf{pad10^*1}$ defined as

$$\mathsf{pad10^*1}(r, \ell) := 1 \parallel \mathtt{0}^{(-\ell-2) \mod r} \parallel 1 \,.$$

The $\mathsf{pad10^*1}$ scheme simply appends a string composed of two 1 bits around as many `0` bits as necessary (from 0 to $r - 1$) to the message. It is easy to see that it satisfies the properties required by the SPONGE construction. Formally, the SHA3-x functions are defined as:

$$\text{SHA3-}x(\mathsf{m}) = \\ \text{SPONGE}_{2\cdot x}[\text{KECCAK-}p, \mathsf{pad10^*1}, 1600 - 2 \cdot x](\mathsf{m} \parallel \mathtt{01}, x) \,,$$

where `01` denote two domain separation bits.

The SHA-3 standard also defines two XOFs, SHAKE128 and SHAKE256–collectively referred to as SHAKEx, that accept arbitrary bitstrings as input, and produce a caller-chosen-length prefix of an infinite bitstream deterministically defined by the input. On input a bitstring m and an output length d, the SHAKEx XOFs are defined as

$$\text{SHAKE}x(\mathsf{m}, d) = \\ \text{SPONGE}_{2\cdot x}[\text{KECCAK-}p, \mathsf{pad10^*1}, 1600 - 2 \cdot x](\mathsf{m} \parallel \mathsf{ss} \parallel 11, d) \,,$$

[2] We note that the standard in fact defines a family of permutations, indexed by state size and number of rounds, but only approves KECCAK-p[1600, 24] for use in SHA-3 and other standards. All discussions related to the permutation in this paper focus on KECCAK-p[1600, 24] unless otherwise specified.

[3] Figure 2 is as close to the standard as we could make it with a structured programming language. Our specification differs slightly in that we do not squeeze at all when 0 bits of output are requested. We prove both specifications equivalent.

Game $\text{Real}^{\mathcal{D}}_{c,\text{pad},r}$	Game $\text{Ideal}^{\mathcal{D}}_{\mathcal{S}}$
$b \leftarrow\$ \mathcal{D}^{\text{Sponge}_c[p,\text{pad},r],p_+,p_-}$; return b;	$b \leftarrow\$ \mathcal{D}^{F_\ell, S_+^{F_\ell}, S_-^{F_\ell}}$; return b;

Figure 3: Games defining the indifferentiability of the Sponge construction from an (extendable output) RO.

where ss denote two suffix bits and 11 denote two domain separation bits. The standard—and our implementations—introduce suffix bits for future compatibility with coding schemes for tree hashing variants of the SHAKE*x* functions. They have no effect on security; in fact, our security proof applies for arbitrary application suffixes (of any length fixed in advance).

2.3 Security of the Sponge Construction

The Sponge construction satisfies a strong security notion known as indifferentiability from a (extendable output) random oracle. The notion of indifferentiability, introduced by Maurer, Renner and Holenstein [26] generalizes over the standard notion of indistinguishability by considering settings where the adversary has oracle access to both the construction and its underlying primitive. It has been used as a way of reducing concerns in the design of block ciphers (with proofs for Feistel networks [20, 21] and substitution-permutation networks [16]) and hash functions (with proofs for the Merkle-Damgård construction [18] and the Sponge construction [14]), in each case formally capturing the intuition that the construction does not introduce any structural vulnerabilities when the underlying primitive is seen as an ideal black-box.

Definition 2.1 (Indifferentiability [26]). A construction C with oracle access to an ideal primitive $\mathcal{F}$ is said to be (q_D, q_S, ϵ)-indifferentiable from an ideal functionality $\mathcal{G}$ if there exists a simulator $\mathcal{S}$ with oracle access to $\mathcal{G}$ such that for any distinguisher $\mathcal{D}$ that makes queries of total cost at most q_D, it holds that

$$\left|\Pr\left[\mathcal{D}^{C^{\mathcal{F}},\mathcal{F}} = 1\right] - \Pr\left[\mathcal{D}^{\mathcal{G},\mathcal{S}^{\mathcal{G}}} = 1\right]\right| < \epsilon$$

and that $\mathcal{S}$ makes at most q_S queries to the ideal functionality $\mathcal{G}$.

Throughout the paper, when discussing the *query cost* of an adversary, we consider the number of primitive calls incurred by an adversary's combined queries to the construction and to the primitive itself.

For concreteness, we give the real experiment and ideal experiments in Figure 3 when the notion of indifferentiability is applied to the Sponge construction, as used in the SHA-3 standard and formalized in our proof. In the real game, p is a permutation sampled uniformly at random from the set of all permutations over bit strings of length 1600. The distinguisher is given access to oracles p_+ and p_- that allow it to query the permutation backwards as well as forwards. In the ideal game, the simulator $\mathcal{S} = (S_+, S_-)$ must fake the outputs of the p_+ and p_- oracles, while oblivious of the calls that the distinguisher places to the construction (which is replaced by a random object in the ideal world). We show the simulator as two different algorithms for clarity, but we allow them to share state. The ideal functionality is an extendable output random oracle. This is implemented as an *infinite random oracle* F that associates to each input an infinite (lazy) bitstring, each element of which is sampled uniformly at random. The distinguisher and simulator are restricted to queries to the ideal functionality of the form (m, ℓ), matching the syntax of the Sponge interface; these queries return prefixes of size ℓ of the random oracle F outputs (denoted using F_ℓ notation in the security games). In our formalization we consider a random function $f \in \mathbb{Z}_2{}^* \times \mathbb{N} \to \mathbb{Z}_2$ and construct the observable prefix of length ℓ of the infinite random oracle F as follows:

$$F(\mathsf{m}, \ell) = f(\mathsf{m}, 0) \parallel f(\mathsf{m}, 1) \parallel \ldots \parallel f(\mathsf{m}, \ell - 1)$$

We actually implement f lazily: representing it as a finite map from $\mathbb{Z}_2^* \times \mathbb{N}$ to $\mathbb{Z}_2$ to which we add new input/output pairs as needed.

Our machine-checked proof establishes the following security result for the EasyCrypt specification of the Sponge, which corresponds to the pseudo code described in Figure 2.

Theorem 2.2 (Indifferentiability of Sponge). *The* Sponge *construction is* $(\sigma, \sigma, \frac{\sigma^2-\sigma}{2^{b+1}} + \frac{\sigma^2}{2^{b-r-2}})$*-indifferentiable from an extendable output random oracle for any* $\sigma < 2^{b-r}$*. Namely, the simulator* Simulator *exhibited in Figure 8 makes at most* σ *queries when the adversary makes queries of total cost at most* σ*.*

$$\left|\Pr\left[\text{Real}^{\mathcal{D}}_{b-r,\text{pad},r} = 1\right] - \Pr\left[\text{Ideal}^{\mathcal{D}}_{\mathcal{S}} = 1\right]\right| \leq \frac{\sigma^2}{2^{b-r-2}} + \frac{\sigma^2 - \sigma}{2^{b+1}}$$

Both simulator and bound are very similar to the original ones given by Bertoni et al. [14].

For our simulator, we use a simplified version of Bertoni et al.'s simulator which, unlike theirs, puts no work into maintaining a permutation. This simplifies the formal handling of the proof, but also yields a slightly higher bound than Bertoni et al.'s $\frac{\sigma^2+\sigma}{2^{b-r+1}} - \frac{\sigma^2-\sigma}{2^{b+1}}$ This is due to our simulator producing a distribution that is further away from that of a truly random permutation, and to the use of the PRP-PRF switching lemma. Beyond this, the main difference is that our formalization of the simulator keeps track of sequences of queries that mimic the behaviour of the Sponge construction as *paths* to specific capacities. By contrast, Bertoni et al.'s reconstructs such paths each time a query is received, saving memory at the cost of computation time. Since security is information-theoretic, this has no effect on the security claim.

We note that Bertoni et al.'s bound is only claimed to hold when σ is "significantly smaller than" the capacity $c = b - r$, a situation in which the difference between our bound and theirs is dwarfed by the bound itself. Further, we believe our bound could be slightly improved through a more precise handling of failure events when bounding their probability (see Section 3.3).

2.4 Security Implications

Indifferentiability implies a strong form of composition for single-stage security games, that is security experiments where the attacker can keep unrestricted state [30], which includes standard definitions for collision-, preimage-, and second preimage-resistance. Composition implies in particular that any single-staged security property that can be established for the ideal functionality is satisfied by the indifferentiable construction, with the caveat that

security holds in an idealized computational model where the underlying primitive is a truly random permutation.

We prove Theorems 2.3, 2.4 and 2.5, which state that the SHA3-x hash functions have all desired security properties in the random permutation model. Similarly, all SHAKEx functions inherit from the underlying Sponge its indifferentiability from an infinite random oracle, but we do not give the details here.

