Verified Correctness and Security of OpenSSL HMAC

Lennart Beringer, *Princeton University;* Adam Petcher, *Harvard University and MIT Lincoln Laboratory;* Katherine Q. Ye and Andrew W. Appel, *Princeton University*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Open access to the Proceedings of
the 24th USENIX Security Symposium
is sponsored by USENIX

Verified correctness and security of OpenSSL HMAC

Lennart Beringer
Princeton Univ.

Adam Petcher
Harvard Univ. and MIT Lincoln Laboratory

Katherine Q. Ye
Princeton Univ.

Andrew W. Appel
Princeton Univ.

Abstract

We have proved, with machine-checked proofs in Coq, that an OpenSSL implementation of HMAC with SHA-256 correctly implements its FIPS functional specification *and* that its functional specification guarantees the expected cryptographic properties. This is the first machine-checked cryptographic proof that combines a source-program implementation proof, a compiler-correctness proof, and a cryptographic-security proof, with no gaps at the specification interfaces.

The verification was done using three systems within the Coq proof assistant: the Foundational Cryptography Framework, to verify crypto properties of functional specs; the Verified Software Toolchain, to verify C programs w.r.t. functional specs; and CompCert, for verified compilation of C to assembly language.

1 Introduction

HMAC is a cryptographic authentication algorithm, the "Keyed-Hash Message Authentication Code," widely used in conjunction with the SHA-256 cryptographic hashing primitive. The sender and receiver of a message m share a secret session key k. The sender computes $s = \mathrm{HMAC}(k,m)$ and appends s to m. The receiver computes $s' = \mathrm{HMAC}(k,m)$ and verifies that $s' = s$. In principle, a third party will not know k and thus cannot compute s. Therefore, the receiver can infer that message m really originated with the sender.

What could go wrong?

Algorithmic/cryptographic problems. The compression function underlying SHA might fail to have the cryptographic property of being a pseudorandom function (PRF); the SHA algorithm might not be the right construction over its compression function; the HMAC algorithm might fail to have the cryptographic property of being a PRF; we might even be considering the wrong crypto properties.

Implementation problems. The SHA program (in C) might incorrectly implement the SHA algorithm; the HMAC program might incorrectly implement the HMAC algorithm; the programs might be correct but permit side channels such as power analysis, timing analysis, or fault injection.

Specification mismatch. The specification of HMAC or SHA used in the cryptographic-properties [15] proof might be subtly different from the one published as the specification of computer programs [28, 27]. The proofs about C programs might interpret the semantics of the C language differently from the C compiler.

Based on Bellare and Rogaway's probabilistic game framework [16] for cryptographic proofs, Halevi [30] advocates creating an "automated tool to help us with the mundane parts of writing and checking common arguments in [game-based] proofs." Barthe *et al.* [13] present such a tool in the form of CertiCrypt, a framework that "enables the machine-checked construction and verification" of proofs using the same game-based techniques, written in code. Barthe *et al.*'s more recent EasyCrypt system [12] is a more lightweight, user-friendly version (but not *foundational,* i.e., the implementation is not proved sound in any machine-checked general-purpose logic). In this paper we use the Foundational Cryptography Framework (FCF) of Petcher and Morrisett [38].

But the automated tools envisioned by Halevi—and built by Barthe *et al.* and Petcher—address only the "algorithmic/cryptographic problems." We also need machine-checked tools for functional correctness of C programs—not just static analysis tools that verify the absence of buffer overruns. And we need the functional-correctness tools to connect, with machine-checked proofs of equivalence, to the crypto-algorithm proofs. By 2015, proof systems for formally reasoning about crypto algorithms and C programs have come far enough that it is now possible to do this.

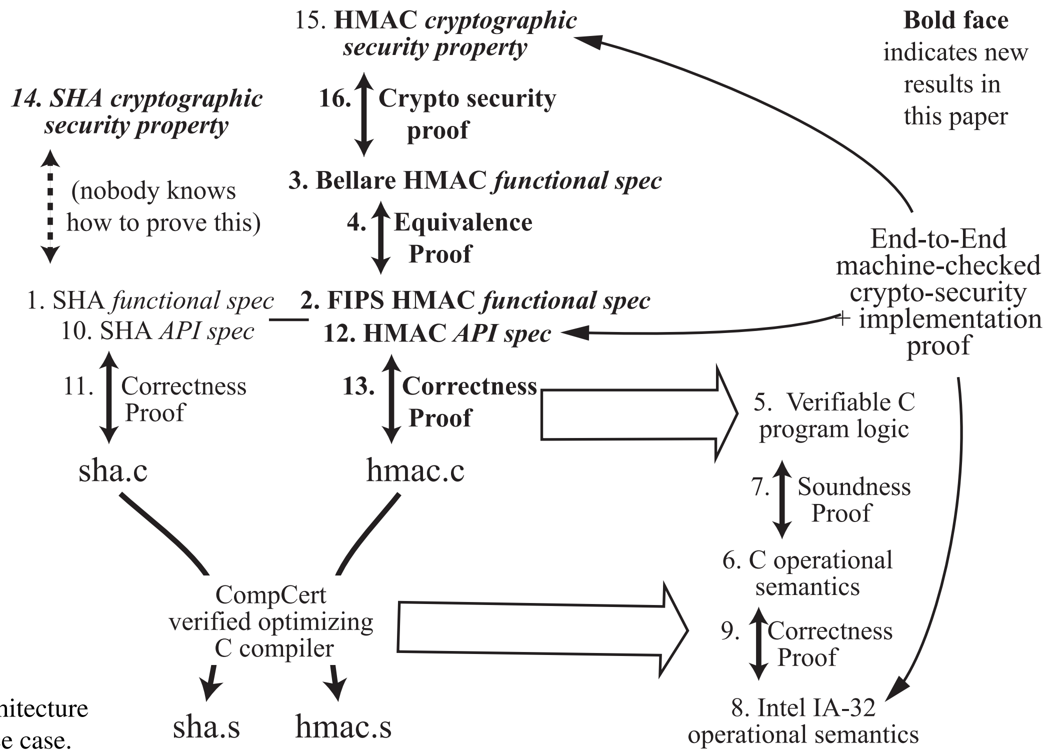

Figure 1: Architecture of our assurance case.

Here we present machine-checked proofs, in Coq, of many components, connected and checked at their specification interfaces so that we get a truly end-to-end result: *Version 0.9.1c of OpenSSL's HMAC and SHA-256 correctly implements the FIPS 198-1 and FIPS 180-4 standards, respectively; and that same FIPS 198-1 HMAC standard is a PRF, subject to certain standard (unproved) assumptions about the SHA-256 algorithm that we state formally and explicitly.*

Software is large, complex, and always under maintenance; if we "prove" something about a real program then the proof (and its correspondence to the syntactic program) had better be checked by machine. Fortunately, as Gödel showed, checking a proof is a simple calculation. Today, proof checkers can be simple trusted (and trustworthy) kernel programs [7].

A *proof assistant* comprises a proof-checking kernel with an untrusted proof-development system. The system is typically *interactive,* relying on the user to build the overall structure of the proof and supply the important invariants and induction hypotheses, with many of the details filled in by tactical proof automation or by decision procedures such as SMT or Omega.

Coq is an open-source proof assistant under development since 1984. In the 21st century it has been used for practical applications such as Leroy's correctness proof of an optimizing C compiler [34]. But note, that compiler was not itself written in C; the proof theory of C makes life harder, and only more recently have people done proofs of substantial C programs in proof assistants [32, 29].

Our entire proof (including the algorithmic/cryptographic proofs, the implementation proofs, and the specification matches) is done in Coq, so that we avoid misunderstandings at interfaces. To prove our main theorem, we took these steps (cf. Figure 1):

1. *Formalized.*[5] We use a Coq formalization of the FIPS 180-4 Secure Hash Standard [28] as a specification of SHA-256. (Henceforth, "formalized" or "proved" implies "in the Coq proof assistant.")

2. *Formalized.** We have formalized the FIPS 198-1 Keyed-Hash Message Authentication Code [27] as a specification of HMAC. (Henceforth, the * indicates new work first reported in this paper; otherwise we provide a citation to previous work.)

3. *Formalized.** We have formalized Bellare's functional characterization of the HMAC algorithm.

