Annales UMCS Informatica AI XI, 2 (2011) 25–36
DOI: 10.2478/v10065-011-0013-8

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Evolution of the StreamHash hash function family

Michał Trojnara*

Faculty of Electronics and Information Technology, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

Abstract

This paper describes the evolution of StreamHash cryptographic hash function family proposed by the author. The first member of the StreamHash family was StreamHash (now called StreamHash1) function, accepted for the first round of SHA-3 competition organized by the US government standards agency NIST†. The competition has been started in order to select a new SHA-3 standard as the successor of SHA-2 family of cryptographic hash functions. Function StreamHash2 mostly addresses security weaknesses identified during the SHA-3 competition, while the sketch of function StreamHash3 attempts to improve resistance to side-channel attacks and performance properties. The paper starts with an overview of basic properties of cryptographic hash functions followed by the description of the StreamHash family design principles and its basic structure. Subsequent sections illustrate the way each subsequent function uses lessons learnt while designing and testing the previous one.

1. Overview of the StreamHash family

1.1. Cryptographic hash functions

The cryptographic hash function is a deterministic function that transforms arbitrary blocks of data into fixed-size values. The hash value for any given

E-mail address: Michal.Trojnara@mirt.net

†National Institute of Standards and Technology

message can be efficiently computed, i.e. $h(m)$ value can be easily computed for any given message m.

The following main security properties are required:

(1) It is not practically feasible to find a message transformed into a given hash (also known as preimage), i.e. for any given $h(m)$ value it is infeasible to find a corresponding message m. This property is called *preimage resistance*.
(2) It is not practically feasible to modify a message without changing its hash, i.e. for any given m_1 message it is infeasible to find another m_2 message (also known as the second preimage) such that $h(m_1) = h(m_2)$. This property is called *second preimage resistance*.
(3) It is not practically feasible to find two different messages with the same hash, i.e. it is infeasible to find two different messages m_1 and m_2 (also known as collision) such that $h(m_1) = h(m_2)$. This property is called *collision resistance*.

Some auxiliary properties are also often required:

(1) The hash function output should be indistinguishable from random numbers, so they can be used as a foundation for keystream generators. For example SSL and TLS [**1**] protocols use a mix of MD5 and SHA-1 to produce a sufficient number of master secret bits from an initial premaster secret and exchanged random values.
(2) The function should be resilient to length-extension attacks: given $h(m_1)$ and $len(m_1)$, but not m_1 itself, it should not be practically feasible to calculate $h(m_1\|\text{padding}\|m_2)$. This property can be used to break naive authentication schemes based on the hash functions. The HMAC[‡] [**2**] construction works around these problems.

Practical infeasibility should not be confused with theoretical computational complexity measures such as time or memory consumption. Theoretical measures cover either best, worst or average complexity. For cryptographic applications it is acceptable to violate any of the above properties as long as the probability of failure is negligible.

Cryptographic hash functions are often mistaken for checksums such as CRC32, only designed to detect accidental and not intentional modification of data.

Applications of cryptographic hash functions include:

- Digital signatures.
- Message authentication codes (MACs).
- User or device authentication.

[‡]keyed-Hash Message Authentication Code

1.2. Design rationale

Commonly used cryptographic hash functions are based on the Merkle-Damgård construction. The input message is processed in blocks. The message needs to be padded, so the length of the padded message is a multiple of the block size. Further processing is performed with a compression function. The function takes two inputs: a chaining variable and a message block. Compression function outputs the next value of the chaining variable. Each block of a padded message is iteratively processed with a compression function, starting with a predefined initial value of the chaining variable.

Compression function is performed in several rounds in order to provide required cryptographic properties. Each round only performs non-trivial (e.g. non-linear) operations on a subset of the chaining variable, while the remaining part is merely shifted. This is why multiple rounds are needed to achieve the avalanche effect, so that every bit of output depends on every bit of input of the compression function.

The approach of the StreamHash family is completely different. Instead of achieving the avalanche effect with multiple rounds, it directly updates the state vector on each octet of the input stream.

