HS1-SIV (v2)

Submitted and designed by
Ted Krovetz[1]
ted@krovetz.net
July 27, 2015

HS1-SIV uses a new PRF called HS1 to provide authenticated encryption via Rogaway and Shrimpton's SIV mode [7]. HS1 (mnemonic for "Hash-Stream 1") is designed for high software speed on systems with good 32-bit processing, including Intel SSE, ARM Neon, and 32-bit architectures without SIMD support. HS1 uses a universal hash function to consume arbitrary strings and a stream cipher to produce its output.

This document defines HS1 and how to use it in an SIV construction. HS1 takes an arbitrary input string and IV and produces a pseudorandom string of any desired length. Each different (input, IV) pair supplied to HS1 yields an independent pseudorandom stream with high probability. SIV, as defined in [7], uses a block-cipher-based PRF to create a synthetic IV (an SIV) from given associated data and plaintext. The SIV is then used to encrypt the plaintext using a block-cipher-based encryption scheme. HS1-SIV instead uses HS1 to instantiate SIV mode. Ignoring many details for the moment, if A is the associated data, M is the plaintext, and N is HS1's IV, then the SIV is defined as the first 16 bytes of HS1($A||M$, N) and ciphertext C is defined as all but the first 16 bytes of HS1(SIV, N) xor'ed with M. The SIV and ciphertext are bundled together to create the final ciphertext. If (A, M, N) is repeated, then an observer knows this fact because the scheme is deterministic and the SIV will be identical, but no security degradation otherwise occurs. Supplying N as a nonce thus improves security by masking repeated encryptions.

HS1 does its work by pairing an almost-universal hash function with a stream cipher. When given an (input, IV) pair, HS1 uses the hash function to hash the input, it then xor's this hash result with the stream cipher's key and uses the HS1 IV as the stream cipher's IV. The stream cipher produces as many bytes as desired. As long as a (hash result, IV) pair is never repeated, and the stream cipher is secure against related-key attacks, the stream cipher will produce independent pseudorandom output streams. We introduce a new hash HS1-Hash which we use for the almost-universal phase of HS1, and Bernstein's Chacha is used as the stream cipher [1].

1 Specification

The figures in this document fully specify HS1-SIV with the exception of the Chacha stream cipher. We use the Chacha version specified in *ChaCha20 and Poly1305 for IETF protocols* [6], and attach it to make this document self-contained. The interface defined in that document has Chacha take four inputs: A 32-byte key, an initial counter value, a 12-byte IV, and a plaintext. The ciphertext produced is the same length as the plaintext and is pseudorandom. This document adopts that interface. $\mathsf{Chacha}[r](K, c, N, X)$ indicates the r-round Chacha encryption of X using key K, initial counter value c, and IV N. When we simply want an n-byte pseudorandom string, we let X be n zero bytes.

[1] Department of Computer Science, California State University, Sacramento CA 95819. This work supported by the generous support of NSF grants CNS-1314592 and CNS-0904380.

HS1-SIV-Encrypt$[b, t, r, \ell](K, M, A, N)$
Inputs:
  K, a non-empty string of up to 32 bytes
  M, a string shorter than 2^{64} bytes
  A, a string shorter than 2^{64} bytes
  N, a 12-byte string
Output:
  (T, C), strings of ℓ and $|M|$ bytes, respectively
Algorithm:
1. $\mathbf{k} = \mathsf{HS1\text{-}subkeygen}[b, t, r, \ell](K)$
2. $M' = \mathsf{pad}(b, A) \,||\, \mathsf{pad}(16, M) \,||\, \mathsf{toStr}(8, |A|) \,||\, \mathsf{toStr}(8, |M|)$
3. $T = \mathsf{HS1}[b, t, r](\mathbf{k}, M', N, \ell)$
4. $C = M \oplus \mathsf{HS1}[b, t, r](\mathbf{k}, T, N, 64 + |M|)[64, |M|]$

Figure 1: Encryption. *The ℓ-byte string T serves as authenticator for A and M, and serves as IV for the encryption of M. If N is a nonce, then repeat encryptions yield different T and C. Algorithm parameters b, t, r, and ℓ effect security and performance.*

1.1 Parameters

There are four parameters used throughout this specification, b, t, r, ℓ. Parameter b specifies the number of bytes used in part of the hashing algorithm (larger b tends to produce higher throughput on longer messages). Parameter t selects the collision level of the hashing algorithm (higher t produces higher security and lower throughput). Parameter r specifies the number of internal rounds used by the stream cipher (higher r produces higher security and lower throughput). Parameter ℓ specifies the byte-length of synthetic IV used (higher ℓ improves security and increases ciphertext lengths by ℓ bytes). The following table names parameter sets.

Name	b	t	r	ℓ
`hs1-siv-lo`	64	2	8	8
`hs1-siv`	64	4	12	16
`hs1-siv-hi`	64	6	20	32

1.2 Notation

The algorithms in this document manipulate integers, vectors of integers, and strings of bytes. Vector and string indices begin with zero. If $\mathbf{v}$ is a vector, then $\mathbf{v}[a]$ is its a-th element and $\mathbf{v}[a, n]$ is the n-element subvector beginning at index a: $(\mathbf{v}[a], \mathbf{v}[a+1], \ldots, \mathbf{v}[a+n-1])$. Similarly, if S is a string, then $S[a, n]$ is the n-byte substring of S beginning at index a. The number of elements in $\mathbf{v}$ and bytes in S is indicated $|\mathbf{v}|$ and $|S|$. Concatenation is accomplished using "$||$" (eg, $(1, 2, 3)||(4, 5) = (1, 2, 3, 4, 5)$ and `0xf0`$||$`0xa8` = `0xf0a8`). The bitwise exclusive-or of same-length strings A and B is $A \oplus B$. A string of k zero-bytes is represented 0^k. $\mathsf{pad}(n, S) = S||0^k$ where k is the smallest non-negative integer making the length of $S||0^k$ a non-negative multiple of n bytes. $\mathsf{toStr}(n, x)$ is the n-byte unsigned little-endian binary representation of integer x (eg, $\mathsf{toStr}(2, 3) =$ `0x0300`). $\mathsf{toInts}(n, S)$ is the vector of integers obtained by breaking S into n-byte chunks and little-endian interpreting each chunk as an unsigned integer (eg, $\mathsf{toInts}(2,$ `0x05000600`$) = (5, 6)$).

$\mathsf{HS1}[b,t,r](\mathbf{k},M,N,y)$
Inputs:
$\mathbf{k}$, a vector $(K_S,\mathbf{k}_N,\mathbf{k}_P,\mathbf{k}_A)$, where
K_S, is a string of 32 bytes,
$\mathbf{k}_N$, is a vector of $b/4+4(t-1)$ integers from $\mathbb{Z}_{2^{32}}$,
$\mathbf{k}_P$, is a vector of t integers from $\mathbb{Z}_{2^{60}}$, and
$\mathbf{k}_A$, is a vector of $3t$ integers from $\mathbb{Z}_{2^{64}}$
M, a string of any length
N, a 12-byte string
y, an integer in $\mathbb{Z}_{2^{38}}$
Output:
Y, a string of y bytes
Algorithm:
1. $A_i = \mathsf{HS1\text{-}Hash}[b,t](\mathbf{k}_N[4i,b/4],\mathbf{k}_P[i],\mathbf{k}_A[3i,3],M)$ for each $0 \le i < t$
2. $Y = \mathsf{Chacha}[r](\mathsf{pad}(32, A_0 \mid\mid A_1 \mid\mid \ldots \mid\mid A_{t-1}) \oplus K_s), 0, N, 0^y)$

Figure 2: HS1 PRF. *Hash M a total of t times with different keys and combine the result with the stream cipher's key.*

Given vectors of integers $\mathbf{v}_1$ and $\mathbf{v}_2$, $\mathsf{NH}(\mathbf{v}_1,\mathbf{v}_2) = \sum_{i=1}^{n/4}((\mathbf{v}_1[4i-3]+\mathbf{v}_2[4i-3]) \times (\mathbf{v}_1[4i-1]+\mathbf{v}_2[4i-1]) + (\mathbf{v}_1[4i-2]+\mathbf{v}_2[4i-2]) \times (\mathbf{v}_1[4i]+\mathbf{v}_2[4i]))$ where $n = \min(|\mathbf{v}_1|,|\mathbf{v}_2|)$ and is always a multiple of four.

2 Security Goals

HS1-SIV provides confidentiality of plaintexts and integrity of ciphertexts, associated data and public message numbers. No private message numbers are employed by HS1-SIV. If we say an adversary can win an attack against HS1-SIV by (i) guessing the key in use or (ii) causing two different encryptions to use the same synthetic IV, then the following table summarizes an adversary's probability of success over n encryptions, each of no more than 2^{32} bytes, or n key guesses.

Name	Key Search	SIV Collision
`hs1-siv-lo`	$n/2^{256}$	$n^2/2^{56} + n^2/2^{64}$
`hs1-siv`	$n/2^{256}$	$n^2/2^{112} + n^2/2^{128}$
`hs1-siv-hi`	$n/2^{256}$	$n^2/2^{168} + n^2/2^{256}$

SIV is designed to provide the maximum possible robustness against nonce reuse. HS1-SIV maintains full integrity and confidentiality, except for leaking collisions of (plaintext, associated data, public message number) via collisions of ciphertexts. If two different (plaintext, associated data, public message number) triples produce the same SIV, then forgeries become possible.

$\mathsf{HS1\text{-}Hash}[b,t](\mathbf{k}_N, k_P, \mathbf{k}_A, M)$
Inputs:
$\mathbf{k}_N$, is a vector of $b/4$ integers from $\mathbb{Z}_{2^{32}}$,
k_P, is an integer from $\mathbb{Z}_{2^{60}}$
$\mathbf{k}_A$, is a vector of 3 integers from $\mathbb{Z}_{2^{64}}$ (Not used when $t \le 4$)
M, a string of any length
Output:
Y, an 8 byte (if $t \le 4$) or 4 byte (if $t > 4$) string
Algorithm:
1. $n = \max(\lceil |M|/b \rceil, 1)$
2. Let $M_1, M_2, \ldots, M_n$ be strings so that $M_1 || M_2 || \cdots || M_n = M$ and $|M_i| = b$ for each $1 \le i < n$.
3. $\mathbf{m}_i = \mathsf{toInts}(4, \mathsf{pad}(16, M_i))$ for each $1 \le i \le n$
4. $a_i = (\mathsf{NH}(\mathbf{k}_N, \mathbf{m}_i) + |M_i| \textbf{ mod } 16) \textbf{ mod } 2^{60}$ for each $1 \le i \le n$
5. $h = k_P^n + a_1 k_P^{n-1} + a_2 k_P^{n-2} + \ldots + a_n k_P^0 \textbf{ mod } (2^{61} - 1)$
6. **if** $(t \le 4)$ $Y = \mathsf{toStr}(8, h)$
7. **else** $Y = \mathsf{toStr}(4, (\mathbf{k}_A[0] + \mathbf{k}_A[1] \times (h \textbf{ mod } 2^{32}) + \mathbf{k}_A[2] \times (h \textbf{ div } 2^{32})) \textbf{ div } 2^{32})$

Figure 3: *The hash family HS1-Hash is* $(1/2^{28} + \ell/b2^{60})$*-AU for all* M *up to* ℓ *bytes, when* $\mathbf{k}_N$ *and* k_P *are chosen randomly and* $t \le 4$*. The hash family is* $(1/2^{28} + 1/2^{32} + \ell/b2^{60})$*-SU when* $\mathbf{k}_A$ *is also randomly chosen and* $t > 4$ *(the extra* $1/2^{32}$ *coming from Line 7, a strongly universal hash developed by Lemire [5]).*

3 Security Analysis

HS1 is a composition of HS1-Hash, an almost-universal hash function, with Chacha, a stream cipher. HS1-Hash is itself a composition of two hashes. Similar to techniques used in UMAC and VMAC [3, 4], the NH hash is used to reduce the input by a fixed ratio to an intermediate string which is then hashed to a fixed size by a polynomial evaluation. It uses well-known and studied techniques. The NH hash hashes inputs of length bm bytes to ones of length $8m$ bytes with a collision probability of no more than 2^{-28}. The resulting $8m$-byte string is hashed using a standard polynomial hash modulo the prime $2^{61} - 1$. Because the polynomial's key is chosen over a restricted set (for performance reasons), the final collision probability is no more than $m2^{-60} + 2^{-28}$. To reduce the chance of collision, this hashing procedure is repeated t times, with different keys, for an ultimate collision probability of no more than $(m2^{-60} + 2^{-28})^t$.

