SHA Formalization

Diana Toma Dominique Borrione
{diana.toma, dominique.borrione}@imag.fr

TIMA Laboratory, Grenoble, France

1 Introduction

Hash functions are among the most widespread cryptographic primitives. They are primarily used together with public-key cryptosystems in digital signature schemes. They are also a basic building block of secret-key Message Authentication Codes (MACs) which are currently used in security protocols such as IPSec and SSL. Other popular applications of hash functions include fast encryption, password storage and verification, computer virus detection, etc. Tens of hash functions have been proposed, the majority of them have been broken and only a few have been standardized.

The most widely accepted hash function is SHA-1 (Secure Hash Algorithm-1), a standard of 1993 [1]. After introducing a new secret-key encryption standard, AES(Advanced Encryption Standard), the security of SHA-1 no longer matched the security guaranteed by the encryption standard. Therefore three new hash functions have been introduced: SHA-256, SHA-384, and SHA-512 [2].

All four algorithms are iterative one-way hash functions that can process a message to produce a condensed representation called a message digest. These algorithms enable the determination of a message integrity: any change to the message will, with a very high probability, result in a different message digest. All functions have a similar internal structure, and process each message block using multiple rounds. The four algorithms differ most significantly in the number of bits of security that are provided for the data being hashed, i.e. the message digest length,and in the size of the blocks and words of data that are used during hashing (Table 1).

Algorithm	Message Size	Block Size	Word Size	Message Digest Size	Security
SHA-1	$< 2^{64}$	512	32	160	2^{80}
SHA-256	$< 2^{64}$	512	32	256	2^{128}
SHA-384	$< 2^{128}$	1024	64	384	2^{192}
SHA-512	$< 2^{128}$	1024	64	512	2^{256}

Table 1: Secure Hash Algorithm Properties

Each algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves padding a message, parsing the padded message into N-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a message schedule from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest. In the following we discuss the verification approach for SHAs using as example SHA-1.

This paper presents a part of an ongoing project in which other partners are designing a chip for secure transmissions. Our contribution is the verification of the hash block which contains the four SHA algorithms. We have chosen theorem proving as verification technique, using ACL2. The first step is formalizing the SHA algorithms in ACL2, the second step is modeling with ACL2 the

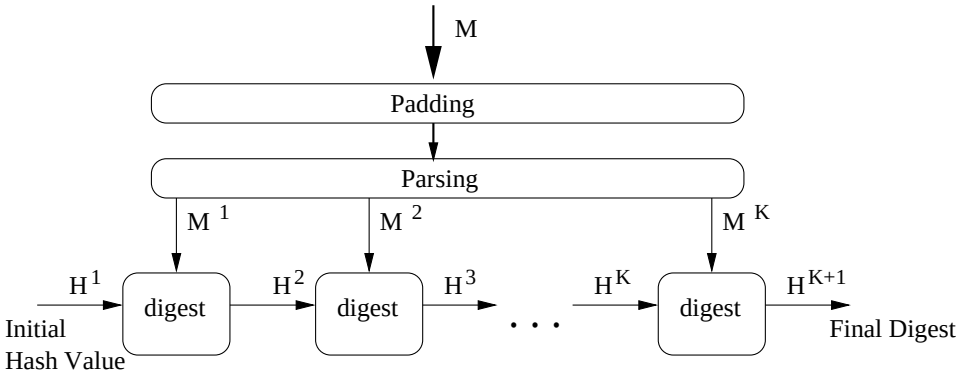

Figure 1: Secure Hash Algorithm

VHDL files describing the hash block, and the last step is verifying that the model extracted from the VHDL corectly implements the algorithms.

In this paper we present the first step, i.e. SHAs formalization. We choose to represent the bit vectors as lists. A word is a list of w bits; In accordance with the SHA algorithms, the big-endian convention is used when expressing words: the most significant bit is stored in the left-most bit position. A set of operations on w-bit words have been defined and their corresponding theorems have been proved.

2 Padding

Let M be a message of length len bits. The purpose of padding is to extend M to a multiple of 512 bits. To obtain the padded message, append the bit 1 to the end of message M, followed by k zero bits, where k is the smallest, non-negative solution to the equation $(len+1+k)\, mod\, 512 = 448$. Then append the 64-bit binary representation of number len.