The structure of the StreamHash family is based on a well-known problem of solving a set of non-linear equations or CSP[§]. Common algorithms for solving CSPs [**3**] include backtracking, constraint propagation, and local search. The StreamHash family is designed, so that these algorithms cannot be applied. This property is ensured by the clear separation of the constraints. Solving a subset of all constraints does not make solving remaining constraints any easier.

No security proof is provided for the StreamHash family. Specifically no reduction from CSP or any other NP-complete problem has been demonstrated.

1.3. *NLF* transformation

The main building block of StreamHash family is a fast non-linear transformation *NLF* (Non-Linear Function).

Figure 1 illustrates inputs and outputs of the *NLF* transformation.

i	–	state vector index
$state_i$	–	previous state vector element
$state_{i+1}$	–	next state vector element
c	–	input octet index (added in StreamHash2)
b_c	–	input octet (StreamHash1, StreamHash2) or word (StreamHash 3)
r_c	–	*PRNG* value (added in Streamhash2)

[§]Constraint Satisfaction Problem

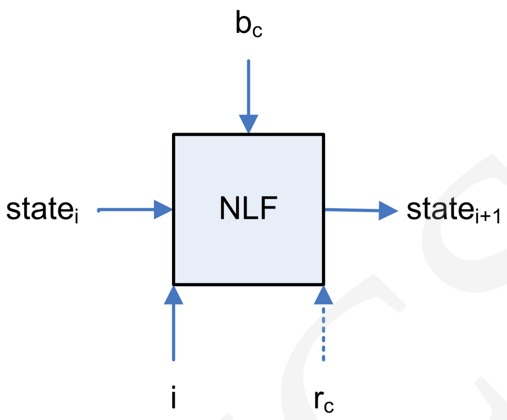

Fig. 1. NLF function.

1.4. Structure

See Figure 2 for the diagram of the StreamHash family structure.

A separate transformation is also applied in the finalization phase. Finalization is designed to prevent the length-extension attacks and to improve statistical properties of the output.

1.5. Advantages of the StreamHash family

The main advantages of the StreamHash family are:

- Clear and easy to analyze design.
- Negligible performance impact of machine endianness.
- High performance on 8-bit and 16-bit architectures.
- Easy to parallelize internal structure with theoretical performance up to a single clock cycle per input octet.
- Fast finalization resulting in low latency. This property is extremely important in real-time (e.g. multimedia) applications.
- Fast finalization resulting in high throughput for short messages.
- Minimal size of code, important for embedded systems.
- Minimal size of variables, important for embedded systems.
- Low size of static data.
- Scalability to use any multiple of 32 bits as the hash value length.

1.6. Limitations of the StreamHash family

The mathematical background is also not well studied in cryptographic applications. While this is not a direct weakness, extensive cryptanalysis is essential to trust a cryptographic primitive.