We conjecture Chacha is secure against related-key attacks: an adversary with reasonably limited resources would have difficulty distinguishing between $f(X, N) = \mathsf{Chacha}(X \oplus K, 0, N, 0^\ell)$ and a randomly chosen function with the same signature $\{0,1\}^{256} \times \{0,1\}^{96} \to \{0,1\}^\ell$ when K is a high-entropy 256-bit string unknown to the adversary and ℓ is any positive number. This conjecture is supported by numerous statements by Bernstein and the way Chacha and Salsa "cores" are deployed in Chacha, Salsa and Rumba. Bernstein's statements have been directed toward Salsa, Chacha's predecessor, but Chacha is simply Salsa with a couple of small improvements, and it is widely believed that Chacha inherits all of Salsa's positive security features. Bernstein's statements and uses are consistent with the belief that the Salsa and Chacha cores are simply pseudorandom functions mapping 48-byte inputs to 64-byte outputs. If this is true, then the conjecture that Chacha is secure against related-key attacks is immediate. In *Salsa20 security*

HS1-Subkeygen$[b, t, r, \ell](K)$

Inputs:

K, a non-empty string of up to 32 bytes

Output:

$(K_S, \mathbf{k}_N, \mathbf{k}_P, \mathbf{k}_A)$, where

K_S, is a 32-byte string,

$\mathbf{k}_N$, is a vector of $b/4 + 4(t-1)$ integers from $\mathbb{Z}_{2^{32}}$,

$\mathbf{k}_P$, is a vector of t integers from $\mathbb{Z}_{2^{60}}$, and

$\mathbf{k}_A$, is a vector of $3t$ integers from $\mathbb{Z}_{2^{64}}$

Algorithm:

1. $\mathit{ChachaLen} = 32$
2. $\mathit{NHLen} = b + 16(t-1)$
3. $\mathit{PolyLen} = 8t$
4. $\mathit{ASULen} = 24t$
5. $y = \mathit{ChachaLen} + \mathit{NHLen} + \mathit{PolyLen} + \mathit{ASULen}$
6. $K' = (K||K||K||K||\ldots)[0, 32]$
7. $N = \mathsf{toStr}(12, b2^{48} + t2^{40} + r2^{32} + \ell 2^{16} + |K|)$
8. $T = \mathsf{Chacha}[r](K', 0, N, 0^y)$
9. $K_S = T[0, \mathit{ChachaLen}]$
10. $\mathbf{k}_N = \mathsf{toInts}(4, T[\mathit{ChachaLen}, \mathit{NHLen}])$
11. $\mathbf{k}_P = \mathsf{map}(\mathbf{mod}\ 2^{60}, \mathsf{toInts}(8, T[\mathit{ChachaLen} + \mathit{NHLen}, \mathit{PolyLen}]))$
12. $\mathbf{k}_A = \mathsf{toInts}(8, T[\mathit{ChachaLen} + \mathit{NHLen} + \mathit{PolyLen}, \mathit{ASULen}])$

Figure 4: *HS1-Subkeygen takes any length key and uses Chacha to produce all internal keys needed by HS1.*

Bernstein writes: “The reader might guess that Salsa is highly resistant to related-key attacks.” In *Response to “On the Salsa core function”* Bernstein claims that the core is designed “to eliminate all visible structure”. In the Rumba compression function, adversaries are allowed to provide any selected 48-byte inputs to the cores, and in Salsa and Chacha the cores are used simply in counter mode to produce their pseudorandom streams.

HS1 is essentially defined $\text{HS1}(K, h, X, N, \ell) = \mathsf{Chacha}(h(X) \oplus K, 0, N, 0^{\ell})$ where K is a high-entropy key and h is from HS1-Hash. As long as $(h(X) \oplus K, N)$ values are distinct, Chacha will always produce independent pseudorandom streams. HS1 is designed to make it unlikely that $(h(X) \oplus K, N)$ pairs ever repeat. A user supplied nonce is used for N with each message, and even if this facility is misused by giving the same nonce repeatedly, $h(X)$ will be distinct with high probability over a sequence of distinct X values.

4 Features

HS1-SIV is designed to have the following features.

Competitive speed on multiple architectures. HS1-SIV is designed to exploit 32-bit multiplication and SIMD processing, which are well-supported on almost all current CPUs. This ensures a consistent performance profile over a wide range of processors, including modern embedded ones.

$\mathsf{HS1\text{-}SIV\text{-}Decrypt}[b,t,r,\ell](K,(T,C),A,N)$
Inputs:
K, a non-empty string of up to 32 bytes
(T,C), an ℓ-byte string and a string shorter than 2^{64} bytes, respectively
A, a string shorter than 2^{64} bytes
N, a 12-byte string
Output:
M, a $|C|$-byte string, or AuthenticationError
Algorithm:
1. $\mathbf{k} = \mathsf{HS1\text{-}subkeygen}[b,t,r,\ell](K)$
4. $M = C \oplus \mathsf{HS1}[b,t,r](\mathbf{k},T,N,64+|C|)[64,|C|]$
2. $M' = \mathsf{pad}(b,A) \mathbin{||} \mathsf{pad}(16,M) \mathbin{||} \mathsf{toStr}(8,|A|) \mathbin{||} \mathsf{toStr}(8,|M|)$
3. $T' = \mathsf{HS1}[b,t,r](\mathbf{k},M',N,\ell)$
4. **if** $(T = T')$ **then return** M
4. **else return** AuthenticationError

Figure 5: Decryption. *The ℓ-byte string T serves as authenticator for A and M, and serves as IV for the decryption of C. Algorithm parameters b, t, r, and ℓ effect security and performance.*

Provable security. HS1-Hash and SIV are based on well-known and proven constructions [4, 7]. The only security assumption needed is that the Chacha stream cipher is a good pseudorandom generator and secure against related-key attacks.

Nonce misuse resistant. No harm is done when a nonce is reused, except that it is revealed whether corresponding (associated data, plaintext) pairs have been repeated. If (associated data, plaintext) pairs are known to never repeat, no nonce need be used at all.

Scalable. Different security levels are available for different tasks, with varying throughput.

General-purpose PRF. The general nature of HS1 makes it useful for a variety of tasks, such as entropy harvesting, random generation, and other IV-based encryption and authentication schemes. A single software library could provide multiple services under a single key by simply partitioning the nonce space and providing access to HS1.

With the exception of provable security, all of the above features are improvements over GCM.

5 Design Rationale

HS1-SIV is designed to exploit a variable-input-length/variable-output-length PRF. This abstraction allows the PRF and its application to be designed separately. Although the PRF could be used in several different ways to realize authenticated encryption, SIV is a natural fit because its definition requires variable input-lengths to be consumed, variable output lengths to be produced, and the intermediate value–the synthetic IV–to be exposed. Furthermore, SIV enjoys provable security, allowing the security focus of HS1-SIV to be entirely on HS1.

HS1 is conceived as a fast almost-universal hash function composed with a fast stream cipher, with little integration overhead. HS1-Hash is heavily influence by VMAC [4], which targets beefy CPUs, but with the assumption that 32-bit multiplication–with or without SIMD acceleration–will be the maximum-size available efficient multiplication. This allows good performance on a wide range of CPU's; from modern embedded CPU's (the 43 thousand gate ARM Cortex-M3 has 32-bit multiplication with a 64-bit result) to modern Intel processors (Intel's current AVX instruction set can process four 32-bit multiplies in a single instruction). The result is a hash that performs nearly as well on 32-bit CPUs as 64-bit ones. Bernstein's Chacha is chosen as the stream cipher for three reasons: it is one of the fastest stream ciphers in software on our target systems, its key setup is simple and fast, and it is secure against related-key attacks.

The designer has not hidden any weaknesses in this cipher.

6 Intellectual Property

The submitter knows of no known patents, patent applications, planned patent applications, or other intellectual-property constraints relevant to use of HS1-SIV. If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

7 Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

8 Changes

Version 1 mistakenly defined Prod and listed *bit*lengths in the Table of Section 1.1. Version 2 replaces Prod with NH, gives *byte*lengths in the table of Section 1.1, and increases the zero-padding on associated data from a multiple of 16 bytes to a multiple of b bytes to accommodate static AD and to simplify message/associated-data bundling.

References

[1] Daniel Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers. ECRYPT, 2008. Also available at `http://cr.yp.to/chacha.html`.

[2] Daniel Bernstein. The Salsa20 family of stream ciphers. In: New stream cipher designs: the eSTREAM finalists. Springer 2008. Also available at `http://cr.yp.to/snuffle.html`

[3] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz and Phillip Rogaway. UMAC: Fast and secure message authentication. Advances in Cryptology (CRYPTO 1999), Springer, 1999. Also available at `http://krovetz.net/csus`.

[4] Ted Krovetz. Message authentication on 64-bit architectures. Selected Areas of Cryptography (SAC 2006), Springer, 2007. Also available at `http://krovetz.net/csus`.

[5] Daniel Lemire and Owen Kaser. Strongly universal string hashing is fast. Computer Journal 57(11), 2014.

[6] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. RFC 7539, IETF, 2015. `https://www.rfc-editor.org/info/rfc7539`.

[7] Phillip Rogaway and Tom Shrimpton. Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Keywrap Problem. EUROCRYPT 2006. Springer, 2006. Also available at `http://www.cs.ucdavis.edu/~rogaway/papers`.

Internet Research Task Force (IRTF)
Request for Comments: 7539
Category: Informational
ISSN: 2070-1721
May 2015

Y. Nir
Check Point
A. Langley
Google, Inc.

ChaCha20 and Poly1305 for IETF Protocols

Abstract

This document defines the ChaCha20 stream cipher as well as the use of the Poly1305 authenticator, both as stand-alone algorithms and as a "combined mode", or Authenticated Encryption with Associated Data (AEAD) algorithm.

