Formally proving equivalence between abstract and concrete specifications of HMAC

Katherine Ye, advised by Andrew Appel[1]

[1] *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

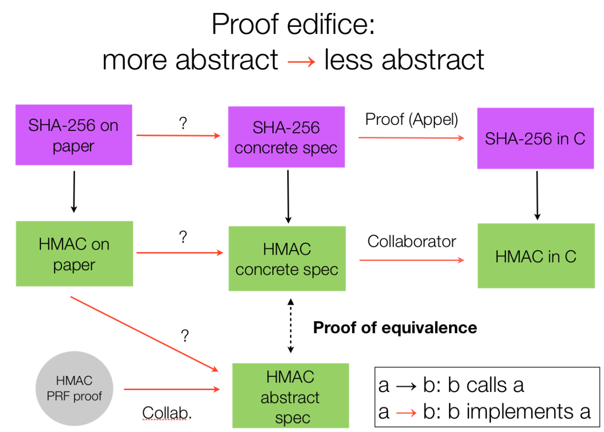

Figure 1. Mind the gap (the dotted arrow).

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

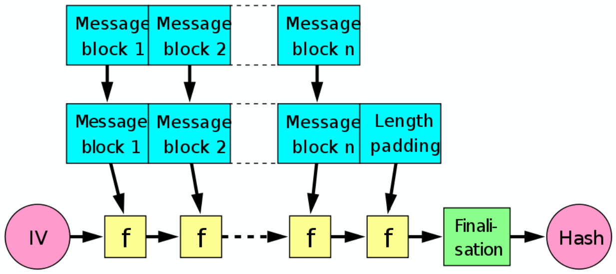

Figure 2. The M-D construction.

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.

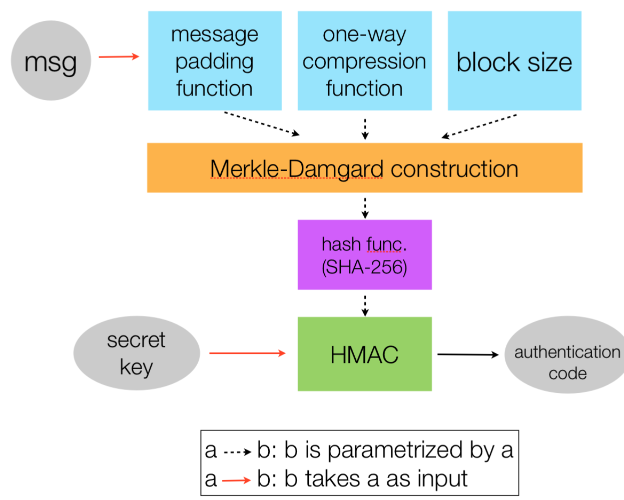

Figure 3. The entire system. It is slightly more correct to think of a "black box" HMAC taking message and key as input, and outputting the authentication code.

E. Prior work

Verification of SHA-256.

II. A TALE OF TWO SPECS

A. The concrete HMAC spec

This spec was written to conform to RFC 2104 (the government's English description of the algorithm); 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)
        .

Definition HMAC_2K (k : Bvector (b + b)) (m : Blist)
    :=
  GHMAC_2K k (splitAndPad m).