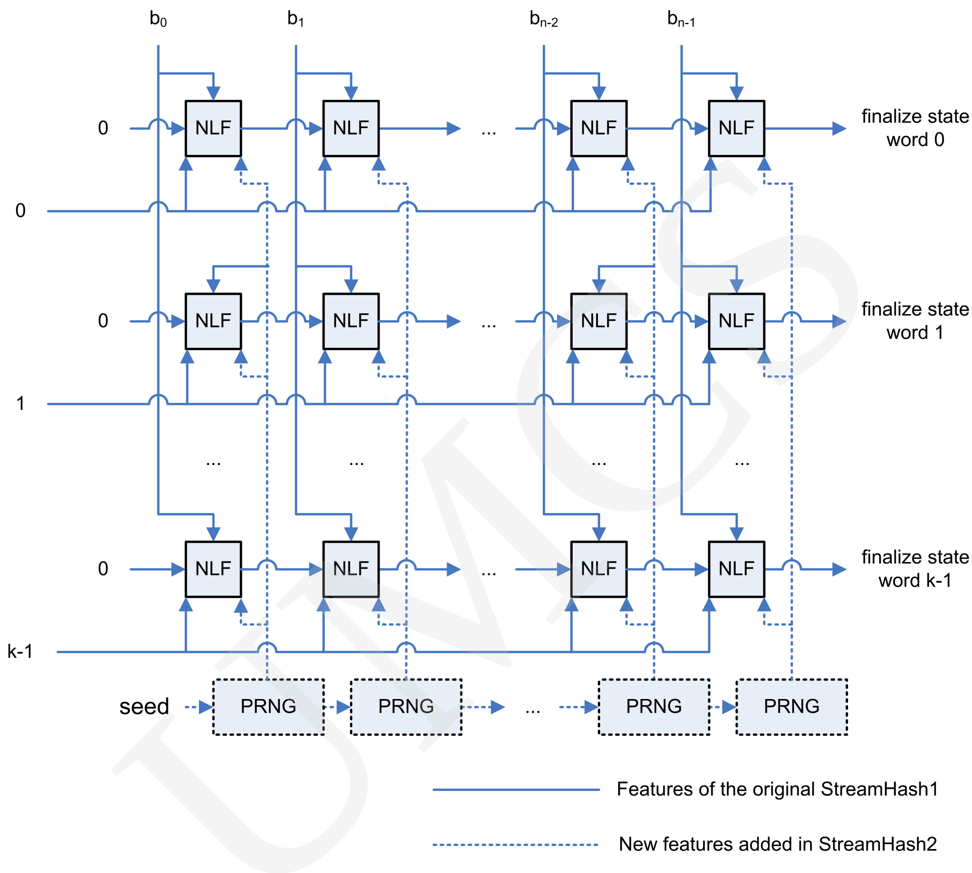

Fig. 2. StreamHash structure.

2. StreamHash1 function

2.1. Motivation

The StreamHash[**4**] (now called StreamHash1) algorithm was accepted for the first round of SHA-3 competition organized[**5**] NIST.

The main motivation for StreamHash1 was to demonstrate security of performance properties of the StreamHash family. The function was designed to be as simple as possible in order to simplify its cryptoanalysis. Specifically, no constants or transformations were included without a clear security rationale.

As an early and immature design, StreamHash suffered from severe security weaknesses.

2.2. State data

StreamHash1 state structure consists of:

- A vector of 32-bit values to hold the state for all processed octets, hereafter referred to as the state vector;
- The value of remaining bits in the last input data octet if it is not full; and
- The number $\{0, 1, \ldots 7\}$ of remaining bits in the last input data octet.

The length of the state vector is equal to the message digest size divided by 32, i.e. 7 for 224-bit digest, 8 for 256-bit digest, 12 for 384-bit digest, and 16 for 512-bit digest.

At initialization the state vector is set to zero.

2.3. State update algorithm

StramHash2 *NLF* transformation works by adding (modulo 2^{32}) an *S-BOX* output to the state vector value. The *S-BOX* index is computed as:

$$LSB(state_i) \oplus b \oplus i \tag{1}$$

The resulting formula to update a state vector value for the index i is:

$$state_i \leftarrow state_i \oplus S\text{-}BOX[LSB(state_i) \oplus b \oplus i] \tag{2}$$

Any remaining input data bits (for input size not being a multiple of 8 bits), and the number of these bits are both saved within the state structure.

Figure 3 illustrates the internal structure of the StreamHash1 *NLF* transformation.

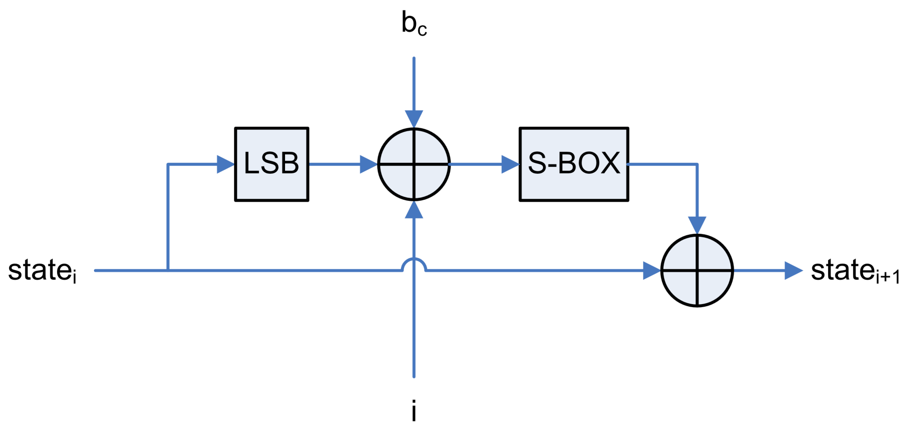

Fig. 3. NLF Function of StreamHash1.

