Performance of Message Authentication Codes for Secure Ethernet

Philipp Hagenlocher
Advisors: Dominik Scholz & Fabien Geyer
Seminar Future Internet SS2018
Chair of Network Architectures and Services
Departments of Informatics, Technical University of Munich
Email: philipp.hagenlocher@in.tum.de

ABSTRACT
Cyclic redundancy checks within Ethernet do not provide data integrity, which can be solved by using message authentication codes (MACs) within Ethernet frames. This paper gives theoretical background on hash functions and hash-based MACs, different schemes are compared for the usage in Ethernet frames by multiple attributes such as performance, output lengths and security. The functions Poly1305, Skein and BLAKE2 are tested for throughput against implementations of a SHA-2 HMAC and SipHash and the results are discussed. Numerical measurements and results show that Poly1305 is a suitable candidate for the usage as a per-packet MAC.

Keywords
Message Authentication Codes, Hash functions, Performance evaluation, Secure Ethernet

1. INTRODUCTION
In daily communication within LANs, MANs and WANs, computers use the Ethernet protocol and furthermore send so called Ethernet frames over the network in order to communicate. These frames possess a checksum calculated with a cyclic redundancy check (CRC), which is a code to detect errors, in the form of flipped bits, within the frame [27]. This ensures that transmission errors are detected.

Using CRC has the advantage of needing only a small amount of space (32 bits) and having a code that can be computed in short time. The problem with CRC is that it does not indicate data integrity of the frame if an attacker tries to maliciously alter it. The attacker can easily recompute the CRC since there is no secret protecting the checksum [31].

In order to combat the problem of data integrity *message authentication codes* (MAC) are often used in cryptographic protocols. MACs based on cryptographic hash functions can use an additional secret key in order to combine the integrity of the frame with a shared secret, thus blocking a potential attacker from altering the frame *and* computing a correct checksum. A popular usage of hash functions in MACs are *hash-based message authentication codes* defined in Section 2.3, although MACs do not have to be constructed with or from hash functions. Another application of hash functions are *TCP SYN cookies*, which are used in order to combat *TCP SYN flooding* attacks [28], where an attacker intentionally doesn't complete the 3-way-handshake of TCP connection establishment in order to waste resources on the server until a denial of service is reached. To combat this attack scheme, a hash function is used to compute a hash combining the IP address of the client, port numbers and a timestamp. This hash is then used as the sequence number for the packet. This way the client is identified by this hash, thus the server can free old resources if a new SYN is sent by the client. Another important usage for hash functions are lookup tables, where hash functions are used to group values into segments that can be retrieved in faster time than a simple lookup in a set of elements.

1.1 Outline
This paper is structured as follows: At first it will explain the theory behind hash functions and hash-based MACs in Section 2. Section 3 will give an overview over possible candidates for the usage as a MAC scheme within Ethernet and Section 4 will explain the testing environment and discuss test results. The conclusion to the test results and further discussion is given in Section 5.

2. HASH FUNCTIONS
A hash function h is defined as a *one-way function*

$$h : \Sigma^* \rightarrow \Sigma^n$$

where $\Sigma = \{0, 1\}$, Σ^n denotes all bitstrings of length n and Σ^* denotes all bitstrings of arbitrary length [6, p. 177]. In order for this function to be cryptographically secure the function needs to have weak and strong collision resistance. Weak collision resistance is defined as the absence of an efficient algorithm that, for a given m, can find a m' such that $h(m) = h(m')$. Strong collision resistance is defined as the absence of an efficient algorithm that can find any m and m' such that $h(m) = h(m')$. These properties are important for the use of these hash functions in a MAC scheme since we don't want an attacker to be able to guess what possible alterations can be made to the content of the message or, even worse, find out what the shared secret key is. Hash functions that possess these properties are known as *cryptographic hash functions*.