4. *Proved.** We have proved the equivalence of FIPS 198-1 with Bellare's functional characterization of HMAC.

5. *Formalized.*[6] We use *Verifiable C*, a program logic (embedded in Coq) for specifying and proving functional correctness of C programs.

6. *Formalized.*[35] Leroy has formalized the operational semantics of the C programming language.

7. *Proved.*[6] Verifiable C has been proved sound. That is, if you specify and prove any input-output property of your C program using Verifiable C, then that property actually holds in Leroy's operational semantics of the C language.

8. *Formalized.*[35] Leroy has formalized the operational semantics of the Intel x86 (and PowerPC and ARM) assembly language.

9. *Proved.*[35] If the CompCert optimizing C compiler translates a C program to assembly language, then input-output property of the C program is preserved in the assembly-language program.

10. *Formalized.*[5] We rely on a formalization (in Verifiable C) of the *API interface* of the OpenSSL header file for SHA-256, including its semantic connection to the formalization of the FIPS Secure Hash Standard.

11. *Proved.*[5] The C program implementing SHA-256, lightly adapted from the OpenSSL implementation, has the input-output (API) properties specified by the formalized API spec of SHA-256.

12. *Formalized.** We have formalized the API interface of the OpenSSL header file for HMAC, including its semantic connection to our FIPS 198-1 formalization.

13. *Proved.** Our C program implementing HMAC, lightly adapted from the OpenSSL implementation, has the input-output (API) properties specified by our formalization of FIPS 198-1.

14. *Formalized.** Bellare *et al.* proved properties of HMAC [15, 14] subject to certain assumptions about the underlying cryptographic compression function (typically SHA). We have formalized those assumptions.

15. *Formalized.** Bellare *et al.* proved that HMAC implements a pseudorandom function (PRF); we have formalized what exactly that means. (Bellare's work is "formal" in the sense of rigorous mathematics and LaTeX; we formalized our work in Coq so that proofs of these properties can be machine-checked.)

16. *Proved.** We prove that, subject to these formalized assumptions about SHA, Bellare's HMAC algorithm is a PRF; this is a mechanization of a variant of the 1996 proof [15] using some ideas from the 2006 proofs [14].

Theorem. *The assembly-language program, resulting from compiling OpenSSL 0.9.1c using CompCert, correctly implements the FIPS standards for HMAC and SHA, and implements a cryptographically secure PRF subject to the usual assumptions about SHA.*
Proof. Machine-checked, in Coq, by chaining together specifications and proofs 1–16. Available open-source at https://github.com/PrincetonUniversity/VST/, subdirectories sha, fcf, hmacfcf.

The *trusted code base* (TCB) of our system is quite small, comprising only items 1, 2, 8, 12, 14, 15. Items 4, 7, 9, 11, 13, 16 need not be trusted, because they are proofs checked by the kernel of Coq. Items 3, 5, 6, 10 need not be trusted, because they are specification interfaces checked on both sides by Coq, as Appel [5, §8] explains.

One needs to trust the Coq kernel and the software that compiles it; see Appel's discussion [5, §12].

We do not analyze timing channels or other side channels. But the programs we prove correct are standard C programs for which standard timing and side-channel analysis tools and techniques can be used.

The HMAC brawl. Bernstein [19] and Koblitz and Menezes [33] argue that the security guarantees proved by Bellare *et al.* are of little value in practice, because these guarantees do not properly account for the power of precomputation by the adversary. In effect, they argue that item 15 in our enumeration is the wrong specification for desired cryptographic properties of a symmetric-key authentication algorithm. This may well be true; here we use Bellare's specification in a demonstration of end-to-end machine-checked proof. As improved specifications and proofs are developed by the theorists, we can implement them using our tools. Our proofs are sufficiently modular that only items 15 and 16 would change.

Which version of OpenSSL. We verified HMAC/SHA from OpenSSL 0.9.1c, dated March 1999, which does not include the home-brew object system "engines" of more recent versions of OpenSSL. We further simplified the code by specializing OpenSSL's use of generic "envelopes" to the specific hash function SHA-256, thus obtaining a statically linked code. Verifiable C is capable of reasoning about function pointers and home-brew object systems [6, Chapter 29]—it is entirely plausible that a formal specification of "engines" and "envelopes" could be written down—but such proofs are more complex.

2 Formalizing functional specifications

(Items 1, 2 of the architecture.) The FIPS 180-4 specification of the SHA function can be formalized in Coq as this mathematical function:

Definition SHA_256 (str : list Z) : list Z :=
intlist_to_Zlist (
hash_blocks init_registers (generate_and_pad str)).

where hash_blocks, init_registers, and generate_and_pad are translations of the FIPS standard. Z is Coq's type for (mathematical) integers; the (list Z) is the contents of a string of bytes, considered as their integer values. SHA-256 works internally in 32-bit unsigned modular arithmetic; intlist_to_Zlist converts a sequence of 32-bit machine ints to the mathematical contents of a byte-sequence. See Appel [5] for complete details. The functional spec of SHA-256, including definitions of all these functions, comes to 169 lines of Coq, all of which is in the trusted base for the security/correctness proof.

In this paper we show the full functional spec for HMAC256, the HMAC construction applied to hash function SHA_256:

Definition mkKey (l:list Z):list Z :=
zeropad (**if** $|l| > 64$ **then** SHA_256 l **else** l).
Definition KeyPreparation (k: list Z):list byte :=
map Byte.repr (mkKey k).
Definition HASH l m := SHA_256 (l++m)
Definition HmacCore m k :=
HASH (opad $\oplus$ k) (HASH (ipad $\oplus$ k) m)
Definition HMAC256 (m k : list Z) : list Z :=
HmacCore m (KeyPreparation k)

where zeropad right-extends[1] its argument to length 64 (i.e. to SHA256's block size, in bytes), ipad and opad are the padding constants from FIPS198-1, $\oplus$ denotes byte-wise XOR, and ++ denotes list concatenation.

3 API specifications of C functions

(Items 10, 12 of the architecture.) Hoare logic [31], dating from 1969, is a method of proving correctness of imperative programs using preconditions, postconditions, and loop invariants. Hoare's original logic did not handle pointer data structures well. Separation logic, introduced in 2001 [37], is a variant of Hoare logic that encapsulates "local actions" on data structures.

[1]The more recent RFC4868 mandates that when HMAC is used for authentication, *a fixed key length equal to the output length of the hash functions MUST be supported, and key lengths other than the output length of the associated hash function MUST NOT be supported.* Our specification clearly separates KeyPreparation from HmacCore, but at the top level follows the more permissive standards RFC2104/FIPS198-1 as well as the implementation reality of even contemporary snapshots of OpenSSL and its clones.

Verifiable C [6] is a separation logic that applies to the real C language. Verifiable C's rules are complicated in some places, to capture C's warts and corner cases.

The FIPS 180 and FIPS 198 specifications—and our definitions of SHA_256 and HMAC256—do not explain how the "mathematical" sequences of bytes are laid out in the arrays and structs passed as parameters to (and used internally by) the C functions. For this we need an *API spec.* Using Verifiable C, one specifies the API behavior of each function: the data structures it operates on, its preconditions (what it assumes about the input data structures available in parameters and global variables), and the postcondition (what it guarantees about its return value and changes to data structures). Appel [5, §7] explains how to build such API specs and shows the API spec for the SHA_256 function.

Here we show the API spec for HMAC. First we define a Coq record type,

Record DATA := { LEN:Z; CONT: list Z }.

If *key* has type DATA, then LEN(*key*) is an integer and CONT(*key*) is "contents" of the key, a sequence of integers. We do not use Coq's dependent types *here* to enforce that LEN corresponds to the length of the CONT field, but see the *has_lengthK* constraint below.

To specify the API of a C-language function in Verifiable C, one writes

DECLARE f WITH $\vec{v}$
PRE[*params*] *Pre*   POST [*ret*] *Post*.

where f is the name of the function, *params* are the formal parameters (of various C-language types), and *ret* is the C return type. The precondition *Pre* and postcondition *Post* have the form PROP P LOCAL Q SEP R, where P is a list of pure propositions (true independent of the current program state), Q is a list of local/global variable bindings, and R is a list of *sep*aration logic predicates that describe the contents of memory. The WITH clause describes *logical* variables $\vec{v}$, abstract mathematical values that can be referred to anywhere in the precondition and postcondition.