2.4. Structure of *S-BOX*

StreamHash *S-BOX* is based on AES[¶] *S-BOX*. The formula to compute the 32-bit *S-BOX* value for the index i is:

$$s(i) \vee (s(s(i)) \ll 8) \vee (s(s(s(i))) \ll 16) \vee (s(s(s(s(i)))) \ll 24) \tag{3}$$

The content of the StreamHash *S-BOX* computed using the above formula is listed in Table 1.

2.5. Cryptanalysis

The third-party cryptanalysis is available for the StreamHash1 function, the first function of the StreamHash family.

Dmitry Khovratovich and Ivica Nikolić from University of Luxembourg reviewed cryptographic properties of StreamHash [**6**]. Joux attack [**7**] was applied with the theoretical complexity of $\frac{n}{2}2^{n/4}$ for finding collisions and $\frac{n}{2}2^{n/2}$ for finding preimages.

Tor E. Bjorstad, a PhD student of Computer Science, University of Bergen, Norway implemented [**8**] a practical collision attack against the StreamHash1 function.

3. StreamHash2 function

3.1. Motivation

The StreamHash2 algorithm was designed to address identified weaknesses of the original StreamHash1 function.

3.2. Algorithm updates

The following changes were implemented in the StreamHash2 function compared to the original StreamHash1:

- *NLF* transformation was modified with a 32-bit output of *PRNG*[‖] in order to prevent from the re-use of any identified collision of a single state word.
- $\oplus$ operation was replaced with $\boxplus$ (addition modulo 2^{32}) in order to propagate changes between the four octets of the 32-octet state word.
- Finalization phase was updated to improve resistance against length-extension attacks and statistical properties.

The StreamHash2 state structure was extended with:

- 64 bits of *PRNG* state;

