

# Formally proving equivalence between abstract and concrete specifications of HMAC

Katherine Ye, advised by Andrew Appel<sup>1</sup>

<sup>1</sup>Princeton University

The OpenSSL implementation of HMAC has been proven to correctly implement its concrete specification. HMAC uses SHA-256 as its hash function, and the SHA-256 program has also been proven to correctly implement its specification. At a higher level, HMAC has been proven “safe to use”: an abstract specification of HMAC has been proven to be a pseudo-random function given that its internal hash function is one as well. We bridge the gap between the abstract and the concrete HMAC spec by formally proving their equivalence. This proof transfers the desirable and necessary property of being a pseudo-random function (with some caveats) to both the concrete spec and the C implementation of HMAC, guaranteeing that the OpenSSL code is “safe to use.”  
TODO shorten abstract (December 26, 2014)

## I. INTRODUCTION

Compelling quote or hook here TODO

There exists a gap between mathematical cryptography (rigorous paper proofs of correctness of an algorithm) and applied cryptography (concrete implementations of those algorithms, plus the field of information security). This gap gives us two reasons to be wary. First, the paper proofs may be flawed, leading us to believe that algorithms are “safe to use” when they are not. (TODO example of wrong crypto proof) Second, even if the proofs are right, they are accompanied by a specification of the algorithm only on paper. The concrete code implementing this specification may contain exploitable bugs, allowing hackers to induce buffer overflows, ... TODO (TODO OpenSSL, Heartbleed)

Recent work aims to close this gap. Barthe (2013, 2014) assert that “cryptographic proofs have become essentially unverifiable,” and present CertiCrypt, a framework that “enables the machine-checked construction and verification of code-based proofs.” This paper continues in the spirit of formal verification.

At a high level, this work is motivated by two purposes. First, the Verified Software Toolchain project has built the framework to do TODO, and HMAC is a natural extension of the existing SHA-256 work. See Appel 2015 for an explanation of the system. Second, encrypted communications between military robots requires HMAC TODO.

Toward this end, we

The rest of this paper assumes introductory knowledge of cryptography and and no knowledge about proof assistants or formal verification.

### A. Formal verification in Coq

Coq is a proof assistant.

Coq has two internal languages, Gallina and the tactic language. Gallina is a purely functional language similar to OCaml. The tactic language is used for doing proofs and defining new proof strategies.

As used here, formal verification of a piece of code

Proof edifice:  
more abstract → less abstract

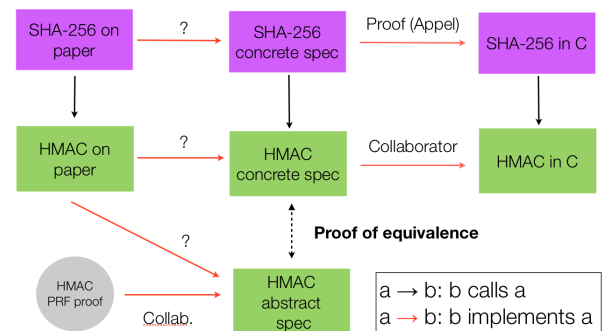


Figure 1. Mind the gap (the dotted arrow).

means proving that this implementation fulfills some kind of high-level specification. The code will usually be written in a low-level language such as C and may contain optimizations and other tricks. The specification, or “spec,” will usually be written in a high-level language such as OCaml or Gallina and is typically more mathematical and abstract. As Appel 2015 notes, “A program without a specification *cannot be incorrect*, it can only be surprising.”

For example, TODO sorting algorithm

TODO: insert diagram of what formal verification means

Coq has been used to

### B. The Merkle-Damgard construction

Say we have a strong cryptographic compression function TODO define. It has certain guarantees. However, it only operates on an input of a fixed length.

The Merkle-Damgard construction (referred to as “M-D” from now on) is a way to extend this function to inputs of any length by iterating the compression function on identically-sized, adjacent blocks of the input. It has been proven to uphold the desirable properties of C

such as

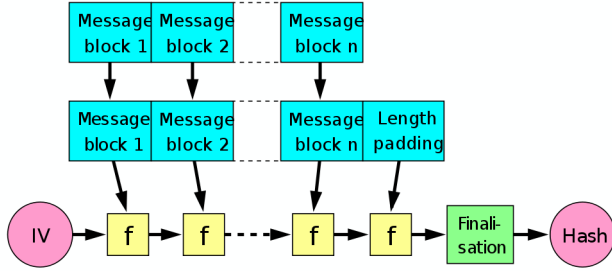


Figure 2. The M-D construction.