In our HMAC256_spec, shown below, the first "abstract mathematical value" listed in this WITH clause is the key-pointer *kp*, whose "mathematical" type is "C-language value'," or val. It represents an address in memory where the HMAC session key is passed. In the LOCAL part of the PREcondition, we say that the formal parameter _key actually contains the value *kp* on entry to the function, and in the SEP part we say that there's a data_block at location *kp* containing the actual *key* bytes. In the postcondition we refer to *kp* again, saying that the data_block at address *kp* is still there, unchanged by the HMAC function.

```
Definition HMAC256_spec :=
 DECLARE _HMAC
  WITH kp: val, key:DATA, KV:val,
        mp: val, msg:DATA, shmd: share, md: val
  PRE [ _key OF tptr tuchar, _key_len OF tint,
        _d OF tptr tuchar, _n OF tint,
        _md OF tptr tuchar ]
   PROP(writable_share shmd;
         has_lengthK (LEN key) (CONT key);
         has_lengthD 512 (LEN msg) (CONT msg))
   LOCAL(temp _md md; temp _key kp; temp _d mp;
          temp _key_len (Vint (Int.repr (LEN key)));
          temp _n (Vint (Int.repr (LEN msg)));
          gvar _K256 KV)
   SEP(`(data_block Tsh (CONT key) kp);
       `(data_block Tsh (CONT msg) mp);
       `(K_vector KV);
       `(memory_block shmd (Int.repr 32) md))
 POST [ tvoid ]
   PROP() LOCAL()
   SEP(`(K_vector KV);
       `(data_block shmd
          (HMAC256 (CONT msg) (CONT key)) md);
       `(data_block Tsh (CONT key) kp);
       `(data_block Tsh (CONT msg) mp)).
```

The next WITH value is *key*, a DATA value, that is, a mathematical sequence of byte values along with its (supposed) length. In the PROP clause of the precondition, we enforce this supposition with *has_lengthK* (LEN *key*) (CONT *key*).

The function Int.repr injects from the mathematical integers into 32-bit signed/unsigned numbers. So temp _n (Vint (Int.repr (LEN *msg*))) means, take the mathematical integer (LEN *msg*), smash it into a 32-bit signed number, inject that into the space of C values, and assert that the parameter _n contains this value on entry to the function. This makes reasonable sense if $0 \leq$ LEN *msg* $< 2^{32}$, which is elsewhere enforced by *has_lengthD*. Such 32-bit range constraints are part of C's "warts and all," which are rigorously accounted for in Verifiable C. Both *has_lengthK* and *has_lengthD* are user-defined predicates within the HMAC API spec.

The precondition contains an uninitialized 32-byte memory_block at address *md*, and the _md parameter of the C function contains the value *md*. In the postcondition, we find that at address *md* the memory block has become an initialized data block containing a representation of HMAC256 (CONT *msg*) (CONT *key*).

For stating and proving these specifications, the following characteristics of separation logic are crucial:

1. The SEP lists are interpreted using the *separating conjunction* $*$ which (in contrast to ordinary conjunction $\wedge$) enforces disjointness of the memory regions specified by each conjunct. Thus, the precondition requires—and the postcondition guarantees—that keys, messages, and digests do not overlap.

2. Implicit in the semantic interpretation of a separation logic judgment is a *safety guarantee* of the absence of memory violations and other runtime errors, apart from memory exhaustion. In particular, verified code is guaranteed to respect the specified *footprint*: it will neither read from, nor modify or free any memory outside the region specified by the SEP clause of PRE. Moreover, all heap that is locally allocated is either locally freed, or is accounted for in POST. Hence, memory leaks are ruled out.

3. As a consequence of these locality principles, separation logic specifications enjoy a *frame property*: a verified judgment remains valid whenever we add an arbitrary additional separating conjunct to *both* SEP-clauses. The corresponding proof rule, the *frame rule*, is crucial for modular verification, guaranteeing, for example, that when we call SHA-256, the HMAC data structure remains unmodified.

The HMAC API spec has the 25 lines shown here plus a few more for definitions of auxiliary predicates (has_lengthK 3 lines, has_lengthD 3 lines, etc.); *plus* the API spec for SHA-256, all in the trusted base.

Incremental hashing. OpenSSL's HMAC and SHA functions are incremental. One can initialize the hasher with a key, then incrementally append message-fragments (not necessarily block-aligned) to be hashed, then finalize to produce the message digest. We fully support this incremental API in our correctness proofs. For simplicity we did not present it here, but Appel [5] presents the incremental API for SHA-256. The API spec for fully incremental SHA-256 is 247 lines of Coq; the simple (nonincremental) version has a *much* smaller API spec, similar to the 25+6 lines shown here for the nonincremental HMAC.

Once every function is specified, we use Verifiable C to prove that each function's body satisfies its specification. See Section 6.

4 Cryptographic properties of HMAC

(Items 14, 15, 16 of the architecture.) This section describes a mechanization of a cryptographic proof of security of HMAC. The final result of this proof is similar to the result of Bellare et al. [15], though the structure of the proof and some of the definitions are influenced

by Bellare's later proof [14]. This proof uses a more abstract model of HMAC (compared to the functional spec in §2) in which keys are in $\{0,1\}^b$ (the set of bit vectors of length b), inputs are in $\{0,1\}^*$ (bit lists), and outputs are in $\{0,1\}^c$ for arbitrary b and c such that $c \leq b$. An implementation of HMAC would require that b and c are multiples of some word size, and the input is an array of words, but these issues are typically not considered in cryptographic proofs.

In the context of the larger proof described in this paper, we refer to this model of HMAC in which sizes are arbitrary as the *abstract specification* of HMAC. In order to use security results related to this specification, we must show that this specification is appropriately related to the specification provided in §2. We chose to prove the security of the abstract specification, rather than directly proving the security of a more concrete specification, because there is significant value in this organization. Primarily, this organization allows us to use the exact definitions and assumptions from the cryptography literature, and we therefore gain greater assurance that the definitions are correct and the assumptions are reasonable. Also, this approach demonstrates how an existing mechanized proof of cryptographic security can be used in a verification of the security of an implementation. This organization also helps decompose the proof, and it allows us to deal with issues related to the implementation in isolation from issues related to cryptographic security.

We address the "gap" between the abstract and concrete HMAC specifications by proving that they are equivalent. Section 5 outlines the proof and states the equivalence theorem.

4.1 The Foundational Cryptography Framework

This proof of security was completed using the Foundational Cryptography Framework (FCF), a Coq library for reasoning about the security of cryptographic schemes in the computational model [38]. FCF provides a probabilistic programming language for describing all cryptographic constructions, security definitions, and problems that are assumed to be hard. Probabilistic programs are described using Gallina, the purely functional programming language of Coq, extended with a computational monad that adds sampling uniformly random bit vectors. The type of probabilistic computations that return values of type A is Comp A. The code uses {0,1}^n to describe sampling a bit vector of length n. Arrows (<-$) denote sequencing (i.e. bind) in the monad.

Listing 1 contains an example program implementing a one-time pad on bit vectors of length c (for any natural number c). The program produces a random bit vector and stores it in p, then returns the *xor* (using the standard Coq function BVxor) of p and the argument x.

```
Definition OTP c (x : Bvector c) : Comp (Bvector c)
  := p <-$ {0, 1}^c; ret (BVxor c p x)
```

Listing 1: Example Program: One-Time Pad.

The language of FCF has a denotational semantics that relates programs to discrete, finite probability distributions. A distribution on type A is modeled as a function in $A \to \mathbb{Q}$ which should be interpreted as a probability mass function. FCF provides a theory of distributions, a program logic, and a library of tactics that can be used to complete proofs without appealing directly to the semantics. We can use FCF to prove that two distributions are equivalent, that the distance between the probabilities of two events is bounded by some value, or that the probability of some event is less than some value. Such claims enable cryptographic proofs in the "sequence of games" style [16].

In some cryptographic definitions and proofs, an adversary is allowed to interact with an "oracle" that maintains state while accepting queries and providing responses. In FCF, an oracle has type S → A → Comp (B ∗ S) for types S, A, and B, of state, input, and output, respectively. The OracleComp type is provided to allow an adversary to interact with an oracle without viewing or modifying its state. By combining an OracleComp with an oracle and a value for the initial state of the oracle, we obtain a computation returning a pair of values, where the first value is produced by the OracleComp at the end of its interaction with the oracle, and the second value is the final state of the oracle.