This document does not introduce any new crypto, but is meant to serve as a stable reference and an implementation guide. It is a product of the Crypto Forum Research Group (CFRG).

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7539.

Copyright Notice

Table of Contents

1. Introduction

The Advanced Encryption Standard (AES -- [FIPS-197]) has become the gold standard in encryption. Its efficient design, widespread implementation, and hardware support allow for high performance in many areas. On most modern platforms, AES is anywhere from four to ten times as fast as the previous most-used cipher, Triple Data Encryption Standard (3DES -- [SP800-67]), which makes it not only the best choice, but the only practical choice.

There are several problems with this. If future advances in cryptanalysis reveal a weakness in AES, users will be in an unenviable position. With the only other widely supported cipher being the much slower 3DES, it is not feasible to reconfigure deployments to use 3DES. [Standby-Cipher] describes this issue and the need for a standby cipher in greater detail. Another problem is that while AES is very fast on dedicated hardware, its performance on platforms that lack such hardware is considerably lower. Yet another problem is that many AES implementations are vulnerable to cache-collision timing attacks ([Cache-Collisions]).

This document provides a definition and implementation guide for three algorithms:

1. The ChaCha20 cipher. This is a high-speed cipher first described in [ChaCha]. It is considerably faster than AES in software-only implementations, making it around three times as fast on platforms that lack specialized AES hardware. See Appendix B for some hard numbers. ChaCha20 is also not sensitive to timing attacks (see the security considerations in Section 4). This algorithm is described in Section 2.4

2. The Poly1305 authenticator. This is a high-speed message authentication code. Implementation is also straightforward and easy to get right. The algorithm is described in Section 2.5.

3. The CHACHA20-POLY1305 Authenticated Encryption with Associated Data (AEAD) construction, described in Section 2.8.

This document does not introduce these new algorithms for the first time. They have been defined in scientific papers by D. J. Bernstein, which are referenced by this document. The purpose of this document is to serve as a stable reference for IETF documents making use of these algorithms.

These algorithms have undergone rigorous analysis. Several papers discuss the security of Salsa and ChaCha ([LatinDances], [LatinDances2], [Zhenqing2012]).

This document represents the consensus of the Crypto Forum Research Group (CFRG).

1.1. 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].

The description of the ChaCha algorithm will at various time refer to the ChaCha state as a "vector" or as a "matrix". This follows the use of these terms in Professor Bernstein's paper. The matrix notation is more visually convenient and gives a better notion as to why some rounds are called "column rounds" while others are called "diagonal rounds". Here's a diagram of how the matrices relate to vectors (using the C language convention of zero being the index origin).

```
 0   1   2   3
 4   5   6   7
 8   9  10  11
12  13  14  15
```

The elements in this vector or matrix are 32-bit unsigned integers.

The algorithm name is "ChaCha". "ChaCha20" is a specific instance where 20 "rounds" (or 80 quarter rounds -- see Section 2.1) are used. Other variations are defined, with 8 or 12 rounds, but in this document we only describe the 20-round ChaCha, so the names "ChaCha" and "ChaCha20" will be used interchangeably.

2. The Algorithms

The subsections below describe the algorithms used and the AEAD construction.

2.1. The ChaCha Quarter Round

The basic operation of the ChaCha algorithm is the quarter round. It operates on four 32-bit unsigned integers, denoted a, b, c, and d. The operation is as follows (in C-like notation):

```
1.  a += b; d ^= a; d <<<= 16;
2.  c += d; b ^= c; b <<<= 12;
3.  a += b; d ^= a; d <<<= 8;
4.  c += d; b ^= c; b <<<= 7;
```

Where "+" denotes integer addition modulo 2^32, "^" denotes a bitwise Exclusive OR (XOR), and "<<< n" denotes an n-bit left rotation (towards the high bits).

For example, let's see the add, XOR, and roll operations from the fourth line with sample numbers:

- a = 0x11111111
- b = 0x01020304
- c = 0x77777777
- d = 0x01234567
- c = c + d = 0x77777777 + 0x01234567 = 0x789abcde
- b = b ^ c = 0x01020304 ^ 0x789abcde = 0x7998bfda
- b = b <<< 7 = 0x7998bfda <<< 7 = 0xcc5fed3c

2.1.1. Test Vector for the ChaCha Quarter Round

For a test vector, we will use the same numbers as in the example, adding something random for c.

- a = 0x11111111
- b = 0x01020304
- c = 0x9b8d6f43
- d = 0x01234567

After running a Quarter Round on these four numbers, we get these:

- a = 0xea2a92f4
- b = 0xcb1cf8ce
- c = 0x4581472e
- d = 0x5881c4bb

2.2. A Quarter Round on the ChaCha State

The ChaCha state does not have four integer numbers: it has 16. So the quarter-round operation works on only four of them -- hence the name. Each quarter round operates on four predetermined numbers in the ChaCha state. We will denote by QUARTERROUND(x,y,z,w) a quarter-round operation on the numbers at indices x, y, z, and w of the ChaCha state when viewed as a vector. For example, if we apply QUARTERROUND(1,5,9,13) to a state, this means running the quarter-round operation on the elements marked with an asterisk, while leaving the others alone:

```
 0  *a   2   3
 4  *b   6   7
 8  *c  10  11
12  *d  14  15
```

Note that this run of quarter round is part of what is called a "column round".

2.2.1. Test Vector for the Quarter Round on the ChaCha State

For a test vector, we will use a ChaCha state that was generated randomly:

Sample ChaCha State

```
    879531e0  c5ecf37d  516461b1  c9a62f8a
    44c20ef3  3390af7f  d9fc690b  2a5f714c
    53372767  b00a5631  974c541a  359e9963
    5c971061  3d631689  2098d9d6  91dbd320
```

We will apply the QUARTERROUND(2,7,8,13) operation to this state. For obvious reasons, this one is part of what is called a "diagonal round":

After applying QUARTERROUND(2,7,8,13)

```
    879531e0  c5ecf37d *bdb886dc  c9a62f8a
    44c20ef3  3390af7f  d9fc690b *cfacafd2
   *e46bea80  b00a5631  974c541a  359e9963
    5c971061 *ccc07c79  2098d9d6  91dbd320
```

Note that only the numbers in positions 2, 7, 8, and 13 changed.

2.3. The ChaCha20 Block Function

The ChaCha block function transforms a ChaCha state by running multiple quarter rounds.

The inputs to ChaCha20 are:

- A 256-bit key, treated as a concatenation of eight 32-bit little-endian integers.
- A 96-bit nonce, treated as a concatenation of three 32-bit little-endian integers.
- A 32-bit block count parameter, treated as a 32-bit little-endian integer.

The output is 64 random-looking bytes.

The ChaCha algorithm described here uses a 256-bit key. The original algorithm also specified 128-bit keys and 8- and 12-round variants, but these are out of scope for this document. In this section, we describe the ChaCha block function.

Note also that the original ChaCha had a 64-bit nonce and 64-bit block count. We have modified this here to be more consistent with recommendations in Section 3.2 of [RFC5116]. This limits the use of a single (key,nonce) combination to 2^32 blocks, or 256 GB, but that is enough for most uses. In cases where a single key is used by multiple senders, it is important to make sure that they don't use the same nonces. This can be assured by partitioning the nonce space so that the first 32 bits are unique per sender, while the other 64 bits come from a counter.

The ChaCha20 state is initialized as follows:

- o The first four words (0-3) are constants: 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.

- o The next eight words (4-11) are taken from the 256-bit key by reading the bytes in little-endian order, in 4-byte chunks.

- o Word 12 is a block counter. Since each block is 64-byte, a 32-bit word is enough for 256 gigabytes of data.

- o Words 13-15 are a nonce, which should not be repeated for the same key. The 13th word is the first 32 bits of the input nonce taken as a little-endian integer, while the 15th word is the last 32 bits.

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

c=constant k=key b=blockcount n=nonce

ChaCha20 runs 20 rounds, alternating between "column rounds" and "diagonal rounds". Each round consists of four quarter-rounds, and they are run as follows. Quarter rounds 1-4 are part of a "column" round, while 5-8 are part of a "diagonal" round:

```
1.  QUARTERROUND ( 0, 4, 8,12)
2.  QUARTERROUND ( 1, 5, 9,13)
3.  QUARTERROUND ( 2, 6,10,14)
4.  QUARTERROUND ( 3, 7,11,15)
5.  QUARTERROUND ( 0, 5,10,15)
```

6. QUARTERROUND (1, 6,11,12)
7. QUARTERROUND (2, 7, 8,13)
8. QUARTERROUND (3, 4, 9,14)

At the end of 20 rounds (or 10 iterations of the above list), we add the original input words to the output words, and serialize the result by sequencing the words one-by-one in little-endian order.

Note: "addition" in the above paragraph is done modulo 2^32. In some machine languages, this is called carryless addition on a 32-bit word.

2.3.1. The ChaCha20 Block Function in Pseudocode

Note: This section and a few others contain pseudocode for the algorithm explained in a previous section. Every effort was made for the pseudocode to accurately reflect the algorithm as described in the preceding section. If a conflict is still present, the textual explanation and the test vectors are normative.

```
inner_block (state):
   Qround(state, 0, 4, 8,12)
   Qround(state, 1, 5, 9,13)
   Qround(state, 2, 6,10,14)
   Qround(state, 3, 7,11,15)
   Qround(state, 0, 5,10,15)
   Qround(state, 1, 6,11,12)
   Qround(state, 2, 7, 8,13)
   Qround(state, 3, 4, 9,14)
   end

chacha20_block(key, counter, nonce):
   state = constants | key | counter | nonce
   working_state = state
   for i=1 upto 10
      inner_block(working_state)
      end
   state += working_state
   return serialize(state)
   end
```

2.3.2. Test Vector for the ChaCha20 Block Function

For a test vector, we will use the following inputs to the ChaCha20 block function:

- o Key = 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f. The key is a sequence of octets with no particular structure before we copy it into the ChaCha state.

- o Nonce = (00:00:00:09:00:00:00:4a:00:00:00:00)

- o Block Count = 1.

