Why narrow-pipe cryptographic hash functions are not a match to wide-pipe cryptographic hash functions?

Danilo Gligoroski

Danilo.Gligoroski@item.ntnu.no

Faculty of Information Technology, Mathematics and Electrical Engineering
Institute of Telematics
Norwegian University of Science and Technology
Trondheim, Norway

Vlastimil Klima

v.klima@volny.cz

Independent Cryptologist - Consultant
Prague, Czech Republic

Abstract

In the last 7-8 months me and Klima have discovered several deficiencies of narrow-pipe cryptographic hash designs. It all started with a note to the hash-forum list that narrow-pipe hash functions are giving outputs that are pretty different than the output that we would expect from a random oracle that is mapping messages of arbitrary length to hash values of n-bits. Then together with Klima we have investigated the consequences of that aberration to some practical protocols for key derivation that are using iterative and repetitive calls to a hash function. Finally, during the third SHA-3 conference I have shown that narrow-pipe hash functions cannot offer n-bits of security against the length-extension attack (a requirement that NIST has put as one of the conditions for the SHA-3 competition). In this paper we collect in one place and explain in details all these problems with narrow-pipe hash designs and we explain why wide-pipe hash functions such as Blue Midnight Wish do not suffer from the mentioned deficiencies.

1 Introduction

The importance of cryptographic functions with arbitrary input-length have been confirmed and reconfirmed numerous times in hundreds of scenarios in information security. The most important properties that these functions have to have are collision-resistance, preimage-resistance and second-preimage resistance. However, several additional properties such as multi-collision resistance, being pseudo-random function, or being a secure MAC, are also considered important.

All practical cryptographic hash function constructions have iterative design and they use a supposed (or conjectured to be close to) ideal finite-input random function (called compression function) $C : \{0,1\}^m \to \{0,1\}^l$ where $m > l$, and then the domain of the function C is extended to the domain $\{0,1\}^*$ in some predefined iterative chaining manner.[1]

The way how the domain extension is defined reflects directly to the properties that the whole cryptographic function has. For example domain extension done by the well known Merkle-Damgård construction transfers the collision-resistance of the compression function to the extended function. However, as it was shown in recent years, some other properties of this design clearly show non-random behavior (such as length-extension vulnerability, vulnerability on multi-collisions e.t.c.).

[1] The infinite domain $\{0,1\}^*$ in all practical implementations of cryptographic hash functions such as SHA-1 or SHA-2 or the next SHA-3 is replaced by some huge practically defined finite domain such as the domain $\mathcal{D} = \bigcup_{i=0}^{\text{maxbitlength}} \{0,1\}^i$, where maxbitlength $= 2^{64} - 1$ or maxbitlength $= 2^{128} - 1$.

The random oracle model has been proposed to be used in cryptography in 1993 by Bellare and Rogaway [1]. Although it has been shown that there exist some bogus and impractical (but mathematically correct) protocols that are provably secure under the random oracle model, but are completely insecure when the ideal random function is instantiated by any concretely designed hash function [2], in the cryptographic practice the random oracle model gained a lot of popularity. It has gained that popularity during all these years, by the simple fact that protocols proved secure in the random oracle model when instantiated by concrete "good" cryptographic hash functions, are sound and secure and broadly employed in practice.

In a series of 4 notes [9, 10, 11, 12] we have shown that four of the SHA-3 second round candidates: BLAKE [4], Hamsi [5], SHAvite-3 [6] and Skein [7] and the current standard SHA-2 [8], act pretty differently than an ideal random function $H : \mathcal{D} \rightarrow \{0,1\}^n$ where $\mathcal{D} = \bigcup_{i=0}^{\text{maxbitlength}} \{0,1\}^i$ and "maxbitlength" is the maximal bit length specified for the concrete functions i.e. $2^{64} - 1$ bits for BLAKE-32, Hamsi, SHAvite-3-256 and SHA-256, $2^{128} - 1$ bits for BLAKE-64, SHAvite-3-512 and SHA-512 and $2^{99} - 8$ bits for Skein.

2 Some basic mathematical facts for ideal random functions

We will discuss the properties of ideal random functions over finite and infinite domains.[2] More concretely we will pay our attention for:

Finite narrow domain: Ideal random functions $C : X \rightarrow Y$ mapping the domain of n-bit strings $X = \{0,1\}^n$ to itself i.e. to the domain $Y = \{0,1\}^n$, where $n > 1$ is a natural number;

Finite wide domain: Ideal random functions $W : X \rightarrow Y$ mapping the domain of $n + m$-bit strings $X = \{0,1\}^{n+m}$ to the domain $Y = \{0,1\}^n$, where $m \geq n$;

Proposition 1.. ([9]) Let $\mathcal{F}_C$ be the family of all functions $C : X \rightarrow Y$ and let for every $y \in Y$, $C^{-1}(y) \subseteq X$ be the set of preimages of y i.e. $C^{-1}(y) = \{x \in X \mid C(x) = y\}$. For a function $C \in \mathcal{F}_C$ chosen uniformly at random and for every $y \in Y$ the probability that the set $C^{-1}(y)$ is empty is approximately e^{-1} i.e.

$$Pr\{C^{-1}(y) = \varnothing\} \approx e^{-1}. \tag{1}$$

□

Corollary 1.. ([9]) If the function $C \in \mathcal{F}_C$ is chosen uniformly at random, then there exists a set $Y_{\varnothing}^C \subseteq Y$ such that for every $y \in Y_{\varnothing}^C$, $C^{-1}(y) = \varnothing$ and

$$|Y_{\varnothing}^C| \approx e^{-1}|Y| \approx 0.36|Y|.$$

□

Proposition 2.. ([9]) Let $\mathcal{F}_W$ be the family of all functions $W : X \rightarrow Y$ where $X = \{0,1\}^{n+m}$ and $Y = \{0,1\}^n$. Let for every $y \in Y$, $W^{-1}(y) \subseteq X$ be the set of preimages of y i.e. $W^{-1}(y) = \{x \in X \mid W(x) = y\}$. For a function $W \in \mathcal{F}_W$ chosen uniformly at random and for every $y \in Y$ the probability that the set $W^{-1}(y)$ is empty is approximately e^{-2^m} i.e.

$$Pr\{C^{-1}(y) = \varnothing\} \approx e^{-2^m}. \tag{2}$$

□

In what follows for the sake of clarity we will work on bit-strings of length which is multiple of n. Namely we will be interested on strings $M = M_1||\ldots||M_i$ where every $|M_j| = n, j = 1, \ldots, i$. Further, we will be interested in practical constructions of cryptographic hash functions that achieve a domain extension from a narrow-domain to the full infinite domain. We will need the following Lemma:

[2] The infinite domain $\{0,1\}^*$ in all practical implementations of cryptographic hash functions such as SHA-1 or SHA-2 or the next SHA-3 is replaced by some huge practically defined finite domain such as the domain $\mathcal{D} = \bigcup_{i=0}^{\text{maxbitlength}} \{0,1\}^i$, where maxbitlength $= 2^{64} - 1$ or maxbitlength $= 2^{128} - 1$.