4.2 HMAC Security

We mechanized a proof of the following fact. If h is a compression function, and h^* is a Merkle-Damgård hash function constructed from h, then HMAC based on h^* is a pseudorandom function (PRF) assuming:

1. h is a PRF.
2. h^* is weakly collision-resistant (WCR).
3. The dual family of h (denoted $\bar{h}$) is a PRF against $\oplus$-related-key attacks.

The formal definition of a PRF is shown in Listing 2. In this definition, f is a function in K → D → R that should be a PRF. That is, for a key k : K, an adversary who does not know k cannot gain much *advantage* in distinguishing f k from a random function in D → R.

The adversary A is an OracleComp that interacts with either an oracle constructed from f or with randomFunc, a random function constructed by producing random values for outputs and memoizing them so they can be repeated the next time the same input is provided. The randomFunc oracle uses a list of pairs as its state, so an empty list is provided as its initial state. The value tt is the "unit" value, where unit is a placeholder type much like "void" in the C language. This definition uses alternative arrows (such as <-$2) to construct sequences in which the first computation produces a tuple, and a name is given to each value in the tuple. The size of the tuple is provided in the arrow in order to assist the parser.

```
Definition f_oracle (k : K) (x : unit) (d : D)
                : Comp (R × unit) :=
  ret (f k d, tt).

Definition PRF_G0 : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt; ret b.

Definition PRF_G1 : Comp bool :=
  [b, _] <-$2 A (randomFunc) nil; ret b.

Definition PRF_Advantage : Rat :=
  | Pr[PRF_G0] -Pr[PRF_G1] |.
```

Listing 2: Definition of a PRF. The f_oracle function wraps the function f (closed over key k) and turns it into an oracle. A is an adversary. Comp bool is the type of probabilistic computations that produce a bool. Rat is the type of (unary, nonnegative) rational numbers.

This security definition is provided in the form of a "game" in which the adversary tries to determine whether the oracle is f (in game 0) or a random function (in game 1). After interacting with the oracle, the adversary produces a bit, and the adversary "wins" if this bit is likely to be different in the games. We define the *advantage* of the adversary to be the difference between the probability that it produces "true" in game 0 and in game 1. We can conclude that f is a PRF if this advantage is sufficiently small.

```
Definition Adv_WCR_G : Comp bool :=
  k <-$ RndKey;
  [d1, d2, _] <-$3 A (f_oracle k) tt;
  ret ((d1 != d2) && ((f k d1) ?= (f k d2))).

Definition Adv_WCR : Rat := Pr[Adv_WCR_G].
```

Listing 3: Definition of Weak Collision-Resistance.

Listing 3 defines a weakly collision-resistant function. This definition uses a single game in which the adversary is allowed to interact with an oracle defined by a keyed function f. At the end of this interaction, the adversary attempts to produce a collision, or a pair of different input values that produce the same output. In this game, we use ?= and != to denote tests for equality and inequality, respectively. The advantage of the adversary is the probability with which it is able to locate a collision.

Finally, the security proof assumes that a certain keyed function is a PRF against $\oplus$-related-key attacks (RKA). This definition (Listing 4) is similar to the definition of a PRF, except the adversary is also allowed to provide a value that will be xored with the unknown key before the PRF is called. Note that this assumption is applied to the *dual family* of h, in which the roles of inputs and keys are reversed. So a single input value is chosen at random and fixed, and the adversary queries the oracle by providing values which are used as keys.

```
Definition RKA_F (k: Bvector b) (s: unit)
          (p: Bvector b × Bvector c)
      : (Bvector c × unit) :=
  ret (f ((fst p) xor k) (snd p), tt).

Definition RKA_R (k: Bvector b)
        (s : list (Bvector c × Bvector c))
        (p: Bvector b × Bvector c)
        : (Bvector c × list (Bvector c × Bvector c) :=
  randomFunc s ((fst p) xor k, (snd p))

Definition RKA_G0 : Comp bool :=
  k <-$ RndKey; [b, _] <-$2 A (RKA_F k) tt; ret b.

Definition RKA_G1 : Comp bool :=
  k <-$ RndKey; [b, _] <-$2 A (RKA_R k) nil; ret b.

Definition RKA_Advantage : Rat :=
  | Pr[RKA_G0] -Pr[RKA_G1] |.
```

Listing 4: Definition of Security against $\oplus$ Related-Key Attacks. b is the key length of the compression function, c is the input length of the compression function; Bvector b is the type of bit-vectors of length b.

The proof of security has the same basic structure (Figure 2) as Bellare's more recent HMAC proof [14], though we simplify the proof significantly by assuming h^* is WCR. The proof makes use of a nested MAC (NMAC) construction that is similar to HMAC, but it uses h^* in a way that is not typically possible in implementations of hash functions. The proof begins by showing that NMAC is a PRF given that h is a PRF and h^* is WCR. Then we show that NMAC and HMAC are "close" (that no adversary can effectively distinguish them) under the assumption that $\bar{h}$ is a $\oplus$-RKA-secure PRF. Finally, we combine these two results to derive that HMAC is a PRF.

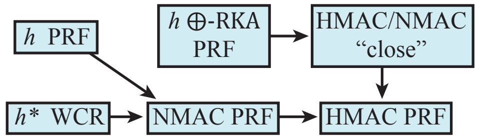

Figure 2: HMAC Security Proof Structure

We also mirror Bellare's proof by reasoning about slightly generalized forms of HMAC and NMAC (called GHMAC and GNMAC) that require the input to be a list of bit vectors of length b. The proof also makes use of a "two-key" version of HMAC that uses a bit vector of length $2b$ as the key. To simplify the development of this proof, we build HMAC on top of these intermediate constructions in the abstract specification (Listing 5).

```
Definition h_star k (m : list (Bvector b))
  := fold_left h m k.
Definition hash_words := h_star iv.

Definition GNMAC k m :=
  let (k_Out, k_In) := splitVector c c k in
  h k_Out (app_fpad (h_star k_In m)).

Definition GHMAC_2K k m :=
  let (k_Out, k_In) := splitVector b b k in
  let h_in := (hash_words (k_In :: m)) in
  hash_words (k_Out :: (app_fpad h_in) :: nil).

Definition HMAC_2K k (m : list bool) :=
  GHMAC_2K k (splitAndPad m).

Definition HMAC (k : Bvector b) :=
  HMAC_2K ((k xor opad) ++ (k xor ipad)).
```

Listing 5: HMAC Abstract Specification.

splitAndPad produces a list of bit-vectors from a list of bits (padding the last bit-vector as needed), and app_fpad is a padding function that produces a bit vector of length b from a bit vector of length c. In the HMAC function, we use constants opad and ipad to produce a key of length $2b$ from a key of length b.

The statement of security for HMAC is shown in Listing 6. We show that HMAC is a PRF by giving an expression that bounds the advantage of an arbitrary adversary A. This expression is the sum of three terms, where each term represents the advantage of some adversary against some other security definition.

The listing describes all the parameters to each of the security definitions. In all these definitions, the first parameter is the computation that produces random keys, and in PRF_Advantage and RKA_Advantage, the second parameter is the computation that produces random values in the range of the function. In all definitions, the penultimate parameter is the function of interest, and the final parameter is some constructed adversary. The descriptions of these adversaries are omitted for brevity, but only their computational complexity is relevant (e.g. all adversaries are in ZPP assuming adversary A is in ZPP).

```
Theorem HMAC_PRF:
  PRF_Advantage ({0, 1}^b) ({0, 1}^c) HMAC A <=
  PRF_Advantage ({0, 1}^c) ({0, 1}^c) h B1 +
  Adv_WCR ({0, 1}^c) h_star B2 +
  RKA_Advantage ({0, 1}^b) ({0, 1}^c)
    (BVxor b) (dual_f h) B3.
```

Listing 6: Statement of Security for HMAC.