After setting up the ChaCha state, it looks like this:

ChaCha state with the key setup.

```
61707865  3320646e  79622d32  6b206574
03020100  07060504  0b0a0908  0f0e0d0c
13121110  17161514  1b1a1918  1f1e1d1c
00000001  09000000  4a000000  00000000
```

After running 20 rounds (10 column rounds interleaved with 10 "diagonal rounds"), the ChaCha state looks like this:

ChaCha state after 20 rounds

```
837778ab  e238d763  a67ae21e  5950bb2f
c4f2d0c7  fc62bb2f  8fa018fc  3f5ec7b7
335271c2  f29489f3  eabda8fc  82e46ebd
d19c12b4  b04e16de  9e83d0cb  4e3c50a2
```

Finally, we add the original state to the result (simple vector or matrix addition), giving this:

ChaCha state at the end of the ChaCha20 operation

```
e4e7f110  15593bd1  1fdd0f50  c47120a3
c7f4d1c7  0368c033  9aaa2204  4e6cd4c3
466482d2  09aa9f07  05d7c214  a2028bd9
d19c12b5  b94e16de  e883d0cb  4e3c50a2
```

After we serialize the state, we get this:

```
Serialized Block:
000  10 f1 e7 e4 d1 3b 59 15 50 0f dd 1f a3 20 71 c4  .....;Y.P.... q.
016  c7 d1 f4 c7 33 c0 68 03 04 22 aa 9a c3 d4 6c 4e  ....3.h.."....lN
032  d2 82 64 46 07 9f aa 09 14 c2 d7 05 d9 8b 02 a2  ..dF............
048  b5 12 9c d1 de 16 4e b9 cb d0 83 e8 a2 50 3c 4e  ......N......P<N
```

2.4. The ChaCha20 Encryption Algorithm

ChaCha20 is a stream cipher designed by D. J. Bernstein. It is a refinement of the Salsa20 algorithm, and it uses a 256-bit key.

ChaCha20 successively calls the ChaCha20 block function, with the same key and nonce, and with successively increasing block counter parameters. ChaCha20 then serializes the resulting state by writing the numbers in little-endian order, creating a keystream block.

Concatenating the keystream blocks from the successive blocks forms a keystream. The ChaCha20 function then performs an XOR of this keystream with the plaintext. Alternatively, each keystream block can be XORed with a plaintext block before proceeding to create the next block, saving some memory. There is no requirement for the plaintext to be an integral multiple of 512 bits. If there is extra keystream from the last block, it is discarded. Specific protocols MAY require that the plaintext and ciphertext have certain length. Such protocols need to specify how the plaintext is padded and how much padding it receives.

The inputs to ChaCha20 are:

- o A 256-bit key
- o A 32-bit initial counter. This can be set to any number, but will usually be zero or one. It makes sense to use one if we use the zero block for something else, such as generating a one-time authenticator key as part of an AEAD algorithm.
- o A 96-bit nonce. In some protocols, this is known as the Initialization Vector.
- o An arbitrary-length plaintext

The output is an encrypted message, or "ciphertext", of the same length.

Decryption is done in the same way. The ChaCha20 block function is used to expand the key into a keystream, which is XORed with the ciphertext giving back the plaintext.

2.4.1. The ChaCha20 Encryption Algorithm in Pseudocode

```
chacha20_encrypt(key, counter, nonce, plaintext):
   for j = 0 upto floor(len(plaintext)/64)-1
      key_stream = chacha20_block(key, counter+j, nonce)
      block = plaintext[(j*64)..(j*64+63)]
      encrypted_message +=  block ^ key_stream
      end
   if ((len(plaintext) % 64) != 0)
      j = floor(len(plaintext)/64)
      key_stream = chacha20_block(key, counter+j, nonce)
      block = plaintext[(j*64)..len(plaintext)-1]
      encrypted_message += (block^key_stream)[0..len(plaintext)%64]
      end
   return encrypted_message
   end
```

2.4.2. Example and Test Vector for the ChaCha20 Cipher

For a test vector, we will use the following inputs to the ChaCha20 block function:

- o Key = 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f.

- o Nonce = (00:00:00:00:00:00:00:4a:00:00:00:00).

- o Initial Counter = 1.

We use the following for the plaintext. It was chosen to be long enough to require more than one block, but not so long that it would make this example cumbersome (so, less than 3 blocks):

```
Plaintext Sunscreen:
000  4c 61 64 69 65 73 20 61 6e 64 20 47 65 6e 74 6c  Ladies and Gentl
016  65 6d 65 6e 20 6f 66 20 74 68 65 20 63 6c 61 73  emen of the clas
032  73 20 6f 66 20 27 39 39 3a 20 49 66 20 49 20 63  s of '99: If I c
048  6f 75 6c 64 20 6f 66 66 65 72 20 79 6f 75 20 6f  ould offer you o
064  6e 6c 79 20 6f 6e 65 20 74 69 70 20 66 6f 72 20  nly one tip for
080  74 68 65 20 66 75 74 75 72 65 2c 20 73 75 6e 73  the future, suns
096  63 72 65 65 6e 20 77 6f 75 6c 64 20 62 65 20 69  creen would be i
112  74 2e                                            t.
```

The following figure shows four ChaCha state matrices:

1. First block as it is set up.

2. Second block as it is set up. Note that these blocks are only two bits apart -- only the counter in position 12 is different.

3. Third block is the first block after the ChaCha20 block operation.

4. Final block is the second block after the ChaCha20 block operation was applied.

After that, we show the keystream.

```
First block setup:
    61707865  3320646e  79622d32  6b206574
    03020100  07060504  0b0a0908  0f0e0d0c
    13121110  17161514  1b1a1918  1f1e1d1c
    00000001  00000000  4a000000  00000000

Second block setup:
    61707865  3320646e  79622d32  6b206574
    03020100  07060504  0b0a0908  0f0e0d0c
    13121110  17161514  1b1a1918  1f1e1d1c
    00000002  00000000  4a000000  00000000

First block after block operation:
    f3514f22  e1d91b40  6f27de2f  ed1d63b8
    821f138c  e2062c3d  ecca4f7e  78cff39e
    a30a3b8a  920a6072  cd7479b5  34932bed
    40ba4c79  cd343ec6  4c2c21ea  b7417df0

Second block after block operation:
    9f74a669  410f633f  28feca22  7ec44dec
    6d34d426  738cb970  3ac5e9f3  45590cc4
    da6e8b39  892c831a  cdea67c1  2b7e1d90
    037463f3  a11a2073  e8bcfb88  edc49139

Keystream:
22:4f:51:f3:40:1b:d9:e1:2f:de:27:6f:b8:63:1d:ed:8c:13:1f:82:3d:2c:06
e2:7e:4f:ca:ec:9e:f3:cf:78:8a:3b:0a:a3:72:60:0a:92:b5:79:74:cd:ed:2b
93:34:79:4c:ba:40:c6:3e:34:cd:ea:21:2c:4c:f0:7d:41:b7:69:a6:74:9f:3f
63:0f:41:22:ca:fe:28:ec:4d:c4:7e:26:d4:34:6d:70:b9:8c:73:f3:e9:c5:3a
c4:0c:59:45:39:8b:6e:da:1a:83:2c:89:c1:67:ea:cd:90:1d:7e:2b:f3:63
```

Finally, we XOR the keystream with the plaintext, yielding the ciphertext:

```
Ciphertext Sunscreen:
000  6e 2e 35 9a 25 68 f9 80 41 ba 07 28 dd 0d 69 81  n.5.%h..A..(..i.
016  e9 7e 7a ec 1d 43 60 c2 0a 27 af cc fd 9f ae 0b  .~z..C`..'......
032  f9 1b 65 c5 52 47 33 ab 8f 59 3d ab cd 62 b3 57  ..e.RG3..Y=..b.W
048  16 39 d6 24 e6 51 52 ab 8f 53 0c 35 9f 08 61 d8  .9.$.QR..S.5..a.
064  07 ca 0d bf 50 0d 6a 61 56 a3 8e 08 8a 22 b6 5e  ....P.jaV....".^
080  52 bc 51 4d 16 cc f8 06 81 8c e9 1a b7 79 37 36  R.QM.........y76
096  5a f9 0b bf 74 a3 5b e6 b4 0b 8e ed f2 78 5e 42  Z...t.[......x^B
112  87 4d                                            .M
```

2.5. The Poly1305 Algorithm

Poly1305 is a one-time authenticator designed by D. J. Bernstein. Poly1305 takes a 32-byte one-time key and a message and produces a 16-byte tag. This tag is used to authenticate the message.

The original article ([Poly1305]) is titled "The Poly1305-AES message-authentication code", and the MAC function there requires a 128-bit AES key, a 128-bit "additional key", and a 128-bit (non-secret) nonce. AES is used there for encrypting the nonce, so as to get a unique (and secret) 128-bit string, but as the paper states, "There is nothing special about AES here. One can replace AES with an arbitrary keyed function from an arbitrary set of nonces to 16-byte strings."

Regardless of how the key is generated, the key is partitioned into two parts, called "r" and "s". The pair (r,s) should be unique, and MUST be unpredictable for each invocation (that is why it was originally obtained by encrypting a nonce), while "r" MAY be constant, but needs to be modified as follows before being used: ("r" is treated as a 16-octet little-endian number):

- o r[3], r[7], r[11], and r[15] are required to have their top four bits clear (be smaller than 16)

- o r[4], r[8], and r[12] are required to have their bottom two bits clear (be divisible by 4)

The following sample code clamps "r" to be appropriate:

```
/*
Adapted from poly1305aes_test_clamp.c version 20050207
D. J. Bernstein
Public domain.
*/

#include "poly1305aes_test.h"

void poly1305aes_test_clamp(unsigned char r[16])
{
  r[3] &= 15;
  r[7] &= 15;
  r[11] &= 15;
  r[15] &= 15;
  r[4] &= 252;
  r[8] &= 252;
  r[12] &= 252;
}
```

The "s" should be unpredictable, but it is perfectly acceptable to generate both "r" and "s" uniquely each time. Because each of them is 128 bits, pseudorandomly generating them (see Section 2.6) is also acceptable.

The inputs to Poly1305 are:

- A 256-bit one-time key

- An arbitrary length message

The output is a 128-bit tag.

First, the "r" value should be clamped.

Next, set the constant prime "P" be 2^130-5: 3fffffffffffffffffffffffffffffffb. Also set a variable "accumulator" to zero.

Next, divide the message into 16-byte blocks. The last one might be shorter:

- Read the block as a little-endian number.

- Add one bit beyond the number of octets. For a 16-byte block, this is equivalent to adding 2^128 to the number. For the shorter block, it can be 2^120, 2^112, or any power of two that is evenly divisible by 8, all the way down to 2^8.

- If the block is not 17 bytes long (the last block), pad it with zeros. This is meaningless if you are treating the blocks as numbers.

- Add this number to the accumulator.

- Multiply by "r".

- Set the accumulator to the result modulo p. To summarize: Acc = ((Acc+block)*r) % p.

Finally, the value of the secret key "s" is added to the accumulator, and the 128 least significant bits are serialized in little-endian order to form the tag.