For example, the (8-bit ASCII) message "abc" has the length $8 \times 3 = 24$, so the message is padded with one bit, then $448 - (24 + 1) = 423$ zero bits, and then the message length, to become the 512-bit padded message

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ 1\ \overbrace{00..00}^{423}\ \overbrace{00..011000}^{64}$$

We model the padding function in $ACL2$ as follows:

```
(defun padding (M)
    (if (and (bvp M) (< (len M) (expt 2 64)))
        (if (<= (mod (1+ (len M)) 512) 448)
            (append M (list 1)
              (make-list (- 448 (mod (1+ (len M)) 512)) :initial-element 0)
              (bv-to-n (int-bv-big-endian (len M)) 64))
            (append M (list 1)
              (make-list (- 960 (mod (1+ (len M)) 512)) :initial-element 0)
              (bv-to-n (int-bv-big-endian (len M)) 64)))
        nil))
```

Where *int-bv-big-endian (i)* transforms the integer i into the corresponding bit vector, with the most significant bit on the leftmost-bit position, the function *bv-to-n (m i)* forces the bit vector m to the length i and *bvp (m)* is a predicate that recognizes a bit vector.

Now, we prove that the length of the padded message is a multiple of 512, and greater or equal to 512.

```
(defthm len-padding-mod-512=0
```

```
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
```

So, after padding we obtain a message that has the following properties:

- the padded message is a vector of bits.

```
(defthm bvp-padding
    (bvp (padding m)))
```

- the last 64 bits of the padded message represent the length of the initial message.

```
(defthm last64-padding=len
  (implies (and (bvp M) (< (len M) (expt 2 64)))
     (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
```

- the first $len(M)$ bits of the padded message represent the initial message M.

```
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
```

- the next bit after M in the padded message marks the end of message M (i.e. 1).

```
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
```

- the bits in the padded message between the end-of-the-message bit and the last 64 bits are all 0.

```
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
      (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding m))
             (make-list (- (len (padding M)) (+ 65 (len M)))
                        :initial-element 0))))
```

This padding function is also used for the SHA-256 algorithm. It slightly changes for the other two algorithms, SHA-384 and SHA-512.

3 Parsing

In this step we have to parse the padded message into 512-bit blocks. We define a general function which parses a list l into blocks of length n.

```
(defun parsing (l n)
   (if (and (integerp n) (<= 0 n) (true-listp l))
       (cond ((endp l) nil)
             ((zp n) nil)
             (t (cons (firstn n l) (parsing (nthcdr n l) n))))
       nil))
```

Note that *firstn (n,l)* returns the first n elements of l if $n < len(l)$, else it returns l, so:

```
ACL2 !>(parsing '(1 2 3 4 5 6 7 8 9) 4)
((1 2 3 4) (5 6 7 8) (9))
```

Instead, if *len (l)* is a multiple of n, the result of parsing l to n is a list L of blocks of equal length, where the length of L is the result of dividing *len (l)* by n.

```
(defthm parsing-right-len
 (implies (and (true-listp l) (integerp n) (< 0 n) (equal (mod (len l) n) 0))
          (el-of-eq-len (parsing l n))))
```

where the function *el-of-eq-len (l)* verifies that the elements of l have the same length.

```
(defthm len-parsing
 (implies (and (true-listp l)(integerp n) (< 0 n) (equal (mod (len l) n) 0))
          (equal (len (parsing l n)) (/ (len l) n))))
```

If we parse a bit vector m to n, where *len (m)* is a multiple of n, the result is a vector of words, each of length n.

```
(defthm wvp-parsing
  (implies (and (bvp m) (integerp n) (< 0 n) (equal (mod (len m) n) 0))
           (wvp (parsing m n) n)))
```

Now we apply the last theorems to the padded message knowing that its length is a multiple of 512 in the case of SHA-1 and SHA-256 and a multiple of 1024 in the case of SHA-384 and SHA-512. For SHA-1:

```
(defthm wvp-parsing-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (wvp (parsing (padding M) 512) 512)))
```

4 Message Digest

After the preprocessing is complete, the message blocks, $M^1\, M^2 ...,\, M^K$, are processed in order. The message digest operation uses a sequence of logical functions, a sequence of 32-bit constants, a buffer of five 32-bit working variables, a buffer of five 32-bit intermediate hash values, a single word buffer $TEMP$ and a sequence of eighty 32-bit words. The message digest algorithm differs for each SHA. Before computation begins the initial hash value must be set. For SHA-1 it consists of five 32-bit words. The computation of a message block for SHA-1 follows the next steps:

1. parse M^i in 16 words $W_i^0, W_i^1, ..., W_i^{15}$, each of 32 bits and compute the words $W_i^j = ROTL^1(W_i^{j-3} \oplus W_i^{j-8} \oplus W_i^{j-14} \oplus W_i^{j-16})$, where $15 < j < 80$.