SHA-256 is an example of an application of M-D.  
Picture of crypto system

### C. SHA-256

SHA-256 is a cryptographic hash algorithm used for TODO. Mention OpenSSL.

It operates on a message of any length by breaking the message into 512-bit blocks. TODO (is this right?) It outputs a 256-bit digest.

Like all such hash functions, it comes with guarantees of pre-image resistance, second pre-image resistance, and collision resistance. Thus, it is very difficult for an adversary to change the input message without changing the digest.

Inner hash function  
PRF

### D. HMAC

SHA-256 provides only a guarantee of integrity; that is, a guarantee the message has not been tampered with. A message authentication code (MAC) is used to guarantee both integrity and authenticity, the latter meaning that the message's origin is the expected sender. Sender and receiver need only exchange a secret key before beginning their communication. In addition, whereas SHA-256 is vulnerable to length-extension attacks, HMAC is not.

To accomplish this, HMAC (a “hash-based message authentication code”) was designed in Bellare 1996. It includes a proof that the HMAC protocol (described below) is a pseudo-random function (PRF) on its inputs given that the underlying hash primitive is a PRF. In SHA-256, the underlying hash primitive would be its compression function.

To compute the authentication code of a message, RFC 2104 defines HMAC as the following action:

$$HMAC_{H,K}(m) = H((k \oplus opad) | H(k \oplus ipad | m))$$

where

- its block length is 512 bits, or 64 bytes,
- $H$  is a cryptographic hash function (here, *SHA-256*),
- $K$  is a secret key padded to the right with extra zeros to the input block size of the hash function, or the hash of the original key if it's longer than that block size,
- $m$  is the message to be authenticated,
- $|$  denotes concatenation,
- $\oplus$  denotes bitwise exclusive or (XOR),
- *opad* is the outer padding (the byte 0x5c repeated 64 times to be the length of one block),
- and *ipad* is the inner padding (0x36 repeated as above).

Note that formalizing HMAC is a natural extension of our work on SHA-256, since HMAC is not much more complicated than applying a cryptographic hash function twice.

OpenSSL includes an implementation of HMAC in C which calls the OpenSSL implementation of SHA-256.

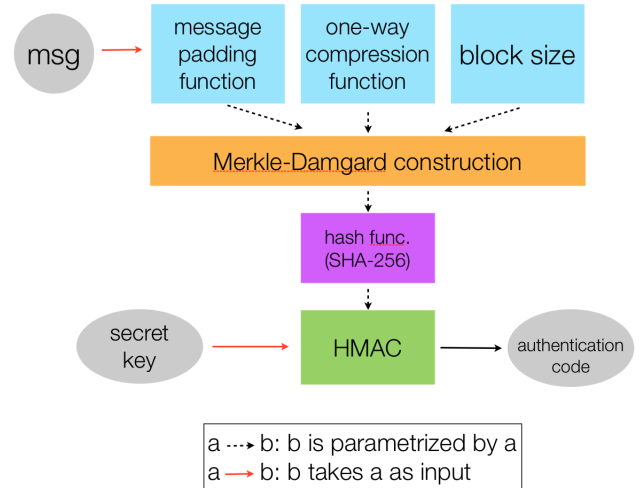


Figure 3. The entire system. It is slightly more correct to think of a “black box” HMAC taking message and key as input, and outputting the authentication code.

### E. Prior work

Verification of SHA-256.

## II. THE PROOF OF EQUIVALENCE

### A. The concrete HMAC spec

This spec was written to conform to RFC 2104; thus, it contains runnable code, operates on byte lists, and returns a byte list. The spec distinguishes between the *Z* type and the *byte* type (values of type *Z* in range  $[0, 256]$ ); however, we will use *Z* synonymously with *byte* with the understanding that all values are in range.

The spec is constructed to work with generic cryptographic hash functions. We will instantiate it with the *SHA-256* functional program, which we treat as a black box that takes care of the message padding and compression function iteration.

The code for this spec and the next is in the Appendix.

### B. The abstract HMAC spec

This spec was written to conform to the HMAC protocol defined in Bellare 1996; thus, to be as general as possible, it operates on bit vectors and returns a bit vector. (A bit vector *Bvector n* is a type dependent on a natural number value *n*, the length of the vector.)