2.5.1. The Poly1305 Algorithms in Pseudocode

```
clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
poly1305_mac(msg, key):
   r = (le_bytes_to_num(key[0..15])
   clamp(r)
   s = le_num(key[16..31])
   accumulator = 0
   p = (1<<130)-5
   for i=1 upto ceil(msg length in bytes / 16)
      n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
      a += n
      a = (r * a) % p
      end
   a += s
   return num_to_16_le_bytes(a)
   end
```

2.5.2. Poly1305 Example and Test Vector

For our example, we will dispense with generating the one-time key using AES, and assume that we got the following keying material:

- Key Material: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8:01:0 3:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b

- s as an octet string: 01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b

- s as a 128-bit number: 1bf54941aff6bf4afdb20dfb8a800301

- r before clamping: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8

- Clamped r as a number: 806d5400e52447c036d555408bed685

For our message, we'll use a short text:

```
Message to be Authenticated:
000  43 72 79 70 74 6f 67 72 61 70 68 69 63 20 46 6f  Cryptographic Fo
016  72 75 6d 20 52 65 73 65 61 72 63 68 20 47 72 6f  rum Research Gro
032  75 70                                            up
```

Since Poly1305 works in 16-byte chunks, the 34-byte message divides into three blocks. In the following calculation, "Acc" denotes the accumulator and "Block" the current block:

Block #1

```
Acc = 00
Block = 6f4620636968706172676f7470797243
Block with 0x01 byte = 016f4620636968706172676f7470797243
Acc + block = 016f4620636968706172676f7470797243
(Acc+Block) * r =
     b83fe991ca66800489155dcd69e8426ba2779453994ac90ed284034da565ecf
Acc = ((Acc+Block)*r) % P = 2c88c77849d64ae9147ddeb88e69c83fc
```

Block #2

```
Acc = 2c88c77849d64ae9147ddeb88e69c83fc
Block = 6f7247206863726165736552206d7572
Block with 0x01 byte = 016f7247206863726165736552206d7572
Acc + block = 437febea505c820f2ad5150db0709f96e
(Acc+Block) * r =
     21dcc992d0c659ba4036f65bb7f88562ae59b32c2b3b8f7efc8b00f78e548a26
Acc = ((Acc+Block)*r) % P = 2d8adaf23b0337fa7cccfb4ea344b30de
```

Last Block

```
Acc = 2d8adaf23b0337fa7cccfb4ea344b30de
Block = 7075
Block with 0x01 byte = 017075
Acc + block = 2d8adaf23b0337fa7cccfb4ea344ca153
(Acc + Block) * r =
     16d8e08a0f3fe1de4fe4a15486aca7a270a29f1e6c849221e4a6798b8e45321f
((Acc + Block) * r) % P = 28d31b7caff946c77c8844335369d03a7
```

Adding s, we get this number, and serialize if to get the tag:

Acc + s = 2a927010caf8b2bc2c6365130c11d06a8

Tag: a8:06:1d:c1:30:51:36:c6:c2:2b:8b:af:0c:01:27:a9

2.6. Generating the Poly1305 Key Using ChaCha20

As said in Section 2.5, it is acceptable to generate the one-time Poly1305 pseudorandomly. This section defines such a method.

To generate such a key pair (r,s), we will use the ChaCha20 block function described in Section 2.3. This assumes that we have a 256-bit session key for the Message Authentication Code (MAC) function, such as SK_ai and SK_ar in Internet Key Exchange Protocol version 2 (IKEv2) ([RFC7296]), the integrity key in the Encapsulating Security Payload (ESP) and Authentication Header (AH), or the client_write_MAC_key and server_write_MAC_key in TLS. Any document that specifies the use of Poly1305 as a MAC algorithm for some protocol must specify that 256 bits are allocated for the integrity key. Note that in the AEAD construction defined in Section 2.8, the same key is used for encryption and key generation, so the use of SK_a* or *_write_MAC_key is only for stand-alone Poly1305.

The method is to call the block function with the following parameters:

- The 256-bit session integrity key is used as the ChaCha20 key.

- The block counter is set to zero.

- The protocol will specify a 96-bit or 64-bit nonce. This MUST be unique per invocation with the same key, so it MUST NOT be randomly generated. A counter is a good way to implement this, but other methods, such as a Linear Feedback Shift Register (LFSR) are also acceptable. ChaCha20 as specified here requires a 96-bit nonce. So if the provided nonce is only 64-bit, then the first 32 bits of the nonce will be set to a constant number. This will usually be zero, but for protocols with multiple senders it may be different for each sender, but should be the same for all invocations of the function with the same key by a particular sender.

After running the block function, we have a 512-bit state. We take the first 256 bits or the serialized state, and use those as the one-time Poly1305 key: the first 128 bits are clamped and form "r", while the next 128 bits become "s". The other 256 bits are discarded.

Note that while many protocols have provisions for a nonce for encryption algorithms (often called Initialization Vectors, or IVs), they usually don't have such a provision for the MAC function. In that case, the per-invocation nonce will have to come from somewhere else, such as a message counter.

2.6.1. Poly1305 Key Generation in Pseudocode

```
poly1305_key_gen(key,nonce):
   counter = 0
   block = chacha20_block(key,counter,nonce)
   return block[0..31]
   end
```

2.6.2. Poly1305 Key Generation Test Vector

For this example, we'll set:

```
Key:
000  80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f  ................
016  90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f  ................
```

```
Nonce:
000  00 00 00 00 00 01 02 03 04 05 06 07              ............
```

The ChaCha state setup with key, nonce, and block counter zero:

```
61707865  3320646e  79622d32  6b206574
83828180  87868584  8b8a8988  8f8e8d8c
93929190  97969594  9b9a9998  9f9e9d9c
00000000  00000000  03020100  07060504
```

The ChaCha state after 20 rounds:

```
8ba0d58a  cc815f90  27405081  7194b24a
37b633a8  a50dfde3  e2b8db08  46a6d1fd
7da03782  9183a233  148ad271  b46773d1
3cc1875a  8607def1  ca5c3086  7085eb87
```

```
Output bytes:
000  8a d5 a0 8b 90 5f 81 cc 81 50 40 27 4a b2 94 71  ....._...P@'J..q
016  a8 33 b6 37 e3 fd 0d a5 08 db b8 e2 fd d1 a6 46  .3.7...........F
```

And that output is also the 32-byte one-time key used for Poly1305.

2.7. A Pseudorandom Function for Crypto Suites based on ChaCha/Poly1305

Some protocols, such as IKEv2 ([RFC7296]), require a Pseudorandom Function (PRF), mostly for key derivation. In the IKEv2 definition, a PRF is a function that accepts a variable-length key and a variable-length input, and returns a fixed-length output. Most commonly, Hashed MAC (HMAC) constructions are used for this purpose, and often the same function is used for both message authentication and PRF.

Poly1305 is not a suitable choice for a PRF. Poly1305 prohibits using the same key twice, whereas the PRF in IKEv2 is used multiple times with the same key. Additionally, unlike HMAC, Poly1305 is biased, so using it for key derivation would reduce the security of the symmetric encryption.

Chacha20 could be used as a key-derivation function, by generating an arbitrarily long keystream. However, that is not what protocols such as IKEv2 require.

For this reason, this document does not specify a PRF and recommends that crypto suites use some other PRF such as PRF_HMAC_SHA2_256 (see Section 2.1.2 of [RFC4868]).

2.8. AEAD Construction

AEAD_CHACHA20_POLY1305 is an authenticated encryption with additional data algorithm. The inputs to AEAD_CHACHA20_POLY1305 are:

- o A 256-bit key
- o A 96-bit nonce -- different for each invocation with the same key
- o An arbitrary length plaintext
- o Arbitrary length additional authenticated data (AAD)

Some protocols may have unique per-invocation inputs that are not 96 bits in length. For example, IPsec may specify a 64-bit nonce. In such a case, it is up to the protocol document to define how to transform the protocol nonce into a 96-bit nonce, for example, by concatenating a constant value.

The ChaCha20 and Poly1305 primitives are combined into an AEAD that takes a 256-bit key and 96-bit nonce as follows:

- o First, a Poly1305 one-time key is generated from the 256-bit key and nonce using the procedure described in Section 2.6.
- o Next, the ChaCha20 encryption function is called to encrypt the plaintext, using the same key and nonce, and with the initial counter set to 1.

- o Finally, the Poly1305 function is called with the Poly1305 key calculated above, and a message constructed as a concatenation of the following:

  * The AAD

  * padding1 -- the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the AAD was already an integral multiple of 16 bytes, this field is zero-length.

  * The ciphertext

  * padding2 -- the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the ciphertext was already an integral multiple of 16 bytes, this field is zero-length.

  * The length of the additional data in octets (as a 64-bit little-endian integer).

  * The length of the ciphertext in octets (as a 64-bit little-endian integer).

The output from the AEAD is twofold:

- o A ciphertext of the same length as the plaintext.

- o A 128-bit tag, which is the output of the Poly1305 function.

Decryption is similar with the following differences:

- o The roles of ciphertext and plaintext are reversed, so the ChaCha20 encryption function is applied to the ciphertext, producing the plaintext.

- o The Poly1305 function is still run on the AAD and the ciphertext, not the plaintext.

- o The calculated tag is bitwise compared to the received tag. The message is authenticated if and only if the tags match.

A few notes about this design:

1. The amount of encrypted data possible in a single invocation is 2^32-1 blocks of 64 bytes each, because of the size of the block counter field in the ChaCha20 block function. This gives a total of 247,877,906,880 bytes, or nearly 256 GB. This should be

enough for traffic protocols such as IPsec and TLS, but may be too small for file and/or disk encryption. For such uses, we can return to the original design, reduce the nonce to 64 bits, and use the integer at position 13 as the top 32 bits of a 64-bit block counter, increasing the total message size to over a million petabytes (1,180,591,620,717,411,303,360 bytes to be exact).

2. Despite the previous item, the ciphertext length field in the construction of the buffer on which Poly1305 runs limits the ciphertext (and hence, the plaintext) size to 2^64 bytes, or sixteen thousand petabytes (18,446,744,073,709,551,616 bytes to be exact).

The AEAD construction in this section is a novel composition of ChaCha20 and Poly1305. A security analysis of this composition is given in [Procter].

Here is a list of the parameters for this construction as defined in Section 4 of RFC 5116:

- K_LEN (key length) is 32 octets.
- P_MAX (maximum size of the plaintext) is 247,877,906,880 bytes, or nearly 256 GB.
- A_MAX (maximum size of the associated data) is set to 2^64-1 octets by the length field for associated data.
- N_MIN = N_MAX = 12 octets.
- C_MAX = P_MAX + tag length = 247,877,906,896 octets.

Distinct AAD inputs (as described in Section 3.3 of RFC 5116) shall be concatenated into a single input to AEAD_CHACHA20_POLY1305. It is up to the application to create a structure in the AAD input if it is needed.

