Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: April 28, 2017

A. Malhotra
S. Goldberg
Boston University
October 25, 2016

Message Authentication Codes for the Network Time Protocol
draft-aanchal4-ntp-mac-02

Abstract

The Network Time Protocol (NTP) RFC 5905 [RFC5905] uses a message authentication code (MAC) to cryptographically authenticate its UDP packets. Currently, NTP packets are authenticated by appending a 128-bit key to the NTP data, and hashing the result with MD5 to obtain a 128-bit tag. However, as discussed in [BCK] and [RFC6151], this is not a secure MAC. As such, this draft considers different secure MAC algorithms for use with NTP, evaluates their performance, and recommends the use of CMAC-AES [RFC4493]. We also suggest deprecating the use of MD5 as defined in [RFC5905] for authenticating NTP packets.

Table of Contents

1. Introduction

NTP uses a message authentication code (MAC) to authenticate its packets. Currently, NTP packets are authenticated by appending a 128-bit key to the NTP data, and hashing the result with MD5 to obtain a 128-bit tag. However, as discussed in [BCK] and [RFC6151], this not a secure MAC. As such, this draft considers different secure MAC algorithms for use with NTP, evaluates their performance, and recommends the use of CMAC-AES [RFC4493]. We also suggest deprecating the use of MD5, as defined in [RFC5905], for authenticating NTP packets.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. MAC Algorithms

We consider five diverse MAC algorithms, which encompass hash-based HMAC-MD5 and HMAC-SHA224 [RFC2104], block cipher-based CMAC-AES [RFC4493], and universal hashing-based Galois MAC (GMAC) [RFC4543] and Poly1305(ChaCha20) as in section 2.6 of [RFC7539]. For completeness we also benchmark the legacy MD5(key||message) from [RFC5905].

Algorithm	Input Key Length (Bytes)	Output Tag Length (Bytes)
legacy MD5	16	16
HMAC-MD5	16	16
HMAC-SHA224	16	16
CMAC(AES)	16	16
GMAC(AES)	16	16
Poly1305(ChaCha20)	32	16

The choice of algorithms evaluated here is motivated, in part, by standardization and availablity of their open source implementations. All algorithms we consider, other than the plain MD5, are standardized. Four out of five algorithms are at least available in the OpenSSL library, while Poly1305(ChaCha20) is implemented in LibreSSL (a fork of OpenSSL) and also in BoringSSL (Google's implementation of OpenSSL).

The output tag length for HMAC-SHA224 is 28 bytes, but we truncate it to 16 bytes as in section 4 of [RFC7630] to fit into the NTP packet. As noted in section 6 of [RFC2104] it is safe to truncate the output of MACs as long as the truncated length is greater than 80-bits and not less than half the length of the hash output.

3. Requirements

3.1. Performance Requirements

In order to accurately compute the time, NTP ideally requires MAC algorithms to have a constant computational latency. However, this is generally not possible, since latency depends on the CPU load, temperature, and other uncontrollable factors. Instead, a MAC algorithm that requires fewer clock cycles for computation is prefered over one that requires more clock cycles, as this directly translates to a reduction in jitter (i.e., the variance of the latency for computing the MAC).

Throughput is another important consideration. NTP servers may have to deal with thousands of client requests per second. A study [NIST] on the usage analysis of NIST's NTP stratum 1 servers shows that these servers cater to 28,000 requests/second on an average, per server.

Most of the Internet is served by stratum 2 and stratum 3 servers, some of which are a part of voluntary NTP pool. These machines may be running old hardware. Generally, while benchmarking MAC algorithms, several optimization techniques on custom specialized hardware are used to get the best results. However, for the reason stated above we choose to benchmark performance on a range of software and hardware platforms with and without optimizations.

3.2. Security Requirements

There are several more constraints specific to NTP that need to be taken into account.

1. NTP servers are stateless, i.e. they do not keep per client state.

2. Per [RFC5905], NTP uses a pre-shared symmetric key. This makes key management difficult because there is no in-band mechanism for distributing keys. As such, to simplify key management, some deployments use the same pre-shared key at many servers (typically at the same stratum). In other words, the same key is used for several client/server associations.

3. [RFC5905] also has no in-band mechanism to refresh keys.

4. Performance Results

The NTP header is 48 bytes long. We therefore consider the latency and throughput for several secure MAC algorithms when computed over 48-byte messages.

We customize the in-built speed utility of OpenSSL-1.0.2g (03 May 2016) version to compute the latency and throughput for each MAC as shown in the tables below. OpenSSL, however, does not implement stream-cipher ChaCha20-based Poly1305 MAC algorithm. To speed test this MAC, we use LibreSSL 2.3.1, a fork of OpenSSL implementation. OpenSSL and LibreSSL are the most widely used cryptographic libraries and are used by the current NTP implementations.

Since the introduction of New Instruction (NI) set for hardware support in Intel chips, certain MACs like CMAC and GMAC have performance advantage on such machines. Based on this, we perform

two different benchmarks: one with AES-NI enabled and the other with it disabled. Benchmarks were taken on an x86_64, Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz with one core CPU.

This table shows throughput in terms of number of 48-byte NTP payload processed per second.

Algorithm	with AES-NI	without AES-NI
legacy MD5	3118K	3165K
HMAC-MD5	2742K	2749K
HMAC-SHA224	1265K	1267K
CMAC(AES)	7567K	4388K
GMAC(AES)	16612K	4627K
Poly1305(ChaCha20)	2598K	2398K

This table shows latency in terms of number of CPU cycles per byte (cpb) when processing a 48-byte NTP payload.

Algorithm	with AES-NI	without AES-NI
legacy MD5	16.0	15.7
HMAC-MD5	18.2	18.1
HMAC-SHA224	39.4	39.0
CMAC(AES)	6.6	11.3
GMAC(AES)	3.0	10.8
Poly1305(ChaCha20)	14.4	15.0

5. Other Hardware Platforms

We also perform tests on the following ARM CPU cores and PowerPC(PPC)core with OpenSSL 1.0.2h released May 2016. These cores are most commonly used in consumer products and Industrial Control Systems (ICS) components. We select these cores to cover the ARM architecture versions 5/6 to 8. The results vary depending on the availability of CPU specific optimizations. For example the used Cortex-A9 and Cortex-A53 CPU have a NEON unit and OpenSSL can utilize it to accelerate AES.

1. Freescale/Apple PPC74xx 1.5GHz

2. NXP i.MX6 1GHz (dual core) ARM Cortex-A9

3. Broadcom BCM2837 1.2GHz (quad core) ARM Cortex-A53

4. Marvell 88F6281 1.2GHz 88FR131 (ARMv5te compliant)

The table below shows throughput in terms of number of 48-byte NTP payload processed per second.

Algorithm	PPC74xx	ARM Cortex-A9	ARM Cortex A-53	Marvell
legacy MD5	600K	543K	748K	383k
HMAC-MD5	463K	415K	864K	438k
HMAC-SHA224	276K	245K	357K	150k
CMAC(AES)	576K	412K	614K	246k
GMAC(AES)	681K	1362K	2193K	453k
Poly1305(ChaCha20)	335K	379K	580K	273k

The table below shows latency in terms of number of CPU cycles per byte (cpb) when processing a 48-byte NTP payload.

Algorithm	PPC74xx	ARM Cortex-A9	ARM Cortex A-53	Marvell
legacy MD5	52.1	38.4	33.4	65.3
HMAC-MD5	67.4	50.3	29.0	57.1
HMAC-SHA224	113.3	85.2	70.1	166.5
CMAC(AES)	54.2	50.5	40.7	101.2
GMAC(AES)	50.0	15.3	11.4	55.1
Poly1305(ChaCha20)	93.1	55.0	43.1	91.7

6. Security Considerations

The MD5 (key||message) "message authentication code" specified in [RFC5905] is vulnerable to length extension attacks, and uses the insecure MD5 hash function, and therefore MUST be deprecated.

Therefore, we consider hash-based MACs (HMAC-MD5, HMAC-SHA224), and cipher-based MACs (CMAC-AES, Poly1305 (ChaCha20)). The upper bound on the security level provided by any MAC against brute-force attacks is min (key-length, tag-length). The security of these MACs can be worse but not better than this bound. All MAC algorithms we consider have comparable key-lengths and output tag-lengths. So the advantage of an adversary that wishes to forge a MAC is lower-bounded by 1/2^{128}.

Assume that an adversary can obtain a valid MAC for q distinct messages. Then the table below describes the advantage of an adversary that wishes to forge a MAC in terms of number of queries (q) it launches.

Algorithm	Advantage
HMAC-MD5 [MB]	q^2/2^{128}
HMAC-SHA224 [BCK]	q^2/2^{224}
CMAC(AES)[IK]	q^2/2^{128}
GMAC(AES) [IOM]	q^2/2^{128}
Poly1305(ChaCha20) [DJB]	{e^{{q^2}/{2^{129}}}}/2^{103}

Poly1305 can easily handle up to q=2^{64} but security degrades pretty rapidly after that.

However, the bounds in the table above are somewhat optimistic, for the following reasons.

1. GMAC has an initialization vector (IV) that [RFC4106] allows to be 1 <= len(IV) <= 2^{64}-1. Per [RFC4106], implementations are optimized to handle a 12-octet IV. With a 12-octet IV, the total number of message invocations is bound to 2^{48}. Moreover, if the IV is reused even once (for the same secret authentication key and different input messages), then [Joux] shows that the secret authetication key can easily be recovered by the adversary. Notice that this attack is even stronger than a message forgery because it recovers the authentication key. This is known as nonce-reuse vulnerability.

2. The other three algorithms evaluated here do not suffer from nonce reuse vulnerabilities where an adversary can recover the authentication key if the nonce is reused just once.

3. The table above suggests that for CMAC, the total number of invocations of the MAC is limited to 2^{64}. However, [NIST-CMAC] recommends, to be on the safe side, that the total number of invocations of the block cipher algorithm during the lifetime of the key is limited to 2^{48}.

6.1. Why is GMAC not suitable for NTP?

[Joux] showed that for GMAC-AES, if the IV is repeated just once, then the authentication key can be fully recovered. None of the other algorithms evaluated here have this vulnerability. Thus, for

GMAC-AES to be secure, we need to make sure that IV is never repeated.

[NIST-GMAC] recommends constructing the 12-byte IV used in GMAC by concatenating a fixed 4-byte salt value concatenate with a variable 8-byte nonce i.e. IV = (salt|| nonce). Here salt is an implicit value established when a session is established, remains fixed for all exchanges in a session (i.e. for all invocations that use the same authenication key) between the sender and the receiver. Meanwhile, the nonce is freshly generated for each authenticated message.

Because NTP servers do not keep per-client state, the nonce can not be a sequential value. Instead, this nonce must be randomly generated 8-bytes value chosen freshly for each authenticated message. According to birthday bound, the nonce value will be repeated, with high probability, after 2^{32} messages sent in a given association. This leads to a repeated IV value and to [Joux]'s attack. Thus, to prevent repeated nonces, we would need to require the authentication key to be refreshed for the association after 2^{32} messages.

On one hand, 2^{32} is a lot of queries for an honest client, assuming that the client queries once per minute (which is NTP's minimum polling interval [RFC5905]). On the other hand, a man-in-the-middle (MiTM) can quickly and easily exhaust this number by replaying old authenticated queries to the NTP server.

The main problem here is that NTP lacks an explict in-band key refresh mechanism that can be invoked automatically (without operator intervention). And a key refresh mechanism is unlikely to be adopted as it would allow denial-of-service (DoS) attacks. The state less nature makes NTP resilient against DoS attacks.

Even if there was a method by which key-refresh could be performed, there is an additional problem. An NTP server does not keep per-client state. Therefore, it cannot keep track of the number of messages it sent in a given association. One idea is to have the client keep this state, and then send an authenicated request for a key refresh. However, a man-in-the-middle could replay old authenticated queries to the NTP server, and then intercept the server's' response before they reach the legitimate client. In this case, the client would never know when to ask for a key refresh.

Alternatively, the server could maintain a global counter (since it can't afford to keep per client counter). And after 2^{32} messages, it can refresh the keys with all its clients. However, a man-in-the-

middle could exhaust this number quickly and the server will have to refresh keys with all the clients very frequently.

Thus, we conclude that a scheme that requires refreshing the key after 2^{32} client queries is not a good idea at all.

Even in the absence of a man-in-the-middle, there is also the problem of multiple servers using the same authentication key. The salt could be used to distinguish IVs across different client/server associations that use the same authenication key. However, this brings us back to the original key management problem. One way to deal with this is to choose the 4-byte salt at random. However, this gives rise to a birthday bound of 2^{16} = 65,000 unique IVs. If we consider 20,000 stratum 3 clients synchronizing to three stratum 2 servers each, all of which are in the same organization and share the same symmetric key, we get very close to the birthday bound. This is another disadvantage of using GMAC with NTP.

7. Use HMAC or CMAC instead

1. CMAC seems to be the next best choice. Leaving out GMAC, it has the best performance with and without hardware support. It is not vulnerable to nonce misuse issues.

2. HMACs are inherently slower because of their structure and also in some cases because of lack of built-in hardware support.

3. On the other hand, it is much easier to get the right implementation for HMAC compared to CMAC.

8. GMAC-SIV - Another Potential MAC Candidate

GMAC-SIV is another possible MAC candidate, which claims to be nonce-misuse resistant [SIV]. There is an IETF Internet draft for the standardization of GCM-SIV AEAD mode.

In terms of security, GCM-SIV (AEAD) achieves usual notion of nonce-based security of an authenticated encryption mode as long as a unique nonce is used per authentication key per message. If, however, the nonce is reused authenticity is still retained (unlike in GMAC).

But there is not many implementations for GCM-SIV available except for the one from the authors. We customized this code for authentication only mode GMAC-SIV and run it on an x86_64, Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz with one core CPU with AES-NI enabled. GMAC-SIV takes ~5.9 CPU cycles/byte to generate a tag of length 16 bytes on a 48-byte NTP payload. The performance efficiency

is far less than GMAC, but is slightly better than CMAC. CMAC, on the other hand is a standardized mode of operation and has several open source implementations.

9. Recommendations

From the tables we clearly see that GMAC(AES) has the best latency and throughput performance in both hardware and software implementations. It is freely available, and there is a flexibilty of changing the underlying block-cipher. However there are several security problems surrounding the use of this mode, as highlighted above, so it is not recommended.

CMAC, on the other hand, is the next best choice in terms of performance and security. So we recommend the use of CMAC (AES).

10. Acknowledgements

The authors wish to acknowledge useful discussions with Leen Alshenibr, Daniel Franke, Ethan Heilman, Kenny Paterson, Leonid Reyzin, Harlan Stenn, Mayank Varia.

11. References

11.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4106] Viega, J. and D. McGrew, "The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)", RFC 4106, DOI 10.17487/RFC4106, June 2005, <http://www.rfc-editor.org/info/rfc4106>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <http://www.rfc-editor.org/info/rfc4493>.

[RFC4543] McGrew, D. and J. Viega, "The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH", RFC 4543, DOI 10.17487/RFC4543, May 2006, <http://www.rfc-editor.org/info/rfc4543>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <http://www.rfc-editor.org/info/rfc5905>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <http://www.rfc-editor.org/info/rfc7539>.

[RFC7630] Merkle, J., Ed. and M. Lochter, "HMAC-SHA-2 Authentication Protocols in the User-based Security Model (USM) for SNMPv3", RFC 7630, DOI 10.17487/RFC7630, October 2015, <http://www.rfc-editor.org/info/rfc7630>.

11.2. Informative References

[BCK] Bellare, M., Canetti, R., and H. Krawczyk, "Keyed Hash Functions and Message Authentication", in Proceedings of Crypto'96, 1996.

[DJB] Bernstein, D., "The Poly1305-AES message-authentication code", in Fast Software Encryption, 2005.

[GK] Gueron, S. and V. Krasnov, "The fragility of AES-GCM authentication algorithm", in Proceedings of 11th International Conference on Information Technology: New Generations 2014, 2014.

[IK] Iwata, T. and K. Kurosawa, "Keyed Hash Functions and Message Authentication", in Progress in Cryptology-INDOCRYPT 2003, 2003.

[IOM] Iwata, T., Ohashi, K., and K. Minematsu, "Breaking and Repairing GCM Security Proofs", in Proceedings of CRYPTO 2012, 2012.

[Joux] Joux, A., "Authentication Failures in NIST version of GCM", <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/GCM/Joux_comments.pdf>.

[MB] Bellare, M., "New Proofs for NMAC and HMAC:Security without Collision-Resistance", in Proceedings of Crypto'96, 1996.

[NIST] Sherman, J. and J. Levine, "Usage Analysis of the NIST Internet Time Service", in Journal of Research of the National Institute of Standards and Technology, 2016.

[NIST-CMAC] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication", in NIST Special Publication 800-38B, 2005.

[NIST-GMAC] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", in NIST Special Publication 800-38D, 2007.

[SIV] Gueron, S., Langley, A., and Y. Lindell, "https://tools.ietf.org/html/draft-gueron-gcmsiv-00", in Work in Progress, 2016.

Authors' Addresses

Aanchal Malhotra
Boston University
111 Cummington St
Boston, MA 02215
US

Email: aanchal4@bu.edu

Sharon Goldberg
Boston University
111 Cummington St
Boston, MA 02215
US

Email: goldbe@cs.bu.edu

CFRG S. Gueron
Internet-Draft University of Haifa and Amazon Web Services
Intended status: Informational A. Langley
Expires: May 23, 2019 Google LLC
Y. Lindell
Bar Ilan University
November 19, 2018

AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption
draft-irtf-cfrg-gcmsiv-09

Abstract

This memo specifies two authenticated encryption algorithms that are nonce misuse-resistant - that is that they do not fail catastrophically if a nonce is repeated.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 23, 2019.

Copyright Notice

Table of Contents

1. Introduction

The concept of "Authenticated encryption with additional data" (AEAD [RFC5116]) couples confidentiality and integrity in a single operation, avoiding the risks of the previously common practice of using ad-hoc constructions of block-cipher and hash primitives. The most popular AEAD, AES-GCM [GCM], is seeing widespread use due to its attractive performance.

However, some AEADs (including AES-GCM) suffer catastrophic failures of confidentiality and/or integrity when two distinct messages are encrypted with the same key and nonce. While the requirements for AEADs specify that the pair of (key, nonce) shall only ever be used once, and thus prohibit this, in practice this is a worry.

Nonce misuse-resistant AEADs do not suffer from this problem. For this class of AEADs, encrypting two messages with the same nonce only discloses whether the messages were equal or not. This is the minimum amount of information that a deterministic algorithm can leak in this situation.

This memo specifies two nonce misuse-resistant AEADs: "AEAD_AES_128_GCM_SIV" and "AEAD_AES_256_GCM_SIV". These AEADs are designed to be able to take advantage of existing hardware support for AES-GCM and can decrypt within 5% of the speed of AES-GCM (for multi-kilobyte messages). Encryption is, perforce, slower than AES-GCM because two passes are required in order to achieve that nonce misuse-resistance property. However, measurements suggest that it can still run at two-thirds of the speed of AES-GCM.

We suggest that these AEADs be considered in any situation where nonce uniqueness cannot be guaranteed. This includes situations where there is no stateful counter or where such state cannot be guaranteed, as when multiple encryptors use the same key. As discussed in Section 9, it is RECOMMENDED to use this scheme with randomly chosen nonces.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. POLYVAL

The GCM-SIV construction is similar to GCM: the block cipher is used in counter mode to encrypt the plaintext and a polynomial authenticator is used to provide integrity. The authenticator in GCM-SIV is called POLYVAL.

POLYVAL, like GHASH (the authenticator in AES-GCM, see [GCM] section 6.4), operates in a binary field of size 2^128. The field is defined by the irreducible polynomial x^128 + x^127 + x^126 + x^121 + 1. The sum of any two elements in the field is the result of XORing them. The product of any two elements is calculated using standard (binary) polynomial multiplication followed by reduction modulo the irreducible polynomial.

We define another binary operation on elements of the field: dot(a, b), where dot(a, b) = a * b * x^-128. The value of the field element x^-128 is equal to x^127 + x^124 + x^121 + x^114 + 1. The result, dot(a, b), of this multiplication is another field element.

Polynomials in this field are converted to and from 128-bit strings by taking the least-significant bit of the first byte to be the coefficient of x^0, the most-significant bit of the first byte to the coefficient of x^7 and so on, until the most-significant bit of the last byte is the coefficient of x^127.

POLYVAL takes a field element, H, and a series of field elements X_1, ..., X_s. Its result is S_s, where S is defined by the iteration S_0 = 0; S_j = dot(S_{j-1} + X_j, H), for j = 1..s

We note that POLYVAL(H, X_1, X_2, ...) is equal to ByteReverse(GHASH(ByteReverse(H) * x, ByteReverse(X_1), ByteReverse(X_2), ...)), where ByteReverse is a function that reverses the order of 16 bytes. See Appendix A for a more detailed explanation.

4. Encryption

AES-GCM-SIV encryption takes a 16- or 32-byte key-generating key, a 96-bit nonce, and variable-length plaintext & additional data byte-strings. It outputs an authenticated ciphertext that will be 16 bytes longer than the plaintext. Both encryption and decryption are only defined on inputs that are a whole number of bytes.

If the key-generating key is 16 bytes long then AES-128 is used throughout. Otherwise AES-256 is used throughout.

The first step of encryption is to generate per-nonce, message-authentication and message-encryption keys. The message-authentication key is 128-bit and the message-encryption key is either 128- (for AES-128) or 256-bit (for AES-256).

These keys are generated by encrypting a series of plaintext blocks that contain a 32-bit, little-endian counter followed by the nonce, and then discarding the second half of the resulting ciphertext. In the AES-128 case, 128 + 128 = 256 bits of key material need to be generated and, since encrypting each block yields 64 bits after discarding half, four blocks need to be encrypted. The counter values for these blocks are 0, 1, 2 and 3. For AES-256, six blocks are needed in total, with counter values 0 through 5 (inclusive).

In pseudocode form, where ++ indicates concatenation and x[:8] indicates taking only the first eight bytes from x:

```
func derive_keys(key_generating_key, nonce) {
  message_authentication_key =
      AES(key = key_generating_key,
          block = little_endian_uint32(0) ++ nonce)[:8] ++
      AES(key = key_generating_key,
          block = little_endian_uint32(1) ++ nonce)[:8]
  message_encryption_key =
      AES(key = key_generating_key,
          block = little_endian_uint32(2) ++ nonce)[:8] ++
      AES(key = key_generating_key,
          block = little_endian_uint32(3) ++ nonce)[:8]

  if bytelen(key_generating_key) == 32 {
    message_encryption_key ++=
        AES(key = key_generating_key,
            block = little_endian_uint32(4) ++ nonce)[:8] ++
        AES(key = key_generating_key,
            block = little_endian_uint32(5) ++ nonce)[:8]
  }

  return message_authentication_key, message_encryption_key
}
```

Define the "length block" as a 16-byte value that is the concatenation of the 64-bit, little-endian encodings of bytelen(additional_data) * 8 and bytelen(plaintext) * 8. Pad the plaintext and additional data with zeros until they are each a multiple of 16 bytes, the AES block size. Then X_1, X_2, ... (the series of field elements that are inputs to POLYVAL) are the concatenation of the padded additional data, the padded plaintext, and the length block.

Calculate S_s = POLYVAL(message-authentication-key, X_1, X_2, ...). XOR the first twelve bytes of S_s with the nonce and clear the most-significant bit of the last byte. Encrypt the result with AES using the message-encryption key to produce the tag.

(It’s worth highlighting a contrast with AES-GCM here: AES-GCM authenticates the encoded additional data and ciphertext, while AES-GCM-SIV authenticates the encoded additional data and _plaintext_.)

The encrypted plaintext is produced by using AES, with the message-encryption key, in counter mode (see [SP800-38A], section 6.5) on the unpadded plaintext. The initial counter block is the tag with the most-significant bit of the last byte set to one. The counter advances by incrementing the first 32 bits interpreted as an unsigned, little-endian integer, wrapping at 2^32. The result of the

encryption is the encrypted plaintext (truncated to the length of the plaintext) followed by the tag.

In pseudo-code form, the encryption process can be expressed as:

```
func right_pad_to_multiple_of_16_bytes(input) {
  while (bytelen(input) % 16 != 0) {
    input = input ++ "\x00"
  }
  return input
}

func AES_CTR(key, initial_counter_block, in) {
  block = initial_counter_block

  output = ""
  while bytelen(in) > 0 {
    keystream_block = AES(key = key, block = block)
    block[0:4] = little_endian_uint32(
        read_little_endian_uint32(block[0:4]) + 1)

    todo = min(bytelen(in), bytelen(keystream_block)
    for j = 0; j < todo; j++ {
      output = output ++ (keystream_block[j] ^ in[j])
    }

    in = in[todo:]
  }

  return output
}

func encrypt(key_generating_key,
             nonce,
             plaintext,
             additional_data) {
  if bytelen(plaintext) > 2**36 {
    fail()
  }
  if bytelen(additional_data) > 2**36 {
    fail()
  }

  message_encryption_key, message_authentication_key =
      derive_keys(key_generating_key, nonce)

  length_block =
      little_endian_uint64(bytelen(additional_data) * 8) ++
```

```
        little_endian_uint64(bytelen(plaintext) * 8)
    padded_plaintext = right_pad_to_multiple_of_16_bytes(plaintext)
    padded_ad = right_pad_to_multiple_of_16_bytes(additional_data)
    S_s = POLYVAL(key = message_authentication_key,
                  input = padded_ad ++ padded_plaintext ++
                          length_block)
    for i = 0; i < 12; i++ {
      S_s[i] ^= nonce[i]
    }
    S_s[15] &= 0x7f
    tag = AES(key = message_encryption_key, block = S_s)

    counter_block = tag
    counter_block[15] |= 0x80
    return AES_CTR(key = message_encryption_key,
                   initial_counter_block = counter_block,
                   in = plaintext) ++
           tag
}
```

5. Decryption

Decryption takes a 16- or 32-byte key-generating key, a 96-bit nonce, and variable-length ciphertext & additional data byte-strings. It either fails, or outputs a plaintext that is 16 bytes shorter than the ciphertext.

To decrypt an AES-GCM-SIV ciphertext, first derive the message-encryption and message-authentication keys in the same manner as when encrypting.