Lemma 1.. ([9]) Let $\mathcal{F}_{C_\nu}$ be a countable family of functions $C_\nu : X \to Y,\ \nu \in \mathbb{N}$ and let $C : X \to Y$ is one particular function, where C_ν and C are chosen uniformly at random. Let us have a function $Rule : \mathbb{N} \times Y \to \mathcal{F}_{C_\nu}$ that chooses some particular random function from the family $\mathcal{F}_{C_\nu}$ according to a given index and a value from Y. If we define a function $H : (\{0,1\}^n)^i \to Y$ that maps the finite strings $M = M_1||\ldots||M_i$ to the set of n-bit strings $Y = \{0,1\}^n$ as a cascade of functions:

$$\begin{aligned} H(M) = H(M_1||\ldots||M_i) = \quad & C_{Rule(1,IV)}(M_1) \circ C_{Rule(2,C_{Rule(1,IV)}(M_1))}(M_2) \circ \\ & \circ \cdots \circ \\ & \circ\, C_{Rule(i,C_{Rule(i-1,\cdot)}(M_{i-1}))}(M_i) \circ \\ & \circ\, C \end{aligned} \tag{3}$$

then for every $y \in Y$ the probability that the set $H^{-1}(y)$ is empty is approximately e^{-1}. □

Proposition 3.. ([10]) Let $C_1 : X \to Y,\ C_2 : X \to Y$ are two particular functions, chosen uniformly at random (where $X = Y = \{0,1\}^n$). If we define a function $C : X \to Y$ as a composition:

$$C = C_1 \circ C_2 \tag{4}$$

then for every $y \in Y$ the probability P_2 that the set $C^{-1}(y)$ is empty is $P_2 = e^{-1+e^{-1}}$.

Proof: We can use the same technique used in the proof of Proposition 1 in [9] but extended to two domains (i.e. one intermediate domain Z) since we have a composition of two functions. Thus let us put the following notation:

$$C \equiv C_1 \circ C_2 : X \xrightarrow{C_1} Z \xrightarrow{C_2} Y$$

From Proposition 1 it follows that for every $z \in Z$ the probability that the set $C_1^{-1}(z)$ is empty is approximately e^{-1} i.e. the probability that z has a preimage is $(1 - Pr\{C_1^{-1}(z) = \varnothing\}) = (1 - e^{-1})$.

Now, for the probability that the set $C^{-1}(y)$ is empty (for every $y \in Y$) we have:

$$Pr\{C^{-1}(y) = \varnothing\} = \left(1 - \frac{1}{2^n}\right)^{2^n(1-Pr\{C_1^{-1}(y)=\varnothing\})} \approx e^{-1+e^{-1}}.$$

□

Lemma 2.. ([10]) $C_1, C_2, \ldots C_k : X \to Y$ are k particular (not necessary different) functions, chosen uniformly at random (where $X = Y = \{0,1\}^n$). If we define a function $C : X \to Y$ as a composition:

$$C = C_1 \circ C_2 \circ \ldots \circ C_k \tag{5}$$

then for every $y \in Y$ the probability P_k that the set $C^{-1}(y)$ is empty is approximately $P_k = e^{-1+P_{k-1}}$, where $P_1 = e^{-1}$.

Proof: The lemma can be proved by using mathematical induction for the value of k and the Proposition 3.. □

The Lemma 2. models the probability of some element in Y to have a preimage if we apply consecutively different random functions defined over the same narrow domain $\{0,1\}^n$. Is the sequence P_k convergent? If yes, what is the limit value and what is the speed of the convergence?

In this paper we will give answers on these questions, but we have to stress that the mathematical proofs for some of those answers will be given elsewhere.[3, 4]

Lemma 3.. ([10]) Let $P_1 = e^{-1}$ and $P_k = e^{-1+P_{k-1}}$. Then the following limit holds:

$$\lim_{i \to \infty} (\log_2(1 - P_{2^i}) + i - 1) = 0 \tag{6}$$

[3] In the initial version of this paper the Lemma 3. was given as a Conjecture, but in the mean time Zoran Šunić from the Department of Mathematics, Texas A&M University, USA has proven it for which we express him an acknowledgement.

[4] After reading our first version of the paper submited to the eprint archive, we got an email from Ernst Schulte-Geers from the German BSI for which we express him an acknowledgement, pointing out that in fact Lemma 3. was known long time ago from the paper of Flajolet and Odlyzko [19].

As a direct consequence of Lemma 3. is the following Corollary:

Corollary 2.. ([10]) The entropy $E(C(X))$ of the set $C(X) = \{C(x) \mid x \in X\}$, where the function C is a composition of 2^i functions mapping the domain $\{0,1\}^n$ to itself, as defined in (5) is:

$$E(C(X)) = n + \log_2(1 - P_{2^i}) \tag{7}$$

□

The last corollary can be interpreted in the following way: With every consecutive mapping of a narrow domain $\{0,1\}^n$ to itself by any random function defined on that domain, the volume of the resulting image is shrinking. The speed of the shrinking is exponentially slow i.e. for shrinking the original volume 2^n of $X = \{0,1\}^n$ to an image set with a volume of 2^{n-i+1} elements, we will need to define a composition of 2^i functions i.e.,

$$C = C_1 \circ C_2 \circ \ldots \circ C_{2^i}.$$

3 Details how BLAKE, Hamsi, SHAvite-3, Skein and SHA-2 aberrate from ideal random functions

3.1 The case of the cryptographic function BLAKE

Let us analyze the iterated procedure defined in BLAKE-32 (and the case for BLAKE-64 is similar). First, a message M is properly padded:

$$M \leftarrow M||1000\ldots0001\langle l_{64}\rangle$$

and then is parsed into N, 512-bit chunks:

$$M \equiv m^0, \ldots, m^{N-1}.$$

The variable l^i is defined as a number of processed bits so far. We quote the description of the padding from [4]:

> For example, if the original (non-padded) message is 600-bit long, then the padded message has two blocks, and $l^0 = 512$, $l^1 = 600$. A particular case occurs when the last block contains no original message bit; for example a 1020-bit message leads to a padded message with three blocks (which contain respectively 512, 508, and 0 message bits), and we set $l^0 = 512$, $l^1 = 1020$, $l^2 = 0$.