2.1 Security of hash functions
When talking about cryptographic strength of a hash function, what is really talked about is the computational complexity of finding a collision. Let h be a perfect hash function with perfect weak and strong collision resistance and

doi: 10.2313/NET-2018-11-1_04

an output length of n. In order to find a collision, an attacker would need to randomly try out different m and m' and check if $h(m) = h(m')$. Since hash functions have a fixed output length the number of possible hash values is limited. Due to the birthday paradox [6, p. 175-180] the complexity of randomly trying out different input values is $2^{\frac{n}{2}}$. Generally, a hash function is considered unsafe once this complexity can be reduced drastically and thus making it practically possible to brute-force collisions or even compute them outright.

This example is known as a *birthday attack*. This attack cannot be improved if the hash function has perfect strong collision resistance. Another attack is the *pre-image attack*, which describes finding an appropriate x to a given y such that $h(x) = y$. Brute-forcing this x has the complexity of 2^n where n is the output length of h. This attack cannot be improved if the hash function has perfect weak or strong collision resistance.

2.2 Construction

Constructing hash functions has many strategies. One of the most popular schemes is the *Merkle-Damgård construction* described by Merkle [24, p. 13-15]. This construction splits the message into blocks and applies a function $f(k, m)$ to these blocks:

$$f(...f(f(s, b_1), b_2)..., ...b_n)$$

where s denotes a fixed starting value and b_n denotes the nth message block. There can be a finalization function applied on the result of this computation. f could be an encryption function that takes a key k and a message m. Popular hash function families like MD, SHA-1 and SHA-2 all use this kind of construction.

Of course there are many ways of constructing hash functions such as using permutation networks, S-Boxes and other parts of block ciphers in order to build a one-way function. JH [15, p. 26], BLAKE [18] and Keccak [17] are examples of that.

The aforementioned *substitution boxes (S-Boxes)* are used by many hash functions in their construction. They describe a mapping from bitstrings of fixed size to other bitstrings of another fixed size. For example, the S-Boxes within the block cipher DES can be represented as a table of values used for substitution, where the choice of columns and rows is done by looking at the outer and inner bits of the bitstring that needs to be substituted [6, p. 76-77].

2.3 Hash-Based Message Authentication Code

A *hash-based message authentication code* (HMAC) is a MAC combining hash functions with a secret key in order to achieve integrity between parties that share said key. RFC 2104 [22] defines the way to compute an HMAC like so: Let H denote a cryptographic hash function with output length of n and internal block size of B, K a secret key, MSG the message to compute an HMAC for, $\|$ the concatenation of byte-strings, $\bigoplus$ the exclusive-or (XOR) operation, *ipad* and *opad* byte-strings consisting of 0x36 and 0x5C repeated B times respectively. Then the HMAC is defined as:

$$H((K \bigoplus opad) \parallel H((K \bigoplus ipad) \parallel MSG))$$

It is easy to see that it is absolutely possible to switch out H with any cryptographic hash function and that a *cryptographic* hash function is needed in order for the HMAC to be secure and that the computational speed of the algorithm is heavily influenced by the speed of the underlying hash function.

3. SURVEY

This section describes and compares suitable candidates for our use-case. Surveying the finalists of the hash function competition by the US National Institute of Standards and Technology (NIST) is a good choice to find hash functions for an HMAC construction since these functions have been preselected due to their performance and security. As well as the finalists of this competition more MAC schemes will be taken into consideration, that have recently found usage in real world applications.

Since performance is an important measure to go by it is advantageous for a candidate to be usable as a standalone MAC scheme. Otherwise an HMAC has to be created out of the function in question, which uses two distinct calls to the function thus slowing down the resulting MAC scheme.

3.1 Use-case

The use case is a per packet message authentication code. Obviously this MAC needs to be secure, but has to have a high performance in order to not slow down traffic too much, since bandwidths of 10 Gbit/s can be achieved and should not be bottlenecked by the CPU and hash computation. Initial tests with an HMAC using SHA-256 and SHA-512 showed bad performance, which can be seen in figure 3 and 4. This performance was increased by switching to an 32-bit variant of SipHash (explained in Section 3.3.7).

3.2 Unsafe functions

Several functions will not be taken into consideration since attacks against these functions have been demonstrated or are theoretically possible. These functions are

- MD4 [36]
- MD5 [32]
- SHA1 [30]
- GOST [11]
- HAVAL-128 [34]
- RIPEMD [35]

3.3 Candidates

In this section possible candidates for the described use-case are described and important features are highlighted.