Collision Resistance under generic attacks. Intuitively, a hash function is collision-resistant if it is unfeasible for a probabilistic polynomial-time adversary, given the ability to compute digests for arbitrary inputs, to find two distinct inputs that produce the same digest. The SHA3-x hash functions are collision-resistant in the random permutation model.

Theorem 2.3 (Collision Resistance for SHA3-x). *For all adversaries $\mathcal{A}$ with oracle access to both a truly random permutation P and the SHA3-x function instantiated with P, and making queries of total cost at most σ, the probability that $\mathcal{A}$ finds a collision is bounded as follows:*

$$\Pr\left[\mathsf{CR}^{\mathcal{A}}_{SHA3\text{-}x} = 1\right] \leq \frac{\sigma^2 - \sigma}{2^{1600+1}} + \frac{\sigma^2}{2^{2\cdot x-2}} + \frac{\sigma^2 - \sigma + 2}{2^{x-1}}$$

Preimage Resistance under generic attacks. Intuitively, a hash function is preimage-resistant if it is infeasible for a probabilistic polynomial time adversary to find a preimage to a given digest, even when given the ability to compute digests for arbitrary inputs. The SHA3-x hash functions are preimage-resistant in the random permutation model.

Theorem 2.4 (Preimage Resistance for SHA3-x). *For all adversaries $\mathcal{A}$ with oracle access to both a truly random permutation P and the SHA3-x function instantiated with P, and queries of total cost at most σ, the probability that $\mathcal{A}$ finds a preimage for any fixed digest is bounded as follows:*

$$\Pr\left[\mathsf{PR1}^{\mathcal{A}}_{SHA3\text{-}x} = 1\right] \leq \frac{\sigma^2 - \sigma}{2^{1600+1}} + \frac{\sigma^2}{2^{2\cdot x-2}} + \frac{\sigma + 1}{2^{x}}$$

Second Preimage Resistance under generic attacks. Intuitively, a hash function is second-preimage-resistant if it is unfeasible, for all possible inputs, for a probabilistic polynomial time adversary given the ability to compute digests for arbitrary inputs, to find a distinct input that produces the same digest. The SHA3-x hash functions are second-preimage-resistant in the random permutation model.

Theorem 2.5 (Second Preimage Resistance for SHA3-x). *For all adversaries $\mathcal{A}$ with oracle access to both a truly random permutation P and the SHA3-x function instantiated with P, and making queries of total cost at most σ, the probability that $\mathcal{A}$ finds a second preimage of arbitrary length for any fixed input m is bounded as follows:*

$$\Pr\left[\mathsf{PR2}^{\mathcal{A}}_{SHA3\text{-}x}(\mathsf{m}) = 1\right] \leq \frac{\sigma^2 - \sigma}{2^{1600+1}} + \frac{\sigma^2}{2^{2\cdot x-2}} + \frac{\sigma + 1}{2^{x}}$$

2.5 Secure and efficient implementations

Reference implementations with varying degrees of readability and efficiency are a crucial part of modern cryptographic standards, and they are very useful side results that illustrate the advantages of the open competition-based process that underlies the selection of new algorithms. In this section we discuss how one can ensure that these reference implementations are correct and secure to the highest level of assurance, using formal verification technology.

Implementation security. In the previous sections we have discussed the theoretical security of the SHA-3 functions in an idealized model of computation where the underlying permutation is replaced by a purely random one. These results do not carry directly to practice, as implementations rely on a fixed permutation. Nevertheless, they provide (heuristic) confidence on the security of the SHA-3 specification.

From a theoretical point of view, this has been discussed (for example) by Rogaway and Shrimpton [31], who have defined notions of security for practical hash functions where all parameters are fixed called *always* preimage resistance and second-preimage resistance. Intuitively, these notions state that the standardised algorithm behaves like a one-way function and that finding second preimages for hashes of high-entropy messages is hard, even though the parameters are common to all applications. This level of security is not directly implied by provable security, and so it is a security assumption on the SHA-3 specification. For collision resistance, the usual assumption is that the algorithm which outputs collisions (which is known to exist) is hard to find.

This raises the question of what it means for a SHA-3 implementation to be secure. In this paper we follow the approach of [3] whereby the security of implementations is defined as a set of sufficient conditions that transfer the (potentially assumed) security properties of high-level specifications to executable code; furthermore, checking the correct deployment of countermeasures against timing attacks, one gets the guarantee that security holds even against implementation attackers that can get precise measurements of the implementation's execution time.

Reference implementation. In Figure 4 we show the entry point for the reference implementation we have constructed in Jasmin as a direct transcription of the standard. The full reference implementation is omitted due to space constraints, but included as supplementary material. In this implementation the emphasis is on readability: we see it as a machine-readable and interpretable incarnation of the SHA-3 standard, when restricted to byte-aligned messages, which is easy to check for compliance by inspection. Note that the reference implementation is general enough to allow instantiations that exactly match all the SHA-3 hash functions and XOFs: the trailing byte abstracts all the possible combinations of domain separation bits and application specific suffixes in the standard.

We have connected this implementation to our theoretical security results, by showing that, assuming the same permutation algorithm, this reference implementation is functionally equivalent to the generic construction that was analysed in the ideal permutation model.[4] This result ensures that whatever (heuristic) security guarantees follow from the theoretical security proofs on the specification will apply to the reference implementation.[5] Furthermore,

[4] Our equivalence proof is restricted to inputs that have a size multiple of 8 bits, since the reference implementation works over sequences of bytes, whereas as our theoretical analysis does not impose this restriction.

[5] Recall that provable security holds in the ideal permutation model, and hence the established security properties can only be assumed to still hold once we fix the

```
fn keccak_1600(
  reg u64 out, // output pointer
  reg u64 outlen, // output length in bytes
  reg u64 in, // input pointer
  reg u64 inlen, // input length in bytes
  stack u8 trail_byte,
  stack u64 rate
) {
  stack u64[25] state;

  state = st0();

  while ( inlen >= rate )
  {
    state = add_full_block(state, in, rate);
    state = keccak_f1600(state);
    inlen = inlen - rate;
    in = in + rate;
  }

  state = add_final_block(state, in, inlen, trail_byte, rate);

  while ( outlen > rate )
  {
    state = keccak_f1600(state);
    xtr_full_block(state, out, rate);
    outlen = outlen - rate;
    out = out + rate;
  }

  state = keccak_f1600(state);
  xtr_bytes(state, out, outlen);
}
```

Figure 4: Reference Implementation

it also guarantees that our security analysis indeed applies to the SHA-3 standard as transcribed in our reference implementation.

Efficient implementations. In Section 5 we present a formally verified library of highly efficient implementations of the SHA-3 functions for x86-64. These implementations follow the state of the art in optimizing SHA-3 in 64-bit architectures, both with and without support for vectorized instructions. In both cases our code essentially matches the best non-verified implementations. However, we prove that the assembly code for these implementations is both functionally correct with respect to our reference implementation, and that its execution time does not depend on user input data (no branching or memory access dependencies on secret data). These results were achieved by applying Jasmin's extraction methodology [5], integrating it—for the first time—with a theoretical security analysis.

In comparison with previous equivalence proofs carried out over Jasmin implementations, we encountered new challenges, as the SHA-3 specification deals with lists of bits, whereas the implementation (and particularly the representation of values in memory) is word-oriented. Conversely, in the SHA-3 case the abstraction gap associated with big number arithmetic, that was faced and dealt with in prior work [5] is not present.

permutation. Our equivalence proof guarantees that whatever security properties are retained by the specification once the permutation is fixed will also hold for the reference implementation.

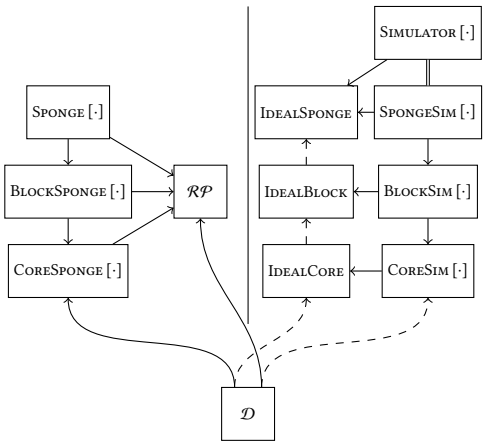

Figure 5: A layered proof for the Sponge construction.

As discussed above, applying the theoretical framework of [3], our results imply that the optimized assembly code inherits the security assumed for the reference implementation without any loss, even in the presence of timing attacks in a model where the adversary is able to observe full execution traces containing all code-memory and data-memory addresses accesses performed by the implementation.

We discuss all proofs carried out over the implementations in Section 4.

3 MACHINE-CHECKED SECURITY PROOF

We now discuss in more detail the machine-checked provable security proofs, which we write in EasyCrypt. After a brief overview of our proof strategy, we give some background on EasyCrypt—introducing some useful notations, before detailing some of the proof's more important aspects.