It defines HMAC via the two-keyed HMAC (*HMAC.2K*) and generalized HMAC (*GHMAC.2K*) structures, rather than straightforwardly as the concrete spec does. It also includes an implementation of generalized NMAC (*GNMAC*), another structure used in the proof.

The spec does not contain runnable code as-is because it leaves several parameters abstract:

```
(* c is the output size, b is the block size
   (larger than the output size),
   and p is the difference between them *)
Variable c p : nat.
Definition b := c + p.

(* 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.
Definition app_fpad (x : Bvector c) : Bvector b :=
  (Vector.append x (fpad x)).

Variable opad ipad : Bvector b.
```

The proof depends on the following assumptions that are explicit in the spec:

1. the key is of the right length (one block)
2. *opad*  $\neq$  *ipad* (they differ in at least one bit)
3. the padding function *splitAndPad* is one-to-one

4. the hash function (e.g. *SHA-256*) is an iterated version of the compression function, a la Merkle-Damgard.

as well as other implicit assumptions explained in Section III.

The fourth assumption can be seen in this definition of the *SHA-256* analogue, *hash\_words*:

```
(* The iteration of the compression function gives
   a keyed hash function on lists of words. *)
Definition h_star k (m : list (Bvector b)) :=
  fold_left h m k.

(* The composition of the keyed hash function with
   the IV gives a hash function on lists of
   words. *)
Definition hash_words := h_star iv.
```

However, *SHA-256* includes a padding function for the message, while this spec's use of dependent types (*Bvector n*) forces the use of two types of ad-hoc padding, the functions *app\_fpad* and *splitAndPad*.

```
Definition GHMAC_2K (k : Bvector (b + b)) 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)
  .
```

### C. Proof outline

Equivalence of specs means

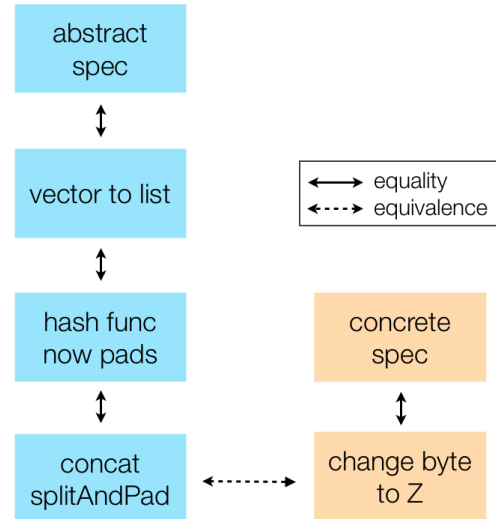


Figure 4. Stages of the equivalence/equality proofs.

Again, the main differences between the specs are that (TODO make a table)

1. *GNMAC* and *GHMAC.2K* structures

2. the abstract spec operates on bits, whereas the concrete spec operates on bytes
3. the abstract spec uses the dependent type `Bvector n`, which is a bit list of length  $n$ , whereas the concrete spec uses byte lists and int lists
4. the abstract spec pads its input twice in an ad-hoc manner, whereas the concrete spec uses the SHA-256 padding function consistently
5. the SHA-256 concrete spec uses the  $Z$  type, whereas the HMAC concrete type uses the byte type, which is  $Z$  constrained to be in  $[0, 255]$ .

We solve some of them by massaging the two specs in the “same direction.”

#### D. Bridging bytes and bits

We would like to prove that the concrete and abstract HMAC specs ( $HMAC_c$  and  $HMAC_a$ ) are extensionally equal. That is, equal inputs always result in equal outputs:

$$k_c = k_a \rightarrow m_c = m_a \rightarrow$$

$$HMAC_c(k_c, m_c) = HMAC_a(k_a, m_a).$$

However,  $HMAC_c$  operates on bits (in fact, vectors of bits) and  $HMAC_a$  operates on bytes (lists of bytes). So the statement  $k_c = k_a$  (for example) does not have meaning because the types are different. To solve this, we generalize equality to equivalence. 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. Both definitions will turn out to be useful. (From now on, we will use  $b$  to name a value of type *Blist* (list of booleans, or bits) and  $B$  to name a value of type *list Z* (list of bytes).)