3.3.1 SHA-2

The SHA-2 family uses the aforementioned *Merkle-Damgård construction* described in section 2.2. The performance was retested in the hash function competition which resulted in ∼11-14 cycles per byte [15, p. 43-44] for SHA-512, the variant of SHA-2 with an output length of 512 bits that generally performs the best. The SHA-2 family was developed by the NSA and standardized by NIST [16]. The family is well known, widely used and extensively studied, thus providing a popular choice for usage as an HMAC.

doi: 10.2313/NET-2018-11-1_04

3.3.2 Keccak

Keccak uses a sponge function combined with permutations and 24 rounds. It is expected to perform well when implemented in hardware [15, p. 6], which holds true when testing the algorithm in an FPGA implementation [12]. One attack found against this function had the complexity of $2^{511.5}$ on a version of Keccak with 8 rounds [15, p. 30]. It won the competition and is now known as the SHA-3 standard [17].

3.3.3 BLAKE

BLAKE is based on the *ChaCha* stream cipher and uses 14 rounds for 224 and 256 bits and 16 rounds for 358 and 512 bits of output [15, p. 17]. It was analyzed heavily during the competition. On performance it was noted that BLAKE performs well in software [15, p. 6]. A collision attack on a version of BLAKE-512 with 2.5 rounds has the complexity of 2^{224} [15, p. 22].

3.3.4 JH

JH uses permutations in combination with XOR operations and S-Boxes with 42 rounds for all output lenghts [15, p. 26]. JH is considered slower than BLAKE, Skein and Keccak [15, p. 6]. An attack was found with a time complexity of 2^{304} [15, p. 28].

3.3.5 Grstl

Grøstl is a combination of a *Merkle-Damgård construction* combined with parts of the block cipher *AES* [15, p. 23]. It uses 10 rounds for an output length of up to 256 bits and 14 rounds for an output length of up to 512 bits. It is considered slower than BLAKE, Skein and Keccak [15, p. 6] and has drastically different performance for 224/256 and 384/512 bits output length [13, p. 23]. The only collision attack on Grøstl with an output length of 512 bits had 3 rounds and a complexity of 2^{192} [15, p. 26].

3.3.6 Skein

Skein uses the block cipher *Threefish* which uses 72 rounds of a substitution-permutation network. Similar to BLAKE, Skein is expected to perform well when implemented in software[15, p. 6]. A lot of the cryptographic analysis went into the block cipher. Almost all of the found attacks on versions of Skein with reduced rounds are impractical. One of the collision attacks on Skein with the output length of 512 bits and 14 rounds has the complexity of $2^{254.5}$[15, p. 33]. Another important feature of Skein is a special MAC mode called Skein-MAC[25, p. 7-8].

3.3.7 SipHash

While not being a hash function SipHash is a keyed *pseudo random function (PRF)*, thus not necessarily having collision resistance but still making it infeasible to compute the hash value if the key is not known. This function was specifically designed to be used as a MAC for short input values and was inspired by BLAKE, Skein and JH [20]. Its performance is a key feature, since it was designed to be resistant against denial of service attacks in schemes such as TCP SYN cookies where the hash function can take up so much computational time that the server is unable to handle other workload. A cryptographic analysis of SipHash concluded that finding a collision has the probability of $2^{-236.3}$ [9].

3.3.8 Poly1305-AES

Poly1305-AES is another MAC scheme that is not a hash function. The author claims high performance, with linearly growing cycles per bytes, and guaranteed security if AES is secure [7]. Another important feature is the output size of 128 bits, which is quite small. It has since been standardized in RFC 7539 [26], has found usage by Google and was incorporated into TLS1.3. Also important to note is that AES can be replaced by any other cipher such as ChaCha20. There has been research on the security of this specific Poly1305 variant [14].

3.3.9 BLAKE2

The successor to BLAKE has been standardized in RFC 7693 [29]. The internals are comparable to BLAKE. The authors claim better performance than MD5, SHA2 and SHA3 [19]. It features special versions for 64- and 32-bit platforms and has a special prefix-MAC mode. It's security has been extensively studied, since most of the theoretical work done on BLAKE is also applicable for its successor [19, p. 4].