Now, let us take that we want to hash just 1020-bit long messages M with BLAKE-32 (the size of 1020 is chosen to fit the example in the original documentation, but it can be also 1024, or any multiple of 512, which is a common action if BLAKE-32 would be used as a PRF or KDF hashing a pool of randomness that is exactly multiple of 512 bits). The iterative procedure will be:

$h^0 = \text{IV}$

for $i = 0, \ldots, 2$

$h^{i+1} = \textbf{compress}(h^i, m^i, s, l^i)$

return h^3

or equivalently:

$$BLAKE\text{-}32(M) = \textbf{compress}(\textbf{compress}(\textbf{compress}(h^0, m^0, s, 512), m^1, s, 1020), m^2, s, 0).$$

Note that $m^2 = const$ does not have any bit from the original 1020-bit message M. So, we have that the final 256-bit hash value that is computed is:

$$BLAKE\text{-}32(M) = \textbf{compress}(h^2, const, s, 0).$$

If we suppose that the compression function of $BLAKE$-32 is ideal, from the Proposition 1. and Lemma 1. we get that there is a huge set $Y_\varnothing \subseteq \{0,1\}^{256}$, with a volume $|Y_\varnothing| \approx 0.36 \times 2^{256} \approx 2^{254.55}$ i.e.

$$Pr\{BLAKE\text{-}32^{-1}(M) = \varnothing\} = e^{-1}.$$

On the other hand, for an ideal random function $W : \{0,1\}^{1020} \to \{0,1\}^{256}$ from Proposition 2. we have that

$$Pr\{W^{-1}(M) = \varnothing\} = e^{-2^{764}}.$$

3.2 The case of the cryptographic function Hamsi

Let us analyze the iterated procedure defined in Hamsi-256 (and the case for Hamsi-512 is similar). Let M be a properly padded message i.e.

$$M = M_1||\dots M_{l-1}||M_l,$$

where the last block M_l does not contain message bits but the 64-bit encoding of the length in bits of the original message M i.e. we have that $M_l = \langle l_{64} \rangle$.

Let us hash messages M of length 1024-bits. Then the padded message will have 35 blocks of 32-bits and will have the following form:

$$M = M_1||\dots M_{32}||1000\dots000||000\dots000||00000000000000000000010000000000$$

The iterative procedure for hashing these messages will be:

$$h_i = (T \circ P \circ C(E(M_i), h_{i-1})) \oplus h_{i-1}, \quad h_0 = iv_{256}, \quad 0 < i < 35,$$

$$\text{Hamsi-256}(M) = (T \circ P_f \circ C(E(M_l), h_{l-1})) \oplus h_{l-1}$$

For the precise definition of the variables used in the iterative process see the Hamsi documentation [5].

From the definition of Hamsi-256 it is obvious that it can act at most as an ideal random function with narrow-domain, but obviously the last call of the compression function for messages of length 1024 bits has no message bits, thus the deviance from an ideal function that maps 1024 bits to 256-bit digest is huge as it is shown in Proposition 1., Proposition 2. and Lemma 1..

3.3 The case of the cryptographic function SHAvite-3

We will analyze 256-bit version of SHAvite-3, SHAvite-3-256 (and the 512-bit version is similar). It uses the HAsh Iterative FrAmework - HAIFA. Hashing with HAIFA has three steps:

1. Message padding, according to the HAIFA padding scheme.
2. Compressing the message using a HAIFA-compatible compression function.
3. Truncating the output to the required length.

Since we will work with 256-bit version of SHAvite-3, the third truncating step will be omitted (which is crucial for our analysis).

We give here the description of the hashing by SHAvite-3 extracted from the documentation in [6]:

The compression is done using a compression function with four inputs:

- A chaining value (of length m_c),
- A message block (of length n),
- The number of bits hashed so far including the current block (a counter of length c),
- A salt (of length s).

In order to compute $HAIFA^C_{salt}(M)$ using the compression function $C : \{0,1\}^{m_c} \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^s \mapsto 0,1^{m_c}$ the message is first padded, and divided into l blocks of n bits each, $pad(M) = M_1||M_2||\dots||M_l$ Now, the user:

1. Sets h_0 as the initial value (according to the procedure defined in Section 3.3).
2. Computes iteratively
$$h_i = C(h_{i-1}, M_i, \#bits, salt).$$
3. Truncates h_l (according to the procedure defined in Section 3.3).
4. Output the truncated value as $HAIFA^C_{salt}(M)$.

The padding rule in SHAvite-3 works to pad the original message such that it is multiple of n bits where $n = 512$ for SHAvite-3-256 or $n = 1024$ for SHAvite-3-256. The padding of a message M has the following steps:

1. Pad with a single bit of 1.
2. Pad with as many 0 bits as needed such that the length of the padded message (with the 1 bit and the 0s) is congruent modulo n to $(n - (t + r))$.
3. Pad with the message length encoded in t bits.
4. Pad with the digest length encoded in r bits.

When a full padding block is added (i.e., the entire original message was already processed by the previous calls to the compression function, and the full message length was already used as an input to the previous call as the #bits parameter), the compression function is called with the #bits parameter set to **zero**.

So, let us hash messages M that are 1024-bits long (this analysis works with messages that are multiple of 512 or 1024 bits) with SHAvite-3-256. The padded message M is:

$$pad(M) = M_1||M_2||M_3,$$

where the final padding block M_3 does not have any message bits and the truncation phase is omitted. Thus,

$$\text{SHAvite-3-256}(M) = HAIFA^C_{salt}(M) = C(C(C(h_0, M_1, 512, salt), M_2, 1024, salt), M_3, 0, salt).$$

Since the final padding block M_3 does not have any message bits, for messages of length 1024 bits, we can treat it as $M_3 = const$ and

$$\text{SHAvite-3-256}(M) = HAIFA^C_{salt}(M) = C(h_2, const, 0, salt).$$

This is exactly the case that is covered by the Proposition 1. (or Lemma 1.). Under the assumption that SHAvite-3-256 compression function acts as ideal finite-narrow-domain random function that maps 256 bits to 256 bits, we conclude that SHAvite-3-256(M) differs significantly from an ideal finite-wide-domain random function that maps strings of 1024 bits to hash values of 256 bits.

3.4 The case of the cryptographic function Skein

The subject of this analysis are the variants Skein-256-256 and Skein-512-512 (which according to the documentation is the primary proposal of the designers [7]). It is an interesting fact that the designs Skein-512-256 and Skein-1024-512 which are double-pipe designs are not suffering from the defects of narrow-pipe compression functions that are extended to the infinite domain and are not affected by this analysis.