2.8.1. Pseudocode for the AEAD Construction

```
pad16(x):
   if (len(x) % 16)==0
      then return NULL
      else return copies(0, 16-(len(x)%16))
   end
```

```
chacha20_aead_encrypt(aad, key, iv, constant, plaintext):
   nonce = constant | iv
   otk = poly1305_key_gen(key, nonce)
   ciphertext = chacha20_encrypt(key, 1, nonce, plaintext)
   mac_data = aad | pad16(aad)
   mac_data |= ciphertext | pad16(ciphertext)
   mac_data |= num_to_4_le_bytes(aad.length)
   mac_data |= num_to_4_le_bytes(ciphertext.length)
   tag = poly1305_mac(mac_data, otk)
   return (ciphertext, tag)
```

2.8.2. Example and Test Vector for AEAD_CHACHA20_POLY1305

For a test vector, we will use the following inputs to the AEAD_CHACHA20_POLY1305 function:

```
Plaintext:
000  4c 61 64 69 65 73 20 61 6e 64 20 47 65 6e 74 6c  Ladies and Gentl
016  65 6d 65 6e 20 6f 66 20 74 68 65 20 63 6c 61 73  emen of the clas
032  73 20 6f 66 20 27 39 39 3a 20 49 66 20 49 20 63  s of '99: If I c
048  6f 75 6c 64 20 6f 66 66 65 72 20 79 6f 75 20 6f  ould offer you o
064  6e 6c 79 20 6f 6e 65 20 74 69 70 20 66 6f 72 20  nly one tip for
080  74 68 65 20 66 75 74 75 72 65 2c 20 73 75 6e 73  the future, suns
096  63 72 65 65 6e 20 77 6f 75 6c 64 20 62 65 20 69  creen would be i
112  74 2e                                            t.

 AAD:
 000  50 51 52 53 c0 c1 c2 c3 c4 c5 c6 c7              PQRS........

Key:
000  80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f  ................
016  90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f  ................

 IV:
 000  40 41 42 43 44 45 46 47                          @ABCDEFG

 32-bit fixed-common part:
 000  07 00 00 00                                      ....

 Setup for generating Poly1305 one-time key (sender id=7):
     61707865  3320646e  79622d32  6b206574
     83828180  87868584  8b8a8988  8f8e8d8c
     93929190  97969594  9b9a9998  9f9e9d9c
     00000000  00000007  43424140  47464544
```

```
   After generating Poly1305 one-time key:
       252bac7b  af47b42d  557ab609  8455e9a4
       73d6e10a  ebd97510  7875932a  ff53d53e
       decc7ea2  b44ddbad  e49c17d1  d8430bc9
       8c94b7bc  8b7d4b4b  3927f67d  1669a432

  Poly1305 Key:
  000  7b ac 2b 25 2d b4 47 af 09 b6 7a 55 a4 e9 55 84  {.+%-.G...zU..U.
  016  0a e1 d6 73 10 75 d9 eb 2a 93 75 78 3e d5 53 ff  ...s.u..*.ux>.S.

  Poly1305 r =  455e9a4057ab6080f47b42c052bac7b
  Poly1305 s = ff53d53e7875932aebd9751073d6e10a

   keystream bytes:
   9f:7b:e9:5d:01:fd:40:ba:15:e2:8f:fb:36:81:0a:ae:
   c1:c0:88:3f:09:01:6e:de:dd:8a:d0:87:55:82:03:a5:
   4e:9e:cb:38:ac:8e:5e:2b:b8:da:b2:0f:fa:db:52:e8:
   75:04:b2:6e:be:69:6d:4f:60:a4:85:cf:11:b8:1b:59:
   fc:b1:c4:5f:42:19:ee:ac:ec:6a:de:c3:4e:66:69:78:
   8e:db:41:c4:9c:a3:01:e1:27:e0:ac:ab:3b:44:b9:cf:
   5c:86:bb:95:e0:6b:0d:f2:90:1a:b6:45:e4:ab:e6:22:
   15:38

  Ciphertext:
  000  d3 1a 8d 34 64 8e 60 db 7b 86 af bc 53 ef 7e c2  ...4d.‘.{...S.˜.
  016  a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7 36 ee 62 d6  ...Q)n......6.b.
  032  3d be a4 5e 8c a9 67 12 82 fa fb 69 da 92 72 8b  =..^..g....i..r.
  048  1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6 7e cd 3b 36  .q.....)....˜.;6
  064  92 dd bd 7f 2d 77 8b 8c 98 03 ae e3 28 09 1b 58  ....-w......(..X
  080  fa b3 24 e4 fa d6 75 94 55 85 80 8b 48 31 d7 bc  ..$...u.U...H1..
  096  3f f4 de f0 8e 4b 7a 9d e5 76 d2 65 86 ce c6 4b  ?....Kz..v.e...K
  112  61 16                                            a.

  AEAD Construction for Poly1305:
  000  50 51 52 53 c0 c1 c2 c3 c4 c5 c6 c7 00 00 00 00  PQRS............
  016  d3 1a 8d 34 64 8e 60 db 7b 86 af bc 53 ef 7e c2  ...4d.‘.{...S.˜.
  032  a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7 36 ee 62 d6  ...Q)n......6.b.
  048  3d be a4 5e 8c a9 67 12 82 fa fb 69 da 92 72 8b  =..^..g....i..r.
  064  1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6 7e cd 3b 36  .q.....)....˜.;6
  080  92 dd bd 7f 2d 77 8b 8c 98 03 ae e3 28 09 1b 58  ....-w......(..X
  096  fa b3 24 e4 fa d6 75 94 55 85 80 8b 48 31 d7 bc  ..$...u.U...H1..
  112  3f f4 de f0 8e 4b 7a 9d e5 76 d2 65 86 ce c6 4b  ?....Kz..v.e...K
  128  61 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00  a...............
  144  0c 00 00 00 00 00 00 00 72 00 00 00 00 00 00 00  ........r.......
```

Note the four zero bytes in line 000 and the 14 zero bytes in line 128

Tag:
1a:e1:0b:59:4f:09:e2:6a:7e:90:2e:cb:d0:60:06:91

3. Implementation Advice

Each block of ChaCha20 involves 16 move operations and one increment operation for loading the state, 80 each of XOR, addition and Roll operations for the rounds, 16 more add operations and 16 XOR operations for protecting the plaintext. Section 2.3 describes the ChaCha block function as "adding the original input words". This implies that before starting the rounds on the ChaCha state, we copy it aside, only to add it in later. This is correct, but we can save a few operations if we instead copy the state and do the work on the copy. This way, for the next block you don't need to recreate the state, but only to increment the block counter. This saves approximately 5.5% of the cycles.

It is not recommended to use a generic big number library such as the one in OpenSSL for the arithmetic operations in Poly1305. Such libraries use dynamic allocation to be able to handle an integer of any size, but that flexibility comes at the expense of performance as well as side-channel security. More efficient implementations that run in constant time are available, one of them in D. J. Bernstein's own library, NaCl ([NaCl]). A constant-time but not optimal approach would be to naively implement the arithmetic operations for 288-bit integers, because even a naive implementation will not exceed 2^288 in the multiplication of (acc+block) and r. An efficient constant-time implementation can be found in the public domain library poly1305-donna ([Poly1305_Donna]).

4. Security Considerations

The ChaCha20 cipher is designed to provide 256-bit security.

The Poly1305 authenticator is designed to ensure that forged messages are rejected with a probability of 1-(n/(2^102)) for a 16n-byte message, even after sending 2^64 legitimate messages, so it is SUF-CMA (strong unforgeability against chosen-message attacks) in the terminology of [AE].

Proving the security of either of these is beyond the scope of this document. Such proofs are available in the referenced academic papers ([ChaCha], [Poly1305], [LatinDances], [LatinDances2], and [Zhenqing2012]).

The most important security consideration in implementing this document is the uniqueness of the nonce used in ChaCha20. Counters and LFSRs are both acceptable ways of generating unique nonces, as is encrypting a counter using a 64-bit cipher such as DES. Note that it is not acceptable to use a truncation of a counter encrypted with a 128-bit or 256-bit cipher, because such a truncation may repeat after a short time.

Consequences of repeating a nonce: If a nonce is repeated, then both the one-time Poly1305 key and the keystream are identical between the messages. This reveals the XOR of the plaintexts, because the XOR of the plaintexts is equal to the XOR of the ciphertexts.

The Poly1305 key MUST be unpredictable to an attacker. Randomly generating the key would fulfill this requirement, except that Poly1305 is often used in communications protocols, so the receiver should know the key. Pseudorandom number generation such as by encrypting a counter is acceptable. Using ChaCha with a secret key and a nonce is also acceptable.

The algorithms presented here were designed to be easy to implement in constant time to avoid side-channel vulnerabilities. The operations used in ChaCha20 are all additions, XORs, and fixed rotations. All of these can and should be implemented in constant time. Access to offsets into the ChaCha state and the number of operations do not depend on any property of the key, eliminating the chance of information about the key leaking through the timing of cache misses.

For Poly1305, the operations are addition, multiplication. and modulus, all on numbers with greater than 128 bits. This can be done in constant time, but a naive implementation (such as using some generic big number library) will not be constant time. For example, if the multiplication is performed as a separate operation from the modulus, the result will sometimes be under 2^256 and sometimes be above 2^256. Implementers should be careful about timing side-channels for Poly1305 by using the appropriate implementation of these operations.

Validating the authenticity of a message involves a bitwise comparison of the calculated tag with the received tag. In most use cases, nonces and AAD contents are not "used up" until a valid message is received. This allows an attacker to send multiple identical messages with different tags until one passes the tag comparison. This is hard if the attacker has to try all 2^128 possible tags one by one. However, if the timing of the tag comparison operation reveals how long a prefix of the calculated and received tags is identical, the number of messages can be reduced significantly. For this reason, with online protocols, implementation MUST use a constant-time comparison function rather than relying on optimized but insecure library functions such as the C language's memcmp().

5. IANA Considerations

IANA has assigned an entry in the "Authenticated Encryption with Associated Data (AEAD) Parameters" registry with 29 as the Numeric ID, "AEAD_CHACHA20_POLY1305" as the name, and this document as reference.

6. References

6.1. Normative References

[ChaCha] Bernstein, D., "ChaCha, a variant of Salsa20", January 2008, <http://cr.yp.to/chacha/chacha-20080128.pdf>.

[Poly1305] Bernstein, D., "The Poly1305-AES message-authentication code", March 2005, <http://cr.yp.to/mac/poly1305-20050329.pdf>.

[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>.

6.2. Informative References

[AE] Bellare, M. and C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", September 2008, <http://dl.acm.org/citation.cfm?id=1410269>.

[Cache-Collisions] Bonneau, J. and I. Mironov, "Cache-Collision Timing Attacks Against AES", 2006, <http://research.microsoft.com/pubs/64024/aes-timing.pdf>.

[FIPS-197] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[LatinDances]
Aumasson, J., Fischer, S., Khazaei, S., Meier, W., and C. Rechberger, "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba", December 2007, <http://cr.yp.to/rumba20/newfeatures-20071218.pdf>.

[LatinDances2]
Ishiguro, T., Kiyomoto, S., and Y. Miyake, "Modified version of 'Latin Dances Revisited: New Analytic Results of Salsa20 and ChaCha'", February 2012, <https://eprint.iacr.org/2012/065.pdf>.

[NaCl] Bernstein, D., Lange, T., and P. Schwabe, "NaCl: Networking and Cryptography library", July 2012, <http://nacl.cr.yp.to>.

[Poly1305_Donna]
Floodyberry, A., "poly1305-donna", February 2014, <https://github.com/floodyberry/poly1305-donna>.

[Procter] Procter, G., "A Security Analysis of the Composition of ChaCha20 and Poly1305", August 2014, <http://eprint.iacr.org/2014/613.pdf>.

[RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, DOI 10.17487/RFC4868, May 2007, <http://www.rfc-editor.org/info/rfc4868>.

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

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <http://www.rfc-editor.org/info/rfc7296>.