If the ciphertext is less than 16 bytes or more than 2^36 + 16 bytes, then fail. Otherwise split the input into the encrypted plaintext and a 16-byte tag. Decrypt the encrypted plaintext with the message-encryption key in counter mode, where the initial counter block is the tag with the most-significant bit of the last byte set to one. Advance the counter for each block in the same way as when encrypting. At this point the plaintext is unauthenticated and MUST NOT be output until the following tag confirmation is complete:

Pad the additional data and plaintext with zeros until they are each a multiple of 16 bytes, the AES block size. Calculate the length block and X_1, X_2, ... as above and compute S_s = POLYVAL(message-authentication-key, X_1, X_2, ...). Compute the expected tag by XORing S_s and the nonce, clearing the most-significant bit of the last byte and encrypting with the message-encryption key. Compare the provided and expected tag values in constant time. Fail the

decryption if they do not match (and do not release the plaintext), otherwise return the plaintext.

In pseudo-code form, the decryption process can be expressed as:

```
   func decrypt(key_generating_key,
                nonce,
                ciphertext,
                additional_data) {
     if bytelen(ciphertext) < 16 || bytelen(ciphertext) > 2**36 + 16 {
       fail()
     }
     if bytelen(additional_data) > 2**36 {
       fail()
     }

     message_encryption_key, message_authentication_key =
         derive_keys(key_generating_key, nonce)

     tag = ciphertext[bytelen(ciphertext)-16:]

     counter_block = tag
     counter_block[15] |= 0x80
     plaintext = AES_CTR(key = message_encryption_key,
                         initial_counter_block = counter_block,
                         in = ciphertext[:bytelen(ciphertext)-16])

     length_block =
         little_endian_uint64(bytelen(additional_data) * 8) ++
         little_endian_uint64(bytelen(plaintext) * 8)
     padded_plaintext = right_pad_to_multiple_of_16_bytes(plaintext)
     padded_ad = right_pad_to_multiple_of_16_bytes(additional_data)
     S_s = POLYVAL(key = message_authentication_key,
                   input = padded_ad ++ padded_plaintext ++
                           length_block)
     for i = 0; i < 12; i++ {
       S_s[i] ^= nonce[i]
     }
     S_s[15] &= 0x7f
     expected_tag = AES(key = message_encryption_key, block = S_s)

     xor_sum = 0
     for i := 0; i < bytelen(expected_tag); i++ {
       xor_sum |= expected_tag[i] ^ tag[i]
     }

     if xor_sum != 0 {
       fail()
     }

     return plaintext
   }
```

6. AEADs

We define two AEADs, in the format of RFC 5116, that use AES-GCM-SIV: AEAD_AES_128_GCM_SIV and AEAD_AES_256_GCM_SIV. They differ only in the size of the AES key used.

The key input to these AEADs becomes the key-generating key. Thus AEAD_AES_128_GCM_SIV takes a 16-byte key and AEAD_AES_256_GCM_SIV takes a 32-byte key.

The parameters for AEAD_AES_128_GCM_SIV are then: K_LEN is 16, P_MAX is 2^36, A_MAX is 2^36, N_MIN and N_MAX are 12 and C_MAX is 2^36 + 16.

The parameters for AEAD_AES_256_GCM_SIV differ only in the key size: K_LEN is 32, P_MAX is 2^36, A_MAX is 2^36, N_MIN and N_MAX are 12 and C_MAX is 2^36 + 16.

7. Field operation examples

Polynomials in this document will be written as 16-byte values. For example, the sixteen bytes 01000000000000000000000000000492 would represent the polynomial x^127 + x^124 + x^121 + x^114 + 1, which is also the value of x^-128 in this field.

If a = 66e94bd4ef8a2c3b884cfa59ca342b2e and b = ff000000000000000000000000000000 then a + b = 99e94bd4ef8a2c3b884cfa59ca342b2e, a * b = 37856175e9dc9df26ebc6d6171aa0ae9 and dot(a, b) = ebe563401e7e91ea3ad6426b8140c394.

8. Worked example

Consider the encryption of the plaintext "Hello world" with the additional data "example" under key ee8e1ed9ff2540ae8f2ba9f50bc2f27c using AEAD_AES_128_GCM_SIV. The random nonce that we'll use for this example is 752abad3e0afb5f434dc4310.

In order to generate the message-authentication and message-encryption keys, a counter is combined with the nonce to form four blocks. These blocks are encrypted with key given above:

```
Counter | Nonce                          Ciphertext
00000000752abad3e0afb5f434dc4310 -> 310728d9911f1f38c40e952ca83d093e
01000000752abad3e0afb5f434dc4310 -> 37b24316c3fab9a046ae90952daa0450
02000000752abad3e0afb5f434dc4310 -> a4c5ae624996327947920b2d2412474b
03000000752abad3e0afb5f434dc4310 -> c100be4d7e2c6edd1efef004305ab1e7
```

The latter halves of the ciphertext blocks are discarded and the remaining bytes are concatenated to form the per-message keys. Thus the message-authentication key is 310728d9911f1f3837b24316c3fab9a0 and the message-encryption key is a4c5ae6249963279c100be4d7e2c6edd.

The length block contains the encoding of the bit-lengths of the additional data and plaintext, respectively. The string "example" is seven characters, thus 56 bits (or 0x38 in hex). The string "Hello world" is 11 characters, or 88 = 0x58 bits. Thus the length block is 38000000000000005800000000000000.

The input to POLYVAL is the padded additional data, padded plaintext and then the length block. This is 6578616d706c650000000000000000004 8656c6c6f20776f726c6400000000003800000000000000580000000000000000, based on the ASCII encoding of "example" (6578616d706c65) and of "Hello world" (48656c6c6f20776f726c64).

Calling POLYVAL with the message-authentication key and the input above results in S_s = ad7fcf0b5169851662672f3c5f95138f.

Before encrypting, the nonce is XORed in and the most-significant bit of the last byte is cleared. This gives d85575d8b1c630e256bb6c2c5f95130f because that bit happened to be one previously. Encrypting with the message-encryption key (using AES-128) gives the tag, which is 4fbcdeb7e4793f4a1d7e4faa70100af1.

In order to form the initial counter block, the most-significant bit of the last byte of the tag is set to one. That doesn't result in a change in this example. Encrypting this with the message key (using AES-128) gives the first block of the keystream: 1551f2c1787e81deac9a99f139540ab5.

The final ciphertext is the result of XORing the plaintext with the keystream and appending the tag. That gives 5d349ead175ef6b1def6fd4fbcdeb7e4793f4a1d7e4faa70100af1.

9. Security Considerations

AES-GCM-SIV decryption involves first producing an unauthenticated plaintext. This plaintext is vulnerable to manipulation by an attacker thus, if an implementation released some or all of the plaintext before authenticating it, other parts of a system may process malicious data as if it were authentic. AES-GCM might be less likely to lead implementations to do this because, there, the ciphertext is generally authenticated before, or concurrently with, the plaintext calculation. Therefore this text requires that implementations MUST NOT release unauthenticated plaintext. Thus system designers should consider memory limitations when picking the

size of AES-GCM-SIV plaintexts: large plaintexts may not fit in the availible memory of some machines, tempting implementations to release unverified plaintext.

A detailed cryptographic analysis of AES-GCM-SIV appears in [AES-GCM-SIV] and the remainder of this section is a summary of that paper.

The AEADs defined in this document calculate fresh AES keys for each nonce. This allows a larger number of plaintexts to be encrypted under a given key. Without this step, AES-GCM-SIV encryption would be limited by the birthday bound like other standard modes (e.g., AES-GCM, AES-CCM [RFC3610], and AES-SIV [RFC5297]). This means that when 2^64 blocks have been encrypted overall, a distinguishing adversary, who is trying to break the confidentiality of the scheme, has an advantage of 1/2. Thus, in order to limit the adversary's advantage to 2^-32, at most 2^48 blocks can be encrypted overall. In contrast, by deriving fresh keys from each nonce, it is possible to encrypt a far larger number of messages and blocks with AES-GCM-SIV.

We stress that nonce-misuse resistant schemes guarantee that if a nonce repeats then the only security loss is that identical plaintexts will produce identical ciphertexts. Since this can also be a concern (as the fact that the same plaintext has been encrypted twice is revealed), we do not recommend using a fixed nonce as a policy. In addition, as we show below, better-than-birthday bounds are achieved by AES-GCM-SIV when the nonce repetition rate is low. Finally, as shown in [BHT18], there is a great security benefit in the multi-user/multi-key setting when each particular nonce is re-used by a small number of users only. We stress that the nonce misuse-resistance property is not intended to be coupled with intentional nonce-reuse; rather, such schemes provide the best possible security in the event of nonce reuse. Due to all of the above, it is RECOMMENDED that AES-GCM-SIV nonces be randomly generated.

Some example usage bounds for AES-GCM-SIV are given below. The adversary's advantage is the "AdvEnc" from [key-derive] and is colloquially the ability of an attacker to distinguish ciphertexts from random bit-strings. The bounds below limit this advantage to 2^-32. For up to 256 uses of the same nonce and key (i.e., where one can assume that nonce misuse is no more than this bound), the following message limits should be respected (this assumes a short AAD, i.e. less than 64 bytes):

2^29 messages, where each plaintext is at most 1GiB

2^35 messages, where each plaintext is at most 128MiB

2^49 messages, where each plaintext is at most 1MiB

2^61 messages, where each plaintext is at most 16KiB

Suzuki et al [multi-birthday] show that even if nonces are selected uniformly at random, the probability that one or more values would be repeated 256 or more times is negligible until the number of nonces reaches 2^102. (Specifically the probability is 1/((2^96)^(255)) * Binomial(q, 256), where q is the number of nonces.) Since 2^102 is vastly greater than the limit on the number of plaintexts per key given above, we don't feel that this limit on the number of repeated nonces will be a problem. This also means that selecting nonces at random is a safe practice with AES-GCM-SIV. The bounds obtained for random nonces are as follows (as above, for these bounds the adversary's advantage is at most 2^-32):

2^32 messages, where each plaintext is at most 8GiB

2^48 messages, where each plaintext is at most 32MiB

2^64 messages, where each plaintext is at most 128KiB

For situations where, for some reason, an even higher number of nonce repeats is possible (e.g. in devices with very poor randomness), the message limits need to be reconsidered. Theorem 7 in [AES-GCM-SIV] contains more details but, for up to 1,024 repeats of each nonce, the limits would be (again assuming a short AAD, i.e. less than 64 bytes):

2^25 messages, where each plaintext is at most 1GiB

2^31 messages, where each plaintext is at most 128MiB

2^45 messages, where each plaintext is at most 1MiB

2^57 messages, where each plaintext is at most 16KiB

In addition to calculating fresh AES keys for each nonce, these AEADs also calculate fresh POLYVAL keys. Previous versions of GCM-SIV did not do this and, instead, used part of the AEAD's key as the POLYVAL key. Bleichenbacher pointed out that this allowed an attacker who controlled the AEAD key to force the POLYVAL key to be zero. If a user of this AEAD authenticated messages with a secret additional-data value then this would be insecure as the attacker could calculate a valid authenticator without knowing the input. This does not violate the standard properties of an AEAD as the additional data is not assumed to be confidential. However, we want these AEADs to be robust to plausible misuse and also to be drop-in replacements for

AES-GCM and so derive nonce-specific POLYVAL keys to avoid this issue.

We also wish to note that the probability of successful forgery increases with the number of attempts that an attacker is permitted. The advantage defined in [key-derive] and used above is specified in terms of the ability of an attacker to distinguish ciphertexts from random bit-strings. It thus covers both confidentiality and integrity and theorem 6.2 in [key-derive] shows that the advantage increases with the number of decryption attempts. (Although much more slowly than with the number of encryptions; the dependence on the number of decryption queries for forgery is actually only linear, not quadratic. The latter is an artifact of the bound in the paper not being tight.) If an attacker is permitted extremely large numbers of attempts then the tiny probability that any given attempt succeeds may sum to a non-trivial chance.

A security analysis of a similar scheme without nonce-based key derivation appears in [GCM-SIV] and a full analysis of the bounds when applying nonce-based key derivation appears in [key-derive]. A larger table of bounds and other information appears at [aes-gcm-siv-homepage].

The multi-user/multi-key security of AES-GCM-SIV was studied by [BHT18] who showed that security is almost like in the single user setting, as long as nonces do not repeat many times across many users. This is the case when nonces are chosen randomly.

10. IANA Considerations

IANA is requested to add two entries to the registry of AEAD algorithms: AEAD_AES_128_GCM_SIV and AEAD_AES_256_GCM_SIV, both referencing this document as their specification.

11. Acknowledgements

The authors would like to thank Uri Blumenthal, Ondrej Mosnacek, Daniel Bleichenbacher, Kenny Paterson, Bart Preneel, John Mattsson, Scott Fluhrer, Tibor Jager, Bjoern Tackmann, Yannick Seurin, Tetsu Iwata and Deb Cooley's team at NSA Information Assurance for their helpful suggestions and review.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[SP800-38A] Dworkin, M., "SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST SP-800-38A, December 2001, <https://csrc.nist.gov/publications/detail/sp/800-38a/final>.

12.2. Informative References

[AES-GCM-SIV] Gueron, S., Langley, A., and Y. Lindell, "AES-GCM-SIV: specification and analysis", July 2017, <https://eprint.iacr.org/2017/168>.

[aes-gcm-siv-homepage] Gueron, S., Langley, A., and Y. Lindell, "Webpage for the AES-GCM-SIV Mode of Operation", 2017, <https://cyber.biu.ac.il/aes-gcm-siv/>.

[BHT18] Bose, P., Hoang, V., and S. Tessaro, "Revisiting AES-GCM-SIV: Multi-user Security, Faster Key Derivation, and Better Bounds", Proceedings of EUROCRYPT 2018 , May 2018, <https://eprint.iacr.org/2018/136.pdf>.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST SP-800-38D, November 2007, <https://csrc.nist.gov/publications/detail/sp/800-38d/final>.

[GCM-SIV] Gueron, S. and Y. Lindell, "GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle Per Byte", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security , 2015, <http://doi.acm.org/10.1145/2810103.2813613>.

[key-derive]
Gueron, S. and Y. Lindell, "Better Bounds for Block Cipher Modes of Operation via Nonce-Based Key Derivation", Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security , 2017, <https://doi.org/10.1145/3133956.3133992>.

[multi-birthday]
Suzuki, K., Tonien, D., Kurosawa, K., and K. Toyota, "Birthday Paradox for Multi-collisions", ICISC 2006: 9th International Conference, Busan, Korea, November 30 - December 1, 2006. Proceedings , 2006, <http://dx.doi.org/10.1007/11927587_5>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <https://www.rfc-editor.org/info/rfc3610>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <https://www.rfc-editor.org/info/rfc5297>.

Appendix A. The relationship between POLYVAL and GHASH

GHASH and POLYVAL both operate in GF(2^128), although with different irreducible polynomials: POLYVAL works modulo x^128 + x^127 + x^126 + x^121 + 1 and GHASH works modulo x^128 + x^7 + x^2 + x + 1. Note that these irreducible polynomials are the "reverse" of each other.

GHASH also has a different mapping between 128-bit strings and field elements. Where as POLYVAL takes the least-significant to most-significant bits of the first byte to be the coefficients of x^0 to x^7, GHASH takes them to be the coefficients of x^7 to x^0. This continues until, for the last byte, POLYVAL takes the least-significant to most-significant bits to be the coefficients of x^120 to x^127 while GHASH takes them to be the coefficients of x^127 to x^120.

The combination of these facts means that it’s possible to "convert" values between the two by reversing the order of the bytes in a 16-byte string. The differing interpretations of bit order takes care of reversing the bits within each byte and then reversing the

bytes does the rest. This may have a practical benefit for implementations that wish to implement both GHASH and POLYVAL.

In order to be clear which field a given operation is performed in, let mulX_GHASH be a function that takes a 16-byte string, converts it to an element of GHASH's field using GHASH's convention, multiplies it by x and converts back to a string. Likewise, let mulX_POLYVAL be a function that converts a 16-byte string to an element of POLYVAL's field using POLYVAL's convention, multiplies it by x and converts back.

Given the 16-byte string 01000000000000000000000000000000, mulX_GHASH of that string is 00800000000000000000000000000000 and mulX_POLYVAL of that string is 02000000000000000000000000000000. As a more general example, given 9c98c04df9387ded828175a92ba652d8, mulX_GHASH of that string is 4e4c6026fc9c3ef6c140bad495d3296c and mulX_POLYVAL of it is 3931819bf271fada0503eb52574ca5f2.

Lastly, let ByteReverse be the function that takes a 16-byte string and returns a copy where the order of the bytes has been reversed.

Now GHASH and POLYVAL can be defined in terms of one another:

POLYVAL(H, X_1, ..., X_n) =
ByteReverse(GHASH(mulX_GHASH(ByteReverse(H)), ByteReverse(X_1), ..., ByteReverse(X_n)))

GHASH(H, X_1, ..., X_n) =
ByteReverse(POLYVAL(mulX_POLYVAL(ByteReverse(H)), ByteReverse(X_1), ..., ByteReverse(X_n)))

As a worked example, let H = 25629347589242761d31f826ba4b757b, X_1 = 4f4f95668c83dfb6401762bb2d01a262 and X_2 = d1a24ddd2721d006bbe45f20d3c9f362. POLYVAL(H, X_1, X_2) = f7a3b47b846119fae5b7866cf5e5b77e. If we wished to calculate this given only an implementation of GHASH then the key for GHASH would be mulX_GHASH(ByteReverse(H)) = dcbaa5dd137c188ebb21492c23c9b112. Then ByteReverse(GHASH(dcba..., ByteReverse(X_1), ByteReverse(X_2))) = f7a3b47b846119fae5b7866cf5e5b77e, as required.

In the other direction, GHASH(H, X_1, X_2) = bd9b3997046731fb96251b91f9c99d7a. If we wished to calculate this given only an implementation of POLYVAL then we would first calculate the key for POLYVAL, mulX_POLYVAL(ByteReverse(H)), which is f6ea96744df0633aec8424b18e26c54a. Then ByteReverse(POLYVAL(f6ea..., ByteReverse(X_1), ByteReverse(X_2))) = bd9b3997046731fb96251b91f9c99d7a.

Appendix B. Additional comparisons with AES-GCM

Some functional properties also differ between AES-GCM and AES-GCM-SIV that are worth noting:

AES-GCM allows plaintexts to be encrypted in a streaming fashion, i.e. the beginning of the plaintext can be encrypted and transmitted before the entire message has been processed. AES-GCM-SIV requires two passes for encryption and so cannot do this.

AES-GCM allows a constant additional-data input to be precomputed in order to save per-message computation. AES-GCM-SIV varies the authenticator key based on the nonce and so does not permit this.

The performance for AES-GCM vs AES-GCM-SIV on small machines can be roughly characterised by the number of AES operations and the number of GF(2^128) multiplications needed to process a message. Let a = (bytelen(additional-data) + 15) / 16 and p = (bytelen(plaintext) + 15) / 16. Then AES-GCM requires p + 1 AES operations and p + a + 1 field multiplications.

Defined similarly, AES-GCM-SIV with AES-128 requires p + 5 AES operations and p + a + 1 field multiplications. With AES-256 that becomes p + 7 AES operations.

With large machines, the available parallelism becomes far more important and such simple performance analysis is no longer representative. For such machines, we find that decryption of AES-GCM-SIV is only about 5% slower then AES-GCM, as long as the message is at least a couple of kilobytes. Encryption tends to run about two-thirds the speed because of the additional pass required.