The main point of our analysis of Skein-256-256 and Skein-512-512 is the fact that Skein is using a final invocation of UBI (Unique Block Iteration) without an input by any message bits. Namely it uses the output function $Output(G, N_0)$ which takes as parameters the final chaining value G and the number of required output bits N_0. The output is simple run of the UBI function in a counter mode:

$$\begin{aligned} O := \quad & UBI(G, ToBytes(0, 8), T_{out}2^{120})|| \\ & UBI(G, ToBytes(1, 8), T_{out}2^{120})|| \\ & UBI(G, ToBytes(2, 8), T_{out}2^{120})|| \\ & \ldots \end{aligned}$$

Now let us use Skein-256-256 (the case for Skein-512-512 is similar). In that case the chaining value G has 256 bits, and the UBI() is called only once. We can treat that one call of UBI() as a finite-narrow-domain mapping that maps 256 bits of G to 256 bits of O. Thus, from the point of view of Proposition 1. and Proposition 2. we have a clear difference between an ideal function that maps the huge domain $\mathcal{D}$ into the set $\{0,1\}^{256}$ and the function of Skein-256-256. Namely, under the assumption that UBI() acts as an ideal finite-narrow-domain function, from Proposition 1. and Lemma 1. we have that there exist a huge set $Y_{\varnothing} \subseteq \{0,1\}^{256}$, with a volume $|Y_{\varnothing}| \approx 0.36 \times 2^{256} \approx 2^{254.55}$ i.e.

$$Pr\{\text{Skein-256-256}^{-1}(M) = \varnothing\} = e^{-1}.$$

3.5 The case of SHA-2

Let us analyze the iterated procedure defined in SHA-256 (and the case for SHA-512 is similar)[8]. First, a message M is properly padded:

$$M \leftarrow M||1000\ldots000\langle l_{64}\rangle$$

where the 64-bit variable $\langle l_{64}\rangle$ is defined as the length of the original message M in bits. Then the padded message is parsed into N, 512-bit chunks:

$$M \equiv m^0, \ldots, m^{N-1}.$$

The iterative procedure for hashing the message M then is defined as:

$h^0 = \text{IV}$

for $i = 0, \ldots, N-1$

$h^{i+1} =$ **CompressSHA256**(h^i, m^i)

return h^N

where **CompressSHA256**() is the compression function for SHA-256.

Now, let us hash messages that are extracted from some pool of randomness with a size of 1024 bits. The padding procedure will make the final block that would be compressed by the **CompressSHA256**() to be always the same i.e. to be the following block of 512 bits:

$$\underbrace{1000\ldots00010000000000}_{512\ bits}$$

If we suppose that the compression function **CompressSHA256**() is ideal, from the Proposition 1. and Lemma 1. we get that there is a huge set $Y_{\varnothing} \subseteq \{0,1\}^{256}$, with a volume $|Y_{\varnothing}| \approx 0.36 \times 2^{256}$ i.e.

$$Pr\{SHA\text{-}256^{-1}(M) = \varnothing\} = e^{-1}.$$

On the other hand, for an ideal random function $W : \{0,1\}^{1024} \rightarrow \{0,1\}^{256}$ from Proposition 2. we have that

$$Pr\{W^{-1}(M) = \varnothing\} = e^{-2^{768}} \approx 0.$$

4 Practical consequences of the observed defects of the narrow-pipe designs

We point out several concrete protocols that are widely used and where the observed abberations of narrow-pipe hash designs from the ideal random function will be amplified due to the iterative use of hash functions in those protocols.

4.1 Reduced entropy outputs from narrow-pipe hash functions

The first practical consequence is by direct application of the Lemma 2..

Let us consider the following scenario: We are using some hash function that gives us 256 bits of output, and we have a pool of randomness of a size of 2^{20} blocks (where the block size is the size of message blocks used in the compression function of that hash function). The pool is constantly updated by actions from the user and from the running operating system. We need random numbers obtained from that pool that will have preferably close to 256 bits of entropy.

If we use narrow-pipe hash design, then depending on the nature of the distribution of the entropy in the randomness pool, we can obtain outputs that can have outputs with entropy as low as 237 bits or outputs with entropy close to 256 bits.

More concretely, if the distribution of the entropy in the pool is somehow concentrated in the first block (or in the first few blocks), then from the Lemma 2. we have that the entropy of the output will not be 256 bits but "just" slightly more than 237 bits. We say "just" because having 237 bits of entropy is really high enough value for any practical use, but it is much smaller than the requested value of 256 bits of entropy. In a case of more uniform distribution of the entropy in the whole pool of randomness, the narrow-pipe hash design will give us outputs with entropies close to 256 bits. The cases where due to different reasons (users habits, user laziness, regularity of actions in the operating system, to name some), the pool is feeded with randomness that is concentrated more on some specific blocks, the outputs will have entropy between 237 and 256 bits.

On the other hand, we want to emphasize, if in all this scenarios we use wide-pipe hash design, the outputs will always have close to 256 bits of entropy, regardless where the distribution of the entropy in the pool will be.

From this perspective, we can say that although the consequences can be just of theoretical interest, there are real and practical scenarios where the abberation of narrow-pipe hash design from ideal random functions can be amplified to some more significant and theoretically visible level.

4.2 Reduced entropy outputs from HMACs produced by narrow-pipe hash functions

HMAC [13] is one very popular scheme for computing MAC - Message Authentication Codes when a shared secret is used by the parties in the communication. We are interested in a possible loss of entropy in the HMAC construction if we use narrow-pipe hash constructions.

Proposition 4.. ([10]) Let a message M be of a size of 256 bits and has a full entropy of 256 and let "*secret*" is shared secret of 256 bits. If in HMAC construction we use a narrow-pipe hash function that parses the hashed messages in 512 blocks, then $mac = HMAC(secret, M)$ has an entropy of 254.58 bits.

Proof: Let we use the hash function SHA256 that has the compression function **CompressSHA256**(). From the definition of HMAC we have that

$$mac = HMAC(secret, M) = hash((secret \oplus opad)||hash((secret \oplus ipad)||M))$$

where $\oplus$ is the operation of bitwise xoring and $||$ is the operation of string concatenation.

Computing of *mac* will use four calls of the compression function **CompressSHA256**() in the following sequence:

1. $h_1 = \mathbf{CompressSHA256}(iv_{256}, (secret \oplus ipad)) \equiv C_1(iv_{256})$
2. $h_2 = \mathbf{CompressSHA256}(h_1, M||CONST256) \equiv C_2(h_1)$, where

$$CONST256 = \underbrace{1000\ldots001000000000}_{256\ bits}.$$

3. $h_3 = \mathbf{CompressSHA256}(iv_{256}, (secret \oplus opad)) \equiv C_3(iv_{256})$
4. $mac = h_4 = \mathbf{CompressSHA256}(h_3, h_2||CONST256) \equiv C_4(h_3)$

For a fixed secret key "*secret*" the value h_1 will be always the same and will be obtained with $C_1(iv_{256})$. The function C_2 depends from the message M that has a full entropy of 256 bits, thus C_2 is not one function but it represent a whole class of 2^{256} random functions mapping 256 bits to 256 bits. Thus, we can consider that any call of the function C_2 decreases the entropy of h_2 to $256 + \log_2(1 - P_1)$.