[SP800-67] National Institute of Standards and Technology, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", NIST 800-67, January 2012, <http://csrc.nist.gov/publications/nistpubs/800-67-Rev1/SP-800-67-Rev1.pdf>.

[Standby-Cipher]
McGrew, D., Grieco, A., and Y. Sheffer, "Selection of Future Cryptographic Standards", Work in Progress, draft-mcgrew-standby-cipher-00, January 2013.

[Zhenqing2012]
Zhenqing, S., Bin, Z., Dengguo, F., and W. Wenling, "Improved Key Recovery Attacks on Reduced-Round Salsa20 and ChaCha*", 2012.

Appendix A. Additional Test Vectors

The subsections of this appendix contain more test vectors for the algorithms in the sub-sections of Section 2.

A.1. The ChaCha20 Block Functions

```
Test Vector #1:
==============

Key:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Nonce:
000  00 00 00 00 00 00 00 00 00 00 00 00              ............

Block Counter = 0

  ChaCha state at the end
      ade0b876  903df1a0  e56a5d40  28bd8653
      b819d2bd  1aed8da0  ccef36a8  c70d778b
      7c5941da  8d485751  3fe02477  374ad8b8
      f4b8436a  1ca11815  69b687c3  8665eeb2

Keystream:
000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
032  da 41 59 7c 51 57 48 8d 77 24 e0 3f b8 d8 4a 37  .AY|QWH.w$.?..J7
048  6a 43 b8 f4 15 18 a1 1c c3 87 b6 69 b2 ee 65 86  jC.........i..e.
```

```
   Test Vector #2:
   ==============

   Key:
   000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

   Nonce:
   000  00 00 00 00 00 00 00 00 00 00 00 00              ............

   Block Counter = 1

     ChaCha state at the end
         bee7079f  7a385155  7c97ba98  0d082d73
         a0290fcb  6965e348  3e53c612  ed7aee32
         7621b729  434ee69c  b03371d5  d539d874
         281fed31  45fb0a51  1f0ae1ac  6f4d794b

   Keystream:
   000  9f 07 e7 be 55 51 38 7a 98 ba 97 7c 73 2d 08 0d  ....UQ8z...|s-..
   016  cb 0f 29 a0 48 e3 65 69 12 c6 53 3e 32 ee 7a ed  ..).H.ei..S>2.z.
   032  29 b7 21 76 9c e6 4e 43 d5 71 33 b0 74 d8 39 d5  ).!v..NC.q3.t.9.
   048  31 ed 1f 28 51 0a fb 45 ac e1 0a 1f 4b 79 4d 6f  1..(Q..E....KyMo

   Test Vector #3:
   ==============

   Key:
   000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................

   Nonce:
   000  00 00 00 00 00 00 00 00 00 00 00 00              ............

   Block Counter = 1

     ChaCha state at the end
         2452eb3a  9249f8ec  8d829d9b  ddd4ceb1
         e8252083  60818b01  f38422b8  5aaa49c9
         bb00ca8e  da3ba7b4  c4b592d1  fdf2732f
         4436274e  2561b3c8  ebdd4aa6  a0136c00

   Keystream:
   000  3a eb 52 24 ec f8 49 92 9b 9d 82 8d b1 ce d4 dd  :.R$..I.........
   016  83 20 25 e8 01 8b 81 60 b8 22 84 f3 c9 49 aa 5a  . %....`."...I.Z
   032  8e ca 00 bb b4 a7 3b da d1 92 b5 c4 2f 73 f2 fd  ......;...../s..
   048  4e 27 36 44 c8 b3 61 25 a6 4a dd eb 00 6c 13 a0  N'6D..a%.J...l..
```

```
   Test Vector #4:
   ==============

   Key:
   000  00 ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

   Nonce:
   000  00 00 00 00 00 00 00 00 00 00 00 00              ............

   Block Counter = 2

     ChaCha state at the end
         fb4dd572  4bc42ef1  df922636  327f1394
         a78dea8f  5e269039  a1bebbc1  caf09aae
         a25ab213  48a6b46c  1b9d9bcb  092c5be6
         546ca624  1bec45d5  87f47473  96f0992e

   Keystream:
   000  72 d5 4d fb f1 2e c4 4b 36 26 92 df 94 13 7f 32  r.M....K6&.....2
   016  8f ea 8d a7 39 90 26 5e c1 bb be a1 ae 9a f0 ca  ....9.&^........
   032  13 b2 5a a2 6c b4 a6 48 cb 9b 9d 1b e6 5b 2c 09  ..Z.l..H.....[,.
   048  24 a6 6c 54 d5 45 ec 1b 73 74 f4 87 2e 99 f0 96  $.lT.E..st......

   Test Vector #5:
   ==============

   Key:
   000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

   Nonce:
   000  00 00 00 00 00 00 00 00 00 00 00 02              ............

   Block Counter = 0

     ChaCha state at the end
         374dc6c2  3736d58c  b904e24a  cd3f93ef
         88228b1a  96a4dfb3  5b76ab72  c727ee54
         0e0e978a  f3145c95  1b748ea8  f786c297
         99c28f5f  628314e8  398a19fa  6ded1b53

   Keystream:
   000  c2 c6 4d 37 8c d5 36 37 4a e2 04 b9 ef 93 3f cd  ..M7..67J.....?.
   016  1a 8b 22 88 b3 df a4 96 72 ab 76 5b 54 ee 27 c7  ..".....r.v[T.'.
   032  8a 97 0e 0e 95 5c 14 f3 a8 8e 74 1b 97 c2 86 f7  .....\....t.....
   048  5f 8f c2 99 e8 14 83 62 fa 19 8a 39 53 1b ed 6d  _......b...9S..m
```

A.2. ChaCha20 Encryption

```
Test Vector #1:
==============

Key:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Nonce:
000  00 00 00 00 00 00 00 00 00 00 00 00              ............

Initial Block Counter = 0

Plaintext:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
032  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Ciphertext:
000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
032  da 41 59 7c 51 57 48 8d 77 24 e0 3f b8 d8 4a 37  .AY|QWH.w$.?..J7
048  6a 43 b8 f4 15 18 a1 1c c3 87 b6 69 b2 ee 65 86  jC.........i..e.

Test Vector #2:
==============

Key:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................

Nonce:
000  00 00 00 00 00 00 00 00 00 00 00 02              ............

Initial Block Counter = 1

Plaintext:
000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft 
112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
```

```
  144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
  160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
  176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
  192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
  208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
  224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
  240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
  256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
  272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
  288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
  304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
  320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
  336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
  352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
  368  73 73 65 64 20 74 6f                              ssed to

  Ciphertext:
  000  a3 fb f0 7d f3 fa 2f de 4f 37 6c a2 3e 82 73 70  ...}../.O7l.>.sp
  016  41 60 5d 9f 4f 4f 57 bd 8c ff 2c 1d 4b 79 55 ec  A`].OOW...,.KyU.
  032  2a 97 94 8b d3 72 29 15 c8 f3 d3 37 f7 d3 70 05  *....r)....7..p.
  048  0e 9e 96 d6 47 b7 c3 9f 56 e0 31 ca 5e b6 25 0d  ....G...V.1.^.%.
  064  40 42 e0 27 85 ec ec fa 4b 4b b5 e8 ea d0 44 0e  @B.'....KK....D.
  080  20 b6 e8 db 09 d8 81 a7 c6 13 2f 42 0e 52 79 50   ........./B.RyP
  096  42 bd fa 77 73 d8 a9 05 14 47 b3 29 1c e1 41 1c  B..ws....G.)..A.
  112  68 04 65 55 2a a6 c4 05 b7 76 4d 5e 87 be a8 5a  h.eU*....vM^...Z
  128  d0 0f 84 49 ed 8f 72 d0 d6 62 ab 05 26 91 ca 66  ...I..r..b..&..f
  144  42 4b c8 6d 2d f8 0e a4 1f 43 ab f9 37 d3 25 9d  BK.m-....C..7.%.
  160  c4 b2 d0 df b4 8a 6c 91 39 dd d7 f7 69 66 e9 28  ......l.9...if.(
  176  e6 35 55 3b a7 6c 5c 87 9d 7b 35 d4 9e b2 e6 2b  .5U;.l\..{5....+
  192  08 71 cd ac 63 89 39 e2 5e 8a 1e 0e f9 d5 28 0f  .q..c.9.^.....(.
  208  a8 ca 32 8b 35 1c 3c 76 59 89 cb cf 3d aa 8b 6c  ..2.5.<vY...=..l
  224  cc 3a af 9f 39 79 c9 2b 37 20 fc 88 dc 95 ed 84  .:..9y.+7 ......
  240  a1 be 05 9c 64 99 b9 fd a2 36 e7 e8 18 b0 4b 0b  ....d....6....K.
  256  c3 9c 1e 87 6b 19 3b fe 55 69 75 3f 88 12 8c c0  ....k.;.Uiu?....
  272  8a aa 9b 63 d1 a1 6f 80 ef 25 54 d7 18 9c 41 1f  ...c..o..%T...A.
  288  58 69 ca 52 c5 b8 3f a3 6f f2 16 b9 c1 d3 00 62  Xi.R..?.o......b
  304  be bc fd 2d c5 bc e0 91 19 34 fd a7 9a 86 f6 e6  ...-.....4......
  320  98 ce d7 59 c3 ff 9b 64 77 33 8f 3d a4 f9 cd 85  ...Y...dw3.=....
  336  14 ea 99 82 cc af b3 41 b2 38 4d d9 02 f3 d1 ab  .......A.8M.....
  352  7a c6 1d d2 9c 6f 21 ba 5b 86 2f 37 30 e3 7c fd  z....o!.[./70.|.
  368  c4 fd 80 6c 22 f2 21                             ...l".!
```

```
  Test Vector #3:
  ==============

  Key:
  000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
  016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

  Nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 02              ............

  Initial Block Counter = 42

  Plaintext:
  000  27 54 77 61 73 20 62 72 69 6c 6c 69 67 2c 20 61  'Twas brillig, a
  016  6e 64 20 74 68 65 20 73 6c 69 74 68 79 20 74 6f  nd the slithy to
  032  76 65 73 0a 44 69 64 20 67 79 72 65 20 61 6e 64  ves.Did gyre and
  048  20 67 69 6d 62 6c 65 20 69 6e 20 74 68 65 20 77   gimble in the w
  064  61 62 65 3a 0a 41 6c 6c 20 6d 69 6d 73 79 20 77  abe:.All mimsy w
  080  65 72 65 20 74 68 65 20 62 6f 72 6f 67 6f 76 65  ere the borogove
  096  73 2c 0a 41 6e 64 20 74 68 65 20 6d 6f 6d 65 20  s,.And the mome
  112  72 61 74 68 73 20 6f 75 74 67 72 61 62 65 2e     raths outgrabe.

  Ciphertext:
  000  62 e6 34 7f 95 ed 87 a4 5f fa e7 42 6f 27 a1 df  b.4....._..Bo'..
  016  5f b6 91 10 04 4c 0d 73 11 8e ff a9 5b 01 e5 cf  _....L.s....[...
  032  16 6d 3d f2 d7 21 ca f9 b2 1e 5f b1 4c 61 68 71  .m=..!...._.Lahq
  048  fd 84 c5 4f 9d 65 b2 83 19 6c 7f e4 f6 05 53 eb  ...O.e...l....S.
  064  f3 9c 64 02 c4 22 34 e3 2a 35 6b 3e 76 43 12 a6  ..d.."4.*5k>vC..
  080  1a 55 32 05 57 16 ea d6 96 25 68 f8 7d 3f 3f 77  .U2.W....%h.}??w
  096  04 c6 a8 d1 bc d1 bf 4d 50 d6 15 4b 6d a7 31 b1  .......MP..Km.1.
  112  87 b5 8d fd 72 8a fa 36 75 7a 79 7a c1 88 d1     ....r..6uzyz...
```

A.3. Poly1305 Message Authentication Code

Notice how, in test vector #2, r is equal to zero. The part of the Poly1305 algorithm where the accumulator is multiplied by r means that with r equal zero, the tag will be equal to s regardless of the content of the text. Fortunately, all the proposed methods of generating r are such that getting this particular weak key is very unlikely.

