DD2448 Foundations of Cryptography
A Presentation of SHA-3 Candidate *Skein*

Daniel Bosk*

May 25, 2010

1 Introduction

The SHA-3 candidate *Skein* is more than just a hash function to replace SHA-2, it is a hash function family. Skein can be used as a simple hash function, which is why it is proposed, but it can also do native tree hashing, natively be used as a MAC, be used in HMAC, native support for randomized hashing, native support as a hash function for digital signatures, natively be used as a key derivation function (KDF) and password-based key derivation function (PBKDF), natively as a pseudo-random number generator (PRNG), natively be used as a stream cipher, supports personalization and finally has native support for different output sizes. All of this because of the extensive set of options which can be used with Skein.

The Skein family of hash functions has internal states of sizes 256, 512 and 1024, which can be compared to the SHA-256 and SHA-512 hash functions. The Skein-256 version is a low memory variant which can be implemented using only about 100 bytes of RAM [1].

The Skein hash function is designed for performance in both software and hardware, it is based on the tweakable block-cipher *Threefish*, and it is thanks to that the cipher is tweakable that Skein can support all these options outlined above. The core design principle of Threefish is that more simple rounds is more secure than fewer complex rounds, and thus it uses only three different mathematical operations: addition, exclusive-or and constant rotations. It is based on 64-bit words to be optimal for use on modern 64-bit processors.

Note that I will in this presentation interchange bitstrings, bytestrings and numbers for the sake of brievity. To do this correctly would involve type-changes from strings to numbers to be able to do arithmetic with them. However, I will assume that computations with bitstrings and numbers can be done (using "automagical" transformations).

2 The Idea

The idea of Skein is that it must be able to be a drop-in replacement for other hash functions such as MD5 and the SHA-variants as well as having some other extensive functionality. In [1, Table 1] there is a table listing the authors' recommended Skein configurations to use Skein as a drop-in replacement for MD5 and SHA.

The construct which allows this high configurability of Skein is the tweakable block-cipher, in this case Threefish. It is the block-size of Threefish, 256, 512 and 1024 bits,

*E-mail: dbosk@kth.se

which determines the size of the internal state of the hash function. The thing used in Skein is that the tweak in Threefish allows for hashing configuration data along with the message data in every block and thus making every block unique. The core being the compression function called *unique block iteration (UBI)*. UBI is a chaining mode which uses Threefish to compress arbitrary sized data into a fixed size. (Skein, or rather UBI, would also work using any other tweakable block-cipher.)

The final part of Skein is the optional argument system. The design makes these three parts of Skein independent of each other, partly because it makes it easier to prove theorems about the independet systems but also to have modularability.

2.1 Threefish

The Threefish cipher is a tweakable block-cipher with block-sizes of 256, 512 or 1024 bits and always a tweak parameter of 128 bits, it works entirely with 64-bit words – which is why it is optimal for a 64-bit processor. The encryption function, $E(K, T, P)$, for Threefish takes three arguments: the keyi K, the tweak T and the plaintexti P. More on this in a moment.

First, the key schedule works as such that you take the 64-bit words in the original key supplied to the function as well as the two 64-bit words supplied as the tweak. Subkeys are generated from this key and tweak using only XOR and addition modulo 2^{64}. So the subkeys depends on both the key and the tweak. (For details, see [1, p. 11-12].)

The encryption, simplified, is done in the following way: the plaintext is added with a subkey using addition modulo 2^{64} (for details, see [1, p. 10]). The result is the run through a mixing function which uses addition modulo 2^{64}, XOR and constant rotation according to a predefined table (for details, see [1, p. 11]). When mixed, the words are permuted according to a predefined permutation (for details, see [1, p. 10-11]). All these steps are repeated are repeated a total of four times before we add the next subkey and repeat all this again. This four-step procedure is repeated 18 times, i.e. a total of 72 mixing and permuting rounds. This applies to the 256- and 512-bit versions of the cipher, the 1024-bit cipher uses a total of 80 rounds.

2.2 Unique Block Iteration (UBI)

Now to the most interesting part of the whole construction – the compression function UBI. UBI takes three inputs: a starting value G, a message M and a starting value for the tweak T_s. The UBI processes the message in blocks of size of the intrernal state and uses a tweak value for each block.

The tweak value, a number of size 128-bits, is divided into certain fields. E.g. *position* (bits 0-95), *tree-level* (bits 112-118), *bit-pad* (bit 119), *type* (bits 120-125), *first* (bit 126)i and finally *final* (bit 127). For a better overview, see [1, Table 5 and Figure 9]. The fields are continuously used during the processing of a message. For instance, if the message is not a multiple of the block-size it must be padded, if it must be padded the *bit-pad* bit must be set to 1 and otherwise 0. The *first* bit will be set to 1 for the first block of the message and 0 otherwise, the *final* bit will be set to 1 for the final block of the message and 0 otherwise. The *position* bit will be continuously increased throughout the process and will thus guarantee uniqueness even if the message is constantly repeating.