For the value h_3 we have a similar situation as for h_1. Similarly as $C_2()$, the function $C_4()$ is a class of random functions that depends of the value h_2. Since we have already determined that the entropy of h_2 is $256 + \log_2(1 - P_1)$, it follows that for computing the entropy of *mac* we can apply the Corollary 2. obtaining that entropy $E(mac)$ is

$$E(mac) = 256 + \log_2(1 - P_2),$$

where $P_1 = \frac{1}{e}$, and $P_2 = e^{-1+\frac{1}{e}}$ which gives us the value $E(mac) = 254.58$. □

What is the difference if we use a double-pipe hash function instead of narrow-pipe in Proposition 4.? The first difference is off course the fact that the initialization variable in the compression function as well as the intermediate variables h_1, h_2, h_3 and h_4 are 512 bits long, and we will need final chopping. Then, under the assumption that the compression function acts as ideal random function mapping 512 bits to 512 bits, and having the entropy of the message M to be 256, we have that the entropy of h_2 is also 256 (not $256 + \log_2(1 - P_1)$). The same applies for the entropy of h_4 which will give us that the entropy of *mac* after the chopping will be 256 bits.

Proposition 5.. ([10]) Let a message M be of a size of 512 bits and has a full entropy of 512 and let "*secret*" is shared secret of 256 bits. If in HMAC construction we use a narrow-pipe hash function that parses the hashed messages in 512 blocks, then $mac = HMAC(secret, M)$ has an entropy of 254.58 bits.

Proof: Let we use the hash function SHA256 that has the compression function **CompressSHA256**(). From the definition of HMAC we have that

$$mac = HMAC(secret, M) = hash((secret \oplus opad)||hash((secret \oplus ipad)||M))$$

where $\oplus$ is the operation of bitwise xoring and $||$ is the operation of string concatenation.

Computing of *mac* will use five calls of the compression function **CompressSHA256**() in the following sequence:

1. $h_1 =$ **CompressSHA256**$(iv_{256}, (secret \oplus ipad)) \equiv C_1(iv_{256})$
2. $h_2 =$ **CompressSHA256**$(h_1, M) \equiv C_2(h_1)$
3. $h_3 =$ **CompressSHA256**$(h_2, CONST512) \equiv C_3(h_2)$, where
$$CONST512 = \underbrace{1000\ldots0001100000000}_{512\ bits}.$$
4. $h_4 =$ **CompressSHA256**$(iv_{256}, (secret \oplus opad)) \equiv C_4(iv_{256})$
5. $mac = h_5 =$ **CompressSHA256**$(h_4, h_3||CONST256) \equiv C_5(h_4)$, where
$$CONST256 = \underbrace{1000\ldots000100000000}_{256\ bits}.$$

Above, we consider the call of the function **CompressSHA256**$(iv_{256}, (secret \oplus ipad))$ as a call to an ideal random function $C_1 : \{0,1\}^{256} \to \{0,1\}^{256}$ that will map the 256-bit value iv_{256} to the 256-bit value h_1. The function C_2 is a specific one. Actually, since it depends from the message M that has a full entropy of 512 bits, C_2 is not one function but it represent a whole class of 2^{512} random functions mapping 256 bits to 256 bits. Thus, we can consider that there is no entropy loss for h_2 i.e. it has a full entropy of 256 bits.

For the value h_3 we start to consider the entropy loss again from the value 256. The call to the function C_3 will decrease the entropy of h_3 to $256+\log_2(1-P_1)$. For a fixed secret key "*secret*" the value h_4 will be always the same and will be mapped with $C_5(h_4)$ to the final value *mac*. Similarly as $C_2()$, the function $C_5()$ is a class of random functions that depends of the value h_3. Since we have already determined that

the entropy of h_3 is $256 + \log_2(1 - P_1)$, it follows that for computing the entropy of *mac* we can apply the Corollary 2. obtaining that entropy $E(mac)$ is

$$E(mac) = 256 + \log_2(1 - P_2),$$

where $P_1 = \frac{1}{e}$, and $P_2 = e^{-1+\frac{1}{e}}$ which gives us the value $E(mac) = 254.58$. □

Again, if we are interested to know what will happen if we use a double-pipe hash function in the Proposition 5., we can say that the entropy of the 512-bit variable h_3 will start to decrease from the value 512 and will be $512 + \log_2(1 - P_1)$, and the entropy of h_5 will be $512 + \log_2(1 - P_2)$, that after the final chopping will give us a *mac* with full entropy of 256.

4.3 Loss of entropy in the pseudo-random function of SSL/TLS 1.2

SSL/TLS 1.2 is one very popular suite of cryptographic algorithms, tools and protocols defined in [14]. Its pseudo-random function PRF which is producing pseudo-random values based on a shared secret value "*secret*", a seed value "*seed*" (and by an optional variable called "*label*") is defined as follows:

$$PRF(secret, label, seed) = P_{<hash>}(secret, label \mid\mid seed), \tag{8}$$

where the function $P_{<hash>}(secret, seed)$ is defined as:

$$\begin{aligned} P_{<hash>}(secret, seed) = {} & HMAC_{<hash>}(secret, A(1) \mid\mid seed) \mid\mid \\ & HMAC_{<hash>}(secret, A(2) \mid\mid seed) \mid\mid \\ & HMAC_{<hash>}(secret, A(3) \mid\mid seed) \mid\mid \\ & \ldots \end{aligned} \tag{9}$$

and where $A(i)$ are defined as:

$$\begin{aligned} A(0) &= seed \\ A(i) &= HMAC_{<hash>}(secret, A(i-1)). \end{aligned} \tag{10}$$

Proposition 6.. ([10]) Let "*secret*" is shared secret of 256 bits. The entropy $E(A(i))$ of the i-th value $A(i)$ as defined in the equation (10) for the hash function SHA-256 can be computed with the following expression:

$$E(A(i)) = 256 + \log_2(1 - P_{2i}) \tag{11}$$

where the values P_{2i} are defined recursively in the Lemma 2..