[¶]Advanced Encryption Standard
[‖]Pseudo-Random Number Generator

Table 1. StreamHash2 *S-BOX*

760ffb63	74ca107c	8ee6f577	54fd217b	5ca789f2	b5d27f6b	25c2a86f	3624a6c5
89f20430	ca107c01	88978567	32a1f12b	87eabbfe	62ab0ed7	acaa62ab	c5073876
4f9274ca	ff7d1382	78c1ddc9	4716ff7d	61d82dfa	c01fcb59	e1e0a047	43648cf0
e52a95ad	005248d4	cd803aa2	4eb679af	a41dde9c	e23b49a4	01094072	bff4bac0
66d3a9b7	b72054fd	4486dc93	4568f726	7f6b0536	5e9d753f	6e4568f7	6db34bcc
95ad1834	a86f06a5	9635d9e5	2332a1f1	670aa371	dfef61d8	b4c6c731	1fcb5915
a789f204	8db4c6c7	68f72623	c7312ec3	2a95ad18	d0609096	d27f6b05	506cb89a
24a6c507	c1ddc912	7abdcd80	5a4698e2	721ee9eb	b34bcc27	b89a37b2	585e9d75
107c0109	8bceec83	0aa3712c	803aa21a	b679af1b	b9db9f6e	e4aebe5a	f8e1e0a0
fb630052	4698e23b	2c42f6d6	eb3c6db3	6f06a529	138211e3	cb59152f	8acf5f84
fc55ed53	37b23ed1	0ffb6300	b0fc55ed	d3a9b720	94e7b0fc	9be8c8b1	c912395b
f577026a	bac01fcb	69e4aebe	ddc91239	42f6d64a	06a5294c	77026a58	f37e8acf
d15170d0	0b9edfef	8191acaa	38760ffb	3aa21a43	8211e34d	312ec333	c4889785
db9f6e45	28ee99f9	e6f57702	d5b5d27f	55ed5350	1ee9eb3c	56b9db9f	3f25c2a8
b23ed151	85670aa3	7c010940	738f738f	5f844f92	6a585e9d	a6c50738	198ee6f5
e34d65bc	152f4eb6	395b57da	2054fd21	9274ca10	a04716ff	0ed70df3	03d5b5d2
da7abdcd	eabbfe0c	16ff7d13	3d8bceec	7e8acf5f	1cc48897	79af1b44	648cf017
de9c1cc4	d64a5ca7	d70df37e	4bcc273d	a21a4364	a5294c5d	5248d419	8f738f73
5170d060	bbfe0c81	cf5f844f	1b4486dc	86dc9322	35d9e52a	70d06090	9c1cc488
aebe5a46	183428ee	53506cb8	d82dfa14	49a41dde	026a585e	a1f12b0b	b156b9db
41f8e1e0	f7262332	bdcd803a	9785670a	98e23b49	c2a86f06	6b053624	f6d64a5c
753f25c2	c33366d3	0c8191ac	91acaa62	fe0c8191	d9e52a95	99f969e4	2f4eb679
932294e7	149be8c8	6cb89a37	e9eb3c6d	294c5d8d	217b03d5	59152f4e	3366d3a9
ed53506c	e8c8b156	3008bff4	f01787ea	11e34d65	5b57da7a	f969e4ae	f2043008
08bff4ba	4d65bc78	9d753f25	c6c7312e	1dde9c1c	053624a6	4c5d8db4	5d8db4c6
fa149be8	bc78c1dd	844f9274	f4bac01f	3c6db34b	57da7abd	cc273d8b	0df37e8a
3ed15170	9a37b23e	7b03d5b5	2ec33366	63005248	fd217b03	712c42f6	aa62ab0e
9edfef61	60909635	12395b57	c8b156b9	af1b4486	65bc78c1	3b49a41d	f12b0b9e
8341f8e1	ec8341f8	be5a4698	7d138211	ee99f969	909635d9	48d4198e	dc932294
2dfa149b	0940721e	8cf01787	40721ee9	273d8bce	e7b0fc55	ad183428	2b0b9edf
1a43648c	262332a1	4a5ca789	ab0ed70d	043008bf	d4198ee6	a3712c42	9f6e4568
ceec8341	3428ee99	ef61d82d	0738760f	2294e7b0	a9b72054	1787eabb	e0a04716

The improved formula to update a state vector value for the index i is:

$$state_i \leftarrow state_i \boxplus S\text{-}BOX[LSB(state_i) \oplus b \oplus i] \boxplus r_c \tag{4}$$

StreamHash2 shares all other parts of the StreamHash2 design described above, e.g. the *S-BOX* table.

Figure 4 illustrates the internal structure of the StreamHash2 *NLF* transformation.

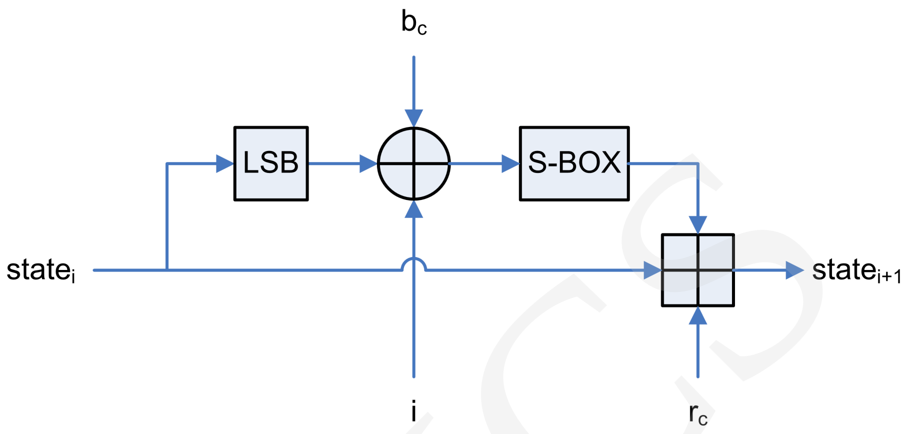

Fig. 4. NLF Function of StreamHash2.