3.1 Proof Outline

Our formalization of definitions and top-level statements is direct: we simply express the Sponge, its ideal functionality, the simulator and the indifferentiability result in EasyCrypt, and express the upper bound on any distinguisher's probability of differentiating as is standard, specialized to a two-oracle primitive interface.

In order to carry out the proof, we layer it as shown in Figure 5 to account for individual aspects of the construction. Our layers separately deal with truncation and padding (Sponge), and squeezing (BlockSponge) over a simplified CoreSponge which outputs only a single block. We discuss the decomposition methodology in Section 3.3. The simulator constructed by layers is not optimal in terms of query cost, and we then show its equivalence to the top-level simulator, allowing us to conclude with a tighter concrete security result.

In addition to layering the proof to separate concerns, we also greatly generalize known formal results on random oracles to allow switching painlessly from eagerly sampled random oracle to lazily sampled random oracles (and back) in a greater variety of cases.

This generalization, discussed in Section 3.5, also contributes to placing this proof, and others, within reach of formalization.

Beyond the proof of indifferentiability for the bare Sponge construction, we then show that the functions defined in the SHA-3 standard [23] inherit specialized version of this indifferentiability, and formally establish concrete security bounds on an adversary's ability to produce collisions, preimages or second preimages on the SHA-3 hash functions without first breaking the Keccak-p permutation. At the highest level, our indifferentiability proof is instantiated to the hash functions SHA3-224, SHA3-256, SHA3-384 and SHA3-512, and extendable output functions SHAKE128 and SHAKE256 *as they are defined in the SHA-3 standard* [23]. We discuss these proofs, which discharge Theorems 2.3, 2.4 and 2.5, in Section 3.6.

3.2 Background on EasyCrypt

EasyCrypt [11] is an interactive proof assistant, which also embeds a simple probabilistic programming language, pWhile, used to model cryptographic primitives, schemes, oracles and experiments, as well as program logics for bounding the probability of events in programs, and for proving equivalences or approximate equivalences between programs. Although EasyCrypt was initially designed to capture Shoup's code-based game-based proof methodology [32], it has since successfully been applied to simulation-based proofs, and to a growing body of standard symmetric primitives [3, 8, 9]. Both of these make it a suitable candidate for this formalization effort.

The pWhile language is a simple imperative programming language with assignments, conditionals (**if-then-else**) and **while** loops, as well as the ability to sample a value—stored in a variable, say x—from an arbitrary (sub-)distribution d, denoted $x \leftarrow\$ d$. We sometimes abuse notation and write $x \leftarrow\$ X$ with X a set, to denote uniform sampling in X. The language of expressions, including distributions, can be user-extended using a simple polymorphic higher-order functional programming language. A rich module system can be used to describe global memories and control access to them (which captures groups of oracles that should share some global state), to define programs that rely on some external functionalities (modelling oracle access), and to universally quantify over all possible programs that implement a set of interfaces (which captures adversaries). In the following, we use different notations for the same mechanism that parameter sizes a module by other modules, in order to clarify the intended semantics. In particular, we denote with $C[P]$ a *modular construction*,[6] where a construction C relies on a primitive (or lower-level construction) P for its functionality, and with $\mathcal{A}^O$ the standard notion of *oracle access* whereby a program $\mathcal{A}$ (often an adversary) is given access to a set of oracles O (whose security it is usually attempting to break).

As mentioned, program logics in EasyCrypt can be used to prove three kinds of statements over programs (or pairs of programs). We use only two in this paper, although the third (approximate equivalence between two programs) is used extensively in the proofs:

(1) perfect equivalence between two programs c_1 and c_2, denoted $c_1 \sim c_2$, indicates that if c_1 and c_2 are run on the same initial memory configuration, then they either both diverge, or both terminate with final memory configurations that follow the same distribution;
(2) bounds or equality on the probability for an event E to occur during execution of a program c, denoted with $\Pr[c : E]$.

[6] We also use $C \circ P$ for the same when trying to emphasize a *decomposition* instead.

3.3 Decomposing Indifferentiability proofs

The decomposition outlined in Figure 5 is made possible by the following general observation. Suppose we have a stateless "upper-level" construction $\mathcal{C}[\mathcal{RP}]$ that we want to prove to be indifferentiable from an upper-level ideal functionality $\mathcal{J}$. Furthermore, let us assume that we already know that a stateless "lower-level" construction $\mathcal{E}[\mathcal{RP}]$ is indifferentiable from a lower-level ideal functionality $\mathcal{I}$, where $\mathcal{S}$ is a lower-level simulator such that no adversary can effectively distinguish between $(\mathcal{E}[\mathcal{RP}], \mathcal{RP})$ and $(\mathcal{I}, \mathcal{S}[\mathcal{I}])$.

We construct a pair of stateless converters $\mathcal{D}$ and $\mathcal{U}$ that works as follows: $\mathcal{D}$ ("down") transforms an upper-level functionality into a lower-level one; and $\mathcal{U}$ ("up") transforms a lower-level functionality into a upper-level one. We define the upper-level simulator $\mathcal{T}$ such that $\mathcal{T}[\mathcal{J}] := \mathcal{S}[\mathcal{D}[\mathcal{J}]]$. And, for any upper-level adversary $\mathcal{A}$ that is asked to differentiate $\mathcal{C}$ from $\mathcal{J}$, let the lower-level adversary $\mathcal{B}[\mathcal{A}]$ be defined as $\mathcal{B}[\mathcal{A}]^{X,Y} := \mathcal{A}^{\mathcal{U}[X],Y}$.

Then, to prove that $\mathcal{C}[\mathcal{RP}]$ is indifferentiable from $\mathcal{J}$, it will suffice to show the following two equivalences.

$$\mathcal{A}^{\mathcal{C}[\mathcal{RP}],\mathcal{RP}} \sim \mathcal{B}[\mathcal{A}]^{\mathcal{E}[\mathcal{RP}],\mathcal{RP}} \quad (1)$$

$$\mathcal{A}^{\mathcal{J},\mathcal{T}[\mathcal{J}]} \sim \mathcal{B}[\mathcal{A}]^{\mathcal{I},\mathcal{S}[\mathcal{I}]} \quad (2)$$

Equivalence (1) relates the "real" games, and simply reflects that the modular construction is correct. Equivalence (2), on the other hand, pertains to the "ideal" games. Since $\mathcal{E}$ is indifferentiable from $\mathcal{I}$ for all adversaries, this holds in particular for $\mathcal{B}[\mathcal{A}]$.

Because $\mathcal{C}$ and $\mathcal{E}$ (and $\mathcal{U}$) are stateless, it is clear both what $\mathcal{C}[\mathcal{RP}] \sim \mathcal{U}[\mathcal{E}[\mathcal{RP}]]$ should mean, and that it will be sufficient to imply that Equivalence (1) holds.

However the situation is more complex for the ideal equivalence (2), since our ideal functionalities have persistent local state (say, query maps) of *different* types. Consequently it is unclear what the statements $\mathcal{J} \sim \mathcal{U}[\mathcal{I}]$ and $\mathcal{I} \sim \mathcal{D}[\mathcal{J}]$ would even mean, and we must instead prove finer-grained equivalence statements where equivalence—over the whole game—also defines how the ideal functionalities' states are related.

We now describe the fine-grained ideal equivalences and their proofs for the two derived layers (Sponge in Section 3.3.1, and BlockSponge in Section 3.3.2), also giving an intuition of the core simulator and proof for CoreSponge in Section 3.3.3.

3.3.1 Sponge. The BlockSponge construction is similar to Sponge, but works on blocks rather than bits, forgoing padding of inputs and truncation of outputs. The IdealBlock ideal functionality is like the infinite random oracle IdealSponge, except that it, too, works on blocks rather than bits. Both the construction and ideal functionality should only be called with lists of blocks that can be successfully unpadded; when called with invalid arguments, they return the empty list.

We prove that, if BlockSponge is indifferentiable from IdealBlock, then Sponge is indifferentiable from IdealSponge by instantiating the generic argument discussed in Section 3.3. In this

instance, $\mathcal{C}$ is Sponge, $\mathcal{E}$ is BlockSponge, $\mathcal{J}$ is IdealSponge, $\mathcal{I}$ is IdealBlock, and $\mathcal{S}$ is the block sponge simulator. We construct the transformers $\mathcal{D}$ and $\mathcal{U}$ as follows. $\mathcal{D}[\mathcal{J}']$ takes in a list of blocks and a requested length n; it returns the empty list if given an input that is not correctly padded. Otherwise, it calls $\mathcal{J}'$ with the unpadded input and $n * r$, and then chunks the resulting bitstring into n blocks. $\mathcal{U}[\mathcal{I}']$ takes in a list of bits and a requested length n. It calls $\mathcal{I}'$ on the padding of its input and $\lceil n/r \rceil$, and then truncates the result of turning the resulting blocks into a list of bits.