Definition HMAC (k : Bvector b) :=
  HMAC_2K (Vector.append (BVxor _ k opad) (BVxor _
       k ipad)).
```

fpad is used to pad the output size c to the block size b. *splitAndPad* is used to split the variable-length message (of type *Blist*) into a list of blocks, each size b, padding it along the way.

C. Proof outline

There are six main differences between the concrete and abstract specs:

1. the abstract spec operates on bit lists, whereas the concrete spec operates on byte lists.
2. 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.
3. due to its use of dependent types, the abstract spec pads 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.
5. the abstract spec does an explicit fold over the message, which is now a list of blocks, not a list of pure bits.
6. the abstract spec defines HMAC via the HMAC_2K and GHMAC_2K structures, not directly.

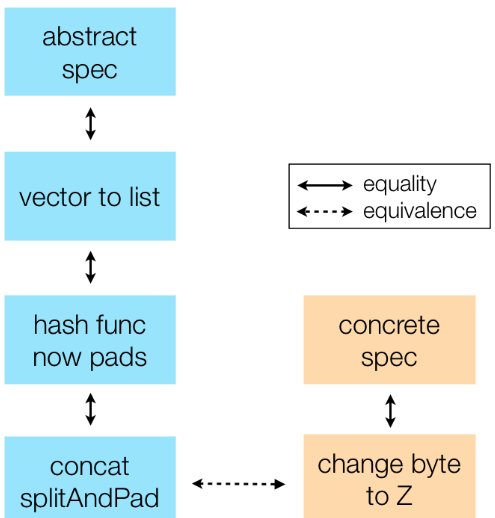

Figure 4. Stages of the equivalence/equality proofs.

They are ranked in rough order of most difficult to least difficult. No difficulty is too great to be overcome–we believe the specs are not "fundamentally different" and that they can be linked via proof. Indeed, a goal of this project was to find and remove differences in the process of proof.

The last, 6, is easily resolved by unfolding the definitions of HMAC_2K and GHMAC_2K. We solve the other five by changing definitions and massaging the two specs in the "same direction," proving equality each time.

5 is resolved by concatenating the *list* (*Bvector b*) in *hash_words* below. Then it can be seen that iterating *firstn b* and *splitn b* over the concatenated list is equal to left-folding over the list of blocks.

```
Function hash_blocks_bits
        (hash_block_bit : Blist -> Blist -> Blist)
            (r: Blist)
        (msg: Blist) {measure length msg} : Blist
            :=
  match msg with
  | nil => r
  | _ => hash_blocks_bits
          hash_block_bit
          (hash_block_bit r (firstn 512 msg)) (skipn
               512 msg)
  end.
```

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

Definition hash_words : list (Bvector b) ->
    Bvector c := h_star iv.
```

4 is not too difficult. Its explicit fold (as iteration method) can be proven equal to the iteration method using *firstn* and *splitn* in the concrete SHA-256 spec, as seen in 3 above. It suffices to convert the SHA_256 initialization vector and wrap its internal compression function. See 1 and the next section for an elaboration on conversion and wrapping.

Before iterating the compression function on the message, SHA-256 pads it in a standard, one-to-one fashion so its length is a multiple of the block size. It pads it as such:

$$msg \mid [1] \mid [0, 0, \ldots 0] \mid [l_1, l_2]$$

where $\mid$ denotes concatenation and $[l_1, l_2]$ denote the two digits of the length of the message as a 64-bit integer. The number of 0s is calculated such that the length of the entire padded message is a multiple of the block size.

Thus, 3 is resolved by rewriting the abstract spec to incorporate *fpad* and *splitAndPad* into a single padding function included in the hash function, much like SHA-256 does.

$$hash\_words\_padded :=$$
$$hash\_words \circ split\_and\_pad$$

As summarized in the previous section, *fpad* is used to pad the output size c to the block size b. *splitAndPad* is used to split the variable-length message (of type *Blist*) 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 (length of one block) will be pre-pended to it later.

2 is resolved by changing all *Bvector n* to *Blist*, 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.

1 is the most difficult; since the types of each HMAC function differ, it requires an equivalence proof rather than an equality proof. It is discussed in the next section.

A minor difference also exists between the SHA-256 concrete spec, which uses the Z type, and the HMAC concrete spec which uses the *byte* type (range-constrained Z). This is resolved by asserting and proving that $\forall x : Z, x \in [0, 255]$ wherever needed.

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:

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

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*.

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

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

$$b \approx_c B :=$$
$$b = bytesToBits\ B,$$

where $bytesToBits : listZ \to Blist$ is a conversion function.

The inductive relation is

$$b \approx_i B :=$$
$$bytes\_bits\_lists\ b\ B.$$

where $bytes\_bits\_lists : Blist \to list\ Z \to Prop$ 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

$$bytes\_bits\_lists\ b\ B \leftrightarrow b = bytesToBits\ B.$$

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 $Integer$s (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 $intlist \to ... \to intlist$ to $Blist ... \to ... Blist$ by converting their inputs and outputs.