Appendix C. Test vectors

C.1. AEAD_AES_128_GCM_SIV

```
Plaintext (0 bytes) =
AAD (0 bytes) =
Key =                          01000000000000000000000000000000
Nonce =                        030000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input =             00000000000000000000000000000000
POLYVAL result =            00000000000000000000000000000000
POLYVAL result XOR nonce =  03000000000000000000000000000000
... and masked =            03000000000000000000000000000000
Tag =                       dc20e2d83f25705bb49e439eca56de25
Initial counter =           dc20e2d83f25705bb49e439eca56dea5
```

```
   Result (16 bytes) =         dc20e2d83f25705bb49e439eca56de25


   Plaintext (8 bytes) =       0100000000000000
   AAD (0 bytes) =
   Key =                       0100000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             0100000000000000000000000000000000
                               0000000000000000400000000000000000
   POLYVAL result =            eb93b7740962c5e49d2a90a7dc5cec74
   POLYVAL result XOR nonce =  e893b7740962c5e49d2a90a7dc5cec74
   ... and masked =            e893b7740962c5e49d2a90a7dc5cec74
   Tag =                       578782fff6013b815b287c22493a364c
   Initial counter =           578782fff6013b815b287c22493a36cc
   Result (24 bytes) =         b5d839330ac7b786578782fff6013b81
                               5b287c22493a364c


   Plaintext (12 bytes) =      010000000000000000000000
   AAD (0 bytes) =
   Key =                       0100000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             0100000000000000000000000000000000
                               0000000000000000600000000000000000
   POLYVAL result =            48eb6c6c5a2dbe4a1dde508fee06361b
   POLYVAL result XOR nonce =  4beb6c6c5a2dbe4a1dde508fee06361b
   ... and masked =            4beb6c6c5a2dbe4a1dde508fee06361b
   Tag =                       a4978db357391a0bc4fdec8b0d106639
   Initial counter =           a4978db357391a0bc4fdec8b0d1066b9
   Result (28 bytes) =         7323ea61d05932260047d942a4978db3
                               57391a0bc4fdec8b0d106639


   Plaintext (16 bytes) =      01000000000000000000000000000000
   AAD (0 bytes) =
   Key =                       0100000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             0100000000000000000000000000000000
                               0000000000000000800000000000000000
   POLYVAL result =            20806c26e3c1de019e111255708031d6
   POLYVAL result XOR nonce =  23806c26e3c1de019e111255708031d6
   ... and masked =            23806c26e3c1de019e11125570803156
```

```
   Tag =                          303aaf90f6fe21199c6068577437a0c4
   Initial counter =              303aaf90f6fe21199c6068577437a0c4
   Result (32 bytes) =            743f7c8077ab25f8624e2e948579cf77
                                  303aaf90f6fe21199c6068577437a0c4


   Plaintext (32 bytes) =         01000000000000000000000000000000
                                  02000000000000000000000000000000
   AAD (0 bytes) =
   Key =                          01000000000000000000000000000000
   Nonce =                        030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =        4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =                01000000000000000000000000000000
                                  02000000000000000000000000000000
                                  00000000000000000001000000000000
   POLYVAL result =               ce6edc9a50b36d9a98986bbf6a261c3b
   POLYVAL result XOR nonce =     cd6edc9a50b36d9a98986bbf6a261c3b
   ... and masked =               cd6edc9a50b36d9a98986bbf6a261c3b
   Tag =                          1a8e45dcd4578c667cd86847bf6155ff
   Initial counter =              1a8e45dcd4578c667cd86847bf6155ff
   Result (48 bytes) =            84e07e62ba83a6585417245d7ec413a9
                                  fe427d6315c09b57ce45f2e3936a9445
                                  1a8e45dcd4578c667cd86847bf6155ff


   Plaintext (48 bytes) =         01000000000000000000000000000000
                                  02000000000000000000000000000000
                                  03000000000000000000000000000000
   AAD (0 bytes) =
   Key =                          01000000000000000000000000000000
   Nonce =                        030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =        4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =                01000000000000000000000000000000
                                  02000000000000000000000000000000
                                  03000000000000000000000000000000
                                  00000000000000008001000000000000
   POLYVAL result =               81388746bc22d26b2abc3dcb15754222
   POLYVAL result XOR nonce =     82388746bc22d26b2abc3dcb15754222
   ... and masked =               82388746bc22d26b2abc3dcb15754222
   Tag =                          5e6e311dbf395d35b0fe39c2714388f8
   Initial counter =              5e6e311dbf395d35b0fe39c2714388f8
   Result (64 bytes) =            3fd24ce1f5a67b75bf2351f181a475c7
                                  b800a5b4d3dcf70106b1eea82fa1d64d
                                  f42bf7226122fa92e17a40eeaac1201b
                                  5e6e311dbf395d35b0fe39c2714388f8
```

```
   Plaintext (64 bytes) =      01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
   AAD (0 bytes) =
   Key =                       01000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
                               00000000000000000020000000000000
   POLYVAL result =            1e39b6d3344d348f6044f89935d1cf78
   POLYVAL result XOR nonce =  1d39b6d3344d348f6044f89935d1cf78
   ... and masked =            1d39b6d3344d348f6044f89935d1cf78
   Tag =                       8a263dd317aa88d56bdf3936dba75bb8
   Initial counter =           8a263dd317aa88d56bdf3936dba75bb8
   Result (80 bytes) =         2433668f1058190f6d43e360f4f35cd8
                               e475127cfca7028ea8ab5c20f7ab2af0
                               2516a2bdcbc08d521be37ff28c152bba
                               36697f25b4cd169c6590d1dd39566d3f
                               8a263dd317aa88d56bdf3936dba75bb8


   Plaintext (8 bytes) =       0200000000000000
   AAD (1 bytes) =             01
   Key =                       01000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               08000000000000004000000000000000
   POLYVAL result =            b26781e7e2c1376f96bec195f3709b2a
   POLYVAL result XOR nonce =  b16781e7e2c1376f96bec195f3709b2a
   ... and masked =            b16781e7e2c1376f96bec195f3709b2a
   Tag =                       3b0a1a2560969cdf790d99759abd1508
   Initial counter =           3b0a1a2560969cdf790d99759abd1588
   Result (24 bytes) =         1e6daba35669f4273b0a1a2560969cdf
                               790d99759abd1508


   Plaintext (12 bytes) =      020000000000000000000000
   AAD (1 bytes) =             01
   Key =                       01000000000000000000000000000000
   Nonce =                     030000000000000000000000
```

```
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input =             01000000000000000000000000000000
                            02000000000000000000000000000000
                            08000000000000006000000000000000
POLYVAL result =            111f5affb18e4cc1164a01bdc12a4145
POLYVAL result XOR nonce =  121f5affb18e4cc1164a01bdc12a4145
... and masked =            121f5affb18e4cc1164a01bdc12a4145
Tag =                       08299c5102745aaa3a0c469fad9e075a
Initial counter =           08299c5102745aaa3a0c469fad9e07da
Result (28 bytes) =         296c7889fd99f41917f4462008299c51
                            02745aaa3a0c469fad9e075a


Plaintext (16 bytes) =      02000000000000000000000000000000
AAD (1 bytes) =             01
Key =                       01000000000000000000000000000000
Nonce =                     030000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input =             01000000000000000000000000000000
                            02000000000000000000000000000000
                            08000000000000008000000000000000
POLYVAL result =            79745ab508622c8a958543675fac4688
POLYVAL result XOR nonce =  7a745ab508622c8a958543675fac4688
... and masked =            7a745ab508622c8a958543675fac4608
Tag =                       8f8936ec039e4e4bb97ebd8c4457441f
Initial counter =           8f8936ec039e4e4bb97ebd8c4457449f
Result (32 bytes) =         e2b0c5da79a901c1745f700525cb335b
                            8f8936ec039e4e4bb97ebd8c4457441f


Plaintext (32 bytes) =      02000000000000000000000000000000
                            03000000000000000000000000000000
AAD (1 bytes) =             01
Key =                       01000000000000000000000000000000
Nonce =                     030000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input =             01000000000000000000000000000000
                            02000000000000000000000000000000
                            03000000000000000000000000000000
                            08000000000000000001000000000000
POLYVAL result =            2ce7daaf7c89490822051255b12eca6b
POLYVAL result XOR nonce =  2fe7daaf7c89490822051255b12eca6b
... and masked =            2fe7daaf7c89490822051255b12eca6b
Tag =                       e6af6a7f87287da059a71684ed3498e1
Initial counter =           e6af6a7f87287da059a71684ed3498e1
```

```
   Result (48 bytes) =         620048ef3c1e73e57e02bb8562c416a3
                               19e73e4caac8e96a1ecb2933145a1d71
                               e6af6a7f87287da059a71684ed3498e1


   Plaintext (48 bytes) =      020000000000000000000000000000000
                               030000000000000000000000000000000
                               040000000000000000000000000000000
   AAD (1 bytes) =             01
   Key =                       010000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             010000000000000000000000000000000
                               020000000000000000000000000000000
                               030000000000000000000000000000000
                               040000000000000000000000000000000
                               080000000000000080010000000000000
   POLYVAL result =            9ca987715d69c1786711dfcd22f830fc
   POLYVAL result XOR nonce =  9fa987715d69c1786711dfcd22f830fc
   ... and masked =            9fa987715d69c1786711dfcd22f8307c
   Tag =                       6a8cc3865f76897c2e4b245cf31c51f2
   Initial counter =           6a8cc3865f76897c2e4b245cf31c51f2
   Result (64 bytes) =         50c8303ea93925d64090d07bd109dfd9
                               515a5a33431019c17d93465999a8b005
                               3201d723120a8562b838cdff25bf9d1e
                               6a8cc3865f76897c2e4b245cf31c51f2


   Plaintext (64 bytes) =      020000000000000000000000000000000
                               030000000000000000000000000000000
                               040000000000000000000000000000000
                               050000000000000000000000000000000
   AAD (1 bytes) =             01
   Key =                       010000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             010000000000000000000000000000000
                               020000000000000000000000000000000
                               030000000000000000000000000000000
                               040000000000000000000000000000000
                               050000000000000000000000000000000
                               080000000000000000020000000000000
   POLYVAL result =            ffcd05d5770f34ad9267f0a59994b15a
   POLYVAL result XOR nonce =  fccd05d5770f34ad9267f0a59994b15a
   ... and masked =            fccd05d5770f34ad9267f0a59994b15a
   Tag =                       cdc46ae475563de037001ef84ae21744
```

```
   Initial counter =           cdc46ae475563de037001ef84ae217c4
   Result (80 bytes) =         2f5c64059db55ee0fb847ed513003746
                               aca4e61c711b5de2e7a77ffd02da42fe
                               ec601910d3467bb8b36ebbaebce5fba3
                               0d36c95f48a3e7980f0e7ac299332a80
                               cdc46ae475563de037001ef84ae21744


   Plaintext (4 bytes) =       02000000
   AAD (12 bytes) =            010000000000000000000000
   Key =                       01000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               60000000000000002000000000000000
   POLYVAL result =            f6ce9d3dcd68a2fd603c7ecc18fb9918
   POLYVAL result XOR nonce =  f5ce9d3dcd68a2fd603c7ecc18fb9918
   ... and masked =            f5ce9d3dcd68a2fd603c7ecc18fb9918
   Tag =                       07eb1f84fb28f8cb73de8e99e2f48a14
   Initial counter =           07eb1f84fb28f8cb73de8e99e2f48a94
   Result (20 bytes) =         a8fe3e8707eb1f84fb28f8cb73de8e99
                               e2f48a14


   Plaintext (20 bytes) =      03000000000000000000000000000000
                               04000000
   AAD (18 bytes) =            01000000000000000000000000000000
                               0200
   Key =                       01000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
                               90000000000000a000000000000000
   POLYVAL result =            4781d492cb8f926c504caa36f61008fe
   POLYVAL result XOR nonce =  4481d492cb8f926c504caa36f61008fe
   ... and masked =            4481d492cb8f926c504caa36f610087e
   Tag =                       24afc9805e976f451e6d87f6fe106514
   Initial counter =           24afc9805e976f451e6d87f6fe106594
   Result (36 bytes) =         6bb0fecf5ded9b77f902c7d5da236a43
                               91dd029724afc9805e976f451e6d87f6
                               fe106514
```

```
   Plaintext (18 bytes) =      030000000000000000000000000000000000
                               0400
   AAD (20 bytes) =            010000000000000000000000000000000000
                               02000000
   Key =                       010000000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = d9b360279694941ac5dbc6987ada7377
   Record encryption key =     4004a0dcd862f2a57360219d2d44ef6c
   POLYVAL input =             010000000000000000000000000000000000
                               020000000000000000000000000000000000
                               030000000000000000000000000000000000
                               040000000000000000000000000000000000
                               a000000000000009000000000000000000
   POLYVAL result =            75cbc23a1a10e348aeb8e384b5cc79fd
   POLYVAL result XOR nonce =  76cbc23a1a10e348aeb8e384b5cc79fd
   ... and masked =            76cbc23a1a10e348aeb8e384b5cc797d
   Tag =                       bff9b2ef00fb47920cc72a0c0f13b9fd
   Initial counter =           bff9b2ef00fb47920cc72a0c0f13b9fd
   Result (34 bytes) =         44d0aaf6fb2f1f34add5e8064e83e12a
                               2adabff9b2ef00fb47920cc72a0c0f13
                               b9fd

   Plaintext (0 bytes) =
   AAD (0 bytes) =
   Key =                       e66021d5eb8e4f4066d4adb9c33560e4
   Nonce =                     f46e44bb3da0015c94f70887
   Record authentication key = 036ee1fe2d7926af68898095e54e7b3c
   Record encryption key =     5e46482396008223b5c1d25173d87539
   POLYVAL input =             000000000000000000000000000000000
   POLYVAL result =            000000000000000000000000000000000
   POLYVAL result XOR nonce =  f46e44bb3da0015c94f7088700000000
   ... and masked =            f46e44bb3da0015c94f7088700000000
   Tag =                       a4194b79071b01a87d65f706e3949578
   Initial counter =           a4194b79071b01a87d65f706e39495f8
   Result (16 bytes) =         a4194b79071b01a87d65f706e3949578


   Plaintext (3 bytes) =       7a806c
   AAD (5 bytes) =             46bb91c3c5
   Key =                       36864200e0eaf5284d884a0e77d31646
   Nonce =                     bae8e37fc83441b16034566b
   Record authentication key = 3e28de1120b2981a0155795ca2812af6
   Record encryption key =     6d4b78b31a4c9c03d8db0f42f7507fae
   POLYVAL input =             46bb91c3c500000000000000000000000
                               7a806c000000000000000000000000000
                               280000000000000180000000000000000
   POLYVAL result =            43d9a745511dcfa21b96dd606f1d5720
   POLYVAL result XOR nonce =  f931443a99298e137ba28b0b6f1d5720
```

```
   ... and masked =            f931443a99298e137ba28b0b6f1d5720
   Tag =                       711bd85bc1e4d3e0a462e074eea428a8
   Initial counter =           711bd85bc1e4d3e0a462e074eea428a8
   Result (19 bytes) =         af60eb711bd85bc1e4d3e0a462e074ee
                               a428a8


   Plaintext (6 bytes) =       bdc66f146545
   AAD (10 bytes) =            fc880c94a95198874296
   Key =                       aedb64a6c590bc84d1a5e269e4b47801
   Nonce =                     afc0577e34699b9e671fdd4f
   Record authentication key = 43b8de9cea62330d15cccfc84a33e8c8
   Record encryption key =     8e54631607e431e095b54852868e3a27
   POLYVAL input =             fc880c94a9519887429600000000000
                               bdc66f14654500000000000000000000
                               5000000000000000300000000000000
   POLYVAL result =            26498e0d2b1ef004e808c458e8f2f515
   POLYVAL result XOR nonce =  8989d9731f776b9a8f171917e8f2f515
   ... and masked =            8989d9731f776b9a8f171917e8f2f515
   Tag =                       d6a9c45545cfc11f03ad743dba20f966
   Initial counter =           d6a9c45545cfc11f03ad743dba20f9e6
   Result (22 bytes) =         bb93a3e34d3cd6a9c45545cfc11f03ad
                               743dba20f966


   Plaintext (9 bytes) =       1177441f195495860f
   AAD (15 bytes) =            046787f3ea22c127aaf195d1894728
   Key =                       d5cc1fd161320b6920ce07787f86743b
   Nonce =                     275d1ab32f6d1f0434d8848c
   Record authentication key = 8a51df64d93eaf667c2c09bd454ce5c5
   Record encryption key =     43ab276c2b4a473918ca73f2dd85109c
   POLYVAL input =             046787f3ea22c127aaf195d189472800
                               1177441f195495860f00000000000000
                               7800000000000000480000000000000
   POLYVAL result =            63a3451c0b23345ad02bba59956517cf
   POLYVAL result XOR nonce =  44fe5faf244e2b5ee4f33ed5956517cf
   ... and masked =            44fe5faf244e2b5ee4f33ed59565174f
   Tag =                       1d02fd0cd174c84fc5dae2f60f52fd2b
   Initial counter =           1d02fd0cd174c84fc5dae2f60f52fdab
   Result (25 bytes) =         4f37281f7ad12949d01d02fd0cd174c8
                               4fc5dae2f60f52fd2b


   Plaintext (12 bytes) =      9f572c614b4745914474e7c7
   AAD (20 bytes) =            c9882e5386fd9f92ec489c8fde2be2cf
                               97e74e93
   Key =                       b3fed1473c528b8426a582995929a149
   Nonce =                     9e9ad8780c8d63d0ab4149c0
```

```
Record authentication key = 22f50707a95dd416df069d670cb775e8
Record encryption key =     f674a5584ee21fe97b4cebc468ab61e4
POLYVAL input =             c9882e5386fd9f92ec489c8fde2be2cf
                            97e74e9300000000000000000000000
                            9f572c614b4745914474e7c700000000
                            a000000000000000600000000000000
POLYVAL result =            0cca0423fba9d77fe7e2e6963b08cdd0
POLYVAL result XOR nonce =  9250dc5bf724b4af4ca3af563b08cdd0
... and masked =            9250dc5bf724b4af4ca3af563b08cd50
Tag =                       c1dc2f871fb7561da1286e655e24b7b0
Initial counter =           c1dc2f871fb7561da1286e655e24b7b0
Result (28 bytes) =         f54673c5ddf710c745641c8bc1dc2f87
                            1fb7561da1286e655e24b7b0


Plaintext (15 bytes) =      0d8c8451178082355c9e940fea2f58
AAD (25 bytes) =            2950a70d5a1db2316fd568378da107b5
                            2b0da55210cc1c1b0a
Key =                       2d4ed87da44102952ef94b02b805249b
Nonce =                     ac80e6f61455bfac8308a2d4
Record authentication key = 0b00a29a83e7e95b92e3a0783b29f140
Record encryption key =     a430c27f285aed913005975c42eed5f3
POLYVAL input =             2950a70d5a1db2316fd568378da107b5
                            2b0da55210cc1c1b0a00000000000000
                            0d8c8451178082355c9e940fea2f5800
                            c800000000000000780000000000000
POLYVAL result =            1086ef25247aa41009bbc40871d9b350
POLYVAL result XOR nonce =  bc0609d3302f1bbc8ab366dc71d9b350
... and masked =            bc0609d3302f1bbc8ab366dc71d9b350
Tag =                       83b3449b9f39552de99dc214a1190b0b
Initial counter =           83b3449b9f39552de99dc214a1190b8b
Result (31 bytes) =         c9ff545e07b88a015f05b274540aa183
                            b3449b9f39552de99dc214a1190b0b


Plaintext (18 bytes) =      6b3db4da3d57aa94842b9803a96e07fb
                            6de7
AAD (30 bytes) =            1860f762ebfbd08284e421702de0de18
                            baa9c9596291b08466f37de21c7f
Key =                       bde3b2f204d1e9f8b06bc47f9745b3d1
Nonce =                     ae06556fb6aa7890bebc18fe
Record authentication key = 21c874a8bad3603d1c3e8784df5b3f9f
Record encryption key =     d1c16d72651c3df504eae27129d818e8
POLYVAL input =             1860f762ebfbd08284e421702de0de18
                            baa9c9596291b08466f37de21c7f0000
                            6b3db4da3d57aa94842b9803a96e07fb
                            6de7000000000000000000000000000
                            f000000000000000900000000000000
```

```
   POLYVAL result =            55462a5afa0da8d646481e049ef9c764
   POLYVAL result XOR nonce =  fb407f354ca7d046f8f406fa9ef9c764
   ... and masked =            fb407f354ca7d046f8f406fa9ef9c764
   Tag =                       3e377094f04709f64d7b985310a4db84
   Initial counter =           3e377094f04709f64d7b985310a4db84
   Result (34 bytes) =         6298b296e24e8cc35dce0bed484b7f30
                               d5803e377094f04709f64d7b985310a4
                               db84


   Plaintext (21 bytes) =      e42a3c02c25b64869e146d7b233987bd
                               dfc240871d
   AAD (35 bytes) =            7576f7028ec6eb5ea7e298342a94d4b2
                               02b370ef9768ec6561c4fe6b7e7296fa
                               859c21
   Key =                       f901cfe8a69615a93fdf7a98cad48179
   Nonce =                     6245709fb18853f68d833640
   Record authentication key = 3724f55f1d22ac0ab830da0b6a995d74
   Record encryption key =     75ac87b70c05db287de779006105a344
   POLYVAL input =             7576f7028ec6eb5ea7e298342a94d4b2
                               02b370ef9768ec6561c4fe6b7e7296fa
                               859c210000000000000000000000000
                               e42a3c02c25b64869e146d7b233987bd
                               dfc240871d000000000000000000000
                               180100000000000a800000000000000
   POLYVAL result =            4cbba090f03f7d1188ea55749fa6c7bd
   POLYVAL result XOR nonce =  2efed00f41b72ee7056963349fa6c7bd
   ... and masked =            2efed00f41b72ee7056963349fa6c73d
   Tag =                       2d15506c84a9edd65e13e9d24a2a6e70
   Initial counter =           2d15506c84a9edd65e13e9d24a2a6ef0
   Result (37 bytes) =         391cc328d484a4f46406181bcd62efd9
                               b3ee197d052d15506c84a9edd65e13e9
                               d24a2a6e70
```