For the real equivalence $\mathcal{A}^{\mathcal{C}[\mathcal{RP}],\mathcal{RP}} \sim \mathcal{B}[\mathcal{A}]^{\mathcal{E}[\mathcal{RP}],\mathcal{RP}}$, we simply prove that the construction and its modularly-constructed version are equivalent, as $\mathcal{C}[\mathcal{RP}] \sim \mathcal{U}[\mathcal{E}[\mathcal{RP}]]$. This involves inlining and code rewriting, noting that $\mathcal{U}$ simply truncates and pads exactly as the Sponge does.

In contrast, and as discussed in Section 3.3, the proof of the ideal equivalence $\mathcal{A}^{\mathcal{J},\mathcal{T}[\mathcal{J}]} \sim \mathcal{B}[\mathcal{A}]^{\mathcal{I},\mathcal{S}[\mathcal{I}]}$ is more complex. We carry it out in three steps, involving *hybrid infinite random oracles* (*hybrid IROs*), which are midway between $\mathcal{J}$ and $\mathcal{I}$. An input to a hybrid IRO is a well-padded list of *blocks* and a desired number of output *bits*. Internally, they work with finite maps from $(\mathbb{Z}_2^r)^* \times \mathbb{N}$ to $\mathbb{Z}_2$. A hybrid IRO can be raised, for comparison with $\mathcal{J}$, or lowered, for comparison with $\mathcal{I}$. Two hybrid IROs are defined: a *lazy* one, and an *eager* one. The lazy one consults/updates just enough inputs of its finite map to provide the requested bits, whereas the eager one continues up to a block boundary, consulting/updating subsequent inputs of the finite map, as if it had been asked for a multiple of r bits.

The first step of the proof transitions from $\mathcal{A}^{\mathcal{J},\mathcal{T}[\mathcal{J}]}$ to a game involving the lazy IRO. This is done by employing a relational invariant between the maps of $\mathcal{J}$ and the lazy IRO.

The second step of the proof uses the eager sampling facility of Section 3.5 to transition from a game involving the lazy IRO to one involving the eager IRO. The bridge between these games uses the eager sampling theory's `sample` oracle to sample the extra bits needed to take one up to a block boundary. The lazy version of `sample` then gives us the lazy IRO, whereas its eager version gives us the eager IRO.

The third step of the proof takes us from the game involving the eager IRO to $\mathcal{B}[\mathcal{A}]^{\mathcal{I},\mathcal{S}[\mathcal{I}]}$. This is done by employing an invariant between the maps of the eager IRO and $\mathcal{I}$. The proof is rather involved, and makes use of: (a) EasyCrypt's library's support for showing the equivalence of randomly choosing a block versus forming a block out of r randomly chosen bits; and (b) a mathematical induction over a pRHL judgement.

3.3.2 BlockSponge. The next step in our proof is to show that squeezing, the operation through which the Sponge's output is extended to any desired length, also preserves indifferentiability. Consider a functionality CoreSponge that computes only the absorption stage of BlockSponge (lines 3-5 of Figure 2, taking as input a list of blocks, and outputting the final value of s_a).

We define Squeeze as the construction layer that builds BlockSponge from CoreSponge as follows. Given a list of blocks m corresponding to a padded bitstring, and a desired output length i (in blocks), Squeeze$[\mathcal{F}](\mathsf{m}, i)$ iteratively calls $\mathcal{F}$ times, with inputs $(\mathsf{m} \,\|\, 0^{r \cdot j})_{0 \le j < i}$, each call producing a single block of output. An example is shown in Figure 6.

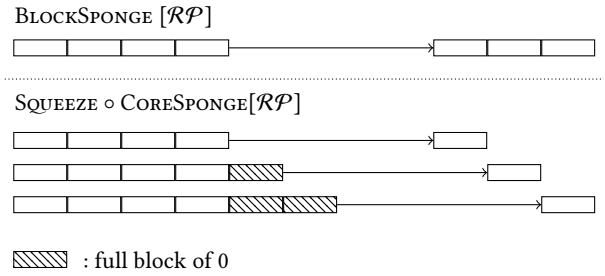

Figure 6: BlockSponge [$\mathcal{RP}$] and Squeeze∘CoreSponge[$\mathcal{RP}$] on inputs that are a list of 4 blocks and output size 3.

Since the primitive $\mathcal{RP}$ appears deterministic (and in particular returns the same output when queried twice on the same input) it is easy to prove in EasyCrypt that the real equivalence Squeeze ∘ CoreSponge[$\mathcal{RP}$] ~ BlockSponge[$\mathcal{RP}$] holds.

To define the BlockSim simulator for BlockSponge from an assumed simulator for CoreSponge, we first need to simulate the IdealCore functionality—a simple random oracle in $(\{0,1\}^r)^* \to \{0,1\}^r$—from IdealBlock. We thus define a construction transformer, called Last, which parses its input m—a non-empty list of blocks—to produce a pair $(y, j) \in (\{0,1\}^r)^* \times \mathbb{N}$ such that $m = y \,\|\, (0^r)^j$ and y does not end with a full block of 0s, calls IdealBlock with input $(y, j+1)$ to obtain a list of blocks, finally outputting only the last block. As before, the ideal equivalence here must be expressed and proved over the whole game, and indeed requires relating the states of ideal functionalities and simulator based on queries done on both interfaces. We thus prove in EasyCrypt that, for all adversaries $\mathcal{A}$, $\mathcal{A}^{\text{Squeeze[IdealCore]},\text{CoreSim[IdealCore]}}$ is equivalent to $\mathcal{A}^{\text{IdealBlock},\text{CoreSim[Last}\circ\text{IdealBlock]}}$. Once again, this proof leverages the generic lazy-eager sampling theory described in Section 3.5 to prove that the intermediate blocks sampled and thrown away by Last[IdealBlock] can instead not be sampled at all.

On the query cost. In this particular application of the decomposition, the cost in number of permutation queries differs greatly between BlockSponge[$\mathcal{RP}$] and Squeeze∘CoreSponge[$\mathcal{RP}$]. Considering the example shown in Figure 6, the cost of that query to BlockSponge is 6 permutations, but the same query to Squeeze ∘ CoreSponge costs $4+5+6 = 15$ permutation calls. Simply applying the decomposition as described would yield a bound dominated by $O\left(\sigma^4/2^r\right)$. We therefore also use this transfer result to refine the way in which the cost of a query in CoreSponge is measured, to avoid double-counting common prefixes.

3.3.3 CoreSponge. All proof steps discussed thus far involve transferring indifferentiability from a simple construction to a more complex one. We now focus on the core of the Sponge construction, a block-oriented construction that, on input a list of blocks, produces a single block of output. As discussed above, our notion of query cost on this CoreSponge construction differs from the top-level notion, and is defined instead as the length of the query without its longest common prefix with any of the previous queries. We call this cost the *prefix cost* of a query. Considering this notion of cost requires us to carefully design the CoreSponge construction to

ensure that the prefix cost and query cost of the same query align when transferring indifferentiability to BlockSponge. In particular, CoreSponge memoizes all query prefixes it has already seen and associates each of them to its final absorption state. We prove in EasyCrypt that CoreSponge is indifferentiable from IdealCore.

As a first simplifying step, we replace the primitive, a random permutation, with a random function. This simplifies some of the formal reasoning—in particular by removing internal dependencies between samplings in the random permutation—but introduces an additional term $\frac{\sigma^2-\sigma}{2^{b+1}}$ in the bound. We note that all results discussed above transfer indifferentiability *without loss*. Therefore, improving the bound for this simpler functionality would be sufficient in improving the bound for the whole Sponge construction at almost no formal cost beyond that of tightening the bound for CoreSponge.

The idea behind the simulator. Indifferentiability requires us to simulate answers to the permutation, in a way that is consistent with what the adversary may have already observed of the ideal functionality, without knowing which queries the distinguisher has made—or will make—to the functionality. To do so, the simulator CoreSim, shown in Figure 7, keeps track of *paths*—which are sequences of blocks that, when fed through CoreSponge, leave its state with a particular value of the capacity—and uses the functionality to simulate its answer to any query that makes use of a capacity to which a path is known, that is, a query that extends a known path through CoreSponge. Queries that are disjoint from path are answered as if by the ideal random function.

When the simulation fails. The simulation fails (and can be distinguished from the true permutation) if either one of the following events occurs:

bcol : A capacity that was previously seen as output of the permutation with a rate s_a has been output again, associated with a different block s'_a.

bext : The adversary has queried the primitive or its inverse with a capacity that has already been, or is later sampled internally by CoreSponge, but to which the simulator does not know a path.