We can view the result in Listing 6 in the asymptotic setting, in which there is a security parameter η, and parameters c and b are polynomial in η. In this setting, it is possible to conclude that the advantage of A against HMAC is negligible in η assuming that each of the other three terms is negligible in η. We can also view this result in the *concrete setting*, and use this expression to obtain *exact security* measures for HMAC when the values of b and c are fixed according the sizes used by the implementation. The latter interpretation is more informative, and probably more appropriate for reasoning about the cryptographic security of an implementation.

5 Equivalence of the two functional specs

(Item 4 of the architecture.) In §2 we described a bytes-and-words specification following FIPS198-1, suited for proving the C program; call that the *concrete specification*. In §4 we described a length-constrained bit-vector specification following Bellare *et al.*'s original papers; call that the *abstract specification*. Here we describe the proof that these two specifications are equivalent.

Proof outline. There are seven main differences between the concrete and abstract specs:

(0) The abstract spec, as its name suggests, leaves several variables as parameters to be instantiated. Thus, in order to compute with the abstract HMAC, one must pass it "converted" variables and "wrapped" functions from the concrete HMAC.

(1) The abstract spec operates on bits, whereas the concrete spec operates on bytes.

(2) The abstract spec uses the dependent type Bvector n, which is a length-constrained bit list of length n, whereas the concrete spec uses byte lists and int lists, whose lengths are unconstrained by definition.

(3) Due to its use of dependent types, the abstract spec must pad its input twice in an ad-hoc manner, whereas the concrete spec uses the SHA-256 padding function consistently.

(4) The concrete spec treats the hash function (SHA-256) as a black box, whereas the abstract spec exposes various parts of its functionality, such as its initialization vector, internal compression function, and manner of iteration. (It does this because the Bellare-style proofs rely on the Merkle-Damgård structure of the hash function.)

(5) The abstract spec pads the message and splits it into a list of blocks so that it can perform an explicit fold over the list of lists. However, the concrete spec leaves the message as a list of bytes and performs an implicit fold over the list, taking a new block at each iteration.

(6) The abstract spec defines HMAC via the HMAC_2K and GHMAC_2K structures, not directly.

Instantiating the abstract specification. The abstract HMAC spec leaves the following parameters abstract:

```
Variable c p : nat.

(* compression function *)
Variable h : Bvector c → Bvector b → Bvector c.

(* initialization vector *)
Variable iv : Bvector c.
Variable splitAndPad : Blist → list (Bvector b).
Variable fpad : Bvector c → Bvector p.
Variable opad ipad : Bvector b.
```

The abstract HMAC spec is also more general than the concrete spec, since it operates on bit vectors, not byte lists, and does not specify a block size or output size. After "replacing" the vectors with lists (see the explanation of difference (2)) and specializing $c = p = 256$ (resulting in $b = 512$), we may instantiate abstract parameters with concrete parameters or functions from SHA-256, wrapped in *bytesToBits* and/or *intlist_to_Zlist* conversion functions. For example, we instantiate the block size to 256 and the output size to 512, and define *iv* and *h* as:

```
Definition intsToBits := bytesToBits ∘ intlist_to_Zlist.
Definition sha_iv : Blist :=
  intsToBits SHA256.init_registers.
Definition sha_h (regs : Blist) (block : Blist) : Blist :=
  intsToBits (SHA256.hash_block (bitsToInts regs)
                                (bitsToInts block)).
```

The *intlist_to_Zlist* conversion function is necessary because portions of the SHA-256 spec operate on lists of *Integer*s, as specified in our bytes-and-words formalization of FIPS 180-4. (Z in Coq denotes arbitrary-precision mathematical integers. Our SHA-256 spec represents byte values as Z. An *Integer* is four byte-Zs packed big-endian into a 32-bit integer.)

We are essentially converting the types of the functions from functions on intlists (*intlist* $\rightarrow \ldots \rightarrow$ *intlist*) to functions on Blists (*Blist* $\rightarrow \ldots \rightarrow$ *Blist*) by converting their inputs and outputs.

Let us denote by HmacAbs256 the instantiation of function HMAC from Listing 5 to these parameters. Since Bellare's proof assumes that the given key is of the right length (the block size), our formal equivalence result relates HmacAbs256 to the function HmacCore from Section 2, i.e. to the part of HMAC256 that is applied after key length normalization. (Unlike Bellare, FIPS 198 includes steps to first truncate or pad the key if it is too long or short.)

Theorem. For key vector kv of type Bvector 256 and message m of type list bool satisfying $|l| \equiv 0\ (mod\ 8)$,

$$\mathsf{HmacAbs256}\ kv\ m \approx \mathsf{HmacCore}\ \overline{m}\ (\mathsf{map\ Bytes.repr}\ \overline{kv}).$$

where $\overline{(\cdot)}$ denotes bitsToBytes conversion, and $\approx$ is equality modulo conversion between lists and vectors.

Reconciling other differences. The last difference (6) is easily resolved by unfolding the definitions of HMAC_2K and GHMAC_2K. We solve the other six problems by changing definitions and massaging the two specs toward each other, proving equality or equivalence each time.

Bridging (5) is basically the proof of correctness of a *deforestation* transformation. Consider a message m as a list of bits b_i. First, split it into 512-bit blocks B_i, then "fold" (the "reduce" operation in map-reduce) the hash operation H over it, starting with the initialization vector iv: $H(H(H(iv,B_0),B_1),\ldots,B_{n-1})$. Alternatively, express this as a recursive function on the original bit-sequence b: grab the first 512 bits, hash with H, then do a recursive call after skipping the first 512 bits:

```
Function F (r: list bool) (b: list bool)
      {measure length b} : list bool :=
match msg with
   nil ⇒ r
 | _ ⇒ F (H r (firstn 512 b)) (skipn 512 b)
end.
```

Provided that $|b|$ is a multiple of 512 (which we prove elsewhere), $F(iv,b) = H(H(H(iv,B_0),B_1),\ldots,B_{n-1})$.

We bridge (4) by using the fact that SHA-256 *is* a Merkle-Damgård construction over a compression function. This is a simple matter of matching the definition of SHA-256 to the definition of an MD hash function.

Bridging (3) is a proof that two different views of the SHA padding function are equivalent. Before iterating the compression function on the message, SHA-256 pads it in a standard, one-to-one fashion such that its length is a multiple of the block size. It pads it as such:

$$msg\,|\,[1]\,|\,[0,0,\ldots 0]\,|\,L$$

where | denotes list concatenation and L denotes the 64-bit representation of the length of the message. The number of 0s is calculated such that the length of the entire padded message is a multiple of the block size.

The abstract spec accomplishes this padding in two ways using the functions *fpad* and *splitAndPad*. *fpad* pads a message of known length of the output size c to the block size b, since c is specified to be less than b. *splitAndPad* breaks a variable-length message (of type list bool) into a list of blocks, each size b, padding it along the way. *fpad* is instantiated as a constant, since we know that the length of the message is $c < b$. *splitAndPad* is instantiated as the normal SHA padding function, but tweaked to add one block size to the length appended in $[l_1, l_2]$, since k_{in} (with a length of one block) will be prepended to the padded message later.

To eliminate these two types of ad-hoc padding, we rewrite the abstract spec to incorporate *fpad* and *splitAndPad* into a single padding function split_and_pad included in the hash function, in the style of SHA-256.

$$\mathsf{hash\_words\_padded} := \mathsf{hash\_words} \circ \mathsf{split\_and\_pad}.$$

We then remove *fpad* and *splitAndPad* from subsequent versions of the specification. We can easily prove equality by unfolding definitions.

Bridging bytes and bits. The abstract and concrete HMAC functions have different types, so we cannot prove them *equal*, only *equivalent*. $HMAC_c$ operates on (lists of) bits and $HMAC_a$ operates on (lists of) bytes. ($HMAC_c$ used to operate on vectors, but recall that we replaced them with lists earlier.) To bridge gap (1) we prove, given that the inputs are equivalent, the outputs will be equivalent:

$$k_c \approx k_a \rightarrow$$
$$m_c \approx m_a \rightarrow$$
$$HMAC_c(k_c, m_c) \approx HMAC_a(k_a, m_a).$$

The equivalence relation $\approx$ can be defined either computationally or inductively, and both definitions are useful.

To reason about the behavior of the wrapped functions with which we instantiated the abstract HMAC spec, we use the computational equivalence relation ($\approx_c$) instantiated with a generic conversion function. This allows us to build a framework for reasoning about the asymmetry of converting from bytes to bits versus from bits to bytes, as well as the behavior of repeatedly applied wrapped functions.