Proof: We can use the same technique described in the previous subsection and in the proof of Proposition 4.. Since we have two volume compressive calls of the compression function, and since the computation of $A(i)$ depends on the value of the previous value $A(i-1)$ in the computation of $A(i)$ we have $2i$ times shrinking of the entropy. □

As a direct consequence of the previous Proposition we have the following:

Corollary 3.. ([10]) Let the size of "$A(i) \mid\mid seed$" is 512 bits, and let "*secret*" is shared secret of 256 bits. For the i-th part $PRF_i = HMAC_{SHA-256}(secret,\ A(i) \mid\mid seed)$ as defined in the equation (9) the entropy $E(PRF_i)$ can be computed with the following expression:

$$E(PRF_i) = E(PRF_i) = 256 + \log_2(1 - P_{2i+3}) \tag{12}$$

Proof: Computing of PRF_i will use five calls of the compression function **CompressSHA256**() in the following sequence:

1. $h_1 = \mathbf{CompressSHA256}(iv_{256}, (secret \oplus ipad)) \equiv C_1(iv_{256})$
2. $h_2 = \mathbf{CompressSHA256}(h_1, A(i) \mid\mid seed) \equiv C_2(h_1)$
3. $h_3 = \mathbf{CompressSHA256}(h_2, CONST1024) \equiv C_3(h_2)$, where

$$CONST1024 = \underbrace{1000\ldots0010000000000}_{512\ bits}.$$

4. $h_4 = \mathbf{CompressSHA256}(iv_{256}, (secret \oplus opad)) \equiv C_4(iv_{256})$
5. $PRF_i = h_5 = \mathbf{CompressSHA256}(h_4, h_3||CONST256) \equiv C_5(h_4)$, where

$$CONST256 = \underbrace{1000\ldots000100000000}_{256\ bits}.$$

Similarly as in Proposition 5. we can see that the function C_2 is a specific one since it depends from $A(i)$ || *seed*. For a given and fixed *seed*, the entropy of "$A(i)$ || *seed*" is the entropy of $A(i)$ and from Proposition 6., it is $E(A(i)) = 256 + \log_2(1 - P_{2i})$ bits. From here it follows that the entropy of h_2 is $E(h_2) = 256 + \log_2(1 - P_{2i+1})$.

For the value h_3 we further have $E(h_2) = 256 + \log_2(1 - P_{2i+2})$. For a fixed secret key "*secret*" the value h_4 will be always the same and will be mapped with $C_5(h_4)$ to the final value PRF_i, with an entropy

$$E(PRF_i) = 256 + \log_2(1 - P_{2i+3}).$$

□

For illustration we can say that the entropy of $E(PRF_1) = 253.463$, but the entropy of $E(PRF_{60}) = 250.00$.

On the other hand, having in mind the discussions about the different attitude of double-pipe hash function in used HMACs, it is clear that with double-pipe hash designs we will not face this kind of entropy loss.

5 "Faster" collision attack on narrow-pipe hash designs

As pointed to us in a private message by Bart Preneel, the "faster" collision attack is very well known in the context of MAC algorithms since there it makes much more sense to evaluate attacks in terms of number of chosen texts rather than in terms of number of compression function calls. That was a concern in the design of Message Authenticator Algorithm (MAA) by Davies and Clayden in the 1980s [15] and is described in details in Preneel's PhD thesis [16] (together with an analysis of the effect). This very same observation was used in the Crypto'95 paper of Paul van Oorschot and Preneel [17] to reduce the number of queries to find a MAC collision and hence a forgery for non-injective MAC functions (for a fixed message block).

Although it is straightforward to conclude that this same observation applies to hash functions, from the point of view of the SHA-3 competition, it is worth to note that there are candidates (wide-pipe candidates) that are not affected by this observation.

In this section we will use the following notation:

- $C(h, m)$ - a compression function C with chaining variable h and message block variable m.
- *hlen* - the length of the chaining variable, i. e. the length of compression function output.
- *mlen* - the length of the message block.
- *hashlen* - the length of the hash function output.

If the compression function has the property, that for every value m the function $C(h, m) \equiv C_m(h)$ is an **i**deal **r**andom **f**unction of the variable h, we denote it as $IRF(h)$.

If the compression function has the property, that for every value h the function $C(h, m) \equiv C_h(m)$ is an **i**deal **r**andom **f**unction of the variable m, we denote it as $IRF(m)$.

The hash function is defined by a narrow-pipe compression function (NPCF), iff $hashlen = hlen = \frac{mlen}{2}$ and the compression function is $IRF(h)$ and $IRF(m)$.

The hash function is defined by a wide-pipe compression function (WPCF), iff $hashlen = \frac{hlen}{2} = \frac{mlen}{2}$ and the compression function is $IRF(h)$ and $IRF(m)$.

Theorem 1.. ([11]) Suppose that the hash function $H : \{0,1\}^* \to \{0,1\}^n$ is defined by a narrow-pipe compression function $C : \{0,1\}^n \times \{0,1\}^{mlen} \to \{0,1\}^n$. Then we can find a collision (M, M') for the hash function H using much less than $2^{n/2}$ calls to the hash function H (the lower bound of the birthday paradox).

Proof: For the sake of simplicity, let us suppose $n = hashlen = 256$. The general case is analogous. In this case, the hashed message is padded and divided into 512-bit blocks. Let us suppose that a message M (for instance the content of a hard disk or a RAM memory) is divided into two parts A and B, i.e. $M = A||B$, where the part A consist of just one message block of 512 bits, and the number of 512-bit blocks in the part B is $N = 2^{35}$ (in case of current 2TByte HDD). Let us denote by hA the intermediate chaining value, obtained after hashing the part A of the message M and let us suppose that the content of the part B is never changing - so it consists of constant message blocks $const_1$, $const_2$, ..., $const_N$. We compute the final hash with the following iterative procedure:

$$\begin{gathered} h_1 = C(hA, const_1) \\ h_2 = C(h_1, const_2) \\ h_3 = C(h_2, const_3) \\ \dots \\ h_N = C(h_{N-1}, const_N) \\ H(M) = h_N \end{gathered}$$

If the compression function C is $IRF(h)$, then the chaining values are loosing the entropy in every of the N steps above. From Corollary 3[10] we obtain that the entropy of the final hash h_N is equal to

$$E(hash) = hashlen + 1 - log_2(N),$$

and for $N = 2^{35}$ it gives

$$E(hash) = 222.$$

If we compute hash values for 2^{111} different parts A (whereas the part B remains unchanged), we will obtain 2^{111} hash values h_N. According to the birthday paradox it is sufficient for finding a collision in the set of these values with probability around $\frac{1}{2}$. Cryptographically strong hash function H should require approximately 2^{128} hash computations. □

Corollary 4.. ([11]) For hash functions $H()$ constructed as in Theorem 1., finding a pair of colliding messages (M, M') that are long $N = 2^k$ blocks, can be done with $O(2^{n/2-k/2})$ calls to the hash function $H()$. □

Note 1: If we count the number of calls to the compression function $C(H_i, M_i)$, then with our collision strategy we are calling actually more times the compression function. Namely, $2^{111} \times 2^{35} = 2^{145}$. So, our finding does not reduces the $\frac{n}{2}$ bits of collision security that narrow-pipe functions are declaring, but we clearly show that narrow-pipe designs have a property when we count the calls to the hash function as a whole, the birthday paradox bound of $2^{n/2}$ calls to the hash function is clearly lowered.