We define the function *prepare (M-i)* which takes a 512-bit block *M-i* and returns a sequence of eighty words, each of 32 bits.

```
(defun prepare-ac (j M-i)
(declare (xargs :measure (acl2-count (- 80 j))))
 (if (and (integerp j) (<= 16 j) (wvp M-i 32))
     (cond ((<= 80 j) M-i)
           ((<= j 79) (prepare-ac (1+ j) (append M-i
                (list (rotl 1 (bv-xor (nth (- j 3) M-i) (nth (- j 8) M-i)
                              (nth (- j 14) M-i) (nth (- j 16) M-i)) 32))))))
      nil))
(defun prepare (M-i)
 (if (wordp M-i 512)
     (prepare-ac 16 (parsing M-i 32))
     nil))
(defthm wvp-prepare
   (implies (wordp M-i 512))
            (wvp (prepare M-i) 32))
(defthm len-prepare
   (implies (wordp M-i 512)
            (equal (len (prepare M-i)) 80)))
```

2. initialize the working variables with the intermediate hash value (for the first block M^1, with initial hash value $(h-1)$)

3. apply eighty times the digest step

```
(defun digest-one-block-ac ( j working-variables M-i-ext)
(declare (xargs :measure (acl2-count (- 80 j))))
   (if (and (wvp working-variables 32) (equal (len working-variables ) 5)
            (integerp j) (<= 0 j)
            (wvp M-i-ext 32) (equal (len M-i-ext) 80))
       (if (<= 80 j) working-variables
            (digest-one-block-ac (+ 1 j)
                  (list (TEMP j working-variables M-i-ext)
                        (nth 0 working-variables)
                        (rotl 30 (nth 1 working-variables) 32)
```

```
                        (nth 2 working-variables)
                        (nth 3 working-variables)) M-i-ext))
        nil))
(defun digest-one-block (hash-values M-i-ext)
   (if (and (wvp hash-values 32) (equal (len hash-values) 5)
            (wvp M-i-ext 32) (equal (len M-i-ext) 80))
       (digest-one-block-ac 0 hash-values M-i-ext)
        nil))
```

4. compute the intermediate hash values.

The intermediate hash value of block M^i is the input hash value for block M^{i+1}. The result after applying steps one to four to all K message blocks represents the message digest of M.

```
(defun digest (M hash-values)
   (if (and (wvp M 512) (wvp hash-values 32) (equal (len hash-values) 5))
       (if (endp M) hash-values
           (digest (cdr M)
              (intermediate-hash hash-values
                   (digest-one-block hash-values (prepare (car M))))))
       nil))
(defun sha-1 ( M)
   (if (and (bvp M) (< (len M) (expt 2 64)))
       (digest (parsing (padding M) 512) (h-1))
        nil))
```

The final result of SHA-1 is a five 32-bit words message digest.

```
(defthm wvp-sha-1
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (and (wvp (sha-1 M) 32) (equal (len (sha-1 M)) 5))))
```

5 Conclusion

This work is part of a larger project, in which other partners are designing a chip for secure transmissions. We have modeled the four algorithms SHA-1, SHA-256, SHA-384 and SHA-512 and we have proven analogous safety theorems on all of them. For the modeling and the verification of SHAs we wrote seventy definitions, a hundred and sixty theorems and we used the ACL2 arithmetic books *equalities*,*inequalities* and *floor-mod*, and the data-structures books *list-defuns* and *list-defthms*. The ACL2 modeling has also permitted numeric execution of the algorithms on the tests provided in the standard document.

Our current work consists in verifing that the VHDL implementations of the SHAs correspond to the ACL2 specifications. To this end, we found it useful to develop a book for bit vectors represented as lists with high order bits on the left, as it is closer to the VHDL bit vectors representation. We also consider that using lists permits to take advantage of the recursivity of list theory. We intend to prove the equivalence between our representation and the integer representation already existing in the IHS book.

References

[1] National Institute of Standards and Technology (NIST). *Secure Hash Standard*. Federal Information Processing Standards Publication 180, 1993.

[2] National Institute of Standards and Technology (NIST). *Secure Hash Standard*. Federal Information Processing Standards Publication 180-2, 2002.