3.3. Pseudo-random number generator

The StreamHash2 function uses a 64-bit version of the pseudo–random number generator Xorshift[**9**] as its *PRNG* transformation. The generator provides the period of $2^{64} - 1$. *PRNG* is not expected to be cryptographically secure, and security of StreamHash2 is not based on the *PRNG* properties other than its period.

The following algorithm is used to generate each 32-bit value of r_c:

(1) $s \leftarrow s \oplus (s \ll 13)$.
(2) $s \leftarrow s \oplus (s \gg 7)$.
(3) $s \leftarrow s \oplus (s \ll 17)$.
(4) Return r_c as the least significant 32 bits of s.

The 64-bit *PRNG* state s is initialized with the seed value of 88172645463325252. This starting value is a constant recommended by the author of the Xorshift algorithm.

3.4. Identified limitations of StreamHash2

Identified disadvantages of StreamHash2 are mostly the result of *S-BOX* lookup:

- Side-channel attacks[**10**] on multiasking software implementations based on the CPU cache timings.
- Not possible to compute with the SIMD** instructions on x86 architecture.
- Expensive hardware implementation (high number of gates).

**Single Instruction, Multiple Data

- 1KB of static data, although it can be reduced to 256 octets with a reasonable performance trade-off.

4. Plans for StreamHash3 function

4.1. Motivation

Daniel J. Bernstein demonstrated[**10**] a practical side-channel attack on the AES algorithm. The attack leverages a weakness of the AES non-linear transformation based on *S-BOX*. Multiple processes running on the same physical machine several resources of the CPU including memory caches. It is possible to force another process to perform cache hit or cache miss depending on the *S-BOX* lookup offset. With accurate time measurements it is possible to infer secret data and subsequently to compute encryption key. The same weaknesses could be used to find preimages of the StreamHash2 algorithm.

Initially, *S-BOX* appeared to be a perfect source of non-linearity for the StreamHash family. It seemed to be extremely fast, as *S-BOX* lookup is implemented with a single CPU instruction. Code profiling tests performed by the author of this paper revealed that a significant amount of CPU time is spent on the lookup instruction, as its lookups cannot be solely computed on registers.

It is also not practical to use the *S-BOX* indices longer than 8 bits for implementations with limited hardware resources. 8-bit *S-BOX* indices, in turn, only allow StreamHash2 to process one octet of input data at a time.

The use of *S-BOX*es is not practical on low-end implementations. For low-end 8-bit CPUs 1KB of static data may represent a substantial amount of memory. The *S-BOX* included in the previous StreamHash family members can, however, be computed on the fly, reducing memory usage with a reasonable performance trade-off.

This issue gets much worse for low-end hardware implementations. For low-power hardware (e.g. RFID[††] tokens) the number of gates required to implement the *S-BOX* of StreamHash2 could be unacceptable.

4.2. Proposed solution

The solution for the planned StreamHash3 is to replace *S-BOX*es with the constructions based on shifts ($\ll$ and $\gg$) and modular addition ($\boxplus$) should allow to process input stream word-by-word instead of octet-by-octet, and to implement non-linearity with the SIMD instructions.

As a result, it may be possible to achieve StreamHash3 performance as good as the performance of StreamHash2, or even better.

[††]Radio-frequency identification

4.3. Support of the x86 CPU architecture

The following instructions, operating on the sets of 32-bit words, could be used on x86 architecture:

PSLLD – Packed Shift Left Logical ($\ll$)
PSRLD – Packed Shift Right Logical ($\gg$)
PADDD – Packed Add ($\boxplus$)

The number of simultaneously processed words depends on the SIMD word size available on the specific architecture [**11**]. The following SIMD register is available on the x86-compatible CPUs:

MMX – 8 64-bit registers $mm_0 - mm_7$.
SSE2 – 8 128-bit registers $xmm_0 - xmm_7$ in 32-bit mode, and 16 128-bit registers $xmm_0 - xmm_{15}$ in 64-bit mode.
AVX – 256-bit registers $ymm_0 - ymm_{15}$ available. The first CPUs supporting AVX architecture are Intel Sandy Bridge (first released on 9 January 2011) and AMD Bulldozer (scheduled for release on Q2 2011).