C.2. AEAD_AES_256_GCM_SIV

```
   Plaintext (0 bytes) =
   AAD (0 bytes) =
   Key =                       010000000000000000000000000000000
                               000000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             000000000000000000000000000000000
   POLYVAL result =            000000000000000000000000000000000
   POLYVAL result XOR nonce =  030000000000000000000000000000000
   ... and masked =            030000000000000000000000000000000
```

```
   Tag =                          07f5f4169bbf55a8400cd47ea6fd400f
   Initial counter =              07f5f4169bbf55a8400cd47ea6fd408f
   Result (16 bytes) =            07f5f4169bbf55a8400cd47ea6fd400f


   Plaintext (8 bytes) =          0100000000000000
   AAD (0 bytes) =
   Key =                          0100000000000000000000000000000000
                                  0000000000000000000000000000000000
   Nonce =                        030000000000000000000000
   Record authentication key =    b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =        b914f4742be9e1d7a2f84addbf96dec3
                                  456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =                0100000000000000000000000000000000
                                  0000000000000000400000000000000000
   POLYVAL result =               05230f62f0eac8aa14fe4d646b59cd41
   POLYVAL result XOR nonce =     06230f62f0eac8aa14fe4d646b59cd41
   ... and masked =               06230f62f0eac8aa14fe4d646b59cd41
   Tag =                          843122130f7364b761e0b97427e3df28
   Initial counter =              843122130f7364b761e0b97427e3dfa8
   Result (24 bytes) =            c2ef328e5c71c83b843122130f7364b7
                                  61e0b97427e3df28


   Plaintext (12 bytes) =         010000000000000000000000
   AAD (0 bytes) =
   Key =                          0100000000000000000000000000000000
                                  0000000000000000000000000000000000
   Nonce =                        030000000000000000000000
   Record authentication key =    b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =        b914f4742be9e1d7a2f84addbf96dec3
                                  456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =                0100000000000000000000000000000000
                                  0000000000000000600000000000000000
   POLYVAL result =               6d81a24732fd6d03ae5af544720a1c13
   POLYVAL result XOR nonce =     6e81a24732fd6d03ae5af544720a1c13
   ... and masked =               6e81a24732fd6d03ae5af544720a1c13
   Tag =                          8ca50da9ae6559e48fd10f6e5c9ca17e
   Initial counter =              8ca50da9ae6559e48fd10f6e5c9ca1fe
   Result (28 bytes) =            9aab2aeb3faa0a34aea8e2b18ca50da9
                                  ae6559e48fd10f6e5c9ca17e


   Plaintext (16 bytes) =         01000000000000000000000000000000
   AAD (0 bytes) =
   Key =                          0100000000000000000000000000000000
                                  0000000000000000000000000000000000
   Nonce =                        030000000000000000000000
```

```
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               00000000000000008000000000000000
   POLYVAL result =            74eee2bf7c9a165f8b25dea73db32a6d
   POLYVAL result XOR nonce =  77eee2bf7c9a165f8b25dea73db32a6d
   ... and masked =            77eee2bf7c9a165f8b25dea73db32a6d
   Tag =                       c9eac6fa700942702e90862383c6c366
   Initial counter =           c9eac6fa700942702e90862383c6c3e6
   Result (32 bytes) =         85a01b63025ba19b7fd3ddfc033b3e76
                               c9eac6fa700942702e90862383c6c366


   Plaintext (32 bytes) =      01000000000000000000000000000000
                               02000000000000000000000000000000
   AAD (0 bytes) =
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               00000000000000000001000000000000
   POLYVAL result =            899b6381b3d46f0def7aa0517ba188f5
   POLYVAL result XOR nonce =  8a9b6381b3d46f0def7aa0517ba188f5
   ... and masked =            8a9b6381b3d46f0def7aa0517ba18875
   Tag =                       e819e63abcd020b006a976397632eb5d
   Initial counter =           e819e63abcd020b006a976397632ebdd
   Result (48 bytes) =         4a6a9db4c8c6549201b9edb53006cba8
                               21ec9cf850948a7c86c68ac7539d027f
                               e819e63abcd020b006a976397632eb5d


   Plaintext (48 bytes) =      01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
   AAD (0 bytes) =
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
```

```
                              03000000000000000000000000000000
                              00000000000000008001000000000000
   POLYVAL result =           c1f8593d8fc29b0c290cae1992f71f51
   POLYVAL result XOR nonce = c2f8593d8fc29b0c290cae1992f71f51
   ... and masked =           c2f8593d8fc29b0c290cae1992f71f51
   Tag =                      790bc96880a99ba804bd12c0e6a22cc4
   Initial counter =          790bc96880a99ba804bd12c0e6a22cc4
   Result (64 bytes) =        c00d121893a9fa603f48ccc1ca3c57ce
                              7499245ea0046db16c53c7c66fe717e3
                              9cf6c748837b61f6ee3adcee17534ed5
                              790bc96880a99ba804bd12c0e6a22cc4


   Plaintext (64 bytes) =     01000000000000000000000000000000
                              02000000000000000000000000000000
                              03000000000000000000000000000000
                              04000000000000000000000000000000
   AAD (0 bytes) =
   Key =                      01000000000000000000000000000000
                              00000000000000000000000000000000
   Nonce =                    030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =    b914f4742be9e1d7a2f84addbf96dec3
                              456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =            01000000000000000000000000000000
                              02000000000000000000000000000000
                              03000000000000000000000000000000
                              04000000000000000000000000000000
                              00000000000000000002000000000000
   POLYVAL result =           6ef38b06046c7c0e225efaef8e2ec4c4
   POLYVAL result XOR nonce = 6df38b06046c7c0e225efaef8e2ec4c4
   ... and masked =           6df38b06046c7c0e225efaef8e2ec444
   Tag =                      112864c269fc0d9d88c61fa47e39aa08
   Initial counter =          112864c269fc0d9d88c61fa47e39aa88
   Result (80 bytes) =        c2d5160a1f8683834910acdafc41fbb1
                              632d4a353e8b905ec9a5499ac34f96c7
                              e1049eb080883891a4db8caaa1f99dd0
                              04d80487540735234e3744512c6f90ce
                              112864c269fc0d9d88c61fa47e39aa08


   Plaintext (8 bytes) =      0200000000000000
   AAD (1 bytes) =            01
   Key =                      01000000000000000000000000000000
                              00000000000000000000000000000000
   Nonce =                    030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =    b914f4742be9e1d7a2f84addbf96dec3
```

```
                              456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =            01000000000000000000000000000000
                              02000000000000000000000000000000
                              08000000000000004000000000000000
   POLYVAL result =           34e57bafe011b9b36fc6821b7ffb3354
   POLYVAL result XOR nonce = 37e57bafe011b9b36fc6821b7ffb3354
   ... and masked =           37e57bafe011b9b36fc6821b7ffb3354
   Tag =                      91213f267e3b452f02d01ae33e4ec854
   Initial counter =          91213f267e3b452f02d01ae33e4ec8d4
   Result (24 bytes) =        1de22967237a813291213f267e3b452f
                              02d01ae33e4ec854


   Plaintext (12 bytes) =     020000000000000000000000
   AAD (1 bytes) =            01
   Key =                      01000000000000000000000000000000
                              00000000000000000000000000000000

   Nonce =                    030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =    b914f4742be9e1d7a2f84addbf96dec3
                              456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =            01000000000000000000000000000000
                              02000000000000000000000000000000
                              08000000000000006000000000000000
   POLYVAL result =           5c47d68a22061c1ad5623a3b66a8e206
   POLYVAL result XOR nonce = 5f47d68a22061c1ad5623a3b66a8e206
   ... and masked =           5f47d68a22061c1ad5623a3b66a8e206
   Tag =                      c1a4a19ae800941ccdc57cc8413c277f
   Initial counter =          c1a4a19ae800941ccdc57cc8413c27ff
   Result (28 bytes) =        163d6f9cc1b346cd453a2e4cc1a4a19a
                              e800941ccdc57cc8413c277f


   Plaintext (16 bytes) =     02000000000000000000000000000000
   AAD (1 bytes) =            01
   Key =                      01000000000000000000000000000000
                              00000000000000000000000000000000

   Nonce =                    030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =    b914f4742be9e1d7a2f84addbf96dec3
                              456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =            01000000000000000000000000000000
                              02000000000000000000000000000000
                              08000000000000008000000000000000
   POLYVAL result =           452896726c616746f01d11d82911d478
   POLYVAL result XOR nonce = 462896726c616746f01d11d82911d478
   ... and masked =           462896726c616746f01d11d82911d478
   Tag =                      b292d28ff61189e8e49f3875ef91aff7
```

```
   Initial counter =           b292d28ff61189e8e49f3875ef91aff7
   Result (32 bytes) =         c91545823cc24f17dbb0e9e807d5ec17
                               b292d28ff61189e8e49f3875ef91aff7


   Plaintext (32 bytes) =      02000000000000000000000000000000
                               03000000000000000000000000000000
   AAD (1 bytes) =             01
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               08000000000000000001000000000000
   POLYVAL result =            4e58c1e341c9bb0ae34eda9509dfc90c
   POLYVAL result XOR nonce =  4d58c1e341c9bb0ae34eda9509dfc90c
   ... and masked =            4d58c1e341c9bb0ae34eda9509dfc90c
   Tag =                       aea1bad12702e1965604374aab96dbbc
   Initial counter =           aea1bad12702e1965604374aab96dbbc
   Result (48 bytes) =         07dad364bfc2b9da89116d7bef6daaaf
                               6f255510aa654f920ac81b94e8bad365
                               aea1bad12702e1965604374aab96dbbc


   Plaintext (48 bytes) =      02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
   AAD (1 bytes) =             01
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000
   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
                               08000000000000000800100000000000
   POLYVAL result =            2566a4aff9a525df9772c16d4eaf8d2a
   POLYVAL result XOR nonce =  2666a4aff9a525df9772c16d4eaf8d2a
   ... and masked =            2666a4aff9a525df9772c16d4eaf8d2a
   Tag =                       03332742b228c647173616cfd44c54eb
   Initial counter =           03332742b228c647173616cfd44c54eb
```

```
   Result (64 bytes) =          c67a1f0f567a5198aa1fcc8e3f213143
                                36f7f51ca8b1af61feac35a86416fa47
                                fbca3b5f749cdf564527f2314f42fe25
                                03332742b228c647173616cfd44c54eb


   Plaintext (64 bytes) =       02000000000000000000000000000000
                                03000000000000000000000000000000
                                04000000000000000000000000000000
                                05000000000000000000000000000000
   AAD (1 bytes) =              01
   Key =                        01000000000000000000000000000000
                                00000000000000000000000000000000
   Nonce =                      030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =      b914f4742be9e1d7a2f84addbf96dec3
                                456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =              01000000000000000000000000000000
                                02000000000000000000000000000000
                                03000000000000000000000000000000
                                04000000000000000000000000000000
                                05000000000000000000000000000000
                                08000000000000000002000000000000
   POLYVAL result =             da58d2f61b0a9d343b2f37fb0c519733
   POLYVAL result XOR nonce =   d958d2f61b0a9d343b2f37fb0c519733
   ... and masked =             d958d2f61b0a9d343b2f37fb0c519733
   Tag =                        5bde0285037c5de81e5b570a049b62a0
   Initial counter =            5bde0285037c5de81e5b570a049b62a0
   Result (80 bytes) =          67fd45e126bfb9a79930c43aad2d3696
                                7d3f0e4d217c1e551f59727870beefc9
                                8cb933a8fce9de887b1e40799988db1f
                                c3f91880ed405b2dd298318858467c89
                                5bde0285037c5de81e5b570a049b62a0


   Plaintext (4 bytes) =        02000000
   AAD (12 bytes) =             010000000000000000000000
   Key =                        01000000000000000000000000000000
                                00000000000000000000000000000000
   Nonce =                      030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =      b914f4742be9e1d7a2f84addbf96dec3
                                456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =              01000000000000000000000000000000
                                02000000000000000000000000000000
                                60000000000000002000000000000000
   POLYVAL result =             6dc76ae84b88916e073a303aafde05cf
   POLYVAL result XOR nonce =   6ec76ae84b88916e073a303aafde05cf
```

```
   ... and masked =            6ec76ae84b88916e073a303aafde054f
   Tag =                       1835e517741dfddccfa07fa4661b74cf
   Initial counter =           1835e517741dfddccfa07fa4661b74cf
   Result (20 bytes) =         22b3f4cd1835e517741dfddccfa07fa4
                               661b74cf


   Plaintext (20 bytes) =      03000000000000000000000000000000
                               04000000
   AAD (18 bytes) =            01000000000000000000000000000000
                               0200
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000

   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
                               9000000000000000a000000000000000
   POLYVAL result =            973ef4fd04bd31d193816ab26f8655ca
   POLYVAL result XOR nonce =  943ef4fd04bd31d193816ab26f8655ca
   ... and masked =            943ef4fd04bd31d193816ab26f86554a
   Tag =                       b879ad976d8242acc188ab59cabfe307
   Initial counter =           b879ad976d8242acc188ab59cabfe387
   Result (36 bytes) =         43dd0163cdb48f9fe3212bf61b201976
                               067f342bb879ad976d8242acc188ab59
                               cabfe307


   Plaintext (18 bytes) =      03000000000000000000000000000000
                               0400
   AAD (20 bytes) =            01000000000000000000000000000000
                               02000000
   Key =                       01000000000000000000000000000000
                               00000000000000000000000000000000

   Nonce =                     030000000000000000000000
   Record authentication key = b5d3c529dfafac43136d2d11be284d7f
   Record encryption key =     b914f4742be9e1d7a2f84addbf96dec3
                               456e3c6c05ecc157cdbf0700fedad222
   POLYVAL input =             01000000000000000000000000000000
                               02000000000000000000000000000000
                               03000000000000000000000000000000
                               04000000000000000000000000000000
                               a000000000000009000000000000000
   POLYVAL result =            2cbb6b7ab2dbffefb797f825f826870c
```

```
   POLYVAL result XOR nonce =  2fbb6b7ab2dbffefb797f825f826870c
   ... and masked =            2fbb6b7ab2dbffefb797f825f826870c
   Tag =                       cfcdf5042112aa29685c912fc2056543
   Initial counter =           cfcdf5042112aa29685c912fc20565c3
   Result (34 bytes) =         462401724b5ce6588d5a54aae5375513
                               a075cfcdf5042112aa29685c912fc205
                               6543

   Plaintext (0 bytes) =
   AAD (0 bytes) =
   Key =                       e66021d5eb8e4f4066d4adb9c33560e4
                               f46e44bb3da0015c94f7088736864200
   Nonce =                     e0eaf5284d884a0e77d31646
   Record authentication key = e40d26f82774aa27f47b047b608b9585
   Record encryption key =     7c7c3d9a542cef53dde0e6de9b580040
                               0f82e73ec5f7ee41b7ba8dcb9ba078c3
   POLYVAL input =             00000000000000000000000000000000
   POLYVAL result =            00000000000000000000000000000000
   POLYVAL result XOR nonce =  e0eaf5284d884a0e77d3164600000000
   ... and masked =            e0eaf5284d884a0e77d3164600000000
   Tag =                       169fbb2fbf389a995f6390af22228a62
   Initial counter =           169fbb2fbf389a995f6390af22228ae2
   Result (16 bytes) =         169fbb2fbf389a995f6390af22228a62


   Plaintext (3 bytes) =       671fdd
   AAD (5 bytes) =             4fbdc66f14
   Key =                       bae8e37fc83441b16034566b7a806c46
                               bb91c3c5aedb64a6c590bc84d1a5e269
   Nonce =                     e4b47801afc0577e34699b9e
   Record authentication key = b546f5a850d0a90adfe39e95c2510fc6
   Record encryption key =     b9d1e239d62cbb5c49273ddac8838bdc
                               c53bca478a770f07087caa4e0a924a55
   POLYVAL input =             4fbdc66f14000000000000000000000000
                               671fdd000000000000000000000000000
                               280000000000000018000000000000000
   POLYVAL result =            b91f91f96b159a7c611c05035b839e92
   POLYVAL result XOR nonce =  5dabe9f8c4d5cd0255759e9d5b839e92
   ... and masked =            5dabe9f8c4d5cd0255759e9d5b839e12
   Tag =                       93da9bb81333aee0c785b240d319719d
   Initial counter =           93da9bb81333aee0c785b240d319719d
   Result (19 bytes) =         0eaccb93da9bb81333aee0c785b240d3
                               19719d


   Plaintext (6 bytes) =       195495860f04
   AAD (10 bytes) =            6787f3ea22c127aaf195
   Key =                       6545fc880c94a95198874296d5cc1fd1
```

```
                          61320b6920ce07787f86743b275d1ab3
   Nonce =                2f6d1f0434d8848c1177441f
   Record authentication key = e156e1f9b0b07b780cbe30f259e3c8da
   Record encryption key =    6fc1c494519f944aae52fcd8b14e5b17
                          1b5a9429d3b76e430d49940c0021d612
   POLYVAL input =            6787f3ea22c127aaf195000000000000
                          195495860f040000000000000000000000
                          500000000000000030000000000000000
   POLYVAL result =           2c480ed9d236b1df24c6eec109bd40c1
   POLYVAL result XOR nonce = 032511dde6ee355335b1aade09bd40c1
   ... and masked =           032511dde6ee355335b1aade09bd4041
   Tag =                      6b62b84dc40c84636a5ec12020ec8c2c
   Initial counter =          6b62b84dc40c84636a5ec12020ec8cac
   Result (22 bytes) =        a254dad4f3f96b62b84dc40c84636a5e
                          c12020ec8c2c


   Plaintext (9 bytes) =      c9882e5386fd9f92ec
   AAD (15 bytes) =           489c8fde2be2cf97e74e932d4ed87d
   Key =                      d1894728b3fed1473c528b8426a58299
                          5929a1499e9ad8780c8d63d0ab4149c0
   Nonce =                    9f572c614b4745914474e7c7
   Record authentication key = 0533fd71f4119257361a3ff1469dd4e5
   Record encryption key =    4feba89799be8ac3684fa2bb30ade0ea
                          51390e6d87dcf3627d2ee44493853abe
   POLYVAL input =            489c8fde2be2cf97e74e932d4ed87d00
                          c9882e5386fd9f92ec00000000000000
                          780000000000000004800000000000000
   POLYVAL result =           bf160bc9ded8c63057d2c38aae552fb4
   POLYVAL result XOR nonce = 204127a8959f83a113a6244dae552fb4
   ... and masked =           204127a8959f83a113a6244dae552f34
   Tag =                      c0fd3dc6628dfe55ebb0b9fb2295c8c2
   Initial counter =          c0fd3dc6628dfe55ebb0b9fb2295c8c2
   Result (25 bytes) =        0df9e308678244c44bc0fd3dc6628dfe
                          55ebb0b9fb2295c8c2


   Plaintext (12 bytes) =     1db2316fd568378da107b52b
   AAD (20 bytes) =           0da55210cc1c1b0abde3b2f204d1e9f8
                          b06bc47f
   Key =                      a44102952ef94b02b805249bac80e6f6
                          1455bfac8308a2d40d8c845117808235
   Nonce =                    5c9e940fea2f582950a70d5a
   Record authentication key = 64779ab10ee8a280272f14cc8851b727
   Record encryption key =    25f40fc63f49d3b9016a8eeeb75846e0
                          d72ca36ddbd312b6f5ef38ad14bd2651
   POLYVAL input =            0da55210cc1c1b0abde3b2f204d1e9f8
                          b06bc47f000000000000000000000000
```

```
                              1db2316fd568378da107b52b00000000
                              a000000000000000600000000000000
   POLYVAL result =           cc86ee22c861e1fd474c84676b42739c
   POLYVAL result XOR nonce = 90187a2d224eb9d417eb893d6b42739c
   ... and masked =           90187a2d224eb9d417eb893d6b42731c
   Tag =                      404099c2587f64979f21826706d497d5
   Initial counter =          404099c2587f64979f21826706d497d5
   Result (28 bytes) =        8dbeb9f7255bf5769dd56692404099c2
                              587f64979f21826706d497d5


   Plaintext (15 bytes) =     21702de0de18baa9c9596291b08466
   AAD (25 bytes) =           f37de21c7ff901cfe8a69615a93fdf7a
                              98cad481796245709f
   Key =                      9745b3d1ae06556fb6aa7890bebc18fe
                              6b3db4da3d57aa94842b9803a96e07fb
   Nonce =                    6de71860f762ebfbd08284e4
   Record authentication key = 27c2959ed4daea3b1f52e849478de376
   Record encryption key =    307a38a5a6cf231c0a9af3b527f23a62
                              e9a6ff09aff8ae669f760153e864fc93
   POLYVAL input =            f37de21c7ff901cfe8a69615a93fdf7a
                              98cad481796245709f00000000000000
                              21702de0de18baa9c9596291b0846600
                              c800000000000000780000000000000
   POLYVAL result =           c4fa5e5b713853703bcf8e6424505fa5
   POLYVAL result XOR nonce = a91d463b865ab88beb4d0a8024505fa5
   ... and masked =           a91d463b865ab88beb4d0a8024505f25
   Tag =                      b3080d28f6ebb5d3648ce97bd5ba67fd
   Initial counter =          b3080d28f6ebb5d3648ce97bd5ba67fd
   Result (31 bytes) =        793576dfa5c0f88729a7ed3c2f1bffb3
                              080d28f6ebb5d3648ce97bd5ba67fd


   Plaintext (18 bytes) =     b202b370ef9768ec6561c4fe6b7e7296
                              fa85
   AAD (30 bytes) =           9c2159058b1f0fe91433a5bdc20e214e
                              ab7fecef4454a10ef0657df21ac7
   Key =                      b18853f68d833640e42a3c02c25b6486
                              9e146d7b233987bddfc240871d7576f7
   Nonce =                    028ec6eb5ea7e298342a94d4
   Record authentication key = 670b98154076ddb59b7a9137d0dcc0f0
   Record encryption key =    78116d78507fbe69d4a820c350f55c7c
                              b36c3c9287df0e9614b142b76a587c3f
   POLYVAL input =            9c2159058b1f0fe91433a5bdc20e214e
                              ab7fecef4454a10ef0657df21ac70000
                              b202b370ef9768ec6561c4fe6b7e7296
                              fa850000000000000000000000000000
                              f000000000000000900000000000000
```

```
POLYVAL result =            4e4108f09f41d797dc9256f8da8d58c7
POLYVAL result XOR nonce =  4ccfce1bc1e6350fe8b8c22cda8d58c7
... and masked =            4ccfce1bc1e6350fe8b8c22cda8d5847
Tag =                       454fc2a154fea91f8363a39fec7d0a49
Initial counter =           454fc2a154fea91f8363a39fec7d0ac9
Result (34 bytes) =         857e16a64915a787637687db4a951963
                            5cdd454fc2a154fea91f8363a39fec7d
                            0a49


Plaintext (21 bytes) =      ced532ce4159b035277d4dfbb7db6296
                            8b13cd4eec
AAD (35 bytes) =            734320ccc9d9bbbb19cb81b2af4ecbc3
                            e72834321f7aa0f70b7282b4f33df23f
                            167541
Key =                       3c535de192eaed3822a2fbbe2ca9dfc8
                            8255e14a661b8aa82cc54236093bbc23
Nonce =                     688089e55540db1872504e1c
Record authentication key = cb8c3aa3f8dbaeb4b28a3e86ff6625f8
Record encryption key =     02426ce1aa3ab31313b0848469a1b5fc
                            6c9af9602600b195b04ad407026bc06d
POLYVAL input =             734320ccc9d9bbbb19cb81b2af4ecbc3
                            e72834321f7aa0f70b7282b4f33df23f
                            16754100000000000000000000000000
                            ced532ce4159b035277d4dfbb7db6296
                            8b13cd4eec00000000000000000000000
                            18010000000000000a800000000000000
POLYVAL result =            ffd503c7dd712eb3791b7114b17bb0cf
POLYVAL result XOR nonce =  97558a228831f5ab0b4b3f08b17bb0cf
... and masked =            97558a228831f5ab0b4b3f08b17bb04f
Tag =                       9d6c7029675b89eaf4ba1ded1a286594
Initial counter =           9d6c7029675b89eaf4ba1ded1a286594
Result (37 bytes) =         626660c26ea6612fb17ad91e8e767639
                            edd6c9faee9d6c7029675b89eaf4ba1d
                            ed1a286594
```

C.3. Counter wrap tests

The tests in this section use AEAD_AES_256_GCM_SIV and are crafted to test correct wrapping of the block counter.

```
   Plaintext (32 bytes) =      0000000000000000000000000000000000
                               4db923dc793ee6497c76dcc03a98e108
   AAD (0 bytes) =
   Key =                       0000000000000000000000000000000000
                               0000000000000000000000000000000000
   Nonce =                     000000000000000000000000
   Record authentication key = dc95c078a24089895275f3d86b4fb868
   Record encryption key =     779b38d15bffb63d39d6e9ae76a9b2f3
                               75d11b0e3a68c422845c7d4690fa594f
   POLYVAL input =             0000000000000000000000000000000000
                               4db923dc793ee6497c76dcc03a98e108
                               0000000000000000000010000000000000
   POLYVAL result =            7367cdb411b730128dd56e8edc0eff56
   POLYVAL result XOR nonce =  7367cdb411b730128dd56e8edc0eff56
   ... and masked =            7367cdb411b730128dd56e8edc0eff56
   Tag =                       ffffffff0000000000000000000000000
   Initial counter =           ffffffff0000000000000000000000080
   Result (48 bytes) =         f3f80f2cf0cb2dd9c5984fcda908456c
                               c537703b5ba70324a6793a7bf218d3ea
                               ffffffff0000000000000000000000000


   Plaintext (24 bytes) =      eb3640277c7ffd1303c7a542d02d3e4c
                               0000000000000000
   AAD (0 bytes) =
   Key =                       0000000000000000000000000000000000
                               0000000000000000000000000000000000
   Nonce =                     000000000000000000000000
   Record authentication key = dc95c078a24089895275f3d86b4fb868
   Record encryption key =     779b38d15bffb63d39d6e9ae76a9b2f3
                               75d11b0e3a68c422845c7d4690fa594f
   POLYVAL input =             eb3640277c7ffd1303c7a542d02d3e4c
                               0000000000000000000000000000000000
                               0000000000000000c000000000000000
   POLYVAL result =            7367cdb411b730128dd56e8edc0eff56
   POLYVAL result XOR nonce =  7367cdb411b730128dd56e8edc0eff56
   ... and masked =            7367cdb411b730128dd56e8edc0eff56
   Tag =                       ffffffff0000000000000000000000000
   Initial counter =           ffffffff0000000000000000000000080
   Result (40 bytes) =         18ce4f0b8cb4d0cac65fea8f79257b20
                               888e53e72299e56dffffffff00000000
                               0000000000000000
```

Authors' Addresses

Shay Gueron
University of Haifa and Amazon Web Services
Abba Khoushy Ave 199
Haifa 3498838
Israel

Email: shay@math.haifa.ac.il

Adam Langley
Google LLC
345 Spear St
San Francisco, CA 94105
US

Email: agl@google.com

Yehuda Lindell
Bar Ilan University
Ramat Gan
5290002
Israel

Email: Yehuda.Lindell@biu.ac.il

Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: July 11, 2019

D. McGrew
M. Curcio
S. Fluhrer
Cisco Systems

January 7, 2019

Hash-Based Signatures
draft-mcgrew-hash-sigs-15

Abstract

This note describes a digital signature system based on cryptographic hash functions, following the seminal work in this area of Lamport, Diffie, Winternitz, and Merkle, as adapted by Leighton and Micali in 1995. It specifies a one-time signature scheme and a general signature scheme. These systems provide asymmetric authentication without using large integer mathematics and can achieve a high security level. They are suitable for compact implementations, are relatively simple to implement, and naturally resist side-channel attacks. Unlike most other signature systems, hash-based signatures would still be secure even if it proves feasible for an attacker to build a quantum computer.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Table of Contents

1. Introduction

One-time signature systems, and general purpose signature systems built out of one-time signature systems, have been known since 1979 [Merkle79], were well studied in the 1990s [USPTO5432852], and have benefited from renewed attention in the last decade. The characteristics of these signature systems are small private and public keys and fast signature generation and verification, but large signatures and moderately slow key generation (in comparison with RSA and ECDSA). Private keys can be made very small by appropriate key generation, for example, as described in Appendix A. In recent years there has been interest in these systems because of their post-quantum security and their suitability for compact verifier implementations.

This note describes the Leighton and Micali adaptation [USPTO5432852] of the original Lamport-Diffie-Winternitz-Merkle one-time signature system [Merkle79] [C:Merkle87][C:Merkle89a][C:Merkle89b] and general signature system [Merkle79] with enough specificity to ensure interoperability between implementations.

A signature system provides asymmetric message authentication. The key generation algorithm produces a public/private key pair. A message is signed by a private key, producing a signature, and a message/signature pair can be verified by a public key. A One-Time

Signature (OTS) system can be used to sign one message securely, but will become insecure if more than one is signed with the same public/private key pair. An N-time signature system can be used to sign N or fewer messages securely. A Merkle tree signature scheme is an N-time signature system that uses an OTS system as a component.

In the Merkle scheme, a binary tree of height h is used to hold 2^h OTS key pairs. Each interior node of the tree holds a value which is the hash of the values of its two children nodes. The public key of the tree is the value of the root node (a recursive hash of the OTS public keys), while the private key of the tree is the collection of all the OTS private keys, together with the index of the next OTS private key to sign the next message with.

In this note we describe the Leighton-Micali Signature (LMS) system, which is a variant of the Merkle scheme, and a Hierarchical Signature System (HSS) built on top of it that can efficiently scale to larger numbers of signatures. In order to support signing a large number of messages on resource constrained systems, the Merkle tree can be subdivided into a number of smaller trees. Only the bottom-most tree is used to sign messages, while trees above that are used to sign the public keys of their children. For example, in the simplest case with 2 levels with both levels consisting of height h trees, the root tree is used to sign 2^h trees with 2^h OTS key pairs, and each second level tree has 2^h OTS key pairs, for a total of 2^(2h) bottom level key pairs, and so can sign 2^(2h) messages. The advantage of this scheme is that only the active trees need to be instantiated, which saves both time (for key generation) and space (for key storage). On the other hand, using a multilevel signature scheme inceases the size of the signature, as well as the signature verification time.

This note is structured as follows. Notes on postquantum cryptography are discussed in Section 1.1. Intellectual Property issues are discussed in Section 1.2. The notation used within this note is defined in Section 3, and the public formats are described in Section 3.3. The LM-OTS signature system is described in Section 4, and the LMS and HSS N-time signature systems are described in Section 5 and Section 6, respectively. Sufficient detail is provided to ensure interoperability. The rationale for the design decisions is given in Section 7. The IANA registry for these signature systems is described in Section 8. Security considerations are presented in Section 9. Comparison with another hash based signature algorithm (XMSS) is in Section 10.

This document represents the rough consensus of the CFRG.

1.1. CFRG Note on Post-Quantum Cryptography

All post-quantum algorithms documented by the Crypto Forum Research Group (CFRG) are today considered ready for experimentation and further engineering development (e.g., to establish the impact of performance and sizes on IETF protocols). However, at the time of writing, we do not have significant deployment experience with such algorithms.

Many of these algorithms come with specific restrictions, e.g., change of classical interface or less cryptanalysis of proposed parameters than established schemes. CFRG has consensus that all documents describing post-quantum technologies include the above paragraph and a clear additional warning about any specific restrictions, especially as those might affect use or deployment of the specific scheme. That guidance may be changed over time via document updates.

Additionally, for LMS:

CFRG consensus is that we are confident in the cryptographic security of the signature schemes described in this document against quantum computers, given the current state of the research community's knowledge about quantum algorithms. Indeed, we are confident that the security of a significant part of the Internet could be made dependent on the signature schemes defined in this document, if developers take care of the following.

In contrast to traditional signature schemes, the signature schemes described in this document are stateful, meaning the secret key changes over time. If a secret key state is used twice, no cryptographic security guarantees remain. In consequence, it becomes feasible to forge a signature on a new message. This is a new property that most developers will not be familiar with and requires careful handling of secret keys. Developers should not use the schemes described here except in systems that prevent the reuse of secret key states.

Note that the fact that the schemes described in this document are stateful also implies that classical APIs for digital signatures cannot be used without modification. The API MUST be able to handle a secret key state; in particular, this means that the API MUST allow to return an updated secret key state.

1.2. Intellectual Property

This draft is based on U.S. patent 5,432,852, which was issued over twenty years ago and is thus expired.

1.2.1. Disclaimer

This document is not intended as legal advice. Readers are advised to consult with their own legal advisers if they would like a legal interpretation of their rights.

The IETF policies and processes regarding intellectual property and patents are outlined in [RFC3979] and [RFC4879] and at https://datatracker.ietf.org/ipr/about.

1.3. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Interface

The LMS signing algorithm is stateful; it modifies and updates the private key as a side effect of generating a signature. Once a particular value of the private key is used to sign one message, it MUST NOT be used to sign another.

The key generation algorithm takes as input an indication of the parameters for the signature system. If it is successful, it returns both a private key and a public key. Otherwise, it returns an indication of failure.

The signing algorithm takes as input the message to be signed and the current value of the private key. If successful, it returns a signature and the next value of the private key, if there is such a value. After the private key of an N-time signature system has signed N messages, the signing algorithm returns the signature and an indication that there is no next value of the private key that can be used for signing. If unsuccessful, it returns an indication of failure.

The verification algorithm takes as input the public key, a message, and a signature, and returns an indication of whether or not the signature and message pair is valid.

A message/signature pair is valid if the signature was returned by the signing algorithm upon input of the message and the private key

corresponding to the public key; otherwise, the signature and message pair is not valid with probability very close to one.

3. Notation

3.1. Data Types

Bytes and byte strings are the fundamental data types. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string with a length of three. An array of byte strings is an ordered set, indexed starting at zero, in which all strings have the same length.

Unsigned integers are converted into byte strings by representing them in network byte order. To make the number of bytes in the representation explicit, we define the functions u8str(X), u16str(X), and u32str(X), which take a non-negative integer X as input and return one, two, and four byte strings, respectively. We also make use of the function strTou32(S), which takes a four-byte string S as input and returns a non-negative integer; the identity u32str(strTou32(S)) = S holds for any four-byte string S.

3.1.1. Operators

When a and b are real numbers, mathematical operators are defined as follows:

^ : a ^ b denotes the result of a raised to the power of b

* : a * b denotes the product of a multiplied by b

/ : a / b denotes the quotient of a divided by b

% : a % b denotes the remainder of the integer division of a by b (with a and b being restricted to integers in this case)

+ : a + b denotes the sum of a and b

- : a - b denotes the difference of a and b

AND : a AND b denotes the bitwise AND of the two nonnegative integers a and b (represented in binary notation)

The standard order of operations is used when evaluating arithmetic expressions.

When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation by i bit positions. Similarly, B << i denotes the logical left-shift operation.

If S and T are byte strings, then S || T denotes the concatenation of S and T. If S and T are equal length byte strings, then S AND T denotes the bitwise logical and operation.

The i-th element in an array A is denoted as A[i].

3.1.2. Functions

If r is a non-negative real number, then we define the following functions:

ceil(r) : returns the smallest integer greater than or equal to r

floor(r) : returns the largest integer less than or equal to r

lg(r) : returns the base-2 logarithm of r

3.1.3. Strings of w-bit elements

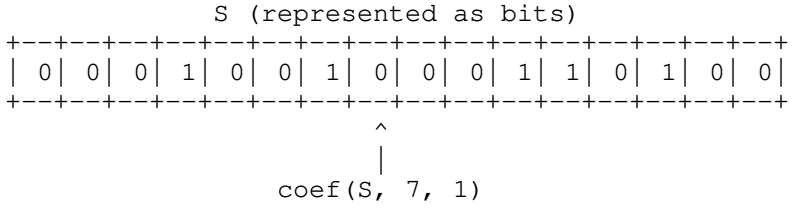
If S is a byte string, then byte(S, i) denotes its i-th byte, where the index starts at 0 at the left. Hence, byte(S, 0) is the leftmost byte of S, byte(S, 1) is the second byte at the left and (assuming S is n bytes long) byte(S, n-1) is the rightmost byte of S. In addition, bytes(S, i, j) denotes the range of bytes from the i-th to the j-th byte, inclusive. For example, if S = 0x02040608, then byte(S, 0) is 0x02 and bytes(S, 1, 2) is 0x0406.

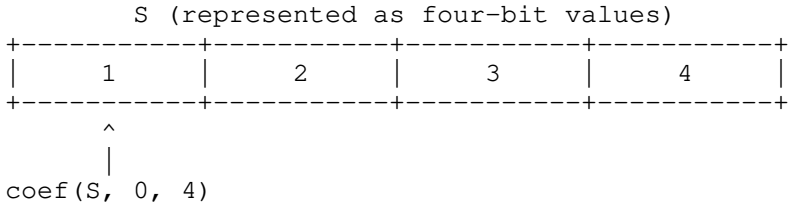
A byte string can be considered to be a string of w-bit unsigned integers; the correspondence is defined by the function coef(S, i, w) as follows:

If S is a string, i is a positive integer, and w is a member of the set { 1, 2, 4, 8 }, then coef(S, i, w) is the i-th, w-bit value, if S is interpreted as a sequence of w-bit values. That is,

```
coef(S, i, w) = (2^w - 1) AND
                ( byte(S, floor(i * w / 8)) >>
                  (8 - (w * (i % (8 / w)) + w)) )
```

For example, if S is the string 0x1234, then coef(S, 7, 1) is 0 and coef(S, 0, 4) is 1.





The return value of coef is an unsigned integer. If i is larger than the number of w-bit values in S, then coef(S, i, w) is undefined, and an attempt to compute that value MUST raise an error.

3.2. Typecodes

A typecode is an unsigned integer that is associated with a particular data format. The format of the LM-OTS, LMS, and HSS signatures and public keys all begin with a typecode that indicates the precise details used in that format. These typecodes are represented as four-byte unsigned integers in network byte order; equivalently, they are XDR enumerations (see Section 3.3).