3.4 Comparison

In this section the aforementioned hash functions and MAC schemes are compared with the already tested algorithms and their suitability for the use case is rated. Performance, output length and security are the key factors that will be compared.

	Performance (approx. cycles/byte)	Output length (bits)	Stand-alone MAC
Keccak	6.7-19	224-512	No
BLAKE	8.4	224-512	No
JH	15-20	224-512	No
Grøstl	13-92 / 18-126	224-512	No
Skein	6.1	Any	Yes
Poly1305	4-15	128	Yes
BLAKE2	3-12	8-2048	Yes
SHA-2	11-14	224-512	No
SipHash	4-10	64	Yes

Table 1: Summary of candidates

3.4.1 Performance

There have been a lot of performance tests done for the NIST hash function competition finalists and their performances have been compared to SHA-256 and SHA-512. In order to have a comprehensive comparison, the functions were tested in three different processors. JH and Grøstl generally perform worse than functions of the SHA-2 family and the other finalists [15, p. 43-44]. The performance of Keccak depends greatly on the output length. For sizes of 224 or 256 bits it is described as *"reasonably fast"* [15, p. 46], but for the output size of 512 bits it is rather slow. Skein and BLAKE are generally faster than SHA-2, while BLAKE seems to be the fastest function. Skein is the only function that has the same performance for all output lengths [15, p. 43-46]. The performance of SipHash is comparable with SHA-512 (∼10

doi: 10.2313/NET-2018-11-1_04

cycles/byte) [15, p. 43-44] and for a message length of 64 bytes it beats out BLAKE with 4 cycles/byte [20, p. 11]. A comparable performance is achieved by BLAKE2 [19, p. 14]. Poly1305 is claimed to have $3.1n + 780$ cycles for a n byte message. For a 64 byte message this would result in ∼15 cycles/byte and for a 1024 byte message this would result in ∼4 cycles/byte. So its performance can be compared to SipHash. The main difference is mainly that Poly1305 profits from long message lengths due to the constant overhead.

3.4.2 *Output lengths*

All NIST hash function competition finalists have output lengths of 224, 256, 384 and 512 bits. Skein is very flexible by having an arbitrary output length and three different internal block sizes (256, 512 and 1024 bits) [25, p. 1]. BLAKE2 features digest sizes from 1 to 256 bytes [19, p. 6]. SipHash computes a MAC of fixed length (64 bit) [20, p. 6]. Poly1305 doubles this length for its output of 128 bits [7, p. 1].

3.4.3 *Security*

None of the functions taken into consideration are considered broken and all attacks known against them are impractical. Still, it is important to note that some functions have not been studied as rigorously as others. BLAKE, Skein and Grøstl have had extensive analysis done on them in the hash function competition, while Keccak and JH had less work done on them [15, p. 33]. Also, in real world applications Skein, JH and Grøstl are not getting as much usage as the other functions. BLAKE2 is implemented in many cryptographic libraries such as libsodium [2] and has found usage in the password hashing scheme Argon2 [8]. Poly1305 has been adopted by Google as an RC4 replacement and Poly1305 in combination with the ChaCha20 block cipher was incorporated into OpenSSH [5]. It is important to note that this implementation of Poly1305 is susceptible to side channel attacks [21]. SipHash is the hash function for the hash table implementations within Python and Rust and is the *"shorthash"* function in libsodium [2].

3.4.4 *Final Choice*

The most suitable MACs should be generally better suited than SHA-2 and SipHash. Looking at Table 1 one can see that JH and Grøstl are generally to slow to be suitable choices. Keccak and BLAKE are fast, but suffer from poor flexibility in output lengths. Since all the algorithms can be considered secure the three final candidates are Skein (as Skein-MAC), BLAKE2 (with its MAC-mode) and Poly1305. Not only do Skein and BLAKE2 have good performance, but they also have designated MAC schemes, that are designed to be faster than traditional HMACs. It is important to note that hardware performance did not factor into the selection process, because it cannot be tested in the used testenvironment, which is why Keccak will not be tested even though showing outstanding performance.