We can use the computational equivalence relation defined earlier ($\approx_c$), instantiated with a generic conversion function, to reason abstractly about the behavior of such wrapped functions. Let's define the framework as such (letting $. = \circ$, function composition):

```
(* B corresponds to bytes, A corresponds to bits *)
Parameter A B : Type.
Parameter convert_BA : B -> A.
Parameter convert_AB : A -> B.

Definition wrap (F : B -> B) : A -> A :=
  convert_BA . F . convert_AB.
```

Note that the types B and A are not symmetric, in the sense that

Define two relations as such:

$$x \approx_c X :=$$
$$x = convert\_BA\ X.$$

$$f \approx_w F :=$$
$$f = wrap\ F.$$

We ask: what assumptions are needed such that application of equivalent (via wrapping) functions to equivalent (via conversion) inputs result in equivalent (via conversion) outputs?

$$\begin{gathered}\textit{Lemma once\_eq}:\\ \forall\,(x:A)\,(X:B)\,(f:A\to A)\,(F:B\to B),\\ x\approx_c X\to\\ f\approx_w F\to\\ f\ x\approx_c F\ X\end{gathered}$$

The necessary and sufficient assumption (together with the other two assumptions) is that we have output equivalence exactly when a "round-trip" of composing the two conversion functions results in the identity function. (The reader is invited to finish this short proof.)

$$\textit{Lemma roundtrip}: \textit{convert\_AB}\circ\textit{convert\_BA} = \textit{id}$$

Indeed, we have proven that $\textit{bitsToBytes}\circ\textit{bytesToBits} = \textit{id}$. Note that it does not hold the other way around. $\textit{bytesToBits}\circ\textit{bitsToBytes}$ is false if the length of the bit list input is not a multiple of 8.

A natural extension is to prove output equivalence on repeated application of the wrapped functions, or iteration:

```
Fixpoint iterate {A : Type} (n : nat) (f : A -> A) (
    x : A) :=
  match n with
    | O => x
    | S n' => f (iterate n' f x)
  end.
```

$$\begin{gathered}\textit{Lemma iterate\_equiv}:\\ \forall\,(x:A)\,(X:B)\\ (f:A\to A)\,(F:B\to B)\,(n:\textit{nat}),\\ x\approx_c X\to\\ f\approx_w F\to\\ \textit{iterate}\ n\ f\ x=\textit{convert\_BA}\ (\textit{iterate}\ n\ F\ X).\end{gathered}$$

Here we need both *once_eq* and our newly admitted assumption *roundtrip*, which can be rephrased as $\forall (X:B), X = \textit{roundtrip}\ X$. The proof is completed by induction on n.

This framework is not just an academic exercise. *once_equiv* and *iterate_equiv* correspond directly to several lemmas in the bytes/bits equivalence proof. In particular, *iterate* corresponds directly to the SHA-256 operation of hashing blocks.

```
Function hash_blocks (r: registers) (msg: list int)
    {measure length msg} : registers :=
  match msg with
  | nil => r
  | _ => hash_blocks (hash_block r (firstn 16 msg))
      (skipn 16 msg)
  end.
```

The proof of equivalence works the same way. One can almost directly substitute the parameters into *iterate_equiv*.

$$\begin{gathered}\textit{Lemma fold\_equiv\_blocks}:\\ \forall(l:\textit{Blist})\,(\textit{acc}:\textit{Blist})\,(L:\textit{list int})\,(\textit{ACC}:\textit{list int}),\\ \textit{InBlocks}\ 16\ L\to\\ l\approx_c L\to\\ \textit{acc}\approx_c \textit{ACC}\to\\ \textit{hash\_blocks\_bits sha\_h acc}\ l\ \approx_c \textit{SHA}256.\textit{hash\_blocks ACC}\ L.\end{gathered}$$

The only differences are that the functions in question take multiple inputs, the length of L must be a multiple of 16 (the block size for the *Integer* type, 512 bits), and the method of iteration is slightly modified to be parametrized by the length of the input list (*firstn*, *skipn*). *sha_h* contains the wrapped *SHA*256.*hash_block* function. *hash_blocks_bits* is a version of *SHA*256.*hash_blocks* modified to use 512 as its block size. The code for the proof is in *HMAC_spec_concat.v*.