The computational relation is

$$b \approx_c B :=$$

$$b = \text{bytesToBits } B,$$

where  $\text{bytesToBits} : \text{list } Z \rightarrow \text{Blist}$  is a conversion function.

The inductive relation is

$$b \approx_i B :=$$

$$\text{bytes\_bits\_lists } b B.$$

where  $\text{bytes\_bits\_lists} : \text{Blist} \rightarrow \text{list } Z \rightarrow \text{Prop}$  creates an “assertion” that  $b$  and  $B$  are related in this way, together with two constructors used to provide evidence to prove the assertion. (For more on inductively defined propositions, see the *Prop* chapter in the Coq textbook “Software Foundations.”)

```
Inductive bytes_bits_lists : Blist -> list Z -> Prop
:=
| eq_empty : bytes_bits_lists nil nil
| eq_cons : forall (bits : Blist) (bytes : list Z)
  (b0 b1 b2 b3 b4 b5 b6 b7 : bool)
  (byte : Z),
  bytes_bits_lists bits bytes ->
  convertByteBits [b0; b1; b2; b3; b4;
    b5; b6; b7] byte ->
  bytes_bits_lists (b0 :: b1 :: b2 ::
    b3 :: b4 :: b5 :: b6 :: b7 ::
    bits)
    (byte :: bytes).
```

The inductive and computational definitions have been proven equivalent in the sense that

$$\text{bytes\_bits\_lists } b B \leftrightarrow b = \text{bytesToBits } B.$$

For several other ways to convert between the definitions, see *ByteBitRelations.v*.

#### E. Instantiating the abstract specification

We instantiate the block sizes and wrap the concrete functions in *byteToBit* and/or *intlist\_to\_Zlist* conversion functions. The latter is necessary because portions of the *SHA-256* spec operate on lists of *Integers* (four bytes, or  $Z$ , combined into 32 bits), as specified in FIPS Pub. 180-2.

Definition c := (SHA256.DigestLength \* 8)%nat.

Definition p := (32 \* 8)%nat.

```
Definition intsToBits (l : list int) : list bool
:=
  bytesToBits (SHA256.intlist_to_Zlist l).
```

```
Definition bitsToInts (l : Blist) : list int :=
  SHA256.Zlist_to_intlist (bitsToBytes l).
```

```
Definition sha_iv : Blist :=
  intsToBits SHA256.init_registers.
```

```
Definition sha_h (regs : Blist) (block : Blist) :
  Blist :=
  intsToBits (SHA256.hash_block (bitsToInts regs)
    (bitsToInts block)).
```

```
Definition sha_splitandpad (msg : Blist) : Blist
:=
  bytesToBits (sha_padding_lemmas.pad (bitsToBytes
    msg)).
```

Note that we are essentially converting the type of the values from *intlist*  $\rightarrow \dots \rightarrow \text{intlist}$  to *Blist*  $\rightarrow \dots \rightarrow \text{Blist}$  by converting their inputs and outputs.

We can use the computational equivalence relation, instantiated with a generic conversion function, to reason abstractly about the behavior of such wrapped functions.

$$x \approx_c X :=$$

$$x = \text{convert\_BA } X.$$

1. roundtrip
2. iteration

We reasoned about the bytes/bits proof via the following framework.

- (Round-trip and conversion properties. Galois connections or other algebraic structure?)
- Wrapped function equivalence on application. Interestingly, we have equivalence when a “round-trip” of composing the two transportation functions results in the identity function.
- Wrapped function equivalence on repeated application: depends on the previous one, and is completed by induction on the number of applications. This corresponds to the SHA-256 operation of hashing blocks.

This is the usefulness of the computational definition.

## F. Main lemmas

Now, the usefulness of the inductive definition

It allows us to breaking it up into 4 main lemmas: main proof

## G. Other proof techniques

The following techniques may be useful in future equivalence proofs.

- Many theorems are true for lists of any length (e.g. some involving map and zip). We found it difficult to do dependent type induction. Instead, we found it easy to prove theorems by induction on a Blist, implying that the list may be any length, then specialize it to Bvector *n*, a Blist of length *n*.
- Likewise, we found it easier to prove theorems about lists of any length (or a certain length given by an assumption), then prove that the functions involved preserve the length. This is equivalent to working with Bvector *n*.