4. EVALUATION

This section describes the testing environment and testing procedure. It then discusses the results from the testing.

```c
#include "blake2.h"
#include "skein.h"
#include "poly1305-donna.h"

static inline uint32_t stream2int(const uint8_t
↪  *stream) {
    return (((uint32_t) stream[0]) << 24 |
        ((uint32_t) stream[1]) << 16 |
        ((uint32_t) stream[2]) <<  8 |
        ((uint32_t) stream[3]) <<  0);
}

uint32_t calculate_blake2b(const blake2b_state
↪  *ctx, const void *msg, uint16_t length) {
        uint8_t mac[4];
        blake2b_update(ctx, msg, length);
        blake2b_final(ctx, mac, 4);
        return stream2int(mac);
}

uint32_t calculate_blake2s(const blake2s_state
↪  *ctx, const void *msg, uint16_t length) {
        uint8_t mac[4];
        blake2s_update(ctx, msg, length);
        blake2s_final(ctx, mac, 4);
        return stream2int(mac);
}

uint32_t calculate_skein(const Skein_256_Ctxt_t
↪  *ctx, const void *msg, uint16_t length) {
        uint8_t mac[4];
        Skein_256_Update(ctx, msg, length);
        Skein_256_Final(ctx, mac);
        return stream2int(mac);
}

uint32_t calculate_poly1305(uint8_t *key, const
↪  void *msg, uint16_t length) {
        /* key must be 32 bytes and UNIQUE */
        uint8_t mac[16];
        poly1305_auth(mac, msg, length, key);
        /* only using first 32 bit */
        return stream2int(mac);
}
```

Figure 1: Implementation of the tested MACs

4.1 Setup

Skein, BLAKE2 and Poly1305 are tested in an artificial environment with a P4 switch. A load generator generates a stream of Ethernet frames with increasing bandwidth. Both the switch and load generator are equipped with a Xeon E3-1230, 15.6 GB of memory and run the Linux 4.9.0-5 kernel. For P4 compilation the environment uses *t4p4s*, a fork of *p4c*[23]. A configuration for the compiler defines which algorithm is used for which test case, even though the internal controller and remaining configuration stays the same in order to keep testing neutral. The load generator uses MoonGen [10] in order to create packets at high rates. The switch is tasked with computing the MAC for each frame and will respond with the finished packet. In order to keep the workload realistic frames are sent in lengths from 64 to 1500 bytes with rates between 50 and 10000 Mbit/s.

doi: 10.2313/NET-2018-11-1_04

TX	Poly1305	Skein	BLAKE2s	BLAKE2b
50	48.19	48.19	48.19	48.19
100	99.98	99.77	99.80	99.80
150	144.29	144.33	144.33	144.63
200	200.04	199.93	199.54	199.57
250	240.49	240.43	240.00	240.46
500	498.84	498.81	500.04	498.81
1000	961.49	529.52	961.64	963.24
2000	1977.02	561.63	1588.61	1849.23
3000	1774.09	529.58	1410.94	1662.78
4000	1973.52	561.35	1572.47	1849.95
5000	1768.91	529.56	1420.38	1657.34
7500	1974.78	562.86	1585.03	1850.57
10000	1783.94	529.47	1420.20	1666.51

(a) Bandwidths for 64 bytes per packet (in Mbit/s)