One wonders whether this "one-way roundtrip" property has been formalized elsewhere, perhaps in a Coq abstract algebra library. We found that the concepts of Galois connections and setoids were similar, but not useful.

In sum, the computational relation $\approx_c$ makes abstract reasoning easy precisely because it is computational, so one can rewrite using equality in proofs. It can loosely be called "computationally compositional" since we start with a If no conversion function existed, one could use the inductive relation $\approx_i$, but it does not easily allow rewrites.

F. The proof of equivalence

The main equivalence theorem is as follows. The symbol $\approx$ will be used to represent either relation $(\approx_c, \approx_i)$ interchangeably.

$$\begin{gathered}\textit{Theorem HMAC\_spec\_equiv}:\\ \forall(k\ m\ h:\textit{Blist})(\textit{op}\ \textit{ip}:\textit{Blist})\\ (K\ M\ H:\textit{list}\ Z)(\textit{OP}\ \textit{IP}:\textit{list}\ Z),\\ \textit{length}\ K=\textit{SHA}256.\textit{BlockSize}\to\\ k\approx K\to\\ m\approx M\to\\ \textit{op}\approx \textit{OP}\to\\ \textit{ip}\approx \textit{IP}\to\\ \textit{HMAC c p sha\_h sha\_iv sha\_splitandpad op ip}\ k\ m=h\to\\ \textit{HMAC\_SHA}256.\textit{HMAC IP OP}\ M\ K=H\to\\ h\approx H.\end{gathered}$$

Note that the following are implicit:

$$sha\_iv \approx_c SHA256.init\_registers$$
$$sha\_h \approx_w SHA256.hash\_block$$
$$sha\_splitandpad \approx_w sha\_padding\_lemmas.pad$$

$sha\_padding\_lemmas.pad$ is a version of $SHA256.generate\_and\_pad$, the padding function defined in FIPS 180-4, modified here to separate the padding from the $Z \to Integer$ conversion. $hash\_blocks\_bits$ is a version of $SHA256.hash\_blocks$ modified to use 512 as its block size.

As a refresher, here is the HMAC code:

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

Section IIc discussed the series of transformations applied to the abstract spec. Its structure was originally not close to the HMAC and SHA code, but after some massaging, it and its internal hash function become structurally similar enough to the high-level code for the proof to be decomposed by function.

Three functions occur in the definition of HMAC: H (hashing), $|$ (concatenation), and $\oplus$ (bitwise xor). Thus, the proof breaks up naturally and modularly into three theorems. For each function, its theorem states that the output is equivalent given that the input is equivalent.

$$\begin{aligned} &Lemma\ f\_equiv : \forall (b : bits)(B : bytes),\\ &b \approx B \to\\ &f \approx_w F \to\\ &f\ b \approx F\ B. \end{aligned}$$

The proof is completed by backward-chaining these implications. A diagram of the chain may be found at the end of the Appendix. The diagram makes it clear that in the end, the leaves of the tree are the sole propositions that need to be proved, and these are exactly the givens: $k \approx K, m \approx M, op \approx OP$, and $ip \approx IP$.

The theorem about $|$, $concat\_equiv$, is easy to prove by induction.

The theorem about $\oplus$, xor_equiv_Z, is slightly harder. It uses the inductive relation and depends on several lemmas in $XorCorrespondence$, including a large proof by brute force.

Lastly, the theorem about H, SHA_equiv_pad, depends on $fold\_equiv$. $fold\_equiv$ requires a messy double induction that requires definition and use of the $InBlocks\ n$ inductive proposition. This is necessary because one list is in blocks of 512 bits and the other is in blocks of 16 integers.

In addition to the main theorems, the equivalence proof involves about 45 more lemmas.

G. Inductive relation

The strength of the inductive relation $\approx_i$ is that it comes with a stronger induction principle that helps with low-level bit/byte proofs, especially those involving xor.

