ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions

Revision 5: February 11, 2008

1. Overview

This document specifies the ANSI C interface profile for implementations of SHA-3 candidate algorithms. C implementations shall support the syntax and parameterization of the interface profile messages as described in this API. The API consists of a few data definitions, one data structure, and four functions to compute hashes. The functions specified in this API have return values listed that are largely used to supply error codes in the event of incomplete execution of the routines. The error values listed are not meant to be an exhaustive list. If additional error codes are useful for your implementation, please provide them.

2. Data Definitions

The following typedef is used to specify the arrays that will hold the data to be hashed and the resulting hash value.

```
typedef unsigned char BitSequence;
```

The byte length, n, of a BitSequence data item of length *bitlen* will be $n = \lceil bitlen/8 \rceil$, e.g., an 8-bit message will require 1 BitSequence element and a 13-bit message will require 2 BitSequence elements. BitSequence arrays will be indexed from 0 to n-1. Sequences of bits are enumerated from 0 to (*bitlen*-1). The i^{th} bit of the sequence will be stored in array element $\lfloor i/8 \rfloor$. Within a BitSequence array element, the bits are indexed from 0 to 7 with bit 0 being the Most Significant Bit (MSB), i.e., the bit with the largest numerical value. Therefore, the i^{th} bit of the BitSequence will be found in the i % 8 bit position of the $\lfloor i/8 \rfloor$ bitSequence element.

The following typedef is used to provide the data length of the message to be hashed. It should be set to the largest integral data type that the target platform and compiler can understand. Preferably this will be an unsigned 64-bit integer. If the target platform and compiler cannot handle a 64-bit data type, use a 32-bit unsigned data type instead.

```
typedef unsigned long long DataLength;    // a typical 64-bit value
```

The following enumeration is to provide return values for the API Hash function. Additional return values may be added. These values shall be documented.

```
typedef enum { SUCCESS = 0, FAIL = 1, BAD_HASHBITLEN = 2 } HashReturn;
```

3. Data Structure

The ***hashState*** structure contains all information necessary to describe the current state of the SHA-3 candidate algorithm. The only required field, ***hashbitlen***, indicates the output size of this particular

instantiation of the hash algorithm. Algorithm specific fields may be placed anywhere in the structure below. These include things like data storage needed to hold intermediate values, tables, unprocessed data, etc. **All implementations must be sure to document any algorithm-specific parameters and their use.**

```
typedef struct {
     //hashbitlen + algorithm specific parameters
     } hashState;
```

4. Function Calls

There are four function calls specified in this API. The first three provide a method for performing incremental hashing with the candidate algorithm. The fourth provides a method to perform all-at-once hashing of the supplied data.

4.1. Init()

Each SHA-3 submitter is required to implement this interface because NIST anticipates that some candidate algorithms will have unique requirements to initialize the *hashState* structure.

This API uses a function called *Init()* to initialize the *hashState* structure. As stated above, the *hashState* structure contains the *hashbitlen* of this particular instantiation, as well as any algorithm specific parameters that are needed. Implementations shall support, at a minimum, *hashbitlen* values of 224, 256, 384, and 512-bits. Additionally, if an algorithm can support other hash lengths, these digest sizes should be supported in this code as well.

The initialization function, *Init(),* is called with the appropriate parameters which get loaded into the *hashState* structure. These parameters are then used to perform any data independent setup that is necessary, e.g., initialization of any intermediate values, initialization of any tables, etc.

- **Init()**

```
HashReturn Init(hashState *state, int hashbitlen);
```

Initializes a *hashState* with the intended hash length of this particular instantiation. Additionally, any data independent setup is performed.

Parameters:
state: a structure that holds the *hashState* information
hashbitlen: an integer value that indicates the length of the hash output in bits.

Returns:
SUCCESS - on success
BAD_HASHBITLEN - hashbitlen is invalid (e.g., unknown value)

4.2. Update()

This API uses a function called *Update()* to process data using the algorithm's compression function. Whatever integral amount of data the *Update()* routine can process through the compression function is handled. Any remaining data must be stored for future processing. For example, SHA-1 has an internal structure of 512-bit data blocks. If the *Update()* function is called with 768-bits of data the first 512-bits will be processed through the compression function (with appropriate updating of the chaining values) and 256-bits will be retained for future processing. If 2048-bits of data were provided, all 2048-bits would be processed immediately. If incremental hashing is being performed, all calls to update will contain data lengths that are divisible by 8, except, possibly, the last call.

The *Update()* function is called with a pointer to the appropriate *hashState* structure, the *data* to be processed, and the length of the data to be processed (*databitlen*). The *Update()* routine processes as much data as it can, updating all appropriate intermediate values, and returns a status code.

❖ **Update()**

```
HashReturn Update(hashState *state, const BitSequence *data,
                  DataLength databitlen);
```

Process the supplied data.

Parameters:
state: a structure that holds the *hashState* information
data: the data to be hashed
databitlen: the length, in bits, of the data to be hashed

Returns:
SUCCESS - on success

4.3. Final()

This API uses a function called *Final()* to process any remaining partial block of the input data and to perform any output filtering that may be needed to produce the final hash value. For example, SHA-1 requires appending a "1"-bit to the end of the message followed by an appropriate number of "0"-bits and the length field. This is all processed through the compression function to produce the final hash value for the message.

The *Final()* function is called with pointers to the appropriate *hashState* structure and the storage for the final hash value to be returned (*hashval*). The *Final()* routine performs any post processing that is necessary, including the handling of any partial blocks, and places the final hash value in *hashval*. Lastly, an appropriate status value is returned.

❖ **Final()**

```
HashReturn Final(hashState *state, BitSequence *hashval);
```

Perform any post processing and output filtering required and return the final hash value.

Parameters:
`state`: a structure that holds the *`hashState`* information
`hashval`: the storage for the final hash value to be returned

Returns:
`SUCCESS - on success`

4.4. Hash()

This API uses a function called *`Hash()`* to provide a method to perform all-at-once processing of data using the candidate algorithm and to return the resulting hash value. The *`Hash()`* function is called with a pointer to the *`data`* to be processed, the length of the data to be processed (*`databitlen`*), a pointer to the storage for the resulting hash value (*`hashval`*), and a length of the desired hash value (*`hashbitlen`*). This function shall utilize the previous three function calls, namely *`Init()`*, *`Update()`*, and *`Final()`*.

❖ **Hash()**

```
HashReturn Hash(int hashbitlen, const BitSequence *data,
                DataLength databitlen, BitSequence *hashval);
```

Hash the supplied data and provide the resulting hash value. Set return code as appropriate.

Parameters:
`hashbitlen`: the length in bits of the desired hash value
`data`: the data to be hashed
`databitlen`: the length, in bits, of the data to be hashed
`hashval`: the resulting hash value of the provided data

Returns:
```
SUCCESS - on success
FAIL – arbitrary failure
BAD_HASHBITLEN – unknown hashbitlen requested
...
```