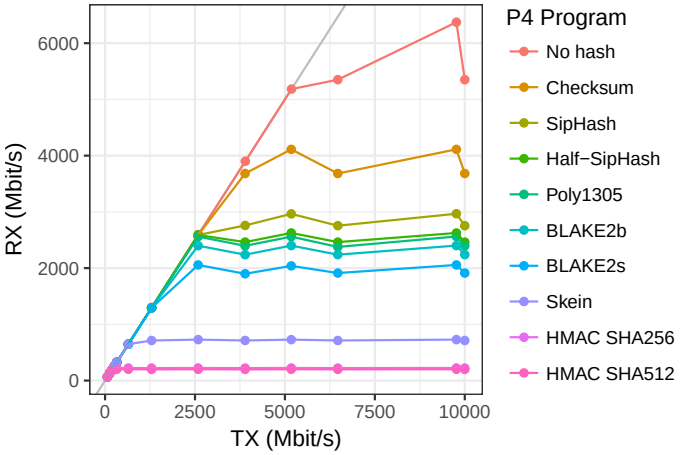
TX	Poly1305	Skein	BLAKE2s	BLAKE2b
50	50.03	50.03	50.03	49.94
100	96.55	96.55	96.36	96.39
150	150.04	149.77	150.04	150.04
200	192.90	192.91	192.47	192.56
250	249.53	249.98	249.03	250.04
500	480.97	481.00	480.98	480.86
1000	999.62	997.67	999.63	997.86
2000	1923.14	1282.15	1926.51	1923.32
3000	3001.89	1333.83	2394.66	3001.83
4000	3852.96	1282.29	2298.26	3425.08
5000	4986.97	1334.10	2399.72	3578.64
7500	6025.09	1284.78	2312.45	3417.14
10000	6428.82	1334.35	2394.06	3579.34

(b) Bandwidths for 512 bytes per packet (in Mbit/s)

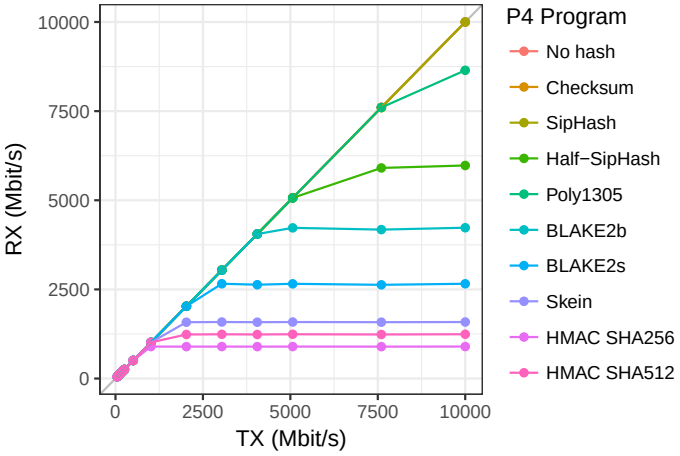
TX	Poly1305	Skein	BLAKE2s	BLAKE2b
50	50.03	50.03	50.03	50.03
100	96.55	96.57	96.59	96.57
150	150.04	149.92	150.04	150.04
200	192.74	192.85	192.77	192.98
250	250.04	249.64	250.04	249.62
500	481.14	481.23	482.12	481.28
1000	997.96	997.59	999.73	1000.07
2000	1923.15	1499.35	1922.85	1923.34
3000	2999.31	1560.45	2622.18	2993.16
4000	3845.81	1502.14	2502.63	3846.05
5000	4996.43	1562.82	2615.59	4160.39
7500	7224.33	1500.01	2498.69	3963.35
10000	8527.97	1560.32	2616.91	4164.24

(c) Bandwidths for 1500 bytes per packet (in Mbit/s)

Figure 2: Bandwidth measurements



(a) Bandwidths for 64 bytes per packet



(b) Bandwidths for 1500 bytes per packet

Figure 3: Throughput comparison

4.2 Implementation

In order to keep testing neutral, there were no optimized libraries used since the test concerns the raw performance of the algorithms, not their optimization. For Skein the reference implementation [4] is used as well as for BLAKE2 [1]. For the first the internal size is set to 256 bit and for the later there are two testing candidates namely *BLAKE2b* and *BLAKE2s* the implementations meant for 64 bit systems and 32 bit systems respectively. The Poly1305 implementation [3] is a portable implementation that is kept close to the reference implementation. For every MAC the same key is used, which should be avoided in real world usage of Poly1305, since it needs a new key for every new message. This test does not factor in the overhead that computing new keys would create even though for every MAC the needed initialization function is called every time. Figure 1 shows the relevant source code for the tested functions. The context for BLAKE2 and Skein are initialized beforehand.