- When dealing with lists whose lengths must be a multiple of a block size (e.g. 512 bits or 64 bytes), we found it useful to define an inductive proposition InWords *n* 1 that would allow one to do proofs by inducting in the block size, cons'ing elements to the front. We then proved this equivalent to the computational version (using the length function).
- Likewise, we found it useful to define two things: a function to compute conversions between byte lists and bit lists, and an inductive proposition stating that the lists correspond in this way.
- When it comes to theorems that involve tricky math, we exploited the fact that range of a byte is [0,255] and proved them by brute force instead. The same technique also works in reverse. Proving something true for 8 booleans is as simple as checking that the statement is true for each of the 256 cases.

## H. Problems encountered

- We found it difficult to work with dependent types, induction, and John Major equality.
- We encountered problems converting between many machine representations: byte/Z, byte/bit, int/Z, and even little-endian vs. big-endian.
- We did not find much prior work on this sort of equivalence proof, except for related functions in Coq.Strings.Ascii.

## III. THE PROOF OF HMAC'S SAFETY

### A. The 1996 (?) Bellare proof

Let *f* be the compression function and *f\** the iteration of *f*.

Given the assumption that *f* is a pseudo-random function (PRF), that implies that *f\** is computationally almost universal (*cAU*). *cAU* is a slightly weaker property than collision-resistant.

The *f\*cAU* property is used to prove that generalized NMAC (GNMAC) using *f\** is a PRF. The GNMACH property is used to prove that generalized HMAC with two keys (GHMAC\_2k) using *f\** is a PRF.

Using that fact and the assumption that *f\** is RkA-resistant (?), we prove that GHMAC (single-keyed) using *f\** is a PRF. This, plus the fact that the padding function for *f* is one-to-one, leads finally to the proof that HMAC using *f\** is a PRF.

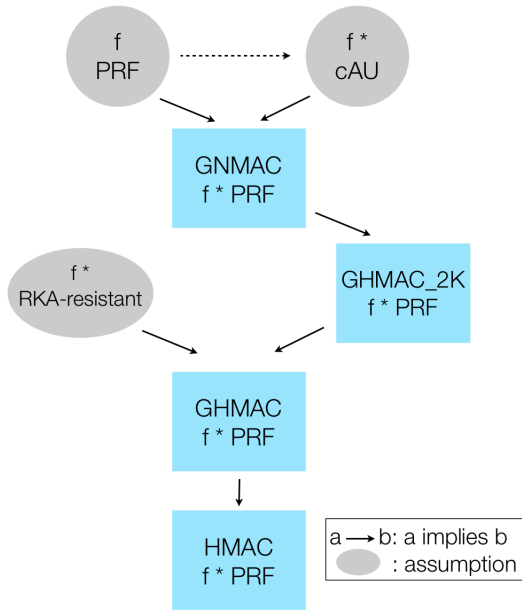


Figure 5. The structure of the 1996 (?) Bellare HMAC proof.

### B. Formalization in Coq

See Adam Petcher’s paper on MCF. Random oracles, monadic syntax.

### C. Transfer of properties

In order for the the proof to hold, the following assumptions must be true:

- the padding function (which? TODO) is one-to-one
- opad and ipad differ in at least one bit.
- (anything else? TODO)

Examining the final spec, these are true. TODO

Thus, the property TODO transfers. (termination and well-definedness of C code? Adam’s email?)

Interfaces that drop out of the proof are TODO, leaving an end-to-end proof of correctness.

## IV. CONCLUSION

Contribution to cryptography: bridging the gap between concrete and abstract HMAC specs allows the desirable property of being a PRF to transfer to the OpenSSL implementation of HMAC. So it’s “safe to use.”

Contribution to reasoning about equivalence: many bytes/bits computational and inductive definitions. Lemmas for correspondence between them.

Contribution to proof techniques: results of trying to use dependent types. Brute force. Inductive propositions

for length, plus InBlocks correspondence with division, mod, and existence.

### A. Future work

Dependent types, type-level programming in Coq.

Libraries could be written to aid reasoning about equivalence and lengths.

More crypto formalization! EasyCrypt, CertiCrypt, IMDEA. MCF

SHA PRF, of course

How to transmit HMAC key? Need RSA. Work being done at Princeton.

## ACKNOWLEDGMENTS

Andrew Appel, Lennart Beringer, Adam Petcher, and Qinxiang Cao.