Bridging vectors and lists. We bridge (2) by changing all Bvector n to list bool, then proving that all functions preserve the length of the list when needed. This maintains the Bvector n invariant that its length is always n. In general, the use of lists (of bytes, or Z values) is motivated by the desire to reuse Appel [5]'s prior work on SHA literally, whereas the use of Bvector enables a more elegant proof of the proof of cryptographic properties.

Injectivity of splitAndPad. The security proof relies on the fact that splitAndPad is injective, in the sense that $b_1 = b_2$ should hold whenever splitAndPad(b_1) = splitAndPad(b_2). Indeed, this property is violated if we naively instantiate splitAndPad with the bitlists-to-bytelists roundtrip conversion of SHA-256's padding+length function, due to the non-injectivity of bitlists-to-bytelists conversion. On the other hand, as the C programs interpret all length informations as referring to lengths in bytes, attackers that attempt to send messages whose length is not divisible by 8 are effectively ruled out. To verify this property formally, we make the abstract specification (and the proof of **Theorem** HMAC_PRF) parametric in the type of messages. Instantiating the development to the case where messages are bitlists of length $8n$ allows us to establish the desired injectivity condition along the the lines of the following informal argument.

Given a message m, SHA's splitAndPad appends a 1 bit, then k zero bits, then a 64-bit integer representing the length of the message $|m|$; k is the smallest number so that $|\text{splitAndPad}(m_1)|$ is a multiple of the block size. Injective means that if $m_1 \neq m_2$ then splitAndPad$(m_1) \neq$ splitAndPad(m_2). The proof has five cases:

- $m_1 = m_2$, then by contradiction.
- $|m_1| = |m_2|$, then splitAndPad(m_1) must differ from splitAndPad(m_2) in their first $|m_1|$ bits.
- $|m_1| \neq |m_2|$, $|m_1| \leq 2^{64}$, $|m_2| \leq 2^{64}$, then the last 64 bits (representation of length) will differ.
- $(|m_1| - |m_2|) \bmod 2^{64} \neq 0$, then the last 64 bits (representation of length) will differ.
- $|m_1| \neq |m_2|$, and $(|m_1| - |m_2|) \bmod 2^{64} = 0$; then the lengths $|m_1|, |m_2|$ must differ by at least 2^{64}, so the variation in k_1 and k_2 (which must each be less than twice the block size) cannot make up the difference, so the padded messages will have different lengths.

Our machine-checked proof of injectivity is somewhat more comprehensive than Bellare et al.'s [15], which reads in its entirety, "Notice that a way to pad messages to an exact multiple of b bits needs to be defined, in particular, MD5 and SHA pad inputs to always include an encoding of their length."

Preservation of cryptographic security. Once the equivalence between the two functional programs has been established, and injectivity of the padding function is proved, it is straightforward to prove the applicability of **Theorem** HMAC_PRF (Listing 6) to the API spec.

6 Specifying and verifying the C program

(Items 11, 13 of the architecture.) We use Verifiable C to prove that each function's body satisfies its specification. As in a classic Hoare logic, each kind of C-language statement has one or more proof rules. Appel [6, Ch. 24-26] presents these proof rules, and explains how *tactics*—programmed in the Ltac language of Coq—apply the proof rules to the abstract syntax trees of C programs. The ASTs are obtained by applying the front-end phase of the CompCert compiler to the C program. The HMAC proof (item 13 in §1) is 2832 lines of Coq (including blanks and comments), none of which is in the trusted base because it is all machine-checked.

Just like OpenSSL's implementation of SHA-256, the C code implementing HMAC is *incremental*: the one-shot HMAC function is obtained by composing auxiliary functions hmacInit, hmacUpdate, hmacFinish, and hmacCleanup that are all exposed in the header file. They allow a client to reuse a key for the authentication of multiple messages, and also to provide each individual message in chunks, by repeatedly invoking hmacUpdate. To this end, the auxiliary functions employ the hash function's incremental interface and are formulated over a client-visible struct, HMAC_CTX. Specializing OpenSSL's original header file to the hash function SHA-256 yields the following:[2]

```
typedef struct hmac_ctx_st {
        SHA256_CTX md_ctx; // workspace
        SHA256_CTX i_ctx; // inner SHA structure
        SHA256_CTX o_ctx; // outer SHA structure
        unsigned int key_length;
        unsigned char key[64];
        } HMAC_CTX;

void HMAC_Init(HMAC_CTX *ctx,
               unsigned char *key, int len);

void HMAC_Update(HMAC_CTX *ctx,
                 const void *data, size_t len);

void HMAC_Final(HMAC_CTX *ctx,
                unsigned char *md);

void HMAC_cleanup(HMAC_CTX *ctx);

unsigned char *HMAC(unsigned char *key,
                    int key_len,
                    unsigned char *d, int n,
                    unsigned char *md);
```

[2]During the verification, we observed that the fields key_length and key can be eliminated from hmac_ctx_st, for the price of minor alterations to the code, API specification, and proof. A similar modification has recently (and independently) been implemented in boringssl.

Fields i_ctx and o_ctx store partially constructed SHA data structures that are initialized during HMAC_Init to hold the $\oplus$ of the normalized key and ipad/opad, respectively, and are copied to the workspace md_ctx where the inner and outer hashing applications are performed.

Omitting the implementations of the other functions, the one-shot HMAC invokes the incremental functions on a freshly stack-allocated HMAC_CTX, where 32 is the digest length of SHA-256:

```
unsigned char *HMAC(unsigned char *key,
                    int key_len, unsigned char *d,
                    int n, unsigned char *md) {
   HMAC_CTX c; static unsigned char m[32];
   if (md == NULL) md=m;
   HMAC_Init(&c, key, key_len);
   HMAC_Update(&c,d,n);
   HMAC_Final(&c,md);
   HMAC_cleanup(&c);
   return(md);
}
```

In order to verify that this code satisfies the specification HMAC256_spec from Section 2, each incremental function is equipped with its individual Verifiable C specification. Each specification is formulated with reference to a suitable Coq function (or alternatively a propositional relation, as extractability is not required) that expresses the function's effect on the HMAC_CTX structure abstractly, without reference to the concrete memory layout.

More precisely, the logical counterpart of an HMAC_CTX structure is given by the Coq type

```
Inductive hmacabs :=
 HMACabs: ∀(ctx iSha oSha: s256abs)
                (keylen: Z) (key: list Z), hmacabs.
```

That is, an *HMAC abstract state* has five components: ctx, iSha, and oSha are *SHA abstract states*, keylen is an integer, and key is a list of (integer) byte values. Appel [5] defines SHA abstract states; if you initialize a SHA module and dump the first n bytes of a message into it, you get a value of type s256abs representing the abstract state of the incremental-mode SHA-256 program.

Appel also defines a relation, update_abs $a\ c_1\ c_2$, saying that adding another (incremental mode) message fragment s to abstract state c_1 yields state c_2.

We define abstract states for HMAC, and the incremental-mode HMAC update relation, in terms of the SHA s256abs type and update_abs relation.

```
Definition hmacUpdate (data: list Z)
                        (h1 h2: hmacabs) : Prop :=
  match h1 with HMACabs ctx1 iS oS klen k
  ⇒ ∃ctx2, update_abs data ctx1 ctx2
     ∧ h2 = HMACabs ctx2 iS oS klen k
  end.
```

To connect these definitions to the *upper* parts of our verification architecture, we prove that the composition of these counterparts of the incremental functions (i.e. the counterpart of the one-shot HMAC) coincides with HMAC256 the FIPS functional specification from Section 3.

```
Definition hmacIncremental (k data dig:list Z) :=
∃hInit hUpd, hmacInit k hInit ∧
        hmacUpdate data hInit hUpd ∧
        hmacFinal hUpd dig.

Lemma hmacIncremental_sound k data dig:
       hmacIncremental k data dig →
       dig = HMAC256 data k.
Proof. ... Qed.
```

Downward, we connect hmacabs and HMAC_CTX by a separation logic representation predicate:

```
Definition hmacstate_ (h:hmacabs) (c:val): mpred:=
    EX r:hmacstate,
     !! hmac_relate h r
     && data_at Tsh t_struct_hmac_ctx_st r c.
```

where hmac_relate is a pure proposition specifying that each component of a concrete struct r has precisely the content prescribed by h.

