CYGNACOM SOLUTIONS

Improving Hash Function Padding

Don B. Johnson
Entrust Cygnacom

CYGNACOM SOLUTIONS

Truth in Advertising

This is the **LEAST** technical talk at this NIST Hash workshop

Please do not have high expectations

On the plus side, this talk should be among the easiest to understand

CYGNACOM SOLUTIONS

NIST Secure Hash padding:

1. **Append a binary 1** pad bit to message.
2. **Append binary zero pad bits** up to the block length minus the size of the message length field.

   a. SHA-256 & less: block length is 512 bits and message length field is 64 bits.

   b. SHA-384 & greater: block length is 1024 bits and message length field is 128 bits.
3. **Append the message length field** containing a value consisting of the bit length of the message.

CYGNACOM SOLUTIONS

NIST Secure Hash Padding (Simplified)

Data to be hashed	'1'	'0...0'	Postfix length field

CYGNACOM SOLUTIONS

Unambiguous Padding

- An Insight: The message length field (by itself) ensures that all inputs are mapped unambiguously after padding
- This means that appending a binary 1 pad bit is not needed
- **Recommendation: Remove the appending of a binary 1 pad bit in padding for future hash designs.**

CYGNACOM SOLUTIONS

Proposal 1 Secure Hash Padding

Data to be hashed	'0…0'	Postfix length field

CYGNACOM SOLUTIONS

Cost of appending '1' pad bit

- Some (small) additional code space
- 1% to 2% additional performance cost for processing an additional block for messages with random lengths (See paper for details)
- Software design principle: worth it to **highly optimize** a commonly-used routine

CYGNACOM SOLUTIONS

Moral of Story

- "Every little bit counts"
- Or you can make up your own

CYGNACOM SOLUTIONS

Postfix Length Padding Concerns

- Well known: postfix padding has possible concerns because of length extension
- A new block can be added with a new length field
- Solution: Put length field in the same fixed location
- The fixed location that always exists is before any data to be hashed: prefix

CYGNACOM SOLUTIONS

Why Postfix Length Padding?

- The rationale for postfix padding is that SOME applications will not know the total length to be hashed at the start of hash processing
- This means every hash has postfix concern
- Can prefix length outside of hash design, but this means short messages pay more
- Better: Prefix length field with an indicator that postfix length field will be coming

CYGNACOM SOLUTIONS

Postfix Padding Extension Concern

Data to be hashed	'0…0'	Original postfix length field	'0…0'	New postfix length field

CYGNACOM SOLUTIONS

Proposal 2 Secure Hash Padding

Length Field	Data to be hashed	'0…0'	Postfix length field, if needed

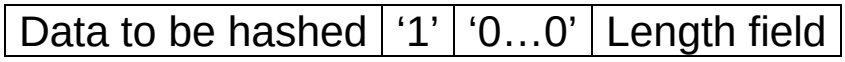
This way, only hashes that must have the postfix block extension concern will have it.

Thanks for listening!

Improving Hash Function Padding

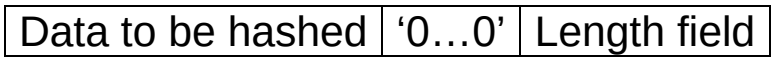
September 30, 2005
Don B. Johnson, Entrust CygnaCom, djohnson@cygnacom.com

The NIST Secure Hash Functions perform padding by doing the following:
1. Append a binary '1' pad bit.
2. Append binary zero pad bits up to the block length minus the size of the message length field. For SHA-256 and less the block length is 512 bits and the message length field is 64 bits. For SHA-384 and greater the block length is 1024 bits and the message length field is 128 bits.
3. Append the message length field containing a value consisting of the bit length of the input data.



The message length field (by itself) ensures that all inputs are mapped unambiguously after padding. This means that the appending of the binary 1 is not needed.

This paper recommends that appending a binary 1 pad bit be removed in future hash designs.



It may seem that appending a binary 1 is a minor cost and not worth removing. However, as a hash function is used extensively, one should try to avoid ANY unnecessary cost in using it. This follows the well-known software design principle that an extensively used subroutine should be highly optimized.

The cost includes the size of the additional code to append the binary 1, which while it may be small, might unnecessarily use up code space in a constrained environment where every byte is critical.

The performance cost for a hash function is significantly increased when an additional block is needed to be processed, this occurs when the input is exactly (448 modulo 512) bits for the smaller hash function outputs or (896 modulo 1024) bits otherwise. As essentially all inputs to a hash function will be on byte boundaries, we can simplify the above calculation to (56 modulo 64) bytes or (112 modulo 128) bytes. This means for inputs with random byte lengths, about 1/64 or 1/128 of the time an additional block will need to be processed. This averages out to a 1% to 2% additional cost for messages with random lengths that can be avoided.

The specific proposal of this paper is as follows for future Secure Hash padding:
1. Append binary zero pad bits up to the block length minus the size of the message length field. For SHA-256 and less the block length is 512 bits and the message length field is 64 bits. For SHA-384 and greater the block length is 1024 bits and the message length field is 128 bits.
2. Append the message length field containing a value consisting of the bit length of the input data.

The moral to this story is: Every little bit counts!

There is another aspect of hash function padding that at least should be discussed and that is prefix versus postfix padding of the length field. The reason for postfix length padding is claimed to be for those cases when the application does not know the final length of the data being hashed, for example, when a large file is being hashed. However, it is well known that postfix length padding has a security concern in that it is susceptible to a length extension attack/concern by adding a new block to the old data with the new longer length. This concern goes away if the length padding is always in the same location; for variable length data the only location that is always the same is the front as is done with prefix length padding.

Data to be hashed	‘0…0’	Postfix length field

Data to be hashed	‘0…0’	Postfix length field	‘0…0’	New postfix length field

The situation we find ourselves is that almost all hashes are done on short messages or files where the application DOES know the length when it begins processing. Why should all applications need to accept the potential risks of postfix length padding? Of course they do not, one can simply ALSO do prefix length padding and be done with it, but that solution seems inefficient for the vast majority of cases.

A more efficient solution is to realize that there are 2 cases, the common one where the length of the data to be hashed is known ahead of time and a rare situation where it is not. As the rare case is (essentially always) due to large data being hashed, it is reasonable to allow for a slight performance inefficiency in this case. One way to have an integrated single solution is to define a prefix length padding field with a special code to tell the hash routine that the final length is not known when the hashing is begun and that it will be input with the final chunk of data. This special code needs to be unambiguous different from valid length codes so that correct processing can be done.

Prefix length field	Data to be hashed	‘0…0’	Postfix length field, if used

Whether this proposal or another is appropriate needs thought, but at least we should acknowledge the question so that we are confident of the resulting answer.