## V. REFERENCES

- RFC 2104 (HMAC, 1997):  
<https://tools.ietf.org/html/rfc2104>  
 FIPS Publication 180-4 (Secure Hash Standard, containing SHA-256):  
<http://csrc.nist.gov/publications/PubsFIPS.html>  
 “Verification of SHA-256,” Appel 2015  
 “New Proofs for NMAC and HMAC: Security without Collision-Resistance,” Bellare (1996)  
 “Keying Hash Functions for Message Authentication,” Bellare (2004)  
 “MCF,” Petcher (unpublished)  
 Barthe 2009  
 “Merkle-Damgard in EasyCrypt,” IMDEA  
 “Certified Programming with Dependent Types,” Chlipala  
 “Software Foundations,” Pierce et al.  
 “Verified Functional Algorithms,” Appel, OPLSS  
 “HMAC” Wikipedia page  
 “SHA-256” Wikipedia page  
 Coq.Strings.Ascii  
 OpenSSL HMAC and SHA-256 code

## VI. APPENDIX

### A. The abstract HMAC spec

Set Implicit Arguments.

Require Import Bvector.

Require Import List.

Require Import Arith.

Definition Blist := list bool.

```
Fixpoint splitVector(A : Set)(n m : nat) : Vector.t
  A (n + m) -> (Vector.t A n * Vector.t A m) :=
  match n with
  | 0 =>
    fun (v : Vector.t A (0 + m)) => (@Vector.nil A
      , v)
  | S n' =>
    fun (v : Vector.t A (S n' + m)) =>
      let (v1, v2) := splitVector _ _ (Vector.tl v
        ) in
      (Vector.cons _ (Vector.hd v) _ v1, v2)
  end.
```

Section HMAC.

(\* c is the output size, b is the block size (
 larger than the output size),
 and p is the difference between them \*)

Variable c p : nat.

Definition b := c + p.

(\* The compression function \*)

Variable h : Bvector c -> Bvector b -> Bvector c.

(\* The initialization vector is part of the spec
 of the hash function. \*)

Variable iv : Bvector c.

(\* The iteration of the compression function gives
 a keyed hash function on lists of words. \*)

Definition h\_star k (m : list (Bvector b)) :=
 fold\_left h m k.

(\* The composition of the keyed hash function with
 the IV gives a hash function on lists of
 words. \*)

Definition hash\_words := h\_star iv.

Variable splitAndPad : Blist -> list (Bvector b).

Hypothesis splitAndPad\_1\_1 :

```
forall b1 b2,
  splitAndPad b1 = splitAndPad b2 ->
  b1 = b2.
```

Variable fpad : Bvector c -> Bvector p.

Definition app\_fpad (x : Bvector c) : Bvector b :=
 (Vector.append x (fpad x)).

Definition h\_star\_pad k x :=
 app\_fpad (h\_star k x).

Definition GNMAC k m :=

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

(\* The "two-key" version of GHMAC and HMAC. \*)

Definition GHMAC\_2K (k : Bvector (b + b)) 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 : Bvector (b + b)) (m :
 Blist) :=
 GHMAC\_2K k (splitAndPad m).

(\* opad and ipad are constants defined in the HMAC
 spec. \*)

Variable opad ipad : Bvector b.

Hypothesis opad\_ne\_ipad : opad <> ipad.

Definition GHMAC (k : Bvector b) :=

```
GHMAC_2K (Vector.append (BVxor _ k opad) (BVxor
  _ k ipad)).
```

Definition HMAC (k : Bvector b) :=

```
HMAC_2K (Vector.append (BVxor _ k opad) (BVxor _
  k ipad)).
```

End HMAC.

## B. The concrete HMAC spec

Require Import Integers.

Require Import Coqlib.

Require Import Coq.Strings.String.

Require Import Coq.Strings.Ascii.

Require Import List. Import ListNotations.

(\*SHA256: blocksize = 64bytes

corresponds to

#define SHA\_LBLOCK 16

#define SHA256\_CBLOCK (SHA\_LBLOCK\*4) \*)

Module Type HASH\_FUNCTION.

Parameter BlockSize:nat. (\*measured in bytes; 64
 in SHA256\*)

Parameter DigestLength: nat. (\*measured in bytes;
 32 in SHA256\*)

Parameter Hash : list Z -> list Z.

End HASH\_FUNCTION.

Module Type HMAC\_Module.

Parameter HMAC: byte -> byte -> list Z -> list Z
 -> list Z.

End HMAC\_Module.

Module HMAC\_FUN (HF:HASH\_FUNCTION) <: HMAC\_Module.