Using these constructions, we obtain API specifications of the incremental functions such as HMAC_Update.

```
Definition HMAC_Update_spec :=
 DECLARE _HMAC_Update
 WITH h1: hmacabs, c : val, d:val, len:Z,
       data:list Z, KV:val
 PRE [ _ctx OF tptr t_struct_hmac_ctx_st,
       _data OF tptr tvoid, _len OF tuint]
   PROP(has_lengthD (s256a_len (absCtxt h1))
                          len data)
   LOCAL(temp _ctx c; temp _data d;
           temp _len (Vint (Int.repr len));
           gvar _K256 KV)
   SEP(`(K_vector KV); `(hmacstate_ h1 c);
       `(data_block Tsh data d))
 POST [ tvoid ]
   EX h2: hmacabs,
     PROP(hmacUpdate data h1 h2)
     LOCAL()
     SEP(`(K_vector KV); `(hmacstate_ h2 c);
        `(data_block Tsh data d)).
```

7 Proof effort

It is difficult to estimate the proof effort, as we used this case study to learn where to make improvements to the usability and automation of our toolset. However, we can give some numbers: size, in commented lines of code, of the specifications and proofs. Where relevant, we give the size of the corresponding C API or function.

Functional correctness proof of the C program:

C lines	Coq lines	SHA-256 component
	169	FIPS-180 functional spec of SHA
71	247	API spec of SHA-256
	1022	Lemmas about the functional spec
10	229	Proof of addlength function
81	1640	sha256_block_data_order()
10	43	SHA256_Init()
38	1682	SHA256_Update()
31	1484	SHA256_Final()
7	58	SHA256()
248	*6574*	*Total SHA-256*
	159	FIPS-198 functional spec of HMAC
25	374	API spec
25	*533*	*Total HMAC spec*
	875	Supporting lemmas
74	1530	HMAC_Init proof
7	101	HMAC_Update proof
27	196	HMAC_Final proof
5	31	HMAC_Cleanup proof
21	99	HMAC proof
134	*2832*	*Total HMAC proof*

FCF proof that HMAC is a PRF:

Coq lines	component
70	Bellare-style functional spec of HMAC
25	Statement, HMAC is a PRF
377	Proof, HMAC is a PRF
472	*Total*

Connecting Verifiable C proof to FCF proof:

Coq lines	component
3017	General equivalence proof of the two functional specs for any compression function
993	Specialization to SHA-256
4010	*Total*

8 Related work

We have presented a *foundational, end-to-end* verification. All the relevant aspects of cryptographic proofs or of the C programming language are defined and checked with respect to the foundations of logic. We say a reasoning engine for crypto is *foundational* if it is implemented in, or its implementation is proved correct in, a trustworthy general-purpose mechanized logic. We say a connection to a language implementation is *foundational* if the synthesizer or verifier is connected (with proofs in a trustworthy general-purpose mechanized logic) to the operational semantics compiled by a verified compiler.

Crypto verification. Smith and Dill [40] verify several block-cipher implementations written in Java with respect to a functional spec written either in Java or in ACL2. They compile to bytecode, then use a subset model of the JVM to generate straight-line code. This work is not end-to-end, as the JVM is unverified—and it wouldn't suffice to simply plug in a "verified" JVM, if one existed, without also knowing that the *same* specification of the JVM was used in both proofs. Their method applies only where the number of input bits is fixed and the loops can be completely unrolled. Their verifier would likely be applicable to the SHA-256 block shuffle function, but certainly not to the management code (padding, adding the length, key management, HMAC).

Cryptol [25] generates C or VHDL directly from a functional specification, where the number of input bits is fixed and the loops can be completely unrolled, i.e. the SHA-256 block shuffle, but not the SHA-256 management code or HMAC. The Cryptol synthesizer is not foundational because its semantics is not formally specified, let alone proved.

CAO is a domain specific language for crypto applications, which is compiled into C [11], and equipped with verification technology based on the FRAMA-C tool suite [4].

Libraries of arbitrary-precision arithmetic functions have been verified by Fischer [39] and Berghofer [17] using Isabelle/HOL. Bertot et al. [20] verify GMP's computation of square roots in Coq, based on Filliatre's CORRECTNESS tool for ML-style programs with imperative features [26]. Neither of these formalizations is foundatationally connected to a verified compiler.

Verified assembly implementations of arithmetic functions have been developed by Myreen and Curello [36] and Affeldt [1], who use, respectively, proof-producing (de)compilation and simulation to link assembly code and memory layout to functional specifications at (roughly) the level of our FIPS specifications. Chen et al. [24] verify the Montgomery step in Bernstein's high-speed implementation of elliptic curve 25519 [18], using a combination of SMT solving and Coq to implement a Hoare logic for Bernstein's portable assembly representation qhasm.

The abstraction techniques and representation predicates in these works are compatible with our memory layout predicates. One important future step is to condense commonalities of these libraries into an ontology for crypto-related reasoning principles, reusable across multiple language levels and realised in multiple proof assistants. Doing this is crucial for scaling our work to larger fragments of cryptographic libraries.

Formal verification of protocols is an established research area, and efforts to link abstract protocol verifications to implementations are emerging using automated techniques like model extraction or code generation [8], or interactive proof [2].

Toma and Borrione [41] use ACL2 to prove correctness of a VHDL implementation of the SHA-1 block-shuffle algorithm. There is no connection to (for example) a verified compiler for VHDL.

Backes *et al.* [10] verify mechanically (in EasyCrypt) that Merkle-Damgård constructions have certain security properties.

EasyCrypt. Almeida *et al.* [3] describe the use of their EasyCrypt tool to verify the security of an implementation of the RSA-OAEP encryption scheme. A functional specification of RSA-OAEP is written in EasyCrypt, which then verifies its security properties. An unverified Python script translates the EasyCrypt specification to (an extension of) C, then an extension of CompCert compiles it to assembly language. Finally, a leakage tool verifies that the assembly language program has no more program counter leakage than the source code, i.e. that the compiled program's trace of conditional branches is no more informative to the adversary than the source code's.

The EasyCrypt verifier is not foundational; it is an OCaml program whose correctness is not proved. The translation from C to assembly language is foundational, using CompCert. However, EasyCrypt's C code relies on bignum library functions, but provides no mechanism by which these functions can be proved correct.

CertiCrypt [13] is a system for reasoning about cryptographic algorithms in Coq; it is foundational, but (like EasyCrypt) has no foundational connection to a C semantics. **ZKCrypt**[9] is a synthesizer that generates C code for zero-knowledge proofs, implemented in CertiCrypt, also with no foundational connection to a C semantics.

Bhargavan *et al.* [21] "implement TLS with verified cryptographic security" in F# using the **F7** typechecker. F7 is not capable of reasoning about all of the required cryptographic/probabilistic relationships required to complete the proof. So parts of the proof are completed using EasyCrypt, and there is no formal relationship between the EasyCrypt proofs and the F7 proof; one must inspect the code to ensure that the things admitted in F7 are the same things that are proved in Easy-

Crypt. F7 is also not foundational, because the type checker has a large amount of trusted code and because it depends on the Z3 SMT solver. Another difference between this work and ours is that the code provides a reference implementation in F#, not an efficient implementation in C.

CryptoVerif [22] is a prover (implemented in OCaml) for security protocols in which one can, for example, extract a OCaml program from a CryptoVerif model [23]. CryptoVerif is not foundational, the extraction is not foundational, and the compiler for OCaml is not foundationally verified.

C program verification. There are many program analysis tools for C. Most of them do not address functional specification or functional correctness, and most are unsound and incomplete. They are useful in practice, but cannot be used for an end-to-end verification of the kind we have done.

Foundational formal verification of C programs has only recently been possible. The most significant such works are both operating-system kernels: seL4 [32] and CertiKOS [29]. Both proofs are refinement proofs between functional specifications and operational semantics. Both proofs are done in higher-order logics: seL4 in Isabelle/HOL and CertiKOS in Coq.

Neither of those proof frameworks uses separation logic, and neither can accommodate the use of addressable local variables in C. This means that OpenSSL's HMAC/SHA could not be proved in these frameworks, because it uses addressable local variables.