The SIMD instructions would allow to simultaneously process 2 (for MMX), 4 (for SSE2) or 8 (for AVX) 32-bit StreamHash3 state words.

5. Conclusions

Practical attacks against MD5 [**12**] and SHA-1 [**13**] suggest that collision resistance is the most serious threat to cryptographic hash functions. The StreamHash family was designed specifically to deal with this threat.

The whole StreamHash family can be effortlessly scaled to use any multiple of 32 bits as the state vector size. Applications of this property include not just upscaling for improved security, but also downscaling for the applications with reduced security requirements, e.g. lightweight cryptography. These applications can benefit from fast finalization of the StreamHash family, as well as the reduced number of gates achieved by removing *S-BOX* while designing the StreamHash3 function.

Growing popularity of lightweight cryptography is driven by the increasing number of RFID tags as well as battery-powered wireless sensor agents. Currently many of these devices use plaintext communication protocols in order to reduce circuit chip size/cost and power consumption. StreamHash3 could be used as an efficient cryptographic hash function to implement HMAC-based security layer for these protocols.

Symbols

$\boxplus$	–	arithmetic unsigned addition modulo 2^{32}
$\oplus$	–	bitwise exclusive disjunction, also called *XOR* (EXclusive Or)
$\vee$	–	bitwise *OR* operator
$\lll$	–	bitwise *SHIFT LEFT* operator
$\ggg$	–	bitwise *SHIFT RIGHT* operator
$\leftarrow$	–	substitution
$\Vert$	–	concatenation of octet strings
$S\text{-}BOX[x]$	–	table lookup returns the value at the position x of table *S-BOX*
$LSB(x)$	–	least significant octet of x

References

[1] Dierks T., Allen C., RFC 2246: The TLS protocol version 1, January (1999), ftp://ftp.internic.net/rfc/rfc2246.txt
[2] Krawczyk H., Bellare M., Canetti R., RFC 2104: HMAC: Keyed–hashing for message authentication, February (1997), ftp://fp.internic.net/rfc/rfc2104.txt
[3] Garey M. R., Johnson D. S., Computers and Intractability; A Guide to the Theory of NP–Completeness, W. H. Freeman & Co., New York, NY, USA (1990).
[4] Trojnara M., Streamhash algorithm specifications and supporting documentation, http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/StreamHash.zip (2008).
[5] NIST. First round candidates, http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html
[6] Khovratovich D., Nikolić I., Cryptoanalysis of streamhash, http://lj.streamclub.ru/papers/hash/streamhash.pdf (2009).
[7] Joux A., Multicollisions in iterated hash functions, application to cascaded constructions, In Matthew K. FFranklin, editor CRYPTO 3152 of LNCS (2004): 306.
[8] Bjorstad T. E., Collision for streamhash, http://ehash.iaik.tugraz.at/uploads/7/7b/Streamhash.txt (2009).
[9] Marsaglia G., Xorshift rngs, Journal of Statistical Software 8(14) (2003): 1; http://www.jstatsoft.org/v08//i14
[10] Bernstein D. J., Cache–timing attacks on aes, Technikac report (2005).
[11] Intel, Advanced vector extensions programming reference, http://software.intel.com/file/35247/ (2008)
[12] Xie T., Feng D., Construct md5 collisions using just a single block of message, Cryptology ePrint Archive, Report 2010/643 (2010), http://eprint.iacr.org/2010/643
[13] Manuel S., Classification and generation of disturbance vectors for collision attacks againdt sha-1, Cryptology ePrint Archive, Report 2008/469 (2008); http://eprint.iacr.org/2008/469