When the message-blocks M_i are processed, it is done accordingly:

$$\begin{aligned} H_0 &= G, \\ H_{j+1} &= E(H_j, T_j, M_j), \end{aligned}$$

where T_j is the tweak-value with the bit-fields set correctly for the jth block. When all blocks are processed, the final H is the result of the UBI chaining mode. This means that each block is processed uniquely, they all depend on each other and the original message has now been compressed to a single block-sized result.

3 Functionality

The *type* field of the UBI tweak is used by Skein to run in different types of operation. The different type-values are listen in [1, Table 6], but some of them are used to include a key in the hash, e.g. useful for MAC and KDF, to include the configuration block block in the hash, and so on. The configuration type is mandatory to run, and they must be run in the order they are defined in [1, Table 6].

Skein must be run in the following way. First we run the key through the UBI as $K' = \text{UBI}(0, K, T_{\text{key}}2^{120})$, where K is our key and we must set the 120th bit of the tweak (i.e. the *type* bits) to the type T_{key}. If we have no key, we let $K' = 0$.

After this step comes the configuration. The configuration is done by

$$G_0 = \text{UBI}(K', C, T_{cfg}2^{120}),$$

where C is our configuration. The configuration is a 32-byte long bitstring divided into fields just as the tweak. The contains information such as schema identifier, in this case the byte-representation of "SHA3", version number, output length, and some settings regarding tree hashing. Once this configuration step is done we can start the more general part of the algorithm.

Let L be a list of 2-tuples $(T_0, M_0), \ldots, (T_{t-1}, M_{t-1})$ such that $T_{cfg} < T_0$, $T_i < T_{i+1}$ and $T_{t-1} < T_{out}$, T_{out} being the type of largest value. Now, if we do not intend to do any tree hashing[1], we can start processing this list by letting

$$G_{i+1} = \text{UBI}(G_i, M_i, T_i 2^{120}),$$

for $i = 0, 1, 2, \ldots, t-1$. When the list is processed we find our result H by

$$H = \text{concatenate}(\text{UBI}(G_t, 0, T_{out}2^{120}), \text{UBI}(G_t, 1, T_{out}2^{120}), \ldots)$$

until the desired output size is reached (as set in configuration C).

3.1 Features

Now, all those features mentioned in the introduction, almost every one of them has their own type defined. By adding an appropriate message and the corresponding type you do lots of different things with Skein. (Note however that not all types can be combined.) One interesting type of operation, or application, is the personalization. This application is used to personalize the hash, i.e. if you need two different hash functions you can use Skein at both times and just personalize them differently. The authors [1, p. 21] recommends using the following format for a string: 20100319 dbosk@kth.se APP/test1 and 20100319 dbosk@kth.se APP/test2, i.e. today's date, your valid email (this date), the application name and some functional specifier. The you get two different hash functions using the same API. This could be useful when hashes are used in more than one place in a protocol. This is because if a hash is first computed on some data, this hash might be useful for an attacker because later another hash might be computed on similar or related data.

[1] Tree hashing is not covered in this presentation, it requires some other processing of the messages.

The tree hashing is also an interesting functionality. Although, it requires another algorithm, as mentioned above, which is not covered in this text, the idea is interesting. The purpose of the tree hashing is to be able to hash using parallell processing, e.g. on a multi-core processor. It can also be utilized when rehashing of data is needed, because only parts of the tree must be rehashed thus also increasing performance.

4 Security

The security of Skein is proven by the authors in [2], in the actual specification [1] they describe the results of the proofs. The conclusion of the security claims is that Skein is proven to be secure if the compression function is collision resistant. Also, Skein's property of being a pseudo-random function (PRF) and secure to use for KDF, MAC, HMAC, PRNG and as a stream cipher is proven with the assumption that Threefish is a tweakable pseudo-random permutation. There are some known attacks on Threefish, however they seem to be quite harmless and Threefish should be about as secure as AES [1].

5 Conclusion

The paper was a very interesting read. It is a very interesting algorithm which can be both easily used and easily expanded in many ways, as it is already prepared for futher expantion. It is also interesting that such a complex construction can be built by such simple tools. The separation of the parts, e.g. the tweakable block-cipher and the UBI compression function, makes it very easy to follow and to get an overview of the function family.

References

[1] Bellare, M., Kohno, T., Lucks, S., Ferguson, N., Schneier, B., Whiting, D., Callas, J. and Walker, J. *The Skein Hash Function Family*, version 1.2, 15 September 2009. Specification submitted to NIST SHA-3 competition.

[2] Bellare, M., Kohno, T., Lucks, S., Ferguson, N., Schneier, B., Whiting, D., Callas, J. and Walker, J. *Provable Security Support for the Skein Hash Family*, version 1.0, April 2009. `http://www.skein-hash.info/sites/default/files/skein-proofs.pdf`.