We show in EasyCrypt that these are the only conditions under which the distinguisher can indeed distinguish the construction from the ideal functionality.

Bounding the probability of a simulation failure. It remains to bound the probability that these bad events occur. bcol is a straightforward collision event, whose probability we can bound immediately, since it occurs as values are sampled.

On the other hand, bounding the probability that bext occurs in any particular run is much more complex, as it requires identifying that the adversary has guessed a value that has been sampled in the past, or will be sampled later on in the run. We note, however, that the event is only triggered when the adversary guesses a value that is independent from his view of the system: indeed CoreSponge keeps all capacities internal, and the event does not consider the case where the adversary has obtained the capacity's value through a legitimate sequence of calls to the permutation that mimics CoreSponge's operations. The trick is therefore to

```
CoreSim[F](x1, x2)
if (x1, x2) ∉ m {
    y2 ←$ Z_2^{b−r};
    if x2 ∈ paths {
        (p, v) ← paths[x2];
        y1 ← F (p ||(v ⊕ x1)) ;
        paths[y2] ← (p ||(v ⊕ x1), y1);
    } else {
        y1 ←$ Z_2^r;
    }
    m[(x1, x2)] ← (y1, y2);
    mi[(y1, y2)] ← (x1, x2);
}
return m[(x1, x2)];
```

```
CoreSim[F]^{−1}(x1, x2)
if (x1, x2) ∉ mi {
    y1 ←$ Z_2^r;
    y2 ←$ Z_2^{b−r};
    mi[(x1, x2)] ← (y1, y2);
    m[(y1, y2)] ← (x1, x2);
}
return mi[(x1, x2)];
```

Figure 7: The core simulator CoreSim

delay the sampling of capacities that are used in CoreSponge until the end of the game (at which point we can sample them all, and easily bound the probability that any one of them was used by the adversary), or until the simulator constructs a path to it, at which point bext is no longer triggered.

In order to deploy the lazy-eager sampling theory described in Section 3.5, however, we must first remove the dependencies between capacities and rates introduced by their joint use in the permutation.

A proof trick: indirection. To do so, we deploy an indirection technique similar to that used by Backes et al. [8] in proving indifferentiability of Merkle-Damgård. First, each fresh permutation query (made by the adversary or the functionality) is tagged with its sequence number. The main permutation map is used to keep track, given an input state, of the rate that was returned, and of the sequence number. An auxiliary map is used to translate sequence numbers into capacities. This auxiliary map can then be sampled lazily: on direct permutation queries, both rate and capacities are sampled and returned to the distinguisher; on permutation queries made by the construction, the rate is sampled and associated with a sequence number, but the capacity is not sampled. A loop after the distinguisher has finished running samples all remaining capacities, that is, exactly those that have been used in the construction, but not been observed as part of a path, triggering the bad event *a posteriori* if any one of them collides with an adversarial input that is not part of a path. This last transformation makes heavy use of the lazy-eager sampling theory described in Section 3.5, including in particular the use of programming queries in some cases.

3.4 The Sponge simulator

As described in Figure 5, the simulator resulting from the proof described above is constructed from the modular layers as:

$$\text{SpongeSim}\,[\text{BlockSim}\,[\text{CoreSim}\,[\cdot]]]$$

The final step of our proof is to collapse the layered construction into a final simulator, shown in Figure 8. This allows us to reduce

```
SIMULATOR[F](x1, x2)
1 : if (x1, x2) ∉ m {
2 :   y2 ←$ Z_2^{b−r};
3 :   if x2 ∈ paths {
4 :     (p, v) ← paths[x2];
5 :     (m, k) ← parse(p||(v ⊕ x1));
6 :     if unpad(m) ≠ ⊥ {
7 :       lb ← F(unpad(m), k · r);
8 :       y1 ← last(bits2blocks(lb));
9 :     } else if 0 < k {
10 :      if (m, k − 1) ∉ invalid {
11 :        invalid[m, k − 1] ←$ Z_2^r;
12 :      }
13 :      y1 ← invalid[m, k − 1];
14 :    } else {
15 :      y1 ← 0^r;
16 :    }
17 :    paths[y2] ← (p||x1 ⊕ v, y1);
18 :  } else {
19 :    y1 ←$ Z_2^r;
20 :  }
21 :  m[x1, x2] ← (y1, y2);
22 :  mi[y1, y2] ← (x1, x2);
23 : }
24 : return m[(x1, x2)];
```

```
SIMULATOR[F]^{-1}(x1, x2)
if (x1, x2) ∉ mi {
  y1 ←$ Z_2^r;
  y2 ←$ Z_2^{b−r};
  mi[x1, x2] ← (y1, y2);
  m[y1, y2] ← (x1, x2);
}
return mi[(x1, x2)]
```

Figure 8: The optimized SIMULATOR **for** SPONGE

the cost of simulator queries, aligning them with the announced notion of query cost, and to present the simulator as a single algorithm. However, the layered nature of the simulator can still be seen in this final presentation: the core simulator still appears, keeping track of paths through the simulated permutation that could correspond to functionality calls and extending them as required, or simply simulating the random permutation with a random function. However, the way in which paths are extended in the layered simulator (lines 5-16 in Figure 8 replacing the single line 5 in Figure 7) reflects the different layers required to turn a path through the core sponge (a well-padded bitstring followed by a number of 0^r blocks) into a valid query to the SPONGE (a bitstring that the SPONGE itself pads and a number of desired output bits), also turning the output of the SPONGE (a list of bits of the requested length) into the expected output for the core simulator (a single block). For top-level simulator queries that would yield invalid queries to any intermediate functionalities (for example because they correspond to paths that do not reflect well-padded inputs), the layered simulator simply simulates an independent random function.

In other words, our final simulator is CORESIM where the ideal functionality IDEALCORE itself is further simulated from the top-level IDEALSPONGE ideal functionality.

3.5 Eager Sampling in the Programmable ROM

As described above, all three layers of our decomposition—including its core—rely on an *eager sampling* argument for random oracles. Eager sampling is a standard argument when working with random oracles in cryptographic proofs, which consists in switching between two different views of a random oracle: i. an *eager* view, in which the random oracle is sampled at random in a distribution over the space of functions (of the appropriate type) when the game is initialized; and ii. a *lazy* view, in which the random oracle's responses are individually sampled upon fresh requests. Although it is clear that no adversary that can only query the random oracle can distinguish between these two views, this restriction is often much too strict for the argument to be applicable directly.

For example, the second step in the proof that padding and truncation preserve indifferentiability, discussed in Section 3.3.1 boils down to proving that the oracles E and L shown on the outside of Figure 9 cannot be distinguished—when instantiated with a random oracle $\mathcal{RF}$ with domain $D \times \mathbb{N}$ and range $\mathbb{Z}_2$—by any algorithm (even unbounded) with oracle access to one of them. Note here that it is impossible to simply eagerly sample the random oracle $\mathcal{RF}$, since its domain is countably infinite, preventing us from defining a uniform distribution over the function space.

Rather than forcing the EasyCrypt user to use low-level tactics to reason about this equivalence, we define a new abstraction for random oracles that supports eager arguments, not only in situations like the one of Figure 9—where definitional difficulties arise from the use of infinite domains, but also in the presence of programming queries, such as scenarios in which answers to some queries are deterministically `set` or `removed` by the context. Our new abstraction models a random oracle with input in set D, output in C, and output distribution d_C as 4 oracles that share state, and whose canonical eager and lazy implementations are shown in Figure 10. They differ only in their `sample` oracle: an eager implementation (shown on a grey background), which samples values even though they are not returned; and a lazy implementation, which does nothing.

Even given this much extended interface (over the "traditional" interface which only exposes `get`), we can prove the following lemma, which states that the eager and lazy implementations are strictly equivalent.

LEMMA 3.1 (EAGER SAMPLING FOR PROGRAMMABLE RANDOM ORACLES). *For any map $m_0 \in D \to C$, and for any unbounded adversary $\mathcal{D}$ with unbounded oracle access to* `get`, `set`, `rem` *and* `sample` *oracles as specified above, we have*

$$\mathcal{D}^{\mathsf{Eager}_{m_0}} \sim \mathcal{D}^{\mathsf{Lazy}_{m_0}}$$

Proving this lemma makes heavy use of EasyCrypt's advanced `eager` tactic, formalizing and confirming the standard intuition that, even when oracles can be externally programmed, sampling operations whose results do not influence the adversary's view can safely be delayed, either until the point where they do influence the adversary's view, or until the end of the game's execution.