Note 2: This technique is not applicable to wide-pipe hash functions because the entropy reduction after applying the compression function $C(H_i, M_i)$ to different message blocks starts from the value $hlen$ which is two times bigger than $hashlen$ i.e. $hlen = 2hashlen$. So the final reduction from $hlen$ to $hashlen$ bits will make the technique described in this note ineffective against wide-pipe designs.

6 Length extension attack on narrow-pipe SHA-3 candidates

In their call for the SHA-3 competition [3], NIST has defined several security requirements such as collision resistance of $\frac{n}{2}$ bits, preimage resistance of n bits, resistance against second preimages of 2^{n-k} bits for messages long 2^k bits and resistance against length-extension attack. However, in the SHA-3 call there is no clear statement how many bits of security should SHA-3 candidates provide against length extension attack.

On my request for clarification submitted to the SHA-3 hash forum list on 12 December 2008, I got the following answer by the NIST representative submitted to the hash forum list on 14 January 2009:

> "We expect the winning n-bit hash function to be able to provide n bits of security against length extension attacks. That is, given $H(M)$, with M wholly or partially unknown to the attacker: the cost of finding (Z, x) so that $x = H(M||Z)$ should be greater than or equal to either the cost of guessing M or 2^n times the cost of computing a typical hash compression function."

In this section we will show that four SHA-3 candidates that are narrow-pipe designs do not provide n bits of security.

6.1 A generic modeling of the narrow-pipe iterative finalization

In order to launch a length-extension attack to the narrow-pipe designs we will need to model the finalization of the iterative process that narrow-pipe designs do when they are processing messages. Moreover, we will assume that the length of the digested messages is such that the final processed block does not have any bits from the message but is a constant $PADDING$ that consist only from the bits defined by the padding rule of that hash function.

In that case, the modeling of the narrow-pipe hash designs can be expressed by the following expression:

$$H = f(parameters, \textbf{compress}(H_{chain}, parameters, PADDING)) \tag{13}$$

and is graphically described in Figure 1.

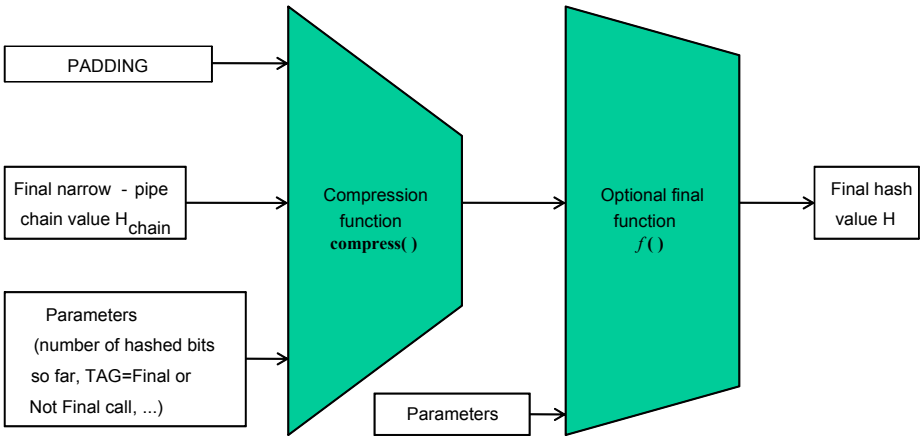

Figure 1: A graphic representation of narrow-pipe hash designs finalization of the iterative process of message digestion.

Note that in the narrow-pipe designs where the final function $f()$ is missing, we can treat it as the identity function in the expression (13) and that although the parts $parameters$ are different for all four designs, it will not change our analysis and our attack.

How narrow-pipe designs are protected from the length-extension attack?

Since the designers of narrow-pipe hash functions are designing the compression function as one-way pseudo-random function, the value H_{chain}, which is the internal state of the hash function, is hidden from the attacker. That means that by just knowing the final hash value H it is infeasible for the attacker to find the preimage H_{chain} that has produced that value H. Consequently, the attacker will have a difficulty to produce a message Z such that only by knowing the value H (where $H = Hash(M)$ and the message M is unknown to the attacker), he/she can produce a valid hash value $x = Hash(M||Z)$.

In what follows our goal will be to show that the attacker can recover that internal state H_{chain} with much less complexity than 2^n calls to the compression function - the complexity that NIST requires in order to claim that the design is offering n bits of security against the length-extension attack.

A generic length-extension attack on narrow-pipe hash functions
1. **One time pre-computation phase** **Step 0.** Fix the length of the messages such that the $PADDING$ block does not posses any message bits. **Step 1.** Produce 2^k pairs (h_{chain}, h) for random values h_{chain} with the expression: $h = f(parameters, \textbf{compress}(h_{chain}, parameters, PADDING))$. This phase has a complexity of 2^k calls to the compression function (or 2^{k+1} calls if the design has a final transformation $f()$). 2. **Query (attack) phase** **Step 2.** Ask the user to produce a hash value $H(M)$ where M is unknown (but its length is fixed in **Step 0**). **Step 3.** If there exists a pre-computed pair (h'_{chain}, h') such that $H(M) = H = h'$, put $H_{chain} = h'_{chain}$, put whatever message block Z and produce a valid $x = H(M\|Z)$.

Table 1: A generic length-extension attack on narrow-pipe hash functions

6.2 The attack

Our attack is based on the old Merkle's observation [18] that when an adversary is given 2^k distinct target hashes, (second) preimages can be found after hashing about 2^{n-k} messages, instead of expected 2^n different messages. In our attack we use the Merkle's observation not on the whole hash function, but on the two final invocations of the compression function. In order our attack to work, we will assume that the length of the messages is such that after the padding, the final padded block is without any message bits (which is usual situation when the length of the message is a multiple of 256, 512 or 1024 bits).

A generic description of the length-extension attack on narrow-pipe hash functions is given in Table 1.

Proposition 7.. ([12]) The probability that the condition in **Step 3** is true is $\frac{1}{2^{n-k}}$.