3.3. Notation and Formats

The signature and public key formats are formally defined using the External Data Representation (XDR) [RFC4506] in order to provide an unambiguous, machine readable definition. For clarity, we also include a private key format as well, though consistency is not needed for interoperability and an implementation MAY use any private key format. Though XDR is used, these formats are simple and easy to parse without any special tools. An illustration of the layout of data in these objects is provided below. The definitions are as follows:

```
/* one-time signatures */
```

```
   enum lmots_algorithm_type {
     lmots_reserved       = 0,
     lmots_sha256_n32_w1  = 1,
     lmots_sha256_n32_w2  = 2,
     lmots_sha256_n32_w4  = 3,
     lmots_sha256_n32_w8  = 4
   };

   typedef opaque bytestring32[32];

   struct lmots_signature_n32_p265 {
     bytestring32 C;
     bytestring32 y[265];
   };

   struct lmots_signature_n32_p133 {
     bytestring32 C;
     bytestring32 y[133];
   };

   struct lmots_signature_n32_p67 {
     bytestring32 C;
     bytestring32 y[67];
   };

   struct lmots_signature_n32_p34 {
     bytestring32 C;
     bytestring32 y[34];
   };

   union lmots_signature switch (lmots_algorithm_type type) {
    case lmots_sha256_n32_w1:
      lmots_signature_n32_p265 sig_n32_p265;
    case lmots_sha256_n32_w2:
      lmots_signature_n32_p133 sig_n32_p133;
    case lmots_sha256_n32_w4:
      lmots_signature_n32_p67  sig_n32_p67;
    case lmots_sha256_n32_w8:
      lmots_signature_n32_p34  sig_n32_p34;
    default:
      void;   /* error condition */
   };


   /* hash based signatures (hbs) */

   enum lms_algorithm_type {
     lms_reserved       = 0,
```

```
     lms_sha256_n32_h5  = 5,
     lms_sha256_n32_h10 = 6,
     lms_sha256_n32_h15 = 7,
     lms_sha256_n32_h20 = 8,
     lms_sha256_n32_h25 = 9,
   };

   /* leighton-micali signatures (lms) */

   union lms_path switch (lms_algorithm_type type) {
    case lms_sha256_n32_h5:
      bytestring32 path_n32_h5[5];
    case lms_sha256_n32_h10:
      bytestring32 path_n32_h10[10];
    case lms_sha256_n32_h15:
      bytestring32 path_n32_h15[15];
    case lms_sha256_n32_h20:
      bytestring32 path_n32_h20[20];
    case lms_sha256_n32_h25:
      bytestring32 path_n32_h25[25];
    default:
      void;     /* error condition */
   };

   struct lms_signature {
     unsigned int q;
     lmots_signature lmots_sig;
     lms_path nodes;
   };

   struct lms_key_n32 {
     lmots_algorithm_type ots_alg_type;
     opaque I[16];
     opaque K[32];
   };

   union lms_public_key switch (lms_algorithm_type type) {
    case lms_sha256_n32_h5:
    case lms_sha256_n32_h10:
    case lms_sha256_n32_h15:
    case lms_sha256_n32_h20:
    case lms_sha256_n32_h25:
         lms_key_n32 z_n32;
    default:
      void;     /* error condition */
   };

   /* hierarchical signature system (hss)  */
```

```
struct hss_public_key {
  unsigned int L;
  lms_public_key pub;
};

struct signed_public_key {
  lms_signature sig;
  lms_public_key pub;
}

struct hss_signature {
  signed_public_key signed_keys<7>;
  lms_signature sig_of_message;
};
```

4. LM-OTS One-Time Signatures

This section defines LM-OTS signatures. The signature is used to validate the authenticity of a message by associating a secret private key with a shared public key. These are one-time signatures; each private key MUST be used at most one time to sign any given message.

As part of the signing process, a digest of the original message is computed using the cryptographic hash function H (see Section 4.1), and the resulting digest is signed.

In order to facilitate its use in an N-time signature system, the LM-OTS key generation, signing, and verification algorithms all take as input parameters I and q. The parameter I is a 16 byte string, which indicates which Merkle tree this LM-OTS is used with. The parameter q is a 32 bit integer which indicates the leaf of the Merkle tree where the OTS public key appears. These parameters are used as part of the security string, as listed in Section 7.1. When the LM-OTS signature system is used outside of an N-time signature system, the value I MAY be used to differentiate this one time signatures from others; however the value q MUST be set to the all-zero value.

4.1. Parameters

The signature system uses the parameters n and w, which are both positive integers. The algorithm description also makes use of the internal parameters p and ls, which are dependent on n and w. These parameters are summarized as follows:

n : the number of bytes of the output of the hash function

w : the width (in bits) of the Winternitz coefficients; that is, the number of bits from the hash or checksum that is used with a single Winternitz chain. It is a member of the set { 1, 2, 4, 8 }

p : the number of n-byte string elements that make up the LM-OTS signature. This is a function of n and w; the values for the defined parameter sets are lited in Table 1; it can also be computed by the algorithm given in Appendix B.

ls : the number of left-shift bits used in the checksum function Cksm (defined in Section 4.4)

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an n-byte string

For more background on the cryptographic security requirements on H, see the Section 9.

The value of n is determined by the hash function selected for use as part of the LM-OTS algorithm; the choice of this value has a strong effect on the security of the system. The parameter w determines the length of the Winternitz chains computed as a part of the OTS signature (which involve 2^w-1 invocations of the hash function); it has little effect on security. Increasing w will shorten the signature, but at a cost of a larger computation to generate and verify a signature. The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix B. A table illustrating various combinations of n, w, p and ls, along with the resulting signature length, is provided in Table 1.

The value of w describes a space/time trade-off; increasing the value of w will cause the signature to shrink (by decreasing the value of p) while increasing the amount of time needed to perform operations with it (generate the public key, generate and verify the signature); in general, the LM-OTS signature is 4+n*(p+1) bytes long, and public key generation will take p*(2^w-1)+1 hash computations (and signature generation and verification will take approximately half that on average).

Parameter Set Name	H	n	w	p	ls	sig_len
LMOTS_SHA256_N32_W1	SHA256	32	1	265	7	8516
LMOTS_SHA256_N32_W2	SHA256	32	2	133	6	4292
LMOTS_SHA256_N32_W4	SHA256	32	4	67	4	2180
LMOTS_SHA256_N32_W8	SHA256	32	8	34	0	1124

Table 1

Here SHA256 denotes the SHA-256 hash fucntion defined in NIST standard [FIPS180].

4.2. Private Key

The format of the LM-OTS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that the private key may consist of a typecode indicating the particular LM-OTS algorithm, an array x[] containing p n-byte strings, and the 16-byte string I and the 4 byte string q. This private key MUST be used to sign (at most) one message. The following algorithm shows pseudocode for generating a private key.

Algorithm 0: Generating a Private Key

1. retrieve the values of q and I (the 16-byte identifier of the LMS public/private keypair) from the LMS tree that this LM-OTS private key will be used with

2. set type to the typecode of the algorithm

3. set n and p according to the typecode and Table 1

4. compute the array x as follows:
   ```
   for ( i = 0; i < p; i = i + 1 ) {
     set x[i] to a uniformly random n-byte string
   }
   ```

5. return u32str(type) || I || u32str(q) || x[0] || x[1] || ... || x[p-1]

An implementation MAY use a pseudorandom method to compute x[i], as suggested in [Merkle79], page 46. The details of the pseudorandom method do not affect interoperability, but the cryptographic strength

MUST match that of the LM-OTS algorithm. Appendix A provides an example of a pseudorandom method for computing the LM-OTS private key.

4.3. Public Key

The LM-OTS public key is generated from the private key by iteratively applying the function H to each individual element of x, for 2^w - 1 iterations, then hashing all of the resulting values.

The public key is generated from the private key using the following algorithm, or any equivalent process.

Algorithm 1: Generating a One Time Signature Public Key From a Private Key

1. set type to the typecode of the algorithm

2. set the integers n, p, and w according to the typecode and Table 1

3. determine x, I and q from the private key

4. compute the string K as follows:

```
for ( i = 0; i < p; i = i + 1 ) {
  tmp = x[i]
  for ( j = 0; j < 2^w - 1; j = j + 1 ) {
     tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
  }
  y[i] = tmp
}
K = H(I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1])
```

5. return u32str(type) || I || u32str(q) || K

where D_PBLC is the fixed two byte value 0x8080, which is used to distinguish the last hash from every other hash in this system.

The public key is the value returned by Algorithm 1.

4.4. Checksum

A checksum is used to ensure that any forgery attempt that manipulates the elements of an existing signature will be detected. This checksum is needed because an attacker can freely advance any of the Winternitz chains. That is, if this checksum were not present, then an attacker who could find a hash that has every digit larger than the valid hash could replace it (and adjust the Winternitz chains). The security property that it provides is detailed in

Section 9. The checksum function Cksm is defined as follows, where S denotes the n-byte string that is input to that function, and the value sum is a 16-bit unsigned integer:

Algorithm 2: Checksum Calculation

```
sum = 0
for ( i = 0; i < (n*8/w); i = i + 1 ) {
  sum = sum + (2^w - 1) - coef(S, i, w)
}
return (sum << ls)
```

ls is the parameter that shifts the significant bits of the checksum into the positions that will actually be used by the coef function when encoding the digits of the checksum. The actual ls parameter is a function of the n and w parameters; the values for the currently defined parameter sets is shown in table 1. It is calculated by the algorithm given in Appendix B.

Because of the left-shift operation, the rightmost bits of the result of Cksm will often be zeros. Due to the value of p, these bits will not be used during signature generation or verification.

4.5. Signature Generation

The LM-OTS signature of a message is generated by first prepending the LMS key identifier I, the LMS leaf identifier q, the value D_MESG (0x8181) and the randomizer C to the message, then computing the hash, and then concatenating the checksum of the hash to the hash itself, then considering the resulting value as a sequence of w-bit values, and using each of the w-bit values to determine the number of times to apply the function H to the corresponding element of the private key. The outputs of the function H are concatenated together and returned as the signature. The pseudocode for this procedure is shown below.

Algorithm 3: Generating a One Time Signature From a Private Key and a Message

1. set type to the typecode of the algorithm

2. set n, p, and w according to the typecode and Table 1

3. determine x, I and q from the private key

4. set C to a uniformly random n-byte string

5. compute the array y as follows:

```
Q = H(I || u32str(q) || u16str(D_MESG) || C || message)
for ( i = 0; i < p; i = i + 1 ) {
  a = coef(Q || Cksm(Q), i, w)
  tmp = x[i]
  for ( j = 0; j < a; j = j + 1 ) {
     tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
  }
  y[i] = tmp
}
```

6. return u32str(type) || C || y[0] || ... || y[p-1]

Note that this algorithm results in a signature whose elements are intermediate values of the elements computed by the public key algorithm in Section 4.3.

The signature is the string returned by Algorithm 3. Section 3.3 specifies the typecode and more formally defines the encoding and decoding of the string.

4.6. Signature Verification

In order to verify a message with its signature (an array of n-byte strings, denoted as y), the receiver must "complete" the chain of iterations of H using the w-bit coefficients of the string resulting from the concatenation of the message hash and its checksum. This computation should result in a value that matches the provided public key.

Algorithm 4a: Verifying a Signature and Message Using a Public Key

1. if the public key is not at least four bytes long,
   return INVALID

2. parse pubtype, I, q, and K from the public key as follows:
   a. pubtype = strTou32(first 4 bytes of public key)

   b. set n according to the pubkey and Table 1; if the public key
      is not exactly 24 + n bytes long, return INVALID

   c. I = next 16 bytes of public key

   d. q = strTou32(next 4 bytes of public key)

   e. K = next n bytes of public key

3. compute the public key candidate Kc from the signature,
   message, pubtype and the identifiers I and q obtained from the
   public key, using Algorithm 4b. If Algorithm 4b returns
   INVALID, then return INVALID.

4. if Kc is equal to K, return VALID; otherwise, return INVALID

Algorithm 4b: Computing a Public Key Candidate Kc from a Signature, Message, Signature Typecode pubtype, and identifiers I, q

1. if the signature is not at least four bytes long, return INVALID

2. parse sigtype, C, and y from the signature as follows:
   a. sigtype = strTou32(first 4 bytes of signature)

   b. if sigtype is not equal to pubtype, return INVALID

   c. set n and p according to the pubtype and Table 1; if the signature is not exactly 4 + n * (p+1) bytes long, return INVALID

   d. C = next n bytes of signature

   e. y[0] = next n bytes of signature
      y[1] = next n bytes of signature
      ...
      y[p-1] = next n bytes of signature

3. compute the string Kc as follows

```
Q = H(I || u32str(q) || u16str(D_MESG) || C || message)
for ( i = 0; i < p; i = i + 1 ) {
  a = coef(Q || Cksm(Q), i, w)
  tmp = y[i]
  for ( j = a; j < 2^w - 1; j = j + 1 ) {
     tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
  }
  z[i] = tmp
}
Kc = H(I || u32str(q) || u16str(D_PBLC) ||
                              z[0] || z[1] || ... || z[p-1])
```

4. return Kc

5. Leighton-Micali Signatures

The Leighton-Micali Signature (LMS) method can sign a potentially large but fixed number of messages. An LMS system uses two cryptographic components: a one-time signature method and a hash function. Each LMS public/private key pair is associated with a perfect binary tree, each node of which contains an m-byte value, where m is the output length of the hash function. Each leaf of the tree contains the value of the public key of an LM-OTS public/private key pair. The value contained by the root of the tree is the LMS public key. Each interior node is computed by applying the hash function to the concatenation of the values of its children nodes.

Each node of the tree is associated with a node number, an unsigned integer that is denoted as node_num in the algorithms below, which is computed as follows. The root node has node number 1; for each node with node number N < 2^h (where h is the height of the tree), its left child has node number 2*N, while its right child has node number 2*N+1. The result of this is that each node within the tree will have a unique node number, and the leaves will have node numbers 2^h, (2^h)+1, (2^h)+2, ..., (2^h)+(2^h)-1. In general, the j-th node at level i has node number 2^i + j. The node number can conveniently be computed when it is needed in the LMS algorithms, as described in those algorithms.

5.1. Parameters

An LMS system has the following parameters:

h : the height of the tree, and

m : the number of bytes associated with each node.

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an m-byte string.

There are 2^h leaves in the tree.

The overall strength of the LMS signatures is governed by the weaker of the hash function used within the LM-OTS and the hash function used within the LMS system. In order to minimize the risk, these two hash functions SHOULD be the same (so that an attacker could not take advantage of the weaker hash function choice).

Name	H	m	h
LMS_SHA256_M32_H5	SHA256	32	5
LMS_SHA256_M32_H10	SHA256	32	10
LMS_SHA256_M32_H15	SHA256	32	15
LMS_SHA256_M32_H20	SHA256	32	20
LMS_SHA256_M32_H25	SHA256	32	25

Table 2

5.2. LMS Private Key

The format of the LMS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that it may consist of an array OTS_PRIV[] of 2^h LM-OTS private keys, and the leaf number q of the next LM-OTS private key that has not yet been used. The q-th element of OTS_PRIV[] is generated using Algorithm 0 with the identifiers I, q. The leaf number q is initialized to zero when the LMS private key is created. The process is as follows:

Algorithm 5: Computing an LMS Private Key.

1. determine h and m from the typecode and Table 2.

2. set I to a uniformly random 16-byte string

3. compute the array OTS_PRIV[] as follows:
   ```
   for ( q = 0; q < 2^h; q = q + 1) {
      OTS_PRIV[q] = LM-OTS private key with identifiers I, q
   }
   ```

4. q = 0

An LMS private key MAY be generated pseudorandomly from a secret value, in which case the secret value MUST be at least m bytes long, be uniformly random, and MUST NOT be used for any other purpose than the generation of the LMS private key. The details of how this process is done do not affect interoperability; that is, the public key verification operation is independent of these details. Appendix A provides an example of a pseudorandom method for computing an LMS private key.

The signature generation logic uses q as the next leaf to use, hence step 4 starts it off at the left-most one. Because the signature proces increments q after the signature operation, the first signature will have q=0.

5.3. LMS Public Key

An LMS public key is defined as follows, where we denote the public key final hash value (namely, the K value computed in Algorithm 1) associated with the i-th LM-OTS private key as OTS_PUB_HASH[i], with i ranging from 0 to (2^h)-1. Each instance of an LMS public/private key pair is associated with a balanced binary tree, and the nodes of that tree are indexed from 1 to 2^(h+1)-1. Each node is associated with an m-byte string, and the string for the r-th node is denoted as T[r] and is defined as

```
if r >= 2^h:
     H(I||u32str(r)||u16str(D_LEAF)||OTS_PUB_HASH[r-2^h])
else
     H(I||u32str(r)||u16str(D_INTR)||T[2*r]||T[2*r+1])
```

where D_LEAF is the fixed two byte value 0x8282, and D_INTR is the fixed two byte value 0x8383, both of which are used to distinguish this hash from every other hash in this system.

When we have r >= 2^h, then we are processing a leaf node (and thus hashing only a single LM-OTS public key). When we have r < 2^h, then we are processing an internal node, that is, a node with two child nodes that we need to combine.

The LMS public key is the string

```
u32str(type) || u32str(otstype) || I || T[1]
```

Section 3.3 specifies the format of the type variable. The value otstype is the parameter set for the LM-OTS public/private keypairs used. The value I is the private key identifier, and is the value used for all computations for the same LMS tree. The value T[1] can be computed via recursive application of the above equation, or by any equivalent method. An iterative procedure is outlined in Appendix C.

5.4. LMS Signature

An LMS signature consists of

the number q of the leaf associated with the LM-OTS signature, as a four-byte unsigned integer in network byte order,

an LM-OTS signature,

a typecode indicating the particular LMS algorithm,

an array of h m-byte values that is associated with the path through the tree from the leaf associated with the LM-OTS signature to the root.

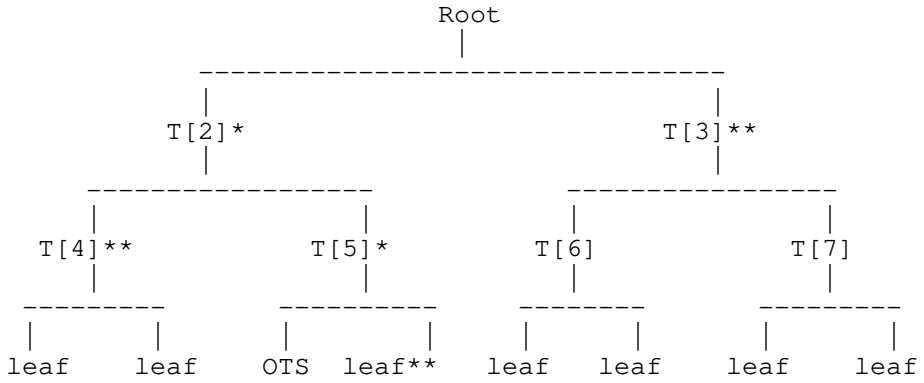
Symbolically, the signature can be represented as

```
    u32str(q) || lmots_signature || u32str(type) ||
              path[0] || path[1] || path[2] || ... || path[h-1]
```

Section 3.3 specifies the typecode and more formally defines the format. The array for a tree with height h will have h values and

contains the values of the siblings of (that is, is adjacent to) the nodes on the path from the leaf to the root, where the sibling to node A is the other node which shares node A's parent. In the signature, 0 is counted from the bottom level of the tree, and so path[0] is the value of the node adjacent to leaf node q; path[1] is the second level node that is adjacent to leaf node q's parent, and so up the tree until we get to path[h-1], which is the value of the next-to-the-top level node that leaf node q does not reside in.

Below is a simple example of the authentication path for h=3 and q=2. The leaf marked OTS is the one time signature which is used to sign the actual message. The nodes on the path from the OTS public key to the root are marked with a *, while the nodes that are used within the path array are marked with a **. The values in the path array are those nodes which are siblings of the nodes on the path; path[0] is the leaf** node that is adjacent to the OTS public key (which is the start of the path); path[1] is the T[4]** node which is the sibling of the second node T[5]* on the path, and path[2] is the T[3]** node which is the sibling of the third node T[2]* on the path.



The idea behind this authentication path is that it allows us to validate the OTS hash with using h path array values and hash computations. What the verifier does is recompute the hashes up the path; first, he hashes the given OTS and path[0] value, giving a tentative T[5]' value. Then, he hashes his path[1] and tentative T[5]' value to get a tentative T[2]' value. Then, he hashes that and the path[2] value to get a tentative Root' value. If that value is the known public key of the Merkle tree, then we can assume that the value T[2]' he got was the correct T[2] value in the original tree, and so the T[5]' value he got was the correct T[5] value in the original tree, and so the OTS public key is the same as in the original, and hence is correct.

5.4.1. LMS Signature Generation

To compute the LMS signature of a message with an LMS private key, the signer first computes the LM-OTS signature of the message using the leaf number of the next unused LM-OTS private key. The leaf number q in the signature is set to the leaf number of the LMS private key that was used in the signature. Before releasing the signature, the leaf number q in the LMS private key MUST be incremented, to prevent the LM-OTS private key from being used again. If the LMS private key is maintained in nonvolatile memory, then the implementation MUST ensure that the incremented value has been stored before releasing the signature. The issue this tries to prevent is a scenario where a) we generate a signature, using one LM-OTS private key, and release it to the application, b) before we update the nonvolatile memory, we crash, and c) we reboot, and generate a second signature using the same LM-OTS private key; with two different signatures using the same LM-OTS private key, someone could potentially generate a forged signature of a third message.

The array of node values in the signature MAY be computed in any way. There are many potential time/storage tradeoffs that can be applied. The fastest alternative is to store all of the nodes of the tree and set the array in the signature by copying them; pseudocode to do so appears in Appendix D. The least storage intensive alternative is to recompute all of the nodes for each signature. Note that the details of this procedure are not important for interoperability; it is not necessary to know any of these details in order to perform the signature verification operation. The internal nodes of the tree need not be kept secret, and thus a node-caching scheme that stores only internal nodes can sidestep the need for strong protections.

Several useful time/storage tradeoffs are described in the 'Small-Memory LM Schemes' section of [USPTO5432852].

5.4.2. LMS Signature Verification

An LMS signature is verified by first using the LM-OTS signature verification algorithm (Algorithm 4b) to compute the LM-OTS public key from the LM-OTS signature and the message. The value of that public key is then assigned to the associated leaf of the LMS tree, then the root of the tree is computed from the leaf value and the array path[] as described in Algorithm 6 below. If the root value matches the public key, then the signature is valid; otherwise, the signature fails.

Algorithm 6: LMS Signature Verification

1. if the public key is not at least eight bytes long, return INVALID

2. parse pubtype, I, and T[1] from the public key as follows:

   a. pubtype = strTou32(first 4 bytes of public key)

   b. ots_typecode = strTou32(next 4 bytes of public key)

   c. set m according to pubtype, based on Table 2

   d. if the public key is not exactly 24 + m bytes long, return INVALID

   e. I = next 16 bytes of the public key

   f. T[1] = next m bytes of the public key

3. compute the LMS Public Key Candidate Tc from the signature, message, identifier, pubtype and ots_typecode using Algorithm 6a.

4. if Tc is equal to T[1], return VALID; otherwise, return INVALID

Algorithm 6a: Computing an LMS Public Key Candidate from a Signature, Message, Identifier, and algorithm typecode

1. if the signature is not at least eight bytes long, return INVALID

2. parse sigtype, q, lmots_signature, and path from the signature as follows:

   a. q = strTou32(first 4 bytes of signature)

   b. otssigtype = strTou32(next 4 bytes of signature)

   c. if otssigtype is not the OTS typecode from the public key, return INVALID

   d. set n, p according to otssigtype and Table 1; if the signature is not at least 12 + n * (p + 1) bytes long, return INVALID

   e. lmots_signature = bytes 4 through 7 + n * (p + 1) of signature

   f. sigtype = strTou32(bytes 8 + n * (p + 1)) through

```
      11 + n * (p + 1) of signature)

   f. if sigtype is not the LM typecode from the public key,
        return INVALID

   g. set m, h according to sigtype and Table 2

   h. if q >= 2^h or the signature is not exactly
      12 + n * (p + 1) + m * h bytes long,
        return INVALID

   i. set path as follows:
        path[0] = next m bytes of signature
        path[1] = next m bytes of signature
        ...
        path[h-1] = next m bytes of signature

3. Kc = candidate public key computed by applying Algorithm 4b
   to the signature lmots_signature, the message, and the
   identifiers I, q

4. compute the candidate LMS root value Tc as follows:
   node_num = 2^h + q
   tmp = H(I || u32str(node_num) || u16str(D_LEAF) || Kc)
   i = 0
   while (node_num > 1) {
     if (node_num is odd):
       tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||path[i]||tmp)
     else:
       tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||tmp||path[i])
     node_num = node_num/2
     i = i + 1
   }
   Tc = tmp

5. return Tc
```

6. Hierarchical signatures

In scenarios where it is necessary to minimize the time taken by the public key generation process, a Hierarchical N-time Signature System (HSS) can be used. This hierarchical scheme, which we describe in this section, uses the LMS scheme as a component. In HSS, we have a sequence of L LMS trees, where the public key for the first LMS tree is included in the public key of the HSS system, and where each LMS private key signs the next LMS public key, and where the last LMS private key signs the actual message. For example, if we have a three level hierarchy (L=3), then to sign a message, we would have:

The first LMS private key (level 0) signs a level 1 LMS public key.

The second LMS private key (level 1) signs a level 2 LMS public key.

The third LMS private key (level 2) signs the message.

The root of the level 0 LMS tree is contained in the HSS public key.

To verify the LMS signature, we would verify all the signatures:

We would verify that the level 1 LMS public key is correctly signed by the level 0 signature.

We would verify that the level 2 LMS public key is correctly signed by the level 1 signature.

We would verify that the message is correctly signed by the level 2 signature.

We would accept the HSS signature only if all the signatures validated.

During the signature generation process, we sign messages with the lowest (level L-1) LMS tree. Once we have used all the leafs in that tree to sign messages, we would discard it, generate a fresh LMS tree, and sign it with the next (level L-2) LMS tree (and when that is used up, recursively generate and sign a fresh level L-2 LMS tree).