Continuing our example, the extended interface can be used as shown in Figure 9 to define a hybrid oracle H^O that uses O.`sample` as well as O.`get` for $O \in \{\mathsf{Eager}, \mathsf{Lazy}\}$. Then E and H^{Eager} are equivalent, since their second loops do nothing with the values they sample: both oracles sample $\lceil n/r \rceil \cdot r$ bits, returning the first

```
E(x ∈ D, n ∈ ℕ)
z ← ε;
for (i = 0; i < n; i++)
  z ← z || RF.get(x, i);
for (i = n; i < ⌈n/r⌉ · r; i++)
  RF.get(x, i);
return z;
```

```
H^O(x ∈ D, n ∈ ℕ)
z ← ε;
for (i = 0; i < n; i++)
  z ← z || O.get(x, i);
for (i = n; i < ⌈n/r⌉ · r; i++)
  O.sample(x, i);
return z;
```

```
L(x ∈ D, n ∈ ℕ)
z ← ε;
for (i = 0; i < n; i++)
  z ← z || RF.get(x, i);
return z;
```

Figure 9: Eager-lazy random sampling example.

```
Eager_{m0}   Lazy_{m0}
Global state: a map m ∈ D → C, initially m_0.

Common oracles
get(x)                  set(x, y)
if (x ∉ m)              m[x] ← y;
  m[x] ←$ d_C;          return;
return m[x];
                        rem(x)
                        m[x] ← ⊥;
                        return;

sample(x)
if (x ∉ m)
  m[x] ←$ d_C;
return;

sample(x)
return;
```

Figure 10: The eager and lazy programmable random oracle

n of them. Also H^{Lazy} is equivalent to L, since the second loop in H^{Lazy} does nothing. Finally, if $\mathcal{D}$ is a distinguisher for E/L, then by Lemma 3.1, we have that $\mathcal{D}^E \sim \mathcal{D}^{H^{\mathsf{Eager}}} \sim \mathcal{D}^{H^{\mathsf{Lazy}}} \sim \mathcal{D}^L$, considering the distinguisher $\mathcal{D}^{H^{\cdot}}$ when applying Lemma 3.1.

3.6 Completing the Proofs

It now remains to close the proofs of Theorems 2.3, 2.4 and 2.5 (and of corresponding theorems, not stated explicitly here, regarding the SHAKEx functions).

As demonstrated by Maurer, Renner and Holenstein [26], indifferentiability naturally extends the intuitive consequences of indistinguishability to adversarial settings where multiple components must be simulated at the same time. In particular, they show that security properties that can be expressed as single-stage games are preserved by indifferentiability, and Theorems 2.3, 2.4 and 2.5 are thus simple corollaries of Theorem 2.2. We formalize this argument—specialized to the notion of indifferentiability from an infinite random oracle—and leverage it in the proofs of our top-level security theorems. We note, however, that Maurer, Renner and Holenstein's results do not cover multi-stage games, for which a stronger notion of indifferentiability—not known to be met by the Sponge construction—is required [30].

3.6.1 Security Definitions and Proofs. We now formally define the security notions referred to in Section 2, and discuss the choices where they are not obvious. Formally, our top-level results on the SHA3-x functions are all expressed in the ideal permutation model, where the adversary always has oracle access to the permutation, and we model the SHA3-x functions as families of hash functions indexed by their permutation.

Collision Resistance. We use the textbook notion of collision resistance. Formally, the game $\mathsf{CR}_H^{\mathcal{A}}$ used in Theorem 2.3 is displayed in Figure 11.

Preimage Resistance. For preimage resistance, we use Rogaway and Shrimpton's *everywhere preimage resistance* [31]. This is a compromise, which allows us to express and prove the absence of *generic* attacks against the SHA3-x functions, while also being stronger than the standard notion of preimage resistance. Formally, the game $\mathsf{PR1}_H^{\mathcal{A}}$ used in Theorem 2.4 is displayed in Figure 12.

Second Preimage Resistance. Finally, for second preimage resistance, we generalize and strengthen Rogaway and Shrimpton's *everywhere second preimage resistance* to remove its parametrization by the challenge length: the advantage of an adversary in breaking our notion of second preimage is the maximum over all possible input m of any length of the probability the adversary finds a second preimage for that input. We formally express the game $\mathsf{PR2}_H^{\mathcal{A}}$ used in Theorem 2.5 in Figure 13.

3.6.2 About Joint Security. Our proofs do not attempt to capture the joint security of SHA3-x and SHAKEx, even with compatible rates. This is due to the strategy we followed to complete the top-level proofs, whereby we proved that fixing the Sponge construction's output length produces a hash function that is collision, preimage and second preimage resistant *before* considering the domain separation bits, which do not affect the security of a hash function. We leave the consideration of domain separation as future work, with proofs carried out modularly following the methodology presented here, and do not foresee any particular difficulty.

Obtaining a proof of security—even informal—for a multi-rate Sponge construction, whereby the same permutation is used to construct several XOFs, remains an open problem of independent interest. We do not tackle it here, and do note that our results therefore do not cover the SHA-3 standard as a whole, but rather each of its components in isolation.

4 MACHINE-CHECKED CORRECTNESS

Jasmin [1] is a language for high-assurance and high-speed cryptography, Jasmin implementations are *predictably* transformed into assembly programs by the Jasmin compiler. Indeed, the Jasmin

$\mathsf{CR}_H^{\mathcal{A}}()$

$P \leftarrow_\$ \mathsf{Perm};$

$(\mathsf{m}, \mathsf{m}') \leftarrow_\$ \mathcal{A}^{H[P],P}();$

return $\mathsf{m} \neq \mathsf{m}' \wedge$

$H[P](\mathsf{m}) = H[P](\mathsf{m}');$

Figure 11: Collision resistance.

$\mathsf{PR1}_H^{\mathcal{A}}()$

$P \leftarrow_\$ \mathsf{Perm};$

$y \leftarrow_\$ C;$

$\mathsf{m} \leftarrow_\$ \mathcal{A}^{H[P],P}(y);$

return $H[P](\mathsf{m}) = y;$

Figure 12: Preimage resistance.

$\mathsf{PR2}_H^{\mathcal{A}}(\mathsf{m})$

$P \leftarrow_\$ \mathsf{Perm};$

$\mathsf{m}' \leftarrow_\$ \mathcal{A}^{H[P],P}(\mathsf{m});$

return $\mathsf{m} \neq \mathsf{m}' \wedge$

$H[P](\mathsf{m}') = H[P](\mathsf{m});$

Figure 13: Second preimage resistance.

language is designed to support "assembly in the head" programming, i.e., it smoothly combines high-level (structured control-flow, variables, etc.) and low-level (assembly instructions, flag manipulation, etc.) constructs. Predictability empowers Jasmin programmers to develop optimized implementations with essentially the same level of control as if they were using assembly or domain-specific languages such as qhasm.

Recently [5], the Jasmin framework has been extended with a formal verification back-end based on EasyCrypt. This back-end makes it possible to reason about the correctness of Jasmin implementations by automatically extracting them into EasyCrypt programs whose semantics is formally defined by an embedding of the Jasmin semantics in the EasyCrypt logic. The EasyCrypt proof assistant supports program logics for reasoning about correctness and equivalence of imperative programs. The approach proposed in [5] takes advantage of this relational reasoning to carry out functional correctness proofs in the "game-hopping" style: one starts from a reference implementation, which is proved correct with respect to a high-level specification, and gradually modifies it until equivalence to high-speed code is trivial to prove. Crucially, this approach mimics the optimization process adopted by cryptographers when writing high-speed code, which means that the intermediate implementations required to bridge reference and target code are side-effects of the optimization procedure itself.[7] The Jasmin compiler is certified via a proof formalized in the Coq proof assistant, thus guarantees are carried to assembly code.

In this paper we build on the work of [5] to show that the embedding into EasyCrypt opens a way for a new type of formal guarantee: in addition to correctness of high-speed implementations, we can connect high-speed implementations to provable security results for the high-level specifications.

Equivalence proofs. In Figure 14 we show the general structure of our correctness proofs. At the top we show the standard connected by arrows that represent correctness by inspection to three artefacts: i. the Sponge specification that was proved indifferentiable from a random oracle in EasyCrypt; ii. the reference implementation of the Sponge construction; and iii. the reference implementation of Keccak-*p*. The latter two implementations are shown as rounded rectangles to denote Jasmin as the implementation language. At the bottom of the figure are the high-speed implementations. Here x denotes either a scalar implementation or a vectorized AVX2 implementation. Both were proved correct with respect to the reference implementation using a common strategy detailed below.

[7] There is a rich body of work in equivalence checking that we do not discuss in this paper, as our contributions are not centred on the verification technology itself. See [5] for a detailed discussion.