doi: 10.2313/NET-2018-11-1_04

4.3 Results

The test results for packet sizes of 64 and 1500 bytes are shown in Figure 3. Figure 3a shows similar performance for Poly1305 and BLAKE2, while Skein is performing worse. For increasing size, Poly1305s performance is rapidly increasing which could be due to the performance of $3.1n+780$ cycles per n bytes, trivially converging to 3.1 cycles per byte for steadily increasing message sizes. The gap between BLAKE2b and BLAKE2s is also increasing for larger packet sizes, while Skein is hitting a bottleneck for a bandwidth of around 1500 Mbit/s. BLAKE2b has a similar bottleneck at 4000 Mbit/s. For small packet sizes, Poly1305 is also hitting a bottleneck, even though it occurs for higher bandwidths as the other candidates as can be seen in the tables of Figure 2. Comparing the three candidates to HMACs built from SHA-2 and SipHash reveals that all of them outperform the HMACs, but non of them outperform SipHash. The most important finding is that Poly1305 increases in performance for packet sizes bigger than 1000 bytes while all other performances decrease as can be seen in Figure 4. This could indicate that Poly1305 will eventually outperform SipHash at a certain packet size, but further testing would need to be done in order to prove this conjecture.

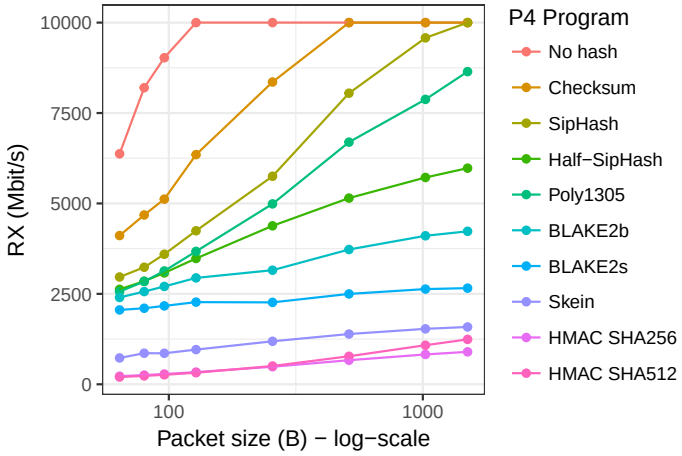


Figure 4: Throughput comparison by packet size

5. CONCLUSION

The paper has shown the motivation behind the usage of hash functions in Ethernet frames and has explained the theory behind hash functions. Multiple functions and MAC schemes have been compared by their qualities for this use case and the three candidates (Skein, BLAKE2 and Poly1305) have been compared in a synthetic test environment.

The test indicates that Poly1305 is a suitable candidate when it comes to throughput due to its great performance for bigger packet lengths. While throughput is important, Poly1305's implementation faces problems that cannot be ignored. It needs a *new* key for every message that has to be 32 bits large. This problem is fixed in an alternative proposal called DPoly1305, which does not require a new key for every message [33], but has not been standardized or thoroughly analysed yet. Another problem that the implementation faces is a fixed output length of 128 bit. The hardware performance of Poly1305 was not tested and for implementation in hardware, an algorithm like Keccak could be far more suitable, but further testing would need to be done before a conclusive statement can be made.

6. REFERENCES

[1] Blake2 github repository. `https://github.com/BLAKE2/BLAKE2`. Accessed: 2018-04-16.

[2] libsodium github repository. `https://github.com/jedisct1/libsodium`. Accessed: 2018-04-16.

[3] Poly1305-donna github repository. `https://github.com/floodyberry/poly1305-donna`. Accessed: 2018-04-16.

[4] Skein github repository. `https://github.com/meteficha/skein/tree/master/c_impl/reference`. Accessed: 2018-04-16.

[5] Super user's bsd cross reference: Protocol.chacha20poly1305. `http://bxr.su/OpenBSD/usr.bin/ssh/PROTOCOL.chacha20poly1305`. Accessed: 2018-04-16.

[6] Albrecht Beutelspacher, Heike B. Neumann, Thomas Schwarzpaul. *Kryptografie in Theorie und Praxis*. Vieweg+Teubner Verlag, 2010.