HSS, in essence, utilizes a tree of LMS trees. There is a single LMS tree at level 0 (the root). Each LMS tree (actually, the private key corresponding to the LMS tree) at level i is used to sign 2^h objects (where h is the height of trees at level i). If i < L-1, then each object will be another LMS tree (actually, the public key) at level i+1; if i = L-1, we've reached the bottom of the HSS tree, and so each object is a message from the application. The HSS public key contains the public key of the LMS tree at the root, and an HSS signature is associated with a path from the root of the HSS tree to the leaf.

Compared to LMS, HSS has a much reduced public key generation time, as only the root tree needs to be generated prior to the distribution of the HSS public key. For example, a L=3 tree (with h=10 at each level) would have 1 level 0 LMS tree, 2^10 level 1 LMS trees (with each such level 1 public key signed by one of the 1024 level 0 OTS public keys), and 2^20 level 2 LMS trees. Only 1024 OTS public keys

need to be computed to generate the HSS public key (as you need to compute only the level 0 LMS tree to compute that value; you can, of course, decide to compute the initial level 1 and level 2 LMS trees). And, the 2^20 level 2 LMS trees can jointly sign a total of over a billion messages. In contrast, a single LMS tree that could sign a billion messages would require a billion OTS public keys to be computed first (even if h=30 were allowed in a supported parameter set).

Each LMS tree within the hierarchy is associated with a distinct LMS public key, private key, signature, and identifier. The number of levels is denoted L, and is between one and eight, inclusive. The following notation is used, where i is an integer between 0 and L-1 inclusive, and the root of the hierarchy is level 0:

prv[i] is the current LMS private key of the i-th level.

pub[i] is the current LMS public key of the i-th level, as described in Section 5.3.

sig[i] is the LMS signature of public key pub[i+1] generated using the private key prv[i].

It is expected that the above arrays are maintained for the course of the HSS key. The contents of the prv[] array MUST be kept private; the pub[] and sig[] array may be revealed, should the implementation find that convienent.

In this section, we say that an N-time private key is exhausted when it has generated N signatures, and thus it can no longer be used for signing.

For i > 0, the values prv[i], pub[i] and (for all values of i) sig[i] will be updated over time, as private keys are exhausted, and replaced by newer keys.

When these keys pairs are updated (or initially generated before the first message is signed), then the LMS key generation processes outlined in sections Section 5.2 and Section 5.3 are performed. If the generated key pairs are for level i of the HSS hierarchy, then we store the public key in pub[i] and the private key in prv[i]. In addition, if i > 0, then we sign the generated public key with the LMS private key at level i-1, placing the signature into sig[i-1]. When the LMS key pair are generated, the key pair and the corresponding identifier MUST be generated indepently of all other keypairs.

HSS allows L=1, in which case the HSS public key and signature formats are essentially the LMS public key and signature formats, prepended by a fixed field. Since HSS with L=1 has very little overhead compared to LMS, all implementations MUST support HSS in order to maximize interoperability.

We specifically allow different LMS levels to use different parameter sets. For example, the 0-th LMS public key (the root) may use the LMS_SHA256_M32_H15 parameter set, while the 1-th public key may use LMS_SHA256_M32_H10. There are practical reasons to allow this; for one, the signer may decide to store parts of the 0-th LMS tree (that it needs to construct while computing the public key) to accelerate later operations. As the 0-th tree is never updated, these internal nodes will never need to be recomputed. In addition, during the signature generation operation, almost all the operations involved with updating the authentication path occurs with the bottom (L-1th) LMS public key; hence it may be useful to make the tree that implements that to be shorter.

A close reading of the HSS verification pseudocode would show that it would allow the parameters of the non-top LMS public keys to change over time; for example, the signer might initially have the 1-th LMS public key to use LMS_SHA256_M32_H10, but when that tree is exhausted, the signer might replace it with LMS_SHA256_M32_H15 LMS public key. While this would work with the example verification pseudocode, the signer MUST NOT change the parameter sets for a specific level. This prohibition is to support verifiers that may keep state over the course of several signature verifications.

6.1. Key Generation

The public key of the HSS scheme consists of the number of levels L, followed by pub[0], the public key of the top level.

The HSS private key consists of prv[0], ... , prv[L-1], along with the associated pub[0], ... pub[L-1] and sig[0], ..., sig[L-2] values. As stated earlier, the values of the pub[] and sig[] arrays need not be kept secret, but may be revealed. The value of pub[0] does not change (and, except for the index q, the value of prv[0] need not change), though the values of pub[i] and prv[i] are dynamic for i > 0, and are changed by the signature generation algorithm.

During the key generation, the public and private keys are initialized. Here is some pseudocode that explains the key generation logic

Algorithm 7: Generating an HSS keypair

1. generate an LMS key pair, as specified in sections 5.2 and 5.3, placing the private key into priv[0], and the public key into pub[0]

2. for i = 1 to L-1 do {
       generate an LMS key pair, placing the private key into priv[i] and the public key into pub[i]

       sig[i-1] = lms_signature(pub[i], priv[i-1])
   }

3. return u32str(L) || pub[0] as the public key, and the priv[], pub[] and sig[] arrays as the private key

In the above algorithm, each LMS public/private keypair generated MUST be generated indepedently.

Note that the value of the public key does not depend on the execution of step 2. As a result, an implementation may decide to delay step 2 until later, for example, during the initial signature generation operation.

6.2. Signature Generation

To sign a message using an HSS keypair, the following steps are performed:

If prv[L-1] is exhausted, then determine the smallest integer d such that all of the private keys prv[d], prv[d+1], ... , prv[L-1] are exhausted. If d is equal to zero, then the HSS key pair is exhausted, and it MUST NOT generate any more signatures. Otherwise, the key pairs for levels d through L-1 must be regenerated during the signature generation process, as follows. For i from d to L-1, a new LMS public and private key pair with a new identifier is generated, pub[i] and prv[i] are set to those values, then the public key pub[i] is signed with prv[i-1], and sig[i-1] is set to the resulting value.

The message is signed with prv[L-1], and the value sig[L-1] is set to that result.

The value of the HSS signature is set as follows. We let signed_pub_key denote an array of octet strings, where signed_pub_key[i] = sig[i] || pub[i+1], for i between 0 and Nspk-1, inclusive, where Nspk = L-1 denotes the number of signed public

keys. Then the HSS signature is u32str(Nspk) || signed_pub_key[0] || ... || signed_pub_key[Nspk-1] || sig[Nspk].

Note that the number of signed_pub_key elements in the signature is indicated by the value Nspk that appears in the initial four bytes of the signature.

Here is some pseudocode of the above logic

Algorithm 8: Generating an HSS signature

1. If the message-signing key prv[L-1] is exhausted, regenerate that key pair, together with any parent key pairs that might be necessary.
   If the root key pair is exhausted, then the HSS key pair is exhausted and it MUST NOT generate any more signatures.

```
d = L
while (prv[d-1].q == 2^(prv[d-1].h)) {
    d = d - 1
    if (d == 0)
        return FAILURE
}
while (d < L) {
    create lms keypair pub[d], prv[d]
    sig[d-1] = lms_signature( pub[d], prv[d-1] )
    d = d + 1
}
```

2. sign the message
   `sig[L-1] = lms_signature( msg, prv[L-1] )`

3. Create the list of signed public keys

```
i = 0;
while (i < L-1) {
    signed_pub_key[i] = sig[i] || pub[i+1]
    i = i + 1
}
```

4. return u32str(L-1) || signed_pub_key[0] || ... || signed_pub_key[L-2] || sig[L-1]

In the specific case of L=1, the format of an HSS signature is

u32str(0) || sig[0]

In the general case, the format of an HSS signature is

```
   u32str(Nspk) || signed_pub_key[0] || ...
                            || signed_pub_key[Nspk-1] || sig[Nspk]
```

which is equivalent to

```
   u32str(Nspk) || sig[0] || pub[1] || ...
                            || sig[Nspk-1] || pub[Nspk] || sig[Nspk]
```

6.3. Signature Verification

To verify a signature S and message using the public key pub, the following steps are performed:

The signature S is parsed into its components as follows:

```
Nspk = strTou32(first four bytes of S)
if Nspk+1 is not equal to the number of levels L in pub:
   return INVALID
for (i = 0; i < Nspk; i = i + 1) {
   siglist[i] = next LMS signature parsed from S
   publist[i] = next LMS public key parsed from S
}
siglist[Nspk] = next LMS signature parsed from S

key = pub
for (i = 0; i < Nspk; i = i + 1) {
   sig = siglist[i]
   msg = publist[i]
   if (lms_verify(msg, key, sig) != VALID):
       return INVALID
   key = msg
}
return lms_verify(message, key, siglist[Nspk])
```

Since the length of an LMS signature cannot be known without parsing it, the HSS signature verification algorithm makes use of an LMS signature parsing routine that takes as input a string consisting of an LMS signature with an arbitrary string appended to it, and returns both the LMS signature and the appended string. The latter is passed on for further processing.

6.4. Parameter Set Recommendations

As for guidance as to the number of LMS level, and the size of each, any discussion of performance is implementation specific. In general, the sole drawback for a single LMS tree is the time it takes

to generate the public key; as every LM-OTS public key needs to be generated, the time this takes can be substantial. For a two level tree, only the top level LMS tree and the initial bottom level LMS tree needs to be generated initially (before the first signature is generated); this will in general be significantly quicker.

To give a general idea on the trade-offs available, we include some measurements taken with the github.com/cisco/hash-sigs LMS implementation, taken on a 3.3 GHz Xeon processor, with threading enabled. We tried various parameter sets, all with W=8 (which minimizes signature size, while increasing time). These are here to give a guideline as to what's possible; for the computational time, your mileage may vary, depending on the computing resources you have. The machine these tests were performed on does not have the SHA-256 extensions; you could possibly do significantly better.

ParmSet	KeyGenTime	SigSize	KeyLifetime
15	6 sec	1616	30 seconds
20	3 min	1776	16 minutes
25	1.5 hour	1936	9 hours
15/10	6 sec	3172	9 hours
15/15	6 sec	3332	12 days
20/10	3 min	3332	12 days
20/15	3 min	3492	1 year
25/10	1.5 hour	3492	1 year
25/15	1.5 hour	3652	34 years

Table 3

ParmSet: this is the height of the Merkle tree(s); parameter sets listed as a single integer have L=1, and consist a single Merkle tree of that height; parameter sets with L=2 are listed as x/y, with x being the height of the top level Merkle tree, and y being the bottom level.

KeyGenTime: the measured key generation time; that is, the time needed to generate the public private key pair.

SigSize: the size of a signature (in bytes)

KeyLifetime: the lifetime of a key, assuming we generated 1000 signatures per second. In practice, we're not likely to get anywhere close to 1000 signatures per second sustained; if you have a more appropriate figure for your scenario, this column is pretty easy to recompute.

As for signature generation or verification times, those are moderately insensitive to the above parameter settings (except for the Winternitz setting, and the number of Merkle trees for verification). Tests on the same machine (without multithreading) gave approximately 4msec to sign a short message, 2.6msec to verify; these tests used a two level ParmSet; a single level would approximately halve the verification time. All times can be significantly improved (by perhaps a factor of 8) by using a parameter set with W=4; however that also about doubles the signature size.

7. Rationale

The goal of this note is to describe the LM-OTS, LMS and HSS algorithms following the original references and present the modern security analysis of those algorithms. Other signature methods are out of scope and may be interesting follow-on work.

We adopt the techniques described by Leighton and Micali to mitigate attacks that amortize their work over multiple invocations of the hash function.

The values taken by the identifier I across different LMS public/private key pairs are chosen randomly in order to improve security. The analysis of this method in [Fluhrer17] shows that we do not need uniqueness to ensure security; we do need to ensure that we don't have a large number of private keys that use the same I value. By randomly selecting 16 byte I values, the chance that, out of 2^64 private keys, 4 or more of them will use the same I value is negligible (that is, has probability less than 2^-128).

The reason 16 bytes I values were selected was to optimize the Winternitz hash chain operation. With the current settings, the value being hashed is exactly 55 bytes long (for a 32 byte hash function), which SHA-256 can hash in a single hash compression operation. Other hash functions may be used in future specifications; all the ones that we will be likely to support (SHA-512/256 and the various SHA-3 hashes) would work well with a 16-byte I value.

The signature and public key formats are designed so that they are relatively easy to parse. Each format starts with a 32-bit enumeration value that indicates the details of the signature algorithm and provides all of the information that is needed in order to parse the format.

The Checksum Section 4.4 is calculated using a non-negative integer "sum", whose width was chosen to be an integer number of w-bit fields such that it is capable of holding the difference of the total possible number of applications of the function H as defined in the signing algorithm of Section 4.5 and the total actual number. In the case that the number of times H is applied is 0, the sum is (2^w - 1) * (8*n/w). Thus for the purposes of this document, which describes signature methods based on H = SHA256 (n = 32 bytes) and w = { 1, 2, 4, 8 }, the sum variable is a 16-bit non-negative integer for all combinations of n and w. The calculation uses the parameter ls defined in Section 4.1 and calculated in Appendix B, which indicates the number of bits used in the left-shift operation.

7.1. Security String

To improve security against attacks that amortize their effort against multiple invocations of the hash function, Leighton and Micali introduce a "security string" that is distinct for each invocation of that function. Whenever this process computes a hash, the string being hashed will start with a string formed from the below fields. These fields will appear in fixed locations in the value we compute the hash of, and so we list where in the hash these fields would be present. These fields that make up this string are:

I - a 16-byte identifier for the LMS public/private key pair. It MUST be chosen uniformly at random, or via a pseudorandom process, at the time that a key pair is generated, in order to minimize the probability that any specific value of I be used for a large number of different LMS private keys. This is always bytes 0-15 of the value being hashed.

r - in the LMS N-time signature scheme, the node number r associated with a particular node of a hash tree is used as an input to the hash used to compute that node. This value is represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

q - in the LMS N-time signature scheme, each LM-OTS signature is associated with the leaf of a hash tree, and q is set to the leaf number. This ensures that a distinct value of q is used for each distinct LM-OTS public/private key pair. This value is

represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

D - a domain separation parameter, which is a two byte identifier that takes on different values in the different contexts in which the hash function is invoked. D occurs in bytes 20, 21 of the value being hashed and takes on the following values:

D_PBLC = 0x8080 when computing the hash of all of the iterates in the LM-OTS algorithm

D_MESG = 0x8181 when computing the hash of the message in the LM-OTS algorithms

D_LEAF = 0x8282 when computing the hash of the leaf of an LMS tree

D_INTR = 0x8383 when computing the hash of an interior node of an LMS tree

i - a value between 0 and 264; this is used in the LM-OTS scheme, when either computing the iterations of the Winternitz chain, or when using the suggested LM-OTS private key generation process. It is represented as a 16-bit (two-byte) unsigned integer in network byte order. If present, it occurs at bytes 20, 21 of the value being hashed.

j - in the LM-OTS scheme, j is the iteration number used when the private key element is being iteratively hashed. It is represented as an 8-bit (one byte) unsigned integer and is present if i is a value between 0 and 264. If present, it occurs at bytes 22 to 21+n of the value being hashed.

C - an n-byte randomizer that is included with the message whenever it is being hashed to improve security. C MUST be chosen uniformly at random, or via a pseudorandom process. It is present if D=D_MESG, and it occurs at bytes 22 to 21+n of the value being hashed.

8. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create two registries: one for OTS signatures, which includes all of the LM-OTS signatures as defined in Section 4, and one for Leighton-Micali Signatures, as defined in Section 5.

Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient detail that interoperability between independent implementations is possible. IANA MUST verify that all applications for additions to these registries hve first been reviewed by the IRTF Crypto Forum Research Group (CFRG).

Each entry in the registry contains the following elements:

a short name, such as "LMS_SHA256_M32_H10",

a positive number, and

a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

The numbers between 0xDDDDDDDD (decimal 3,722,304,989) and 0xFFFFFFFF (decimal 4,294,967,295) inclusive, will not be assigned by IANA, and are reserved for private use; no attempt will be made to prevent multiple sites from using the same value in different (and incompatible) ways [RFC2434].

The LM-OTS registry is as follows.

Name	Reference	Numeric Identifier
Reserved		0x00000000
LMOTS_SHA256_N32_W1	Section 4	0x00000001
LMOTS_SHA256_N32_W2	Section 4	0x00000002
LMOTS_SHA256_N32_W4	Section 4	0x00000003
LMOTS_SHA256_N32_W8	Section 4	0x00000004

Table 4

The LMS registry is as follows.

Name	Reference	Numeric Identifier
Reserved		0x0 - 0x4
LMS_SHA256_M32_H5	Section 5	0x00000005
LMS_SHA256_M32_H10	Section 5	0x00000006
LMS_SHA256_M32_H15	Section 5	0x00000007
LMS_SHA256_M32_H20	Section 5	0x00000008
LMS_SHA256_M32_H25	Section 5	0x00000009

Table 5

An IANA registration of a signature system does not constitute an endorsement of that system or its security.

The LM-OTS and the LMS registries currently occupy a disjoint set of values. This coincidence is a historical accident; the correctness of the system does not depend on this. IANA is not required to maintain this situation.

9. Security Considerations

The hash function H MUST have second preimage resistance: it must be computationally infeasible for an attacker that is given one message M to be able to find a second message M' such that H(M) = H(M').

The security goal of a signature system is to prevent forgeries. A successful forgery occurs when an attacker who does not know the private key associated with a public key can find a message (distinct from all previously signed ones) and signature that is valid with that public key (that is, the Signature Verification algorithm applied to that signature and message and public key will return VALID). Such an attacker, in the strongest case, may have the ability to forge valid signatures for an arbitrary number of other messages.

LMS is provably secure in the random oracle model, where the hash compression function is considered the random oracle, as shown by [Fluhrer17]. Corollary 1 of that paper states:

> If we have no more than 2^64 randomly chosen LMS private keys, allow the attacker access to a signing oracle and a SHA-256 hash

compression oracle, and allow a maximum of 2^120 hash compression computations, then the probability of an attacker being able to generate a single forgery against any of those LMS keys is less than 2^-129.

Many of the objects within the public key and the signature start with a typecode. A verifier MUST check each of these typecodes, and a verification operation on a signature with an unknown type, or a type that does not correspond to the type within the public key MUST return INVALID. The expected length of a variable-length object can be determined from its typecode, and if an object has a different length, then any signature computed from the object is INVALID.

9.1. Hash Formats

The format of the inputs to the hash function H have the property that each invocation of that function has an input that is repeated by a small bounded number of other inputs (due to potential repeats of the I value), and in particular, will vary somewhere in the first 23 bytes of the value being hashed. This property is important for a proof of security in the random oracle model.

The formats used during key generation and signing (including the recommended pseudorandom key generation procedure in Appendix A):

```
I || u32str(q) || u16str(i) || u8str(j) || tmp
I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1]
I || u32str(q) || u16str(D_MESG) || C || message
I || u32str(r) || u16str(D_LEAF) || OTS_PUB_HASH[r-2^h]
I || u32str(r) || u16str(D_INTR) || T[2*r] || T[2*r+1]
I || u32str(q) || u16str(i) || u8str(0xff) || SEED
```

Each hash type listed is distinct; at locations 20, 21 of the value being hashed, there exists either a fixed value D_PBLC, D_MESG, D_LEAF, D_INTR, or a 16 bit value i. These fixed values are distinct from each other, and are large (over 32768), while the 16 bit values of i are small (currently no more than 265; possibly being slightly larger if larger hash functions are supported); hence the range of possible values of i will not collide any of the D_PBLC, D_MESG, D_LEAF, D_INTR identifiers. The only other collision possibility is the Winternitz chain hash colliding with the recommended pseudorandom key generation process; here, at location 22 of the value being hashed, the Winternitz chain function has the value u8str(j), where j is a value between 0 and 254, while location 22 of the recommended pseudorandom key generation process has value 255.

For the Winternitz chaining function, D_PBLC, and D_MESG, the value of I || u32str(q) is distinct for each LMS leaf (or equivalently, for

each q value). For the Winternitz chaining function, the value of u16str(i) || u8str(j) is distinct for each invocation of H for a given leaf. For D_PBLC and D_MESG, the input format is used only once for each value of q, and thus distinctness is assured. The formats for D_INTR and D_LEAF are used exactly once for each value of r, which ensures their distinctness. For the recommended pseudorandom key generation process, for a given value of I, q and j are distinct for each invocation of H.

The value of I is chosen uniformly at random from the set of all 128 bit strings. If 2^64 public keys are generated (and hence 2^64 random I values), there is a nontrivial probability of a duplicate (which would imply duplicate prefixes). However, there will be an extremely high probability there will not be a four-way collision (that is, any I value used for four distinct LMS keys; probability < 2^-132), and hence the number of repeats for any specific prefix will be limited to at most 3. This is shown (in [Fluhrer17]) to have only a limited effect on the security of the system.

9.2. Stateful signature algorithm

The LMS signature system, like all N-time signature systems, requires that the signer maintain state across different invocations of the signing algorithm, to ensure that none of the component one-time signature systems are used more than once. This section calls out some important practical considerations around this statefulness. These issues are discussed in greater detail in [STMGMT].

In a typical computing environment, a private key will be stored in non-volatile media such as on a hard drive. Before it is used to sign a message, it will be read into an application's Random Access Memory (RAM). After a signature is generated, the value of the private key will need to be updated by writing the new value of the private key into non-volatile storage. It is essential for security that the application ensure that this value is actually written into that storage, yet there may be one or more memory caches between it and the application. Memory caching is commonly done in the file system, and in a physical memory unit on the hard disk that is dedicated to that purpose. To ensure that the updated value is written to physical media, the application may need to take several special steps. In a POSIX environment, for instance, the O_SYNC flag (for the open() system call) will cause invocations of the write() system call to block the calling process until the data has been written to the underlying hardware. However, if that hardware has its own memory cache, it must be separately dealt with using an operating system or device specific tool such as hdparm to flush the on-drive cache, or turn off write caching for that drive. Because these details vary across different operating systems and devices,

this note does not attempt to provide complete guidance; instead, we call the implementer's attention to these issues.

When hierarchical signatures are used, an easy way to minimize the private key synchronization issues is to have the private key for the second level resident in RAM only, and never write that value into non-volatile memory. A new second level public/private key pair will be generated whenever the application (re)starts; thus, failures such as a power outage or application crash are automatically accommodated. Implementations SHOULD use this approach wherever possible.

9.3. Security of LM-OTS Checksum

To show the security of LM-OTS checksum, we consider the signature y of a message with a private key x and let h = H(message) and c = Cksm(H(message)) (see Section 4.5). To attempt a forgery, an attacker may try to change the values of h and c. Let h' and c' denote the values used in the forgery attempt. If for some integer j in the range 0 to u, where u = ceil(8*n/w) is the size of the range that the checksum value can cover, inclusive,

a' = coef(h', j, w),

a = coef(h, j, w), and

a' > a

then the attacker can compute F^a'(x[j]) from F^a(x[j]) = y[j] by iteratively applying function F to the j-th term of the signature an additional (a' - a) times. However, as a result of the increased number of hashing iterations, the checksum value c' will decrease from its original value of c. Thus a valid signature's checksum will have, for some number k in the range u to (p-1), inclusive,

b' = coef(c', k, w),

b = coef(c, k, w), and

b' < b

Due to the one-way property of F, the attacker cannot easily compute F^b'(x[k]) from F^b(x[k]) = y[k].

10. Comparison with other work

The eXtended Merkle Signature Scheme (XMSS) [XMSS], [RFC8391] is similar to HSS in several ways. Both are stateful hash based signature schemes, and both use a hierarchical approach, with a Merkle tree at each level of the hierarchy. XMSS signatures are slightly shorter than HSS signatures, for equivalent security and an equal number of signatures.

HSS has several advantages over XMSS. HSS operations are roughly four times faster than the comparable XMSS ones, when SHA256 is used as the underlying hash. This occurs because the hash operation done as a part of the Winternitz iterations dominates performance, and XMSS performs four compression function invocations (two for the PRF, two for the F function) where HSS needs only perform one. Additionally, HSS is somewhat simpler (as each hash invocation is just a prefix followed by the data being hashed).

11. Acknowledgements

Thanks are due to Chirag Shroff, Andreas Huelsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, Russ Housley, Philip Lafrance, Alexander Truskovsky, Mark Peruzel for constructive suggestions and valuable detailed review. We especially acknowledge Jerry Solinas, Laurie Law, and Kevin Igoe, who pointed out the security benefits of the approach of Leighton and Micali [USPTO5432852] and Jonathan Katz, who gave us security guidance, and Bruno Couillard and Jim Goodman for an especially thorough review.

12. References

12.1. Normative References

[FIPS180] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS 180-4, March 2012.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <https://www.rfc-editor.org/info/rfc2434>.

[RFC3979] Bradner, S., Ed., "Intellectual Property Rights in IETF Technology", RFC 3979, DOI 10.17487/RFC3979, March 2005, <https://www.rfc-editor.org/info/rfc3979>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC4879] Narten, T., "Clarification of the Third Party Disclosure Procedure in RFC 3979", RFC 4879, DOI 10.17487/RFC4879, April 2007, <https://www.rfc-editor.org/info/rfc4879>.

[USPTO5432852] Leighton, T. and S. Micali, "Large provably fast and secure digital signature schemes from secure hash functions", U.S. Patent 5,432,852, July 1995.

12.2. Informative References

[C:Merkle87] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Lecture Notes in Computer Science crypto87vol, 1988.

[C:Merkle89a] Merkle, R., "A Certified Digital Signature", Lecture Notes in Computer Science crypto89vol, 1990.

[C:Merkle89b] Merkle, R., "One Way Hash Functions and DES", Lecture Notes in Computer Science crypto89vol, 1990.

[Fluhrer17] Fluhrer, S., "Further analysis of a proposed hash-based signature standard", EPrint https://eprint.iacr.org/2017/553.pdf, 2017.

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 28th ACM Symposium on the Theory of Computing p. 212, 1996.

[Katz16] Katz, J., "Analysis of a proposed hash-based signature standard", Security Standardization Research (SSR) Conference https://www.cs.umd.edu/~jkatz/papers/HashBasedSigs-SSR16.pdf, 2016.

[Merkle79] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/info/rfc8391>.

[STMGMT] McGrew, D., Fluhrer, S., Kampanakis, P., Gazdag, S., Butin, D., and J. Buchmann, "State Management for Hash-based Signatures.", Security Standardization Resarch (SSR) Conference 224., 2016.

[XMSS] Buchmann, J., Dahmen, E., and Andreas Huelsing, "XMSS-a practical forward secure signature scheme based on minimal security assumptions.", International Workshop on Post-Quantum Cryptography Springer Berlin., 2011.