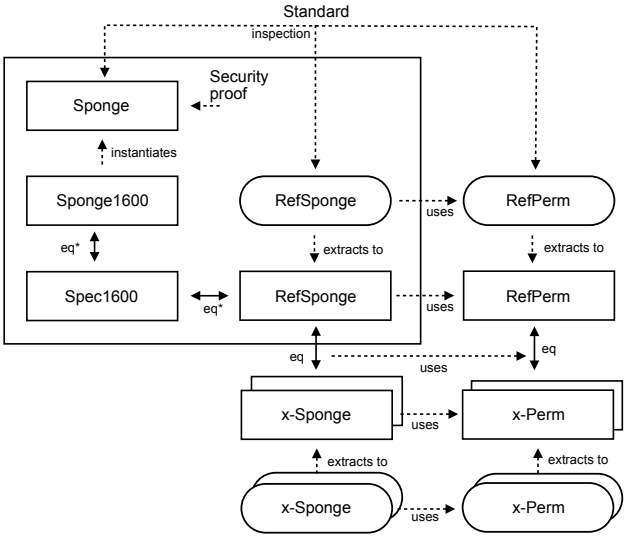

Figure 14: Equivalence proofs

The optimizations used in these implementations are well-known and follow the general approach adopted in OpenSSL, which in turn builds on observations by the proposers of Keccak.[8] The main difference here is that they can be expressed in Jasmin, which permits reasoning about their correctness at a convenient level of abstraction.

Reference implementation correctness. The reference implementation for the Sponge construction in Jasmin is proved equivalent to the provably secure EasyCrypt specification using several steps shown within a large rectangle. In this rectangle, there are no permutation implementations or specifications, as the theoretical security of the Sponge idealizes this component. For this reason, equivalence proofs are denoted eq*, to indicate that they are performed under the assumption that the sponge constructions are instantiated with the same permutation. The Jasmin reference implementation is extracted to EasyCrypt, and linked to the Sponge specification using two intermediate steps: i. Sponge1600 is simply an instantiation of the provably secure construction for a concrete permutation size of 1600-bits; and ii. Spec1600 is an implementation of the Sponge that uses the semantics of Jasmin to define the construction over sequences of bytes, for input messages that are byte-aligned. The main challenges in these proofs relate to both

[8] https://github.com/XKCP/XKCP

data representation (from bit-level, to byte-level and ultimately to word-level manipulation) and to modifications in the control-flow that are needed to deal with lazy memory reading and writing (read when needed, write when ready in order to avoid buffering) and corner cases that arise in the padding stage.

High-speed implementation correctness. The correctness of the high-speed implementations is established modularly. First, the underlying implementation of the permutation is proved equivalent to the reference implementation of Keccak-*p*. Then, the full implementation is proved correct using the permutation equivalence as a stepping stone. All equivalence proofs are carried out over EasyCrypt code extracted from the Jasmin implementations.

The permutation code does not perform memory write operations, as the entire state is maintained in the stack and registers. This greatly simplifies proofs, as stack arrays are treated in the Jasmin semantics as applicative arrays. The main challenges at this level are therefore related with the semantics of low-level optimizations, including vectorization, instruction selection, spilling, and instruction scheduling.

Conversely, the main challenges in proving the full implementations are related to memory accesses. Concretely, some of the permutation implementations rely on memory tables to keep global constants, which means that one must prove that the memory write operations performed by the squeeze stage do not overwrite these memory regions. In the absorb phase, the main difficulty arises in the proof of the vectorized implementation, where the behaviour is significantly different from the reference implementation. In short, the state of the permutation is kept in a (redundant) representation using seven 256-bit registers, where the logical arrangement of 64-bit subwords is optimized for taking advantage of SIMD permutation operations. This means that loading a message block and XOR-ing it with the state is a semantically challenging operation: rather than performing a XOR over only part of the state, the implementation first constructs a dummy message block with the correct word arrangement (padded with zeros up to full permutation input size) in the stack, and then XORs the entire state. Proving equivalence to the reference implementation therefore cannot be proved by matching intermediate state values on a per-instruction basis, and involves complex invariants over the state of the memory and the stack.

Safety verification and side channels. The extended Jasmin framework of [5] also includes a mechanism for the automatic verification of safety—safety is a necessary condition for the soundness of the embedding of Jasmin in EasyCrypt—and another mechanism to check for the absence of timing side-channels. We have checked our implementations for both safety and absence of timing leaks in the constant-time model (input and output lengths are assumed to be public). The constant-time proofs consist of extracting the optimized Jasmin implementations to EasyCrypt using the mechanism introduced in [5], and then proving that the leakage trace explicitly constructed by the extracted code is independent from (or, formally, non-interfering with) the application data processed by the hash function. The leakage trace constructed by the implementation exactly maps the formal definition of constant-time leakage, as defined for example in [4].

We emphasize that, as proved in [3], checking for the constant-time property at the Jasmin level, guarantees that the timing leakage of the implementation can be simulated based solely on public information, i.e., it is independent of user inputs. Together with functional correctness given by our equivalence proofs, this means that whatever security properties are guaranteed by the reference implementations, these are retained by the optimized implementations, even in the presence of timing attackers. Eliminating this type of leakage is critical for applications where hash functions process secret data and the hash function input (albeit not its length) needs to be hidden, as is the case in keyed constructions such as HMAC, or in key derivation functions.

5 PERFORMANCE EVALUATION

In this section we demonstrate the efficiency of our Jasmin implementation of SHA-3. We use SHAKE256 for concreteness, but note that for all the other SHA-3 functions the comparison to other libraries yields similar results, as they all share the same base implementations of the Sponge construction and Keccak-*p*.

Benchmarking Environment. The performance evaluation of the Jasmin implementations of SHAKE256 was performed using SUPERCOP, v. 20190110. All measurements were taken on an Intel i7-6500U (Skylake) processor clocked at 2.5GHz, with Turbo Boost disabled, Ubuntu 16.04, kernel release 4.15.0-46-generic and GCC 8.1.

We compare our code with OpenSSL[9] and HACL*[10]. We configured three different versions of OpenSSL: i. with the no-asm flag to exclude all assembly implementations; ii. without any flags or changes so that the x86_64 assembly implementation of Keccak-*p* was selected; and iii. with an edited version of configuration file crypto/sha/build.info to force the usage of the AVX2 implementation of Keccak-*p*. The resulting three shared libraries were then used in different runs of the supercop benchmarks to produce the presented results. The corresponding binding to the supercop API for hash functions, crypto_hash was already defined. For the HACL* evaluation we produced a static library from the extracted implementations that are present in the directory dist/compact-gcc by using the Makefile provided by the authors. The binding to supercop is a simple function call to the implementation: Hacl_SHA3_shake256_hacl. As a final note, for SHAKE256 supercop defines CRYPTO_BYTES, the output length, as 136 bytes. As such, the presented results correspond to executions of SHAKE256 for that specific output length.

Results. We show our results in Figures 15 and 16. The chart in Figure 15 shows a comparison of our implementations with OpenSSL. We show vectorized (AVX2) and non-vectorized (scalar) implementations, as well as the C implementation in OpenSSL. It is clear from the chart that there are significant advantages in moving from C to assembly, and further performance boosts from vectorization. All of these were known. The novelty in this chart is that we are comparing *verified* Jasmin code to non-verified implementations, and that the verified code matches the performance of the non-verified code. For small message sizes our implementation is faster mostly due to overheads in the library bindings; these overheads lose significance

[9] Branch: OpenSSL_1_1_1-stable; Commit: 4f4d37dacec205066b369b93aa5bacb0553f68d1
[10] Branch: evercrypt-v0.1+; Commit: bc1b759fe5f471c827d2b6d292e530e36a6d3a67

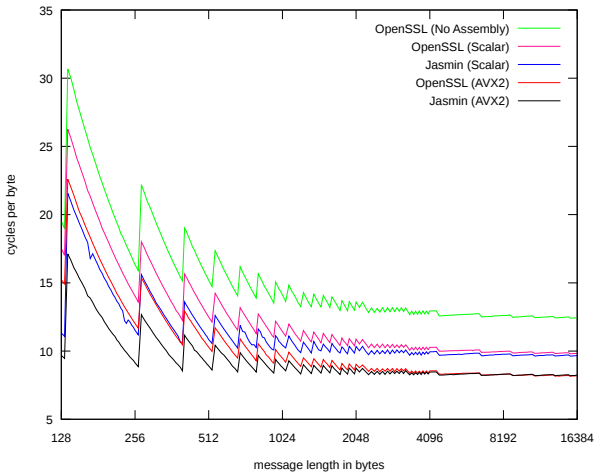

Figure 15: Comparison to non-verified code.