[7] D. J. Bernstein. The poly1305-aes message-authentication code, 2005. `http://cr.yp.to/mac/poly1305-20050329.pdf`.

[8] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302, 2016.

[9] C. Dobraunig, F. Mendel, and M. Schläffer. Differential cryptanalysis of siphash. Cryptology ePrint Archive, Report 2014/722, 2014. `https://eprint.iacr.org/2014/722`.

[10] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[11] Florian Mendel, Norbert Pramstalle1, ,Marcin Kontak, Janusz Szmidt. Cryptanalysis of the gost hash function, 2008. `https://online.tugraz.at/tug_online/voe_main2.getvolltext`.

[12] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five sha-3 finalists using xilinx and altera fpgas. *IACR Cryptology EPrint Archive*, 2012:368, 2012.

[13] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and T. S. S. Grøstl - a sha-3 candidate. In H. Handschuh, S. Lucks, B. Preneel, and P. Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[14] K. Imamura, K. Minematsu, and T. Iwata. Integrity analysis of authenticated encryption based on stream ciphers. *International Journal of Information Security*,

Jun 2017.

[15] Information Technology Laboratory National Institute of Standards and Technology. Nistir 7896 (third-round report of the sha-3 cryptographic hash algorithm competition), 2012. https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf.

[16] Information Technology Laboratory National Institute of Standards and Technology. Fips pub 180-4 (secure hash standard (shs)), 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[17] Information Technology Laboratory National Institute of Standards and Technology. Fips pub 202 (sha-3 standard: Permutation-based hash and extendable-output functions), 2015. http://dx.doi.org/10.6028/NIST.FIPS.202.

[18] W. M.-R. C.-W. P. Jean-Philippe Aumasson, Luca Henzen. Sha-3 proposal blake, 2010. https://131002.net/blake/blake.pdf.

[19] Z. W.-O. C. W. Jean-Philippe Aumasson, Samuel Neves. Blake2: simpler, smaller, fast as md5, 2013. https://blake2.net/blake2.pdf.

[20] Jean-Philippe Aumasson, Daniel J. Bernstein. Siphash: a fast short-input prf, 2010. https://131002.net/siphash/siphash.pdf.

[21] B. Jungk and S. Bhasin. Don't fall into a trap: Physical side-channel analysis of chacha20-poly1305. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1110–1115, March 2017.

[22] D. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Feb. 1997.

[23] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM.

[24] R. C. Merkle, R. C. erkle, R. C. Yerkle, A. Students, S. Pohlig, R. Kahn, and D. Andleman. Secrecy, authentication, and public key systems. Technical report, 1979.

[25] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker. The skein hash function family, 2010. http://www.skein-hash.info/sites/default/files/skein1.3.pdf.

[26] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[27] Philip Koopman. 32-bit cyclic redundancy codes for internet applications, 2002. http://users.ece.cmu.edu/~koopman/networks/dsn02/dsn02_koopman.pdf.

[28] L. Ricciulli, P. Lincoln, and P. Kakkar. Tcp syn flooding defense. CNDS, 1999.

[29] M.-J. O. Saarinen and J.-P. Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, Nov. 2015.

[30] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[31] M. Stigge, H. Plötz, W. Müller, and J.-P. Redlich. Reversing crc–theory and practice. 2006.

[32] Tao Xie and Fanbao Liu and Dengguo Feng. Fast collision attack on md5. Cryptology ePrint Archive, Report 2013/170, 2013. https://eprint.iacr.org/2013/170.

[33] D. Wang, D. Lin, and W. Wu. A variant of poly1305 mac and its security proof. In Y. Hao, J. Liu, Y.-P. Wang, Y.-m. Cheung, H. Yin, L. Jiao, J. Ma, and Y.-C. Jiao, editors, *Computational Intelligence and Security*, pages 375–380, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[34] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. Cryptology ePrint Archive, Report 2004/199, 2004. https://eprint.iacr.org/2004/199.

[35] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions md4 and ripemd. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 1–18, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[36] Yu Sasaki, Lei Wang, Kazuo Ohta and Noboru Kunihiro. New message difference for md4, 2007. https://www.iacr.org/archive/fse2007/45930331/45930331.pdf.