Appendix A. Pseudorandom Key Generation

An implementation MAY use the following pseudorandom process for generating an LMS private key.

SEED is an m-byte value that is generated uniformly at random at the start of the process,

I is LMS key pair identifier,

q denotes the LMS leaf number of an LM-OTS private key,

x_q denotes the x array of private elements in the LM-OTS private key with leaf number q,

i is the index of the private key element, and

H is the hash function used in LM-OTS.

The elements of the LM-OTS private keys are computed as:

x_q[i] = H(I || u32str(q) || u16str(i) || u8str(0xff) || SEED).

This process stretches the m-byte random value SEED into a (much larger) set of pseudorandom values, using a unique counter in each invocation of H. The format of the inputs to H are chosen so that they are distinct from all other uses of H in LMS and LM-OTS. A careful reader will note that this is similar to the hash we perform

when iterating through the Winternitz chain; however in that chain, the iteration index will vary between 0 and 254 maximum (for W=8), while the corresponding value in this formula is 255. This algorithm is included in the proof of security in [Fluhrer17] and hence this method is safe when used within the LMS system; however any other cryptographical secure method of generating private keys would also be safe.

Appendix B. LM-OTS Parameter Options

The LM-OTS one time signature method uses several internal parameters, which are a function of the selected parameter set. These internal parameters set:

p - This is the number of independent Winternitz chains used in the signature; it will be the number of w-bit digits needed to hold the n-bit hash (u in the below equations), along with the number of digits needed to hold the checksum (v in the below equations)

ls - This is the size of the shift needed to move the checksum so that it appears in the checksum digits

ls is needed because, while we express the checksum internally as a 16 bit value, we don't always express all 16 bits in the signature; for example, if w=4, we might use only the top 12 bits. Because we read the checksum in network order, this means that, without the shift, we'll use the higher order bits (which may be always 0), and omit the lower order bits (where the checksum value actually resides). This shift is here to ensure that the parts of the checksum we need to express (for security) actually contribute to the signature; when multiple such shifts are possible, we take the minimal value.

The parameters ls, and p are computed as follows:

```
u = ceil(8*n/w)
v = ceil((floor(lg((2^w - 1) * u)) + 1) / w)
ls = 16 - (v * w)
p = u + v
```

Here u and v represent the number of w-bit fields required to contain the hash of the message and the checksum byte strings, respectively. And as the value of p is the number of w-bit elements of (H(message) || Cksm(H(message))), it is also equivalently the number of byte strings that form the private key and the number of byte strings in the signature. The value 16 in the ls computation of ls corresponds to the 16 bits value used for the sum variable in Algorithm 2 in Section 4.4

A table illustrating various combinations of n and w with the associated values of u, v, ls, and p is provided in Table 6.

Hash Length in Bytes (n)	Winternitz Parameter (w)	w-bit Elements in Hash (u)	w-bit Elements in Checksum (v)	Left Shift (ls)	Total Number of w-bit Elements (p)
32	1	256	9	7	265
32	2	128	5	6	133
32	4	64	3	4	67
32	8	32	2	0	34

Table 6

Appendix C. An iterative algorithm for computing an LMS public key

The LMS public key can be computed using the following algorithm or any equivalent method. The algorithm uses a stack of hashes for data. It also makes use of a hash function with the typical init/update/final interface to hash functions; the result of the invocations hash_init(), hash_update(N[1]), hash_update(N[2]), ... , hash_update(N[n]), v = hash_final(), in that order, is identical to that of the invocation of H(N[1] || N[2] || ... || N[n]).

Generating an LMS Public Key from an LMS Private Key

```
  for ( i = 0; i < 2^h; i = i + 1 ) {
    r = i + num_lmots_keys;
    temp = H(I || u32str(r) || u16str(D_LEAF) || OTS_PUB_HASH[i])
    j = i;
    while (j % 2 == 1) {
      r = (r - 1)/2;
      j = (j-1) / 2;
      left_side = pop(data stack);
      temp = H(I || u32str(r) || u16str(D_INTR) || left_side || temp)
    }
    push temp onto the data stack
 }
 public_key = pop(data stack)
```

Note that this pseudocode expects that all 2^h leaves of the tree have equal depth; that is, num_lmots_keys to be a power of 2. The maximum depth of the stack will be h-1 elements, that is, a total of (h-1)*n bytes; for the currently defined parameter sets, this will never be more than 768 bytes of data.

Appendix D. Method for deriving authentication path for a signature

The LMS signature consists of u32str(q) || lmots_signature || u32str(type) || path[0] || path[1] || ... || path[h-1]. This appendix shows one method of constructing this signature, assuming that the implementation has stored the T[] array that was used to construct the public key. Note that this is not the only possible method; other methods exist which don't assume that you have the entire T[] array in memory. To construct a signature, you perform the following algorithm:

Generating an LMS Signature

1. set type to the typecode of the LMS algorithm

2. extract h from the typecode according to table 2

3. create the LM-OTS signature for the message:
   ots_signature = lmots_sign(message, LMS_PRIV[q])

4. compute the array path as follows:

```
   i = 0
   r = 2^h + q
   while (i < h) {
       temp = (r / 2^i) xor 1
       path[i] = T[temp]
       i = i + 1
  }
```

5. S = u32str(q) || ots_signature || u32str(type) ||
   path[0] || path[1] || ... || path[h-1]

6. q = q + 1

7. return S

where 'xor' is the bitwise exclusive-or operation, and / is integer division (that is, rounded down to an integer value)

Appendix E. Example Implementation

An example implementation can be found online at https://github.com/cisco/hash-sigs.

Appendix F. Test Cases

This section provides test cases that can be used to verify or debug an implementation. This data is formatted with the name of the elements on the left, and the value of the elements on the right, in hexadecimal. The concatenation of all of the values within a public key or signature produces that public key or signature, and values that do not fit within a single line are listed across successive lines.

Test Case 1 Public Key

```
---------------------------------------------
HSS public key
levels      00000002
---------------------------------------------
LMS type    00000005                        # LM_SHA256_M32_H5
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
I           61a5d57d37f5e46bfb7520806b07a1b8
K           50650e3b31fe4a773ea29a07f09cf2ea
            30e579f0df58ef8e298da0434cb2b878
---------------------------------------------
---------------------------------------------
```

Test Case 1 Message

```
---------------------------------------------
Message     54686520706f77657273206e6f742064  |The powers not d|
            656c65676174656420746f2074686520  |elegated to the |
            556e697465642053746174657320627 9 |United States by|
            2074686520436f6e737469747574696f  | the Constitutio|
            6e2c206e6f722070726f686962697465  |n, nor prohibite|
            642062792069742074 6f207468652053  |d by it to the S|
            74617465732c20617265207265736572  |tates, are reser|
            76656420746f2074686520537461746 5  |ved to the State|
            732072657370656374696576 6c792c20  |s respectively, |
            6f7220746f207468652070656f706c65  |or to the people|
            2e0a                              |..|
---------------------------------------------
```

Test Case 1 Signature

```
---------------------------------------------
HSS signature
Nspk        00000001
sig[0]:
---------------------------------------------
LMS signature
q           00000005
---------------------------------------------
LMOTS signature
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
C           d32b56671d7eb98833c49b433c272586
            bc4a1c8a8970528ffa04b966f9426eb9
y[0]        965a25bfd37f196b9073f3d4a232feb6
            9128ec45146f86292f9dff9610a7bf95
y[1]        a64c7f60f6261a62043f86c70324b770
            7f5b4a8a6e19c114c7be866d488778a0
```

```
   y[2]        e05fd5c6509a6e61d559cf1a77a970de
               927d60c70d3de31a7fa0100994e162a2
   y[3]        582e8ff1b10cd99d4e8e413ef469559f
               7d7ed12c838342f9b9c96b83a4943d16
   y[4]        81d84b15357ff48ca579f19f5e71f184
               66f2bbef4bf660c2518eb20de2f66e3b
   y[5]        14784269d7d876f5d35d3fbfc7039a46
               2c716bb9f6891a7f41ad133e9e1f6d95
   y[6]        60b960e7777c52f060492f2d7c660e14
               71e07e72655562035abc9a701b473ecb
   y[7]        c3943c6b9c4f2405a3cb8bf8a691ca51
               d3f6ad2f428bab6f3a30f55dd9625563
   y[8]        f0a75ee390e385e3ae0b906961ecf41a
               e073a0590c2eb6204f44831c26dd768c
   y[9]        35b167b28ce8dc988a3748255230cef9
               9ebf14e730632f27414489808afab1d1
   y[10]       e783ed04516de012498682212b078105
               79b250365941bcc98142da13609e9768
   y[11]       aaf65de7620dabec29eb82a17fde35af
               15ad238c73f81bdb8dec2fc0e7f93270
   y[12]       1099762b37f43c4a3c20010a3d72e2f6
               06be108d310e639f09ce7286800d9ef8
   y[13]       a1a40281cc5a7ea98d2adc7c7400c2fe
               5a101552df4e3cccfd0cbf2ddf5dc677
   y[14]       9cbbc68fee0c3efe4ec22b83a2caa3e4
               8e0809a0a750b73ccdcf3c79e6580c15
   y[15]       4f8a58f7f24335eec5c5eb5e0cf01dcf
               4439424095fceb077f66ded5bec73b27
   y[16]       c5b9f64a2a9af2f07c05e99e5cf80f00
               252e39db32f6c19674f190c9fbc506d8
   y[17]       26857713afd2ca6bb85cd8c107347552
               f30575a5417816ab4db3f603f2df56fb
   y[18]       c413e7d0acd8bdd81352b2471fc1bc4f
               1ef296fea1220403466b1afe78b94f7e
   y[19]       cf7cc62fb92be14f18c2192384ebceaf
               8801afdf947f698ce9c6ceb696ed70e9
   y[20]       e87b0144417e8d7baf25eb5f70f09f01
               6fc925b4db048ab8d8cb2a661ce3b57a
   y[21]       da67571f5dd546fc22cb1f97e0ebd1a6
               5926b1234fd04f171cf469c76b884cf3
   y[22]       115cce6f792cc84e36da58960c5f1d76
               0f32c12faef477e94c92eb75625b6a37
   y[23]       1efc72d60ca5e908b3a7dd69fef02491
               50e3eebdfed39cbdc3ce9704882a2072
   y[24]       c75e13527b7a581a556168783dc1e975
               45e31865ddc46b3c957835da252bb732
   y[25]       8d3ee2062445dfb85ef8c35f8e1f3371
               af34023cef626e0af1e0bc017351aae2
```

```
   y[26]       ab8f5c612ead0b729a1d059d02bfe18e
               fa971b7300e882360a93b025ff97e9e0
   y[27]       eec0f3f3f13039a17f88b0cf808f4884
               31606cb13f9241f40f44e537d302c64a
   y[28]       4f1f4ab949b9feefadcb71ab50ef27d6
               d6ca8510f150c85fb525bf25703df720
   y[29]       9b6066f09c37280d59128d2f0f637c7d
               7d7fad4ed1c1ea04e628d221e3d8db77
   y[30]       b7c878c9411cafc5071a34a00f4cf077
               38912753dfce48f07576f0d4f94f42c6
   y[31]       d76f7ce973e9367095ba7e9a3649b7f4
               61d9f9ac1332a4d1044c96aefee67676
   y[32]       401b64457c54d65fef6500c59cdfb69a
               f7b6dddfcb0f086278dd8ad0686078df
   y[33]       b0f3f79cd893d314168648499898fbc0
               ced5f95b74e8ff14d735cdea968bee74
   ---------------------------------------------
   LMS type    00000005                        # LM_SHA256_M32_H5
   path[0]     d8b8112f9200a5e50c4a262165bd342c
               d800b8496810bc716277435ac376728d
   path[1]     129ac6eda839a6f357b5a04387c5ce97
               382a78f2a4372917eefcbf93f63bb591
   path[2]     12f5dbe400bd49e4501e859f885bf073
               6e90a509b30a26bfac8c17b5991c157e
   path[3]     b5971115aa39efd8d564a6b90282c316
               8af2d30ef89d51bf14654510a12b8a14
   path[4]     4cca1848cf7da59cc2b3d9d0692dd2a2
               0ba3863480e25b1b85ee860c62bf5136
   ---------------------------------------------
   LMS public key
   LMS type    00000005                        # LM_SHA256_M32_H5
   LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
   I           d2f14ff6346af964569f7d6cb880a1b6
   K           6c5004917da6eafe4d9ef6c6407b3db0
               e5485b122d9ebe15cda93cfec582d7ab
   ---------------------------------------------
   final_signature:
   ---------------------------------------------
   LMS signature
   q           0000000a
   ---------------------------------------------
   LMOTS signature
   LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
   C           0703c491e7558b35011ece3592eaa5da
               4d918786771233e8353bc4f62323185c
   y[0]        95cae05b899e35dffd71705470620998
               8ebfdf6e37960bb5c38d7657e8bffeef
   y[1]        9bc042da4b4525650485c66d0ce19b31
```

```
              7587c6ba4bffcc428e25d08931e72dfb
   y[2]       6a120c5612344258b85efdb7db1db9e1
              865a73caf96557eb39ed3e3f426933ac
   y[3]       9eeddb03a1d2374af7bf771855774562
              37f9de2d60113c23f846df26fa942008
   y[4]       a698994c0827d90e86d43e0df7f4bfcd
              b09b86a373b98288b7094ad81a0185ac
   y[5]       100e4f2c5fc38c003c1ab6fea479eb2f
              5ebe48f584d7159b8ada03586e65ad9c
   y[6]       969f6aecbfe44cf356888a7b15a3ff07
              4f771760b26f9c04884ee1faa329fbf4
   y[7]       e61af23aee7fa5d4d9a5dfcf43c4c26c
              e8aea2ce8a2990d7ba7b57108b47dabf
   y[8]       beadb2b25b3cacc1ac0cef346cbb90fb
              044beee4fac2603a442bdf7e507243b7
   y[9]       319c9944b1586e899d431c7f91bcccc8
              690dbf59b28386b2315f3d36ef2eaa3c
   y[10]      f30b2b51f48b71b003dfb08249484201
              043f65f5a3ef6bbd61ddfee81aca9ce6
   y[11]      0081262a00000480dcbc9a3da6fbef5c
              1c0a55e48a0e729f9184fcb1407c3152
   y[12]      9db268f6fe50032a363c9801306837fa
              fabdf957fd97eafc80dbd165e435d0e2
   y[13]      dfd836a28b354023924b6fb7e48bc0b3
              ed95eea64c2d402f4d734c8dc26f3ac5
   y[14]      91825daef01eae3c38e3328d00a77dc6
              57034f287ccb0f0e1c9a7cbdc828f627
   y[15]      205e4737b84b58376551d44c12c3c215
              c812a0970789c83de51d6ad787271963
   y[16]      327f0a5fbb6b5907dec02c9a90934af5
              a1c63b72c82653605d1dcce51596b3c2
   y[17]      b45696689f2eb382007497557692caac
              4d57b5de9f5569bc2ad0137fd47fb47e
   y[18]      664fcb6db4971f5b3e07aceda9ac130e
              9f38182de994cff192ec0e82fd6d4cb7
   y[19]      f3fe00812589b7a7ce51544045643301
              6b84a59bec6619a1c6c0b37dd1450ed4
   y[20]      f2d8b584410ceda8025f5d2d8dd0d217
              6fc1cf2cc06fa8c82bed4d944e71339e
   y[21]      ce780fd025bd41ec34ebff9d4270a322
              4e019fcb444474d482fd2dbe75efb203
   y[22]      89cc10cd600abb54c47ede93e08c114e
              db04117d714dc1d525e11bed8756192f
   y[23]      929d15462b939ff3f52f2252da2ed64d
              8fae88818b1efa2c7b08c8794fb1b214
   y[24]      aa233db3162833141ea4383f1a6f120b
              e1db82ce3630b3429114463157a64e91
   y[25]      234d475e2f79cbf05e4db6a9407d72c6
```

```
              bff7d1198b5c4d6aad2831db61274993
   y[26]      715a0182c7dc8089e32c8531deed4f74
              31c07c02195eba2ef91efb5613c37af7
   y[27]      ae0c066babc69369700e1dd26eddc0d2
              16c781d56e4ce47e3303fa73007ff7b9
   y[28]      49ef23be2aa4dbf25206fe45c20dd888
              395b2526391a724996a44156beac8082
   y[29]      12858792bf8e74cba49dee5e8812e019
              da87454bff9e847ed83db07af3137430
   y[30]      82f880a278f682c2bd0ad6887cb59f65
              2e155987d61bbf6a88d36ee93b6072e6
   y[31]      656d9ccbaae3d655852e38deb3a2dcf8
              058dc9fb6f2ab3d3b3539eb77b248a66
   y[32]      1091d05eb6e2f297774fe6053598457c
              c61908318de4b826f0fc86d4bb117d33
   y[33]      e865aa805009cc2918d9c2f840c4da43
              a703ad9f5b5806163d7161696b5a0adc
   ---------------------------------------------
   LMS type   00000005                        # LM_SHA256_M32_H5
   path[0]    d5c0d1bebb06048ed6fe2ef2c6cef305
              b3ed633941ebc8b3bec9738754cddd60
   path[1]    e1920ada52f43d055b5031cee6192520
              d6a5115514851ce7fd448d4a39fae2ab
   path[2]    2335b525f484e9b40d6a4a969394843b
              dcf6d14c48e8015e08ab92662c05c6e9
   path[3]    f90b65a7a6201689999f32bfd368e5e3
              ec9cb70ac7b8399003f175c40885081a
   path[4]    09ab3034911fe125631051df0408b394
              6b0bde790911e8978ba07dd56c73e7ee

   Test Case 2 Private Key

   ---------------------------------------------
   (note: procedure in Appendix A is used)
   SEED       000102030405060708090a0b0c0d0e0f
              101112131415161718191a1b1c1d1e1f
   I          d08fabd4a2091ff0a8cb4ed834e74534
   ---------------------------------------------
   ---------------------------------------------
```

Test Case 2 Public Key

```
-----------------------------------------
HSS public key
levels      00000002
-----------------------------------------
LMS type    00000006                    # LM_SHA256_M32_H10
LMOTS type  00000003                    # LMOTS_SHA256_N32_W4
I           d08fabd4a2091ff0a8cb4ed834e74534
K           32a58885cd9ba0431235466bff9651c6
            c92124404d45fa53cf161c28f1ad5a8e
-----------------------------------------
-----------------------------------------
```

Test Case 2 Message

```
-----------------------------------------
Message     54686520656e756d65726174696f6e20 The enumeration
            696e2074686520436f6e737469747574 in the Constitut
            696f6e2c206f66206365727461696e20 ion, of certain
            7269676874732c207368616c6c206e6f rights, shall no
            7420626520636f6e7374727565642074 t be construed t
            6f2064656e79206f7220646973706172 o deny or dispar
            616765206f7468657273207265746169 age others retai
            6e656420627920746852070656f706c ned by the peopl
            652e0a                           e..
-----------------------------------------
```