Proof: The proof is a trivial application of the ratio between the volume of the pre-computed pairs (h_{chain}, h) which has a value 2^k and the volume of all possible hash values of n bits which is 2^n. □

Proposition 8.. ([12]) For the conditional probability that the corresponding h'_{chain} is the actual chaining value H_{chain} the following relation holds:

$$P(H_{chain} = h'_{chain} \mid H(M) = h') \geq 0.58 \times 2^{k-n} \approx 2^{k-n-0.780961}. \tag{14}$$

Proof: (Sketch) It is sufficient to notice that for an ideal random function $g : \{0,1\}^n \to \{0,1\}^n$ that maps n bits to n bits, the probability that an n-bit value has m preimages for the first 8 values of m is approximately given in the Table 2 (the precise analytical expressions for the given probabilities can be a nice exercise in the Elementary Probability courses).

Number of preimages m	Probability P
0	0.36787
1	0.36787
2	0.18394
3	0.06131
4	0.01533
5	0.00307
6	0.00051
7	0.00007

Table 2: The probabilities an n bit value to have m preimages for an ideal random function $g : \{0,1\}^n \to \{0,1\}^n$.

The relation (14) follows directly from Proposition 7. and from Table 2. □

Corollary 5.. ([12]) After approximately $2^{n-k+0.780961}$ queries the attacker should expect one successful length extension. □

Corollary 6.. ([12]) The security of narrow-pipe hash designs is upper bounded by the following values:

$$\max(2^{\frac{n}{2}}, 2^{n-k+0.780961}), \tag{15}$$

where $k \leq n$.

Proof: The minimal number of calls to the compression function of the narrow-pipe for which the attack can be successful is achieved approximately for $\frac{n}{2}$. □

The interpretation of the Corollary 6. is that narrow-pipe hash designs do not offer n bits of security against length-extension attack but just $\frac{n}{2}$ bits of security.

6.3 Why wide-pipe designs are resistant to our attack?

A natural question is raising about the security of wide-pipe hash designs and their resistance against the described length-extension attack. The reason of the success of our attack is the narrow size of just n bits of the hidden value H_{chain} that our attack is managing to recover with a generic collision search technique.

Since in the wide-pipe hash designs that internal state of the hash function has at least $2n$ bits, the search for the internal collisions would need at least 2^n calls to the compression function which is actually the value that NIST needs for the resistance against the length-extension attack.

7 Narrow-pipe designs break another NIST requirements

There is one more inconvenience with narrow-pipe hash designs that directly breaks one of the NIST requirements for SHA-3 hash competition [3]. Namely, one of the NIST requirement is: *"NIST also desires that the SHA-3 hash functions will be designed so that a possibly successful attack on the SHA-2 hash functions is unlikely to be applicable to SHA-3."*

Now, from all previously stated in this paper it is clear that if an attack is launched exploiting narrow-pipe weakness of SHA-2 hash functions, then that attack can be directly used also against narrow-pipe SHA-3 candidates.

8 Conclusions and future cryptanalysis directions

We have shown that narrow-pipe designs differ significantly from ideal random functions defined over huge domains. The first consequence from this is that they can not be used as an instantiation in security proofs based on random oracle model.

Several other consequences are also evident such as entropy loss in numerous algorithms and protocols such as HMAC and SSL, the possibility to find "faster" collisions in less than $2^{\frac{n}{2}}$ calls to hash function and the length-extension security of "just" $\frac{n}{2}$ bits.

All these properties of the narrow-pipe hash functions make them a no-match from security point of view to the wide-pipe hash designs.

References

[1] M. Bellare and P. Rogaway: "Random oracles are practical: A paradigm for designing efficient protocols," in CCS 93: Proceedings of the 1st ACM conference on Computer and Communications Security, pp. 6273, 1993.

[2] R. Canetti, O. Goldreich, S. Halevi: "The random oracle methodology, revisited", 30th STOC 1998, pp. 209–218.

[3] National Institute of Standards and Technology: "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family". Federal Register, 27(212):62212–62220, November 2007. Available: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf (2009/04/10).

[4] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan: "SHA-3 proposal BLAKE, Submission to NIST (Round 2)". Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/BLAKE_Round2.zip (2010/05/03).

[5] Özgül Kücük: "The Hash Function Hamsi, Submission to NIST (Round 2)". Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Hamsi_Round2.zip (2010/05/03).

[6] Eli Biham and Orr Dunkelman: "The SHAvite-3 Hash Function, Submission to NIST (Round 2)". Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/SHAvite-3_Round2.zip (2010/05/03).

[7] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker: "The Skein Hash Function Family, Submission to NIST (Round 2)". Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Skein_Round2.zip (2010/05/03).

[8] NIST FIPS PUB 180-2, ""Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce, August 2002.

[9] D. Gligoroski: "Narrow-pipe SHA-3 candidates differ significantly from ideal random functions defined over big domains", NIST hash-forum mailing list, 7 May 2010.

[10] D. Gligoroski and V. Klima: "Practical consequences of the abberation of narrow-pipe hash designs form ideal random functions", IACR eprint archive Report 2010/384, http://eprint.iacr.org/2010/384.pdf

[11] D. Gligoroski and Vlastimil Klima: "Generic Collision Attacks on Narrow-pipe Hash Functions Faster than Birthday Paradox, Applicable to MDx, SHA-1, SHA-2, and SHA-3 Narrow-pipe Candidates", IACR eprint archive Report 2010/430, http://eprint.iacr.org/2010/430.pdf

[12] D. Gligoroski: "Length extension attack on narrow-pipe SHA-3 candidates", NIST hash-forum mailing list, 18 Aug 2010.

[13] H. Krawczyk, M. Bellare and R. Canetti: "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[14] T. Dierks, E. Rescorla: "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, August 2008.

[15] D.W. Davies and D.O. Clayden: "The Message Authenticator Algorithm (MAA) and its Implementation", NPL Report DITC 109/88 February 1988, http://www-users.cs.york.ac.uk/~abhishek/docs/be/chc61/archives/maa.pdf (2010/11/21)

[16] B. Preneel, "Analysis and Design of Cryptographic Hash Functions," PhD thesis, Katholieke Universiteit Leuven, January 1993.

[17] B. Preneel, P.C. van Oorschot: "MDx-MAC and Building Fast MACs from Hash Functions", CRYPTO 1995, pp. 1 - 14

[18] R. C. Merkle - Secrecy, authentication, and public key systems, Ph.D. thesis, Stanford University, 1979, pp. 12 - 13, http://www.merkle.com/papers/Thesis1979.pdf (2010/08/08).

[19] P. Flajolet and A. M. Odlyzko: "Random Mapping Statistics", EUROCRYPT (1989), pp. 329–354