Additionally, neither of those proof frameworks can handle function pointers. OpenSSL uses function pointers in its "engines" mechanism, an object-oriented style of programming that dynamically connects components together, such as HMAC and SHA. The *Verifiable C* program logic is capable of reasoning about such object-oriented patterns in C [6, Chapter 29], although we have not done so in the work described in this paper.

9 Conclusion

Widely used open-source cryptographic libraries such as OpenSSL, operating systems kernels, and the C compilers that build them, form the backbone of the public's communication security. Since 2013 or so, it has become clear that hackers and nation-states (is there a difference anymore?) are willing to invest enormous resources in searching for vulnerabilities and exploiting them. Other authors have demonstrated that compilers [34] and OS kernels [32, 29] can be built to a *provable zero-functional-correctness-defect standard*. Here we have demonstrated the same, in a modular way, for key components of our common cryptographic infrastructure.

Functional correctness implies zero buffer-overrun defects as well. But there are side channels we have not addressed here, such as timing, fault-injection, and leaks through dead memory. Our approach does not solve these problems; but it makes them no worse. Because we can reason about standard C code, other authors' techniques for side channel analysis are applicable without obstruction.

Functional correctness (with respect to a specification) does not always guarantee that a program has abstract security properties. Here, by linking a proof of cryptographic security to a proof of program correctness, we provide that guarantee.

Acknowledgments. Funded in part by DARPA award FA8750-12-2-029 and by a grant from Google ATAP.

References

[1] AFFELDT, R. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering (ISSE) 9*, 2 (2013), 59–77.

[2] AFFELDT, R., AND SAKAGUCHI, K. An intrinsic encoding of a subset of C and its application to TLS network packet processing. *Journal of Formalized Reasoning 7*, 1 (2014), 63–104.

[3] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., AND DUPRESSOIR, F. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications security* (2013), ACM, pp. 1217–1230.

[4] ALMEIDA, J. B., BARBOSA, M., FILLIÂTRE, J., PINTO, J. S., AND VIEIRA, B. CAOVerif: An open-source deductive verification platform for cryptographic software implementations. *Sci. Comput. Program. 91* (2014), 216–233.

[5] APPEL, A. W. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems 37*, 2 (Apr. 2015), 7:1–7:31.

[6] APPEL, A. W., DOCKINS, R., HOBOR, A., BERINGER, L., DODDS, J., STEWART, G., BLAZY, S., AND LEROY, X. *Program Logics for Certified Compilers*. Cambridge, 2014.

[7] APPEL, A. W., MICHAEL, N. G., STUMP, A., AND VIRGA, R. A trustworthy proof checker. *J. Automated Reasoning 31* (2003), 231–260.

[8] AVALLE, M., PIRONTI, A., AND SISTO, R. Formal verification of security protocol implementations: a survey. *Formal Asp. Comput. 26*, 1 (2014), 99–123.

[9] BACELAR ALMEIDA, J., BARBOSA, M., BANGERTER, E., BARTHE, G., KRENN, S., AND ZANELLA BÉGUELIN, S. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 488–500.

[10] BACKES, M., BARTHE, G., BERG, M., GRÉGOIRE, B., KUNZ, C., SKORUPPA, M., AND BÉGUELIN, S. Z. Verified security of Merkle-Damgård. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th* (2012), IEEE, pp. 354–368.

[11] BARBOSA, M., CASTRO, D., AND SILVA, P. F. Compiling CAO: from cryptographic specifications to C implementations. In *Principles of Security and Trust - Third International Conference, POST 2014, Proceedings* (2014), M. Abadi and S. Kremer, Eds., vol. 8414 of *Lecture Notes in Computer Science*, Springer, pp. 240–244.

[12] BARTHE, G., DUPRESSOIR, F., GRÉGOIRE, B., KUNZ, C., SCHMIDT, B., AND STRUB, P.-Y. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*. Springer, 2014, pp. 146–166.

[13] BARTHE, G., GRÉGOIRE, B., AND ZANELLA BÉGUELIN, S. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2009), POPL '09, ACM, pp. 90–101.

[14] BELLARE, M. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology-CRYPTO 2006*. Springer, 2006, pp. 602–619.

[15] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in CryptologyCRYPTO96* (1996), Springer, pp. 1–15.

[16] BELLARE, M., AND ROGAWAY, P. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptology ePrint Archive 2004* (2004), 331.

[17] BERGHOFER, S. Verification of dependable software using SPARK and Isabelle. In *6th International Workshop on Systems Software Verification, SSV 2011* (2011), J. Brauer, M. Roveri, and H. Tews, Eds., vol. 24 of *OASICS*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 15–31.

[18] BERNSTEIN, D. J. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, Proceedings* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *Lecture Notes in Computer Science*, Springer, pp. 207–228.

[19] BERNSTEIN, D. J. The HMAC brawl. cr.yp.to/talks/2012.03.20/slides.pdf, Mar. 2012.

[20] BERTOT, Y., MAGAUD, N., AND ZIMMERMANN, P. A proof of GMP square root. *J. Autom. Reasoning 29*, 3-4 (2002), 225–252.

[21] BHARGAVAN, K., FOURNET, C., KOHLWEISS, M., PIRONTI, A., AND STRUB, P. Implementing TLS with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 445–459.

[22] BLANCHET, B. A computationally sound mechanized prover for security protocols. *Dependable and Secure Computing, IEEE Transactions on 5*, 4 (2008), 193–207.

[23] CADÉ, D., AND BLANCHET, B. Proved generation of implementations from computationally secure protocol specifications. In *Principles of Security and Trust*. Springer, 2013, pp. 63–82.

[24] CHEN, Y., HSU, C., LIN, H., SCHWABE, P., TSAI, M., WANG, B., YANG, B., AND YANG, S. Verifying curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), G. Ahn, M. Yung, and N. Li, Eds., ACM, pp. 299–309.

[25] ERKOK, L., CARLSSON, M., AND WICK, A. Hardware/-software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer-Aided Design, 2009 (FMCAD'09)* (2009), IEEE, pp. 188–191.

[26] FILLIÂTRE, J. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program. 13*, 4 (2003), 709–745.

[27] Keyed-hash message authentication code. Tech. Rep. FIPS PUB 198-1, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, July 2008.

[28] Secure hash standard (SHS). Tech. Rep. FIPS PUB 180-4, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, Mar. 2012.

[29] GU, L., VAYNBERG, A., FORD, B., SHAO, Z., AND COSTANZO, D. CertiKOS: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), APSys'11, ACM, pp. 3:1–3:5.

[30] HALEVI, S. A plausible approach to computer-aided cryptographic proofs. http://eprint.iacr.org/2005/181, 2005.

[31] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (October 1969), 578–580.

[32] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[33] KOBLITZ, N., AND MENEZES, A. Another look at HMAC. *Journal of Mathematical Cryptology 7*, 3 (2013), 225–251.

[34] LEROY, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06* (2006), pp. 42–54.

[35] LEROY, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[36] MYREEN, M. O., AND CURELLO, G. Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Proceedings* (2013), G. Gonthier and M. Norrish, Eds., vol. 8307 of *Lecture Notes in Computer Science*, Springer, pp. 66–81.

[37] O'HEARN, P., REYNOLDS, J., AND YANG, H. Local reasoning about programs that alter data structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic* (Sept. 2001), pp. 1–19. LNCS 2142.

[38] PETCHER, A., AND MORRISETT, G. The foundational cryptography framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Proceedings* (2015), R. Focardi and A. C. Myers, Eds., vol. 9036 of *Lecture Notes in Computer Science*, Springer, pp. 53–72.

[39] SCHMALTZ, S. F. F. Formal verification of a big integer library including division. Master's thesis, Saarland University, 2007. busserver.cs.uni-sb.de/publikationen/Fi08DATE.pdf.

[40] SMITH, E. W., AND DILL, D. L. Automatic formal verification of block cipher implementations. In *Formal Methods in Computer-Aided Design (FMCAD'08)* (2008), IEEE, pp. 1–7.

[41] TOMA, D., AND BORRIONE, D. Formal verification of a SHA-1 circuit core using ACL2. In *Theorem Proving in Higher Order Logics*. Springer, 2005, pp. 326–341.