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."
(December 8, 2014)

I. INTRODUCTION

Motivation and background.

A. Coq

B. The Merkle-Damgard construction

C. SHA-256

D. HMAC

E. Prior work

Verification of SHA-256.

II. THE PROOF OF EQUIVALENCE

A. The concrete specification

B. The abstract specification

It includes the GNMAC/`GHMAC_2K` structures. It leaves the hash function, two padding functions, and the hash function's initialization vector abstract.

C. Instantiating the abstract specification

We wrap the concrete functions in byteToBit and/or `intlist_to_Zlist` conversion functions.

D. Proof outline

The main differences between the specs are that (make a table?)

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 bit list of length n, whereas the concrete spec uses byte lists and int lists
3. 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 SHA-256 concrete spec uses the Z type, whereas the HMAC concrete type uses the byte type, which is Z constrained to be in $[0, 255]$.

E. Proof techniques

We reasoned about the bytes/bits proof via the following two frameworks.

- (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.

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.

F. 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. Bellare's proof

(Insert diagram.) (`goo.gl/wK8OXg`)

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 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

IV. CONCLUSION

A. Future work

ACKNOWLEDGMENTS

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

V. REFERENCES

"Verification of SHA-256," Appel (unpublished?)
"New Proofs for NMAC and HMAC: Security without Collision-Resistance," Bellare (1996)
"Keying Hash Functions for Message Authentication," Bellare (2004)
"MCF," Petcher (unpublished?)
"Merkle-Damgard in EasyCrypt," IMDEA
"Certified Programming with Dependent Types," Chlipala
"Software Foundations," Pierce et al.
Coq.Strings.Ascii

VI. APPENDIX

A. The concrete specification

B. The abstract specification

C. Selected theorems and proofs