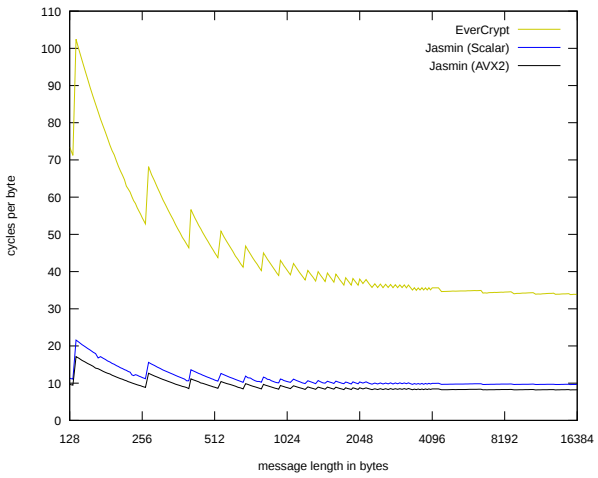

Figure 16: Comparison to verified code.

for large messages (note we report cycles per byte) and one can see that there is an improvement for the scalar implementation due to fine-tuning of the Jasmin implementation, and essentially the same performance for the vectorized implementation.

In Figure 16 we give a comparison to the verified C implementation in EverCrypt, which contains the only formally verified implementation of SHA-3 that we could find. Note that the benchmarked assembly is only high-assurance if one trusts the compiler that was used to produce it. Even so, the advantages of the Jasmin approach are clear: by allowing formal reasoning about correctness at the same level of abstraction as over C code and, at the same time, giving assembly-like control to the programmer, it was possible to improve performance by a factor of 3.

6 CONCLUSION

In this paper, we give concrete evidence that it is feasible to certify standards, from provable security of algorithms to provable and side-channel security of both human-readable and high-performance reference implementations. Although we give this evidence on an already-established standard, we believe that the tools used to produce proofs of functional correctness and implementation security for implementations are accessible enough to be used in ongoing competitions.

Tools and techniques for the formalization of provable security, on the other hand, still require both intense effort and a deep understanding of the mathematical arguments involved in the proof. The process of gaining such understanding in this case has allowed us to develop two contributions of independent interest: i. a new methodology—that we believe could be applied in pen-and-paper proofs—for decomposing proofs of indifferentiability for complex constructions; and ii. an extension of the traditional formal view of random oracles to better support reasoning about patching queries.

ACKNOWLEDGMENTS

This work received support from the National Institute of Standards and Technologies under agreement number 60NANB15D248.

This work was partially supported by Office of Naval Research under projects N00014-12-1-0914, N00014-15-1-2750 and N00014-19-1-2292.

This work was partially funded by national funds via the Portuguese Foundation for Science and Technology (FCT) in the context of project PTDC/CCI-INF/31698/2017. Manuel Barbosa was supported by grant SFRH/BSAB/143018/2018 awarded by the FCT.

This work was supported in part by the National Science Foundation under grant number 1801564.

This work was supported in part by the FutureTPM project of the Horizon 2020 Framework Programme of the European Union, under GA number 779391.

This work was supported by the ANR Scrypt project, grant number ANR-18-CE25-0014.

This work was supported by the ANR TECAP project, grant number ANR-17-CE39-0004-01.

REFERENCES

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1807–1823. https://doi.org/10.1145/3133956.3134078

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2013. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 1217–1230. https://doi.org/10.1145/2508859.2516652

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption – FSE 2016 (Lecture Notes in Computer Science)*, Thomas Peyrin (Ed.), Vol. 9783. Springer, Heidelberg, 163–184. https://doi.org/10.1007/978-3-662-52993-5_9

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security 2016: 25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 53–70.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2019. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. *CoRR* abs/1904.04606 (2019). arXiv:1904.04606 http://arxiv.org/abs/1904.04606

[6] Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. 2. https://doi.org/10.1007/978-3-642-28891-3_2

[7] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. https://doi.org/10.1145/2701415

[8] Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella-Béguelin. 2012. Verified Security of Merkle-Damgård. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. 354–368. https://doi.org/10.1109/CSF.2012.14

[9] Cecile Baritel-Ruet, François Dupressoir, Pierre-Alain Fouque, and Benjamin Grégoire. 2018. Formal Security Proof of CMAC and Its Variants. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. 91–104. https://doi.org/10.1109/CSF.2018.00014

[10] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. 2010. Computational indistinguishability logic. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 375–386. https://doi.org/10.1145/1866307.1866350

[11] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[12] Mihir Bellare. 2006. New Proofs for NMAC and HMAC: Security without Collision-Resistance. In *Advances in Cryptology – CRYPTO 2006 (Lecture Notes in Computer Science)*, Cynthia Dwork (Ed.), Vol. 4117. Springer, Heidelberg, 602–619. https://doi.org/10.1007/11818175_36

[13] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[14] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. 2008. On the Indifferentiability of the Sponge Construction. In *Advances in Cryptology – EUROCRYPT 2008 (Lecture Notes in Computer Science)*, Nigel P. Smart (Ed.), Vol. 4965. Springer, Heidelberg, 181–197. https://doi.org/10.1007/978-3-540-78967-3_11

[15] Benjamin Beurdouche, Franziskus Kiefer, and Tim Taubert. 2017. Verified cryptography for Firefox 57. https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/ Accessed May 14, 2019.

[16] Benoît Cogliati, Yevgeniy Dodis, Jonathan Katz, Jooyoung Lee, John P. Steinberger, Aishwarya Thiruvengadam, and Zhe Zhang. 2018. Provable Security of (Tweakable) Block Ciphers Based on Substitution-Permutation Networks. In *Advances in Cryptology – CRYPTO 2018, Part I (Lecture Notes in Computer Science)*, Hovav Shacham and Alexandra Boldyreva (Eds.), Vol. 10991. Springer, Heidelberg, 722–753. https://doi.org/10.1007/978-3-319-96884-1_24

[17] Pierre Corbineau, Mathilde Duclos, and Yassine Lakhnech. 2011. Certified Security Proofs of Cryptographic Protocols in the Computational Model: An Application to Intrusion Resilience. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.), Vol. 7086. Springer, 378–393. https://doi.org/10.1007/978-3-642-25379-9_27

[18] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. 2005. Merkle-Damgård Revisited: How to Construct a Hash Function. In *Advances in Cryptology – CRYPTO 2005 (Lecture Notes in Computer Science)*, Victor Shoup (Ed.), Vol. 3621. Springer, Heidelberg, 430–448. https://doi.org/10.1007/11535218_26

[19] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*. 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[20] Dana Dachman-Soled, Jonathan Katz, and Aishwarya Thiruvengadam. 2016. 10-Round Feistel is Indifferentiable from an Ideal Cipher. In *Advances in Cryptology – EUROCRYPT 2016, Part II (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9666. Springer, Heidelberg, 649–678. https://doi.org/10.1007/978-3-662-49896-5_23

[21] Yuanxi Dai and John P. Steinberger. 2016. Indifferentiability of 8-Round Feistel Networks. In *Advances in Cryptology – CRYPTO 2016, Part I (Lecture Notes in Computer Science)*, Matthew Robshaw and Jonathan Katz (Eds.), Vol. 9814. Springer, Heidelberg, 95–120. https://doi.org/10.1007/978-3-662-53018-4_4

[22] Marion Daubignard, Pierre-Alain Fouque, and Yassine Lakhnech. 2012. Generic Indifferentiability Proofs of Hash Designs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, Stephen Chong (Ed.). IEEE Computer Society, 340–353. https://doi.org/10.1109/CSF.2012.13

[23] Morris J. Dworkin. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. http://dx.doi.org/10.6028/NIST.FIPS.202

[24] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Security & Privacy*. To appear. Retrieved from http://adam.chlipala.net/papers/FiatCryptoSP19/FiatCryptoSP19.pdf.

[25] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[26] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. 2004. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *TCC 2004: 1st Theory of Cryptography Conference (Lecture Notes in Computer Science)*, Moni Naor (Ed.), Vol. 2951. Springer, Heidelberg, 21–39. https://doi.org/10.1007/978-3-540-24638-1_2

[27] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[28] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. 2019. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. Cryptology ePrint Archive, Report 2019/757. https://eprint.iacr.org/2019/757.

[29] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F*. *PACMPL* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[30] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. 2011. Careful with Composition: Limitations of the Indifferentiability Framework. In *Advances in Cryptology – EUROCRYPT 2011 (Lecture Notes in Computer Science)*, Kenneth G. Paterson (Ed.), Vol. 6632. Springer, Heidelberg, 487–506. https://doi.org/10.1007/978-3-642-20465-4_27

[31] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *Fast Software Encryption – FSE 2004 (Lecture Notes in Computer Science)*, Bimal K. Roy and Willi Meier (Eds.), Vol. 3017. Springer, Heidelberg, 371–388. https://doi.org/10.1007/978-3-540-25937-4_24

[32] Victor Shoup. 2004. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332. http://eprint.iacr.org/2004/332.

[33] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[34] Aaron Tomb. 2016. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security & Privacy* 14, 6 (2016), 26–33. https://doi.org/10.1109/MSP.2016.125

[35] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 2007–2020. https://doi.org/10.1145/3133956.3133974

[36] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1789–1806. https://doi.org/10.1145/3133956.3134043