Test Case 2 Signature

```
-----------------------------------------
HSS signature
Nspk        00000001
sig[0]:
-----------------------------------------
LMS signature
q           00000003
-----------------------------------------
LMOTS signature
LMOTS type  00000003                    # LMOTS_SHA256_N32_W4
C           3d46bee8660f8f215d3f96408a7a64cf
            1c4da02b63a55f62c666ef5707a914ce
y[0]        0674e8cb7a55f0c48d484f31f3aa4af9
            719a74f22cf823b94431d01c926e2a76
y[1]        bb71226d279700ec81c9e95fb11a0d10
            d065279a5796e265ae17737c44eb8c59
y[2]        4508e126a9a7870bf4360820bdeb9a01
            d9693779e416828e75bddd7d8c70d50a
```

```
   y[3]        0ac8ba39810909d445f44cb5bb58de73
               7e60cb4345302786ef2c6b14af212ca1
   y[4]        9edeaa3bfcfe8baa6621ce88480df237
               1dd37add732c9de4ea2ce0dffa53c926
   y[5]        49a18d39a50788f4652987f226a1d481
               68205df6ae7c58e049a25d4907edc1aa
   y[6]        90da8aa5e5f7671773e941d805536021
               5c6b60dd35463cf2240a9c06d694e9cb
   y[7]        54e7b1e1bf494d0d1a28c0d31acc7516
               1f4f485dfd3cb9578e836ec2dc722f37
   y[8]        ed30872e07f2b8bd0374eb57d22c614e
               09150f6c0d8774a39a6e168211035dc5
   y[9]        2988ab46eaca9ec597fb18b4936e66ef
               2f0df26e8d1e34da28cbb3af75231372
   y[10]       0c7b345434f72d65314328bbb030d0f0
               f6d5e47b28ea91008fb11b05017705a8
   y[11]       be3b2adb83c60a54f9d1d1b2f476f9e3
               93eb5695203d2ba6ad815e6a111ea293
   y[12]       dcc21033f9453d49c8e5a6387f588b1e
               a4f706217c151e05f55a6eb7997be09d
   y[13]       56a326a32f9cba1fbe1c07bb49fa04ce
               cf9df1a1b815483c75d7a27cc88ad1b1
   y[14]       238e5ea986b53e087045723ce16187ed
               a22e33b2c70709e53251025abde89396
   y[15]       45fc8c0693e97763928f00b2e3c75af3
               942d8ddaee81b59a6f1f67efda0ef81d
   y[16]       11873b59137f67800b35e81b01563d18
               7c4a1575a1acb92d087b517a8833383f
   y[17]       05d357ef4678de0c57ff9f1b2da61dfd
               e5d88318bcdde4d9061cc75c2de3cd47
   y[18]       40dd7739ca3ef66f1930026f47d9ebaa
               713b07176f76f953e1c2e7f8f271a6ca
   y[19]       375dbfb83d719b1635a7d8a138919579
               44b1c29bb101913e166e11bd5f34186f
   y[20]       a6c0a555c9026b256a6860f4866bd6d0
               b5bf90627086c6149133f8282ce6c9b3
   y[21]       622442443d5eca959d6c14ca8389d12c
               4068b503e4e3c39b635bea245d9d05a2
   y[22]       558f249c9661c0427d2e489ca5b5dde2
               20a90333f4862aec793223c781997da9
   y[23]       8266c12c50ea28b2c438e7a379eb106e
               ca0c7fd6006e9bf612f3ea0a454ba3bd
   y[24]       b76e8027992e60de01e9094fddeb3349
               883914fb17a9621ab929d970d101e45f
   y[25]       8278c14b032bcab02bd15692d21b6c5c
               204abbf077d465553bd6eda645e6c306
   y[26]       5d33b10d518a61e15ed0f092c3222628
               1a29c8a0f50cde0a8c66236e29c2f310
```

```
   y[27]       a375cebda1dc6bb9a1a01dae6c7aba8e
               bedc6371a7d52aacb955f83bd6e4f84d
   y[28]       2949dcc198fb77c7e5cdf6040b0f84fa
               f82808bf985577f0a2acf2ec7ed7c0b0
   y[29]       ae8a270e951743ff23e0b2dd12e9c3c8
               28fb5598a22461af94d568f29240ba28
   y[30]       20c4591f71c088f96e095dd98beae456
               579ebbba36f6d9ca2613d1c26eee4d8c
   y[31]       73217ac5962b5f3147b492e8831597fd
               89b64aa7fde82e1974d2f6779504dc21
   y[32]       435eb3109350756b9fdabe1c6f368081
               bd40b27ebcb9819a75d7df8bb07bb05d
   y[33]       b1bab705a4b7e37125186339464ad8fa
               aa4f052cc1272919fde3e025bb64aa8e
   y[34]       0eb1fcbfcc25acb5f718ce4f7c2182fb
               393a1814b0e942490e52d3bca817b2b2
   y[35]       6e90d4c9b0cc38608a6cef5eb153af08
               58acc867c9922aed43bb67d7b33acc51
   y[36]       9313d28d41a5c6fe6cf3595dd5ee63f0
               a4c4065a083590b275788bee7ad875a7
   y[37]       f88dd73720708c6c6c0ecf1f43bbaada
               e6f208557fdc07bd4ed91f88ce4c0de8
   y[38]       42761c70c186bfdafafc444834bd3418
               be4253a71eaf41d718753ad07754ca3e
   y[39]       ffd5960b033698179572142680359 9ed
               5b2b7516920efcbe32ada4bcf6c73bd2
   y[40]       9e3fa152d9adeca36020fdeeee1b7395
               21d3ea8c0da497003df1513897b0f547
   y[41]       94a873670b8d93bcca2ae47e64424b74
               23e1f078d9554bb5232cc6de8aae9b83
   y[42]       fa5b9510beb39ccf4b4e1d9c0f19d5e1
               7f58e5b8705d9a6837a7d9bf99cd1338
   y[43]       7af256a8491671f1f2f22af253bcff54
               b673199bdb7d05d81064ef05f80f0153
   y[44]       d0be7919684b23da8d42ff3effdb7ca0
               985033f389181f47659138003d712b5e
   y[45]       c0a614d31cc7487f52de8664916af79c
               98456b2c94a8038083db55391e347586
   y[46]       2250274a1de2584fec975fb09536792c
               fbfcf6192856cc76eb5b13dc4709e2f7
   y[47]       301ddff26ec1b23de2d188c999166c74
               e1e14bbc15f457cf4e471ae13dcbdd9c
   y[48]       50f4d646fc6278e8fe7eb6cb5c94100f
               a870187380b777ed19d7868fd8ca7ceb
   y[49]       7fa7d5cc861c5bdac98e7495eb0a2cee
               c1924ae979f44c5390ebedddc65d6ec1
   y[50]       1287d978b8df064219bc5679f7d7b264
               a76ff272b2ac9f2f7cfc9fdcfb6a5142
```

```
   y[51]       8240027afd9d52a79b647c90c2709e06
               0ed70f87299dd798d68f4fadd3da6c51
   y[52]       d839f851f98f67840b964ebe73f8cec4
               1572538ec6bc131034ca2894eb736b3b
   y[53]       da93d9f5f6fa6f6c0f03ce43362b8414
               940355fb54d3dfdd03633ae108f3de3e
   y[54]       bc85a3ff51efeea3bc2cf27e1658f178
               9ee612c83d0f5fd56f7cd071930e2946
   y[55]       beeecaa04dccea9f97786001475e0294
               bc2852f62eb5d39bb9fbeef75916efe4
   y[56]       4a662ecae37ede27e9d6eadfdeb8f8b2
               b2dbccbf96fa6dbaf7321fb0e701f4d4
   y[57]       29c2f4dcd153a2742574126e5eaccc77
               686acf6e3ee48f423766e0fc466810a9
   y[58]       05ff5453ec99897b56bc55dd49b99114
               2f65043f2d744eeb935ba7f4ef23cf80
   y[59]       cc5a8a335d3619d781e7454826df720e
               ec82e06034c44699b5f0c44a8787752e
   y[60]       057fa3419b5bb0e25d30981e41cb1361
               322dba8f69931cf42fad3f3bce6ded5b
   y[61]       8bfc3d20a2148861b2afc14562ddd27f
               12897abf0685288dcc5c4982f8260268
   y[62]       46a24bf77e383c7aacab1ab692b29ed8
               c018a65f3dc2b87ff619a633c41b4fad
   y[63]       b1c78725c1f8f922f6009787b1964247
               df0136b1bc614ab575c59a16d089917b
   y[64]       d4a8b6f04d95c581279a139be09fcf6e
               98a470a0bceca191fce476f9370021cb
   y[65]       c05518a7efd35d89d8577c990a5e1996
               1ba16203c959c91829ba7497cffcbb4b
   y[66]       294546454fa5388a23a22e805a5ca35f
               956598848bda678615fec28afd5da61a
   ---------------------------------------------
   LMS type    00000006                        # LM_SHA256_M32_H10
   path[0]     b326493313053ced3876db9d23714818
               1b7173bc7d042cefb4dbe94d2e58cd21
   path[1]     a769db4657a103279ba8ef3a629ca84e
               e836172a9c50e51f45581741cf808315
   path[2]     0b491cb4ecbbabec128e7c81a46e62a6
               7b57640a0a78be1cbf7dd9d419a10cd8
   path[3]     686d16621a80816bfdb5bdc56211d72c
               a70b81f1117d129529a7570cf79cf52a
   path[4]     7028a48538ecdd3b38d3d5d62d262465
               95c4fb73a525a5ed2c30524ebb1d8cc8
   path[5]     2e0c19bc4977c6898ff95fd3d310b0ba
               e71696cef93c6a552456bf96e9d075e3
   path[6]     83bb7543c675842bafbfc7cdb88483b3
               276c29d4f0a341c2d406e40d4653b7e4
```

```
   path[7]     d045851acf6a0a0ea9c710b805cced46
               35ee8c107362f0fc8d80c14d0ac49c51
   path[8]     6703d26d14752f34c1c0d2c4247581c1
               8c2cf4de48e9ce949be7c888e9caebe4
   path[9]     a415e291fd107d21dc1f084b11582082
               49f28f4f7c7e931ba7b3bd0d824a4570
   ---------------------------------------------
   LMS public key
   LMS type    00000005                        # LM_SHA256_M32_H5
   LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
   I           215f83b7ccb9acbcd08db97b0d04dc2b
   K           a1cd035833e0e90059603f26e07ad2aa
               d152338e7a5e5984bcd5f7bb4eba40b7
   ---------------------------------------------
   final_signature:
   ---------------------------------------------
   LMS signature
   q           00000004
   ---------------------------------------------
   LMOTS signature
   LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
   C           0eb1ed54a2460d512388cad533138d24
               0534e97b1e82d33bd927d201dfc24ebb
   y[0]        11b3649023696f85150b189e50c00e98
               850ac343a77b3638319c347d7310269d
   y[1]        3b7714fa406b8c35b021d54d4fdada7b
               9ce5d4ba5b06719e72aaf58c5aae7aca
   y[2]        057aa0e2e74e7dcfd17a0823429db629
               65b7d563c57b4cec942cc865e29c1dad
   y[3]        83cac8b4d61aacc457f336e6a10b6632
               3f5887bf3523dfcadee158503bfaa89d
   y[4]        c6bf59daa82afd2b5ebb2a9ca6572a60
               67cee7c327e9039b3b6ea6a1edc7fdc3
   y[5]        df927aade10c1c9f2d5ff446450d2a39
               98d0f9f6202b5e07c3f97d2458c69d3c
   y[6]        8190643978d7a7f4d64e97e3f1c4a08a
               7c5bc03fd55682c017e2907eab07e5bb
   y[7]        2f190143475a6043d5e6d5263471f4ee
               cf6e2575fbc6ff37edfa249d6cda1a09
   y[8]        f797fd5a3cd53a066700f45863f04b6c
               8a58cfd341241e002d0d2c0217472bf1
   y[9]        8b636ae547c1771368d9f317835c9b0e
               f430b3df4034f6af00d0da44f4af7800
   y[10]       bc7a5cf8a5abdb12dc718b559b74cab9
               090e33cc58a955300981c420c4da8ffd
   y[11]       67df540890a062fe40dba8b2c1c548ce
               d22473219c534911d48ccaabfb71bc71
   y[12]       862f4a24ebd376d288fd4e6fb06ed870
```

```
              5787c5fedc813cd2697e5b1aac1ced45
   y[13]      767b14ce88409eaebb601a93559aae89
              3e143d1c395bc326da821d79a9ed41dc
   y[14]      fbe549147f71c092f4f3ac522b5cc572
              90706650487bae9bb5671ecc9ccc2ce5
   y[15]      1ead87ac01985268521222fb9057df7e
              d41810b5ef0d4f7cc67368c90f573b1a
   y[16]      c2ce956c365ed38e893ce7b2fae15d36
              85a3df2fa3d4cc098fa57dd60d2c9754
   y[17]      a8ade980ad0f93f6787075c3f680a2ba
              1936a8c61d1af52ab7e21f416be09d2a
   y[18]      8d64c3d3d8582968c2839902229f85ae
              e297e717c094c8df4a23bb5db658dd37
   y[19]      7bf0f4ff3ffd8fba5e383a48574802ed
              545bbe7a6b4753533353d73706067640
   y[20]      135a7ce517279cd683039747d218647c
              86e097b0daa2872d54b8f3e508598762
   y[21]      9547b830d8118161b65079fe7bc59a99
              e9c3c7380e3e70b7138fe5d9be255150
   y[22]      2b698d09ae193972f27d40f38dea264a
              0126e637d74ae4c92a6249fa103436d3
   y[23]      eb0d4029ac712bfc7a5eacbdd7518d6d
              4fe903a5ae65527cd65bb0d4e9925ca2
   y[24]      4fd7214dc617c150544e423f450c99ce
              51ac8005d33acd74f1bed3b17b7266a4
   y[25]      a3bb86da7eba80b101e15cb79de9a207
              852cf91249ef480619ff2af8cabca831
   y[26]      25d1faa94cbb0a03a906f683b3f47a97
              c871fd513e510a7a25f283b196075778
   y[27]      496152a91c2bf9da76ebe089f4654877
              f2d586ae7149c406e663eadeb2b5c7e8
   y[28]      2429b9e8cb4834c83464f079995332e4
              b3c8f5a72bb4b8c6f74b0d45dc6c1f79
   y[29]      952c0b7420df525e37c15377b5f09843
              19c3993921e5ccd97e097592064530d3
   y[30]      3de3afad5733cbe7703c5296263f7734
              2efbf5a04755b0b3c997c4328463e84c
   y[31]      aa2de3ffdcd297baaaacd7ae646e44b5
              c0f16044df38fabd296a47b3a838a913
   y[32]      982fb2e370c078edb042c84db34ce36b
              46ccb76460a690cc86c302457dd1cde1
   y[33]      97ec8075e82b393d542075134e2a17ee
              70a5e187075d03ae3c853cff60729ba4
   ---------------------------------------------
   LMS type   00000005                        # LM_SHA256_M32_H5
   path[0]    4de1f6965bdabc676c5a4dc7c35f97f8
              2cb0e31c68d04f1dad96314ff09e6b3d
   path[1]    e96aeee300d1f68bf1bca9fc58e40323
```

```
              36cd819aaf578744e50d1357a0e42867
   path[2]    04d341aa0a337b19fe4bc43c2e79964d
              4f351089f2e0e41c7c43ae0d49e7f404
   path[3]    b0f75be80ea3af098c9752420a8ac0ea
              2bbb1f4eeba05238aef0d8ce63f0c6e5
   path[4]    e4041d95398a6f7f3e0ee97cc1591849
              d4ed236338b147abde9f51ef9fd4e1c1
```

Authors' Addresses

David McGrew
Cisco Systems
13600 Dulles Technology Drive
Herndon, VA 20171
USA

Email: mcgrew@cisco.com

Michael Curcio
Cisco Systems
7025-2 Kit Creek Road
Research Triangle Park, NC 27709-4987
USA

Email: micurcio@cisco.com

Scott Fluhrer
Cisco Systems
170 West Tasman Drive
San Jose, CA
USA

Email: sfluhrer@cisco.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 24, 2017

S. Smyshlyaev, Ed.
E. Alekseev
I. Oshkin
L. Ahmetzyanova
E. Smyshlyaeva
CryptoPro

October 21, 2016

The ACPKM internal re-keying mechanism for block cipher modes of operation
draft-smyshlyaev-re-keying-00

Abstract

This specification presents an approach to increase the security of block cipher operation modes based on re-keying (with no additional keys needed) during each separate message processing. It provides an internal re-keying mechanism called ACPKM. This mechanism doesn't require additional secret parameters or complicated transforms - for key update only the base encryption function is used.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 24, 2017.

Copyright Notice

Table of Contents

1. Introduction

An important problem related to secure functioning of any cryptographic system is the control of key lifetimes. Regarding symmetric keys, the main concern is constraining the key exposure. It could be done by limiting the maximal amount of data processed with one key. The restrictions can come either from combinatorial properties of the used cipher modes of operation (for example, birthday attack [BDJR]) or from particular cryptographic attacks on the used block cipher (for example, linear cryptanalysis [Matsui]). Moreover, most strict restrictions here follow from the need to resist side-channel attacks. The adversary's opportunity to obtain an essential amount of data processed with a single key leads not only to theoretic but also to real vulnerabilities (see [BL]). Therefore, when the total size of a plaintext processed with the same key reaches threshold values, this key cannot be used anymore and certain procedures on encryption keys are needed. It leads to several operating limitations, e.g. the impossibility to process long messages and processing overhead caused by derivation of additional keys.

This specification presents a mechanism to increase the key lifetime, which is called ACPKM. This solution ("key meshing") transforms the key value each time when the given amount of data, precisely the amount of plaintext section (not the total amount of separate messages), is processed and proceeds with a new transformed key value for a new plaintext section. Such a transformation does not require any additional secret values. It is integrated into the base mode of operation and can be considered as it's extension, therefore it is called "internal re-keying" in this document.

This approach seems to be mostly useful in the case when the total amount of data for an established key is not known beforehand: the performance on useless operations won't be lost if the data size is rather small, and the security won't be lacked when it occurs to be large. The transformed keys are computed only when they are needed.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Basic Terms and Definitions

This document uses the following terms and definitions for the sets and operations on the elements of these sets:

(xor) exclusive-or of two binary vectors of the same length.

V* the set of all strings of a finite length (hereinafter referred to as strings), including the empty string;

V_s the set of all binary strings of length s, where s is a non-negative integer; substrings and string components are enumerated from right to left starting from one;

|X| the bit length of the bit string X;

A|B concatenation of strings A and B both belonging to V*, i.e., a string in V_{|A|+|B|}, where the left substring in V_|A| is equal to A, and the right substring in V_|B| is equal to B;

Z_{2^n} ring of residues modulo 2^n;

Int_s: V_s -> Z_{2^s} the transformation that maps a string a = (a_s, ...,a_1), a in V_s, into the integer Int_s(a) = 2^s*a_s + ... + 2*a_2 + a_1;

Vec_s: Z_{2^s} -> V_s the transformation inverse to the mapping Int_s;

MSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ...,a_1) in V_s, into the string MSB_i(a) = (a_s, ...,a_{s-i+1}) in V_i;

LSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ...,a_1) in V_s, into the string LSB_i(a) = (a_i, ...,a_1) in V_i;

Inc_c: V_s -> V_s the transformation that maps the string a = (a_s, ...,a_1) in V_s, into the string Inc_c(a) = MSB_{|a|-c}(a) | Vec_c(Int_c(LSB_c(a)) + 1(mod 2^c)) in V_s;

0^s denotes the string a in V_s that consists of s '0' bits;

E_K: V_n -> V_n the block cipher permutation under the key K in V_k;

k the key K size (in bits);

n the block size of the block cipher (in bits);

b the total number of data blocks in the plaintext;

N the section size (the number of bits in a data section);

l the number of data sections in the plaintext;

m the message M size (in bits);

phi_i: V_s -> V_s the transformation that maps a string a = (a_s, ...,a_1) into the string phi_i(a) = a' = (a'_s, ...,a'_1), 1 <= i <= s, such that a'_i = 1 and a'_j = a_j for all j in {1,...,s}/{i};

ceil(x) the least integer that is not less than x.

4. CTR and GCM Block Cipher Modes

This section describes the families of block cipher modes of operations that are extended by the ACPKM re-keying mechanisms as described in Section 5.

A plaintext message P and a ciphertext C are divided into b = ceil(m/n) parts (denoted as P = P_1 | P_2 |...| P_b and C = C_1 | C_2 |...| C_b, where P_i and C_i are in V_n, for i = 1, 2, ..., b-1, and P_b, C_b are in V_r, where r <= n).

4.1. CTR Block Cipher Mode

The Counter (CTR) mode is a block cipher mode of operation that applies the block cipher transformation E_K to a sequence of input blocks, called counters, to produce a sequence of output blocks that are XORed with a plaintext to produce a ciphertext, and vice versa. It is defined similar to the one specified in [NIST-CTR].

The ACPKM-CTR re-keying mechanisms described in Section 5.1 can be used with the following block cipher and CTR mode parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 32 <= c <= 3/4 n.

In the current document, the counters for a given message are denoted as CTR_1, CTR_2, ..., CTR_b.

The CTR encryption mode is defined as follows:

Input:
    Initial counter nonce ICN in V_{n-c},
    plaintext P, |P| < n*2^c.

Output:
    Ciphertext C.

CTR Encryption:
    1. CTR_1 = ICN | 0^c.
    2. For j = 1, 2, ..., b-1 do
           CTR_{j+1} = Inc_c(CTR_j).
    3. For j = 1, 2, ..., b do
           G_j = E_K(CTR_j).
    4. C = P (xor) MSB_{|P|}(G_1 |...|G_b).
    5. Return C.

The CTR decryption mode is defined as follows:

```
Input:
    Initial counter nonce ICN in V_{n-c},
    ciphertext C, |C| < n*2^c.

Output:
    Plaintext P.
```

```
CTR Decryption:
    1. CTR_1 = ICN | 0^c.
    2. For j = 1, 2,..., b-1 do
           CTR_{j+1} = Inc_c(CTR_j).
    3. For j = 1, 2, ..., b do
           G_j = E_K(CTR_j)
    4. P = C (xor) MSB_{|C|}(G_1 |...|G_b).
    5. Return P.
```

The initial counter nonce ICN value for each message that is encrypted under the given key must be chosen in a unique manner.

4.2. GCM Block Cipher Mode

TODO: This section describes the family of block cipher modes of operation with both encryption and authentication. It is defined similar to the one specified in [NIST-GCM].

The ACPKM-GCM re-keying mechanisms described in Section 5.2 can be used with the following GCM block cipher mode parameters:

- 128 <= n <= 512;
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 32 <= c <= 3/4 n.

4.2.1. GCM Subprocedures

This section presents three mathematical algorithms that appear in the specification of the authenticated encryption and authenticated decryption functions of the GCM cipher mode described in Section 4.2.2 below.

4.2.1.1. Multiplication Operation on Blocks

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements and is defined by a particular GCM mode.

4.2.1.2. GHASH Function

```
Algorithm 2: GHASH_H(X)
=======================
Input:
    Bit string X = X_1 |...| X_m, where X_i in V_n for i in 1,...,m.
Output:
    Block GHASHH (X) in V_n
_____________________________________________________________________
1. Y_0 = 0^n.
2. For i = 1,..., m do
       Y_i = (Y_{i-1} (xor) X_i)*H.
3. Return Y_m.
```

4.2.1.3. GCTR Function

```
Algorithm 3: GCTR_K(ICB, X)
===========================
Input:
    Initial counter block ICB;
    X = X_1 |...| X_t, X_i in V_n for i = 1,...,n-1 and X_n in V_r,
    where r <= n.
Output:
    Y in V_{|X|}.
_____________________________________________________________________
1. If X in V_0 then return Y, where Y in V_0.
2. t = ceil(|X|/n).
3. GCTR_1 = ICB.
4. For i = 2,...,t do
       GCTR_i = Inc_c(GCTR_{i-1}).
5. For i = 1,...,t do
       G_i = E_K(GCTR_i).
6. Y = X (xor) MSB_{|X|}(G_1 |...| G_t).
7. Return Y.
```

4.2.2. GCM Mode Description

The GCM encryption mode is defined as follows:

```
Input:
    Initialization vector IV in V_{n-c},
    plaintext P, |P| < n*(2^c - 2).
    additional authenticated data A.
Output:
    Ciphertext C,
    authentication tag T.
```

```
GCM Encryption:
    1. H = E_K(0^n).
    2. if c = 32, then J_0 = IV | 0^31 | 1;
       if c!= 32, then s = n*ceil(|IV|/n)-|IV|,
                       J_0 = GHASH_H(IV | 0^{s+n-64} | Vec_64(|IV|)).
    3. C = GCTR_K(Inc_32(J_0),P).
    4. u = n*ceil(|C|/n)-|C|,
       v = n*ceil(|A|/n)-|A|.
    5. S = GHASH_H(A | 0^v | C | 0^u | 0^{n-128} |
                   |Vec_64(|A|) | Vec_64(|C|)).
    6. T = MSB_t(E_K(J_0) (xor) S).
    7. Return C | T.
```

The GCM decryption mode is defined as follows:

```
Input:
    Initialization vector IV in V_{n-c},
    ciphertext C, |C| < n*(2^c - 2),
    authentication tag T,
    additional authenticated data A.
Output:
    Plaintext P or FAIL.
```

```
GCM decryption:
    1. H = E_K(0^n).
    2. if c = 32, then J_0 = IV | 0^31 | 1;
       if c!= 32, then s = n*ceil(|IV|/n)-|IV|,
                       J_0 = GHASH_H(IV | 0^{s+n-64} | Vec_64(|IV|)).
    3. P = GCTR_K(Inc_32(J_0),C).
    4. u = n*ceil(|C|/n)-|C|,
       v = n*ceil(|A|/n)-|A|.
    5. S = GHASH_H(A | 0^v | C | 0^u | 0^{n-128}|
                   |Vec_64(|A|) | Vec_64(|C|)).
    6. T' = MSB_t(E_K(J_0) (xor) S).
    7. IF T=T' then return P; else return FAIL.
```

The initial vector IV value for each message that is encrypted under the given key must be chosen in a unique manner.

N o t e : The encryption part in the GCM-ACPKM mode is the encryption in the CTR-ACPKM mode with several differences: in the CTR mode the counter for the plaintext encryption starts with the first CTR_1 value and in the GCM mode the counter starts with the second GCTR_2 value.

5. ACPKM re-keying mechanisms

This section defines periodical key transformations for long message processing that are considered as extensions of the basic CTR and GCM encryption modes and are called ACPKM-CTR and ACPKM-GCM re-keying mechanisms.

An additional parameter that defines the functioning of CTR and GCM block cipher modes with the ACPKM key transformation algorithm is the section size N. The value of N is fixed within a specific protocol based on the requirements of the system capacity and key lifetime (some recommendations on choosing N will be provided in Section 7). The section size N MUST be divisible by the block size n.

The main idea behind internal re-keying is presented in Fig.1:

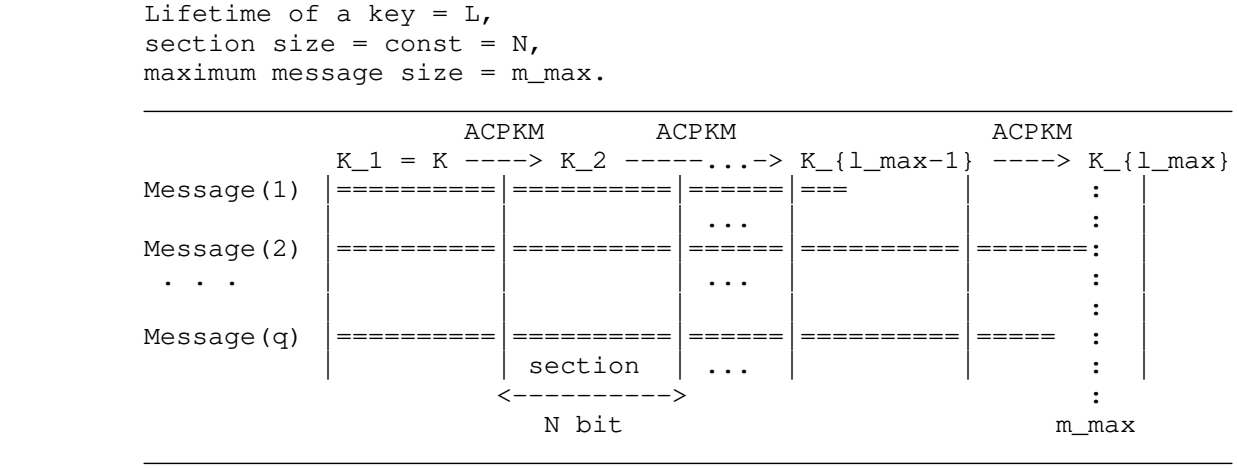


Figure 1: Key meshing approach

For the {i+1}-th section the K_{i+1} value is calculated as follows:

$$K_{i+1} = \text{ACPKM-CTR}(K_i) = \text{MSB}_k(E_{K_i}(W_1)|...|E_{K_i}(W_J)),$$

where J = ceil(k/n), W_t = phi_c(D_t) for any t in {1,...,J} and D_1, D_2,...,D_J are in V_n and are calculated as follows:

D_1 | D_2 |...| D_J = MSB_{J*n}(D),

where D is the following constant in V_1024:

```
D = ( F3 | 74 | E9 | 23 | FE | AA | D6 | DD
    | 98 | B4 | B6 | 3D | 57 | 8B | 35 | AC
    | A9 | 0F | D7 | 31 | E4 | 1D | 64 | 5E
    | 40 | 8C | 87 | 87 | 28 | CC | 76 | 90
    | 37 | 76 | 49 | 9F | 7D | F3 | 3B | 06
    | 92 | 21 | 7B | 06 | 37 | BA | 9F | B4
    | F2 | 71 | 90 | 3F | 3C | F6 | FD | 1D
    | 70 | BB | BB | 88 | E7 | F4 | 1B | 76
    | 7E | 44 | F9 | 0E | 46 | 91 | 5B | 57
    | 00 | BC | 13 | 45 | BE | 0D | BD | C7
    | 61 | 38 | 19 | 3C | 41 | 30 | 86 | 82
    | 1A | A0 | 45 | 79 | 23 | 4C | 4C | F3
    | 64 | F2 | 6A | CC | EA | 48 | CB | B4
    | 0C | B9 | A9 | 28 | C3 | B9 | 65 | CD
    | 9A | CA | 60 | FB | 9C | A4 | 62 | C7
    | 22 | C0 | 6C | E2 | 4A | C7 | FB | 5B).
```

N o t e : The constant D is such that phi_c(D_1),..., phi_c(D_J) are pairwise different for any allowed n, k, c values.

5.1. ACPKM internal re-keying mechanism for CTR encryption mode

This section defines a ACPKM-CTR internal re-keying mechanism for the CTR encryption mode that was described in Section 4.1.

During the processing of the input message M with the length m using ACPKM-CTR internal re-keying algorithm and the key K the message is divided into l = ceil(m*N) parts (denoted as M = M_1 | M_2 |...| M_l, where M_i is in V_N for i = 1, 2,..., l-1 and M_l is in V_r, r <= N). The first section is processed with the initial key K_1 = K. To process the (i+1)-th section the K_{i+1} key value is calculated using ACPKM-CTR transformation of the key K_i. The counter value (CTR_{i+1}) is not changed during this process.

The message size m MUST NOT exceed n*2^{c-1} bits.

5.2. ACPKM internal re-keying mechanism for GCM encryption mode

This section defines a ACPKM-GCM internal re-keying mechanism for the GCM encryption mode that was described in Section 4.2.

During the processing of the input message M with the length m using ACPKM-GCM internal re-keying algorithm and the key K the message is divided into l = ceil(m/N) parts (denoted as M = M_1 | M_2 |...| M_l,

where M_i is in V_N for i = 1, 2,..., l-1 and M_l is in V_r, r <= N). The first section is processed with the initial key K_1 = K. To process the (i+1)-th section the K_{i+1} key value is calculated using ACPKM-GCM transformation of the key K_i.

The message size m MUST NOT exceed n*(2^{c-1}-2) bits.

The key for computing values E_K(J_0) and H is not updated and is equal to the initial key.

6. Acknowledgments

TODO

7. Security Considerations

The ACPKM re-keying mechanisms provide the CTR and GCM encryption modes extensions that have the following property: a compromise of a key of some section does not lead to a compromise of previous keys but leads to a compromise of next keys.

The ACPKM mechanism allows to increase the CTR and GCM encryption modes security in proportion to the frequency of key changing, which is inversely related to the section size N. Thus, the key lifetime can be noticeably increased: an amount of material that is processed with the key K increases quadratically, divided by N.

Since the performance of encryption can slightly decrease for rather small values of N, the parameter of N SHOULD be selected for a particular protocol as maximum possible to provide necessary key lifetime for the adversary models that are considered.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

[BDJR] Bellare M., Desai A., Jokipii E., Rogaway P., "A concrete security treatment of symmetric encryption", In Proceedings of 38th Annual Symposium on Foundations of Computer Science (FOCS '97), pages 394-403. 97, 1997.

[BL] Bhargavan K., Leurent G., "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Cryptology ePrint Archive Report 798, 2016.

[Matsui] Matsui M., "Linear Cryptanalysis Method for DES Cipher", Advanced in Cryptology- EUROCRYPT'93. Lect. Notes in Comp. Sci., Springer. V.765.P. 386-397, 1994.

[NIST-CTR] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, December 2001.

[NIST-GCM] McGrew, D. and J. Viega, "The Galois/Counter Mode of Operation (GCM)", Submission to NIST http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf, January 2004.

Appendix A. Test examples

TODO

Authors' Addresses

Stanislav Smyshlyaev (editor)
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: svs@cryptopro.ru

Evgeny Alekseev
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: alekseev@cryptopro.ru

Igor Oshkin
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: oshkin@cryptopro.ru

Lilia Ahmetzyanova
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: lah@cryptopro.ru

Ekaterina Smyshlyaeva
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: ess@cryptopro.ru