H. 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 l` 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.

I. 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.

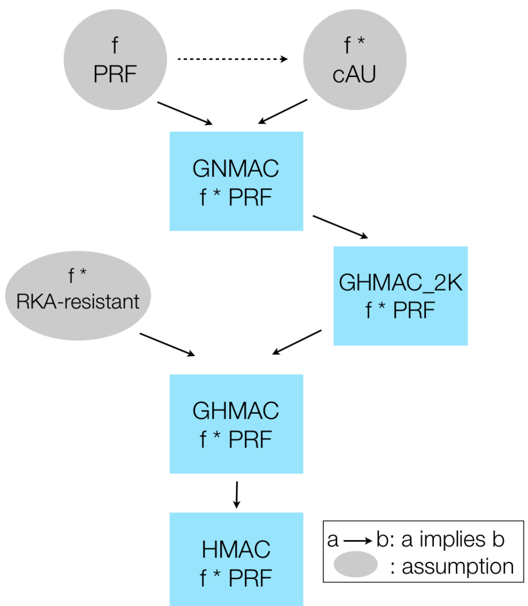

Figure 5. The structure of the 1996 (? TODO) Bellare HMAC proof.

The $f^* cAU$ property is used to prove that generalized NMAC (GNMAC) using f^* is a PRF. The GNMAC 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.

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
    | O =>
      fun (v : Vector.t A (O + 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 O => 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

```
      rewrite H'.
      eexists x.
      omega.

    * 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 *)
Check SHA_equiv_pad.
apply SHA_equiv_pad.
Check concat_equiv.
apply concat_equiv.
SearchAbout bytes_bits_lists.
apply xor_equiv_Z; try assumption.

*
  apply SHA_equiv_pad.
  apply concat_equiv.

- apply xor_equiv_Z; try assumption.
- assumption.
  (* xors preserve length *)
  *
    (* TODO split out this proof as lemma *)
    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.

F. Main HMAC equivalence proof

On next page.

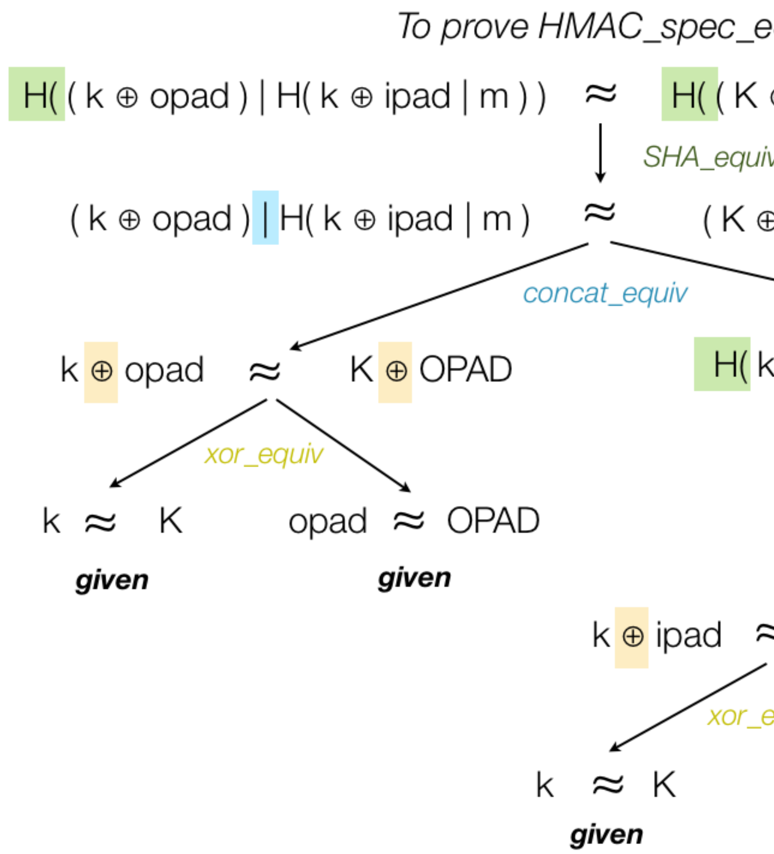

Figure 6. $a \xrightarrow{T} b$ means "By T, b implies a." Note after HMAC has been unpacked, all that is left is our assumptions.