Fixpoint Nlist {A} (i:A) n: list A:=

```
match n with 0 => nil
| S m => i :: Nlist i m
end.
```

Definition sixtyfour {A} (i:A): list A:= Nlist i HF.
 BlockSize.

(\*Reading rfc4231 reveals that padding happens on
 the right\*)

Definition zeroPad (k: list Z) : list Z :=
 k ++ Nlist Z0 (HF.BlockSize-length k).

Definition mkKey (l:list Z) : list Z :=
 if Z.gtb (Zlength l) (Z.of\_nat HF.BlockSize)
 then (zeroPad (HF.Hash l))
 else zeroPad l.

Definition mkArg (key:list byte) (pad:byte): list

```

byte :=
  (map (fun p => Byte.xor (fst p) (snd p))
    (combine key (sixtyfour pad))).
Definition mkArgZ key (pad:byte): list Z :=
  map Byte.unsigned (mkArg key pad).
(*
Definition Ipad := P.Ipad.
Definition Opad := P.Opad.
*)
(*innerArg to be applied to message, (map Byte.repr
  (mkKey password)))*)
Definition innerArg IP (text: list Z) key : list Z
:=
  (mkArgZ key IP) ++ text.

Definition INNER IP k text := HF.Hash (innerArg IP
  text k).

Definition outerArg OP (innerRes: list Z) key: list
  Z :=
  (mkArgZ key OP) ++ innerRes.

Definition OUTER OP k innerRes := HF.Hash (outerArg
  OP innerRes k).

Definition HMAC IP OP txt password: list Z :=
  let key := map Byte.repr (mkKey password) in
  OUTER OP key (INNER IP key txt).

End HMAC_FUN.

Require Import SHA256.
Require Import functional_prog.

Module SHA256_ <: HASH_FUNCTION.
  Definition BlockSize:= 64%nat.
  Definition DigestLength:= 32%nat.
  Definition Hash : list Z -> list Z := SHA_256'.
End SHA256_.

Module HMAC_SHA256 := HMAC_FUN SHA256_.

Definition Ipad := Byte.repr 54. (*0x36*)
Definition Opad := Byte.repr 92. (*0x5c*)

Definition HMAC256 := HMAC_SHA256.HMAC Ipad Opad.

Definition HMACString (txt passwd:string): list Z :=
  HMAC256 (str_to_Z txt) (str_to_Z passwd).

Definition HMACHex (text password:string): list Z :=
  HMAC256 (hexstring_to_Zlist text) (
    hexstring_to_Zlist password).

Definition check password text digest :=
  listZ_eq (HMACString text password) (
    hexstring_to_Zlist digest) = true.

```

### C. Definitions

```

Definition c:nat := (SHA256_.DigestLength * 8)%nat.
Definition p:=(32 * 8)%nat.

```

```

Definition sha_iv : Blist :=
  bytesToBits (SHA256.intlist_to_Zlist SHA256.
    init_registers).

Check SHA256.hash_blocks. (* SHA256.registers ->
  list int -> SHA256.registers *)
Definition sha_h (regs : Blist) (block : Blist) :
  Blist :=
  bytesToBits (SHA256.intlist_to_Zlist
    (SHA256.hash_block (SHA256.
      Zlist_to_intlist (bitsToBytes
        regs))
      (SHA256.
        Zlist_to_intlist
          (bitsToBytes
            block))
    )).

Definition sha_splitandpad (msg : Blist) : Blist :=
  bytesToBits (sha_padding_lemmas.pad (bitsToBytes
    msg)).

Definition convert (l : list int) : list bool :=
  bytesToBits (SHA256.intlist_to_Zlist l).

Definition convertByteBits (bits : Blist) (byte : Z)
  : Prop :=
  exists (b0 b1 b2 b3 b4 b5 b6 b7 : bool),
  bits = [b0; b1; b2; b3; b4; b5; b6; b7] /\
  byte = (1 * (asZ b0) + 2 * (asZ b1) + 4 * (asZ b2)
    ) + 8 * (asZ b3)
    + 16 * (asZ b4) + 32 * (asZ b5) + 64 * (asZ
      b6) + 128 * (asZ b7)).

Inductive bytes_bits_lists : Blist -> list Z -> Prop
:=
  | eq_empty : bytes_bits_lists nil nil
  | eq_cons : forall (bits : Blist) (bytes : list Z)
    (b0 b1 b2 b3 b4 b5 b6 b7 : bool)
    (byte : Z),
    bytes_bits_lists bits bytes ->
    convertByteBits [b0; b1; b2; b3; b4;
      b5; b6; b7] byte ->
    bytes_bits_lists (b0 :: b1 :: b2 ::
      b3 :: b4 :: b5 :: b6 :: b7 ::
        bits)
      (byte :: bytes).

```

### D. The equivalence proof

```

* unfold sha_splitandpad.
  unfold convert.
  rewrite -> pure_lemmas.
    Zlist_to_intlist_to_Zlist.
  f_equal.
  apply bytes_bits_ind_comp in input_eq.
  rewrite -> input_eq.
  reflexivity.
+ admit. (* bytes in range *)
+
  pose proof pad_len_64_nat bytes as
    pad_len_64.
  destruct pad_len_64.

```



```

rewrite -> H.
assert (four : Z.to_nat WORD = 4%nat) by
  reflexivity.
rewrite -> four.
exists (x * 16)%nat.
omega.
+ admit. (* padding in
  range *)

* unfold sha_iv. reflexivity.
Qed.

(* ----- *)

(* MAIN THEOREM *)

Theorem HMAC_spec_equiv : forall
  (K M H : list Z) (OP IP :
    Z)
  (k m h : Blist) (op ip :
    Blist),
  ((length K * 8)%nat = (c + p)%nat ->
  Zlength K = Z.of_nat SHA256.BlockSize ->
  (* TODO: first implies this *)
  (* TODO: might need more hypotheses about lengths
    *)
  bytes_bits_lists k K ->
  bytes_bits_lists m M ->
  bytes_bits_lists op (HMAC_SHA256.sixtyfour OP) ->
  bytes_bits_lists ip (HMAC_SHA256.sixtyfour IP) ->
  HMAC c p sha_h sha_iv sha_splitandpad op ip k m =
    h ->
  HMAC_SHA256.HMAC IP OP M K = H ->
  bytes_bits_lists h H.
Proof.
  intros K M H OP IP k m h op ip.
  intros padded_key_len padded_key_len_byte
    padded_keys_eq msgs_eq ops_eq ips_eq.
  intros HMAC_abstract HMAC_concrete.

  intros.
  unfold p, c in *.
  simpl in *.

  rewrite <- HMAC_abstract. rewrite <- HMAC_concrete
  .

  unfold HMAC. unfold HMAC_SHA256.HMAC. unfold
    HMAC_SHA256.OUTER. unfold HMAC_SHA256.INNER.
  unfold HMAC_SHA256.outerArg. unfold HMAC_SHA256.
    innerArg.

  unfold HMAC_2K. unfold GHMAC_2K. rewrite ->
    split_append_id.

```

```

simpl.

(* Major lemmas *)
apply SHA_equiv_pad.
apply concat_equiv.
apply xor_equiv_Z; try assumption.

*
  apply SHA_equiv_pad.
  apply concat_equiv.

- apply xor_equiv_Z; try assumption.
- assumption.
(* xors preserve length *)
*
  unfold b in *. simpl. unfold BLxor. rewrite ->
    list_length_map.
  rewrite -> combine_length.
  pose proof bytes_bits_length op (HMAC_SHA256.
    sixtyfour OP) as ops_len.
  rewrite -> ops_len.
  pose proof bytes_bits_length k K as keys_len.
  rewrite -> keys_len.
  rewrite -> padded_key_len.
  unfold HMAC_SHA256.sixtyfour.
  rewrite -> length_list_repeat.
  reflexivity.
  apply padded_keys_eq.
  apply ops_eq.

*
  unfold b in *. simpl. unfold BLxor. rewrite ->
    list_length_map.
  rewrite -> combine_length.
  pose proof bytes_bits_length ip (HMAC_SHA256.
    sixtyfour IP) as ips_len.
  rewrite -> ips_len.
  pose proof bytes_bits_length k K as keys_len.
  rewrite -> keys_len.
  rewrite -> padded_key_len.
  unfold HMAC_SHA256.sixtyfour.
  rewrite -> length_list_repeat.
  reflexivity.
  apply padded_keys_eq.
  apply ips_eq.

Qed.

```

## E. Selected theorems and proofs

See the repository at  
[github.com/hypotext/vst-crypto](https://github.com/hypotext/vst-crypto).