Test Vector #1:
==============

```
One-time Poly1305 Key:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Text to MAC:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
032  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Tag:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
   Test Vector #2:
   ==============

   One-time Poly1305 Key:
   000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   016  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....`p...."z.>

   Text to MAC:
   000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
   016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
   032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
   048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
   064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
   080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
   096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft
   112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
   128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
   144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
   160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
   176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
   192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
   208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
   224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
   240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
   256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
   272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
   288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
   304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
   320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
   336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
   352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
   368  73 73 65 64 20 74 6f                             ssed to

   Tag:
   000  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....`p...."z.>
```

```
Test Vector #3:
==============

One-time Poly1305 Key:
000  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....‘p...."z.>
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

Text to MAC:
000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft
112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
368  73 73 65 64 20 74 6f                             ssed to

Tag:
000  f3 47 7e 7c d9 54 17 af 89 a6 b8 79 4c 31 0c f0  .G˜|.T.....yL1..
```

```
Test Vector #4:
==============

One-time Poly1305 Key:
000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

Text to MAC:
000  27 54 77 61 73 20 62 72 69 6c 6c 69 67 2c 20 61  'Twas brillig, a
016  6e 64 20 74 68 65 20 73 6c 69 74 68 79 20 74 6f  nd the slithy to
032  76 65 73 0a 44 69 64 20 67 79 72 65 20 61 6e 64  ves.Did gyre and
048  20 67 69 6d 62 6c 65 20 69 6e 20 74 68 65 20 77   gimble in the w
064  61 62 65 3a 0a 41 6c 6c 20 6d 69 6d 73 79 20 77  abe:.All mimsy w
080  65 72 65 20 74 68 65 20 62 6f 72 6f 67 6f 76 65  ere the borogove
096  73 2c 0a 41 6e 64 20 74 68 65 20 6d 6f 6d 65 20  s,.And the mome
112  72 61 74 68 73 20 6f 75 74 67 72 61 62 65 2e     raths outgrabe.

Tag:
000  45 41 66 9a 7e aa ee 61 e7 08 dc 7c bc c5 eb 62  EAf.~..a...|...b
```

Test Vector #5: If one uses 130-bit partial reduction, does the code handle the case where partially reduced final result is not fully reduced?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
tag:
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #6: What happens if addition of s overflows modulo 2^128?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
data:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #7: What happens if data limb is all ones and there is carry from lower limb?

```
R:
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #8: What happens if final result from polynomial part is exactly 2^130-5?

```
R:
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FB FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
tag:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #9: What happens if final result from polynomial part is exactly 2^130-6?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FD FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
tag:
FA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

```
   Test Vector #10: What happens if 5*H+L-type reduction produces
   131-bit intermediate result?

   R:
   01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
   S:
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   data:
   E3 35 94 D7 50 5E 43 B9 00 00 00 00 00 00 00 00
   33 94 D7 50 5E 43 79 CD 01 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   tag:
   14 00 00 00 00 00 00 00 55 00 00 00 00 00 00 00

   Test Vector #11: What happens if 5*H+L-type reduction produces
   131-bit final result?

   R:
   01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
   S:
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   data:
   E3 35 94 D7 50 5E 43 B9 00 00 00 00 00 00 00 00
   33 94 D7 50 5E 43 79 CD 01 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   tag:
   13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

A.4. Poly1305 Key Generation Using ChaCha20

```
  Test Vector #1:
  ==============

  The key:
  000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

  The nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 00              ............

  Poly1305 one-time key:
  000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
  016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
```

```
Test Vector #2:
==============

The key:
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................

The nonce:
000  00 00 00 00 00 00 00 00 00 00 00 02              ............

Poly1305 one-time key:
000  ec fa 25 4f 84 5f 64 74 73 d3 cb 14 0d a9 e8 76  ..%O._dts......v
016  06 cb 33 06 6c 44 7b 87 bc 26 66 dd e3 fb b7 39  ..3.lD{..&f....9

Test Vector #3:
==============

The key:
000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

The nonce:
000  00 00 00 00 00 00 00 00 00 00 00 02              ............

Poly1305 one-time key:
000  96 5e 3b c6 f9 ec 7e d9 56 08 08 f4 d2 29 f9 4b  .^;...~.V....).K
016  13 7f f2 75 ca 9b 3f cb dd 59 de aa d2 33 10 ae  ...u..?..Y...3..
```

A.5. ChaCha20-Poly1305 AEAD Decryption

Below we see decrypting a message. We receive a ciphertext, a nonce, and a tag. We know the key. We will check the tag and then (assuming that it validates) decrypt the ciphertext. In this particular protocol, we'll assume that there is no padding of the plaintext.

```
The key:
000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

Ciphertext:
000  64 a0 86 15 75 86 1a f4 60 f0 62 c7 9b e6 43 bd  d...u...`.b...C.
016  5e 80 5c fd 34 5c f3 89 f1 08 67 0a c7 6c 8c b2  ^.\.4\....g..l..
032  4c 6c fc 18 75 5d 43 ee a0 9e e9 4e 38 2d 26 b0  Ll..u]C....N8-&.
048  bd b7 b7 3c 32 1b 01 00 d4 f0 3b 7f 35 58 94 cf  ...<2.....;.5X..
064  33 2f 83 0e 71 0b 97 ce 98 c8 a8 4a bd 0b 94 81  3/..q......J....
080  14 ad 17 6e 00 8d 33 bd 60 f9 82 b1 ff 37 c8 55  ...n..3.`....7.U
096  97 97 a0 6e f4 f0 ef 61 c1 86 32 4e 2b 35 06 38  ...n...a..2N+5.8
112  36 06 90 7b 6a 7c 02 b0 f9 f6 15 7b 53 c8 67 e4  6..{j|.....{S.g.
128  b9 16 6c 76 7b 80 4d 46 a5 9b 52 16 cd e7 a4 e9  ..lv{.MF..R.....
144  90 40 c5 a4 04 33 22 5e e2 82 a1 b0 a0 6c 52 3e  .@...3"^.....lR>
160  af 45 34 d7 f8 3f a1 15 5b 00 47 71 8c bc 54 6a  .E4..?..[.Gq..Tj
176  0d 07 2b 04 b3 56 4e ea 1b 42 22 73 f5 48 27 1a  ..+..VN..B"s.H'.
192  0b b2 31 60 53 fa 76 99 19 55 eb d6 31 59 43 4e  ..1`S.v..U..1YCN
208  ce bb 4e 46 6d ae 5a 10 73 a6 72 76 27 09 7a 10  ..NFm.Z.s.rv'.z.
224  49 e6 17 d9 1d 36 10 94 fa 68 f0 ff 77 98 71 30  I....6...h..w.q0
240  30 5b ea ba 2e da 04 df 99 7b 71 4d 6c 6f 2c 29  0[.......{qMlo,)
256  a6 ad 5c b4 02 2b 02 70 9b                       ..\..+.p.

The nonce:
000  00 00 00 00 01 02 03 04 05 06 07 08              ............

The AAD:
000  f3 33 88 86 00 00 00 00 00 00 4e 91              .3........N.

Received Tag:
000  ee ad 9d 67 89 0c bb 22 39 23 36 fe a1 85 1f 38  ...g..."9#6....8
```

First, we calculate the one-time Poly1305 key

```
@@@  ChaCha state with key setup
      61707865  3320646e  79622d32  6b206574
      a540921c  8ad355eb  868833f3  f0b5f604
      c1173947  09802b40  bc5cca9d  c0757020
      00000000  00000000  04030201  08070605

@@@  ChaCha state after 20 rounds
      a94af0bd  89dee45c  b64bb195  afec8fa1
      508f4726  63f554c0  1ea2c0db  aa721526
      11b1e514  a0bacc0f  828a6015  d7825481
      e8a4a850  d9dcbbd6  4c2de33a  f8ccd912

@@@ out bytes:
bd:f0:4a:a9:5c:e4:de:89:95:b1:4b:b6:a1:8f:ec:af:
26:47:8f:50:c0:54:f5:63:db:c0:a2:1e:26:15:72:aa

Poly1305 one-time key:
000  bd f0 4a a9 5c e4 de 89 95 b1 4b b6 a1 8f ec af  ..J.\.....K.....
016  26 47 8f 50 c0 54 f5 63 db c0 a2 1e 26 15 72 aa  &G.P.T.c....&.r.
```

Next, we construct the AEAD buffer

```
Poly1305 Input:
000  f3 33 88 86 00 00 00 00 00 00 4e 91 00 00 00 00  .3........N.....
016  64 a0 86 15 75 86 1a f4 60 f0 62 c7 9b e6 43 bd  d...u...'.b...C.
032  5e 80 5c fd 34 5c f3 89 f1 08 67 0a c7 6c 8c b2  ^.\.4\....g..l..
048  4c 6c fc 18 75 5d 43 ee a0 9e e9 4e 38 2d 26 b0  Ll..u]C....N8-&.
064  bd b7 b7 3c 32 1b 01 00 d4 f0 3b 7f 35 58 94 cf  ...<2.....;.5X..
080  33 2f 83 0e 71 0b 97 ce 98 c8 a8 4a bd 0b 94 81  3/..q......J....
096  14 ad 17 6e 00 8d 33 bd 60 f9 82 b1 ff 37 c8 55  ...n..3.'....7.U
112  97 97 a0 6e f4 f0 ef 61 c1 86 32 4e 2b 35 06 38  ...n...a..2N+5.8
128  36 06 90 7b 6a 7c 02 b0 f9 f6 15 7b 53 c8 67 e4  6..{j|.....{S.g.
144  b9 16 6c 76 7b 80 4d 46 a5 9b 52 16 cd e7 a4 e9  ..lv{.MF..R.....
160  90 40 c5 a4 04 33 22 5e e2 82 a1 b0 a0 6c 52 3e  .@...3"^.....lR>
176  af 45 34 d7 f8 3f a1 15 5b 00 47 71 8c bc 54 6a  .E4..?..[.Gq..Tj
192  0d 07 2b 04 b3 56 4e ea 1b 42 22 73 f5 48 27 1a  ..+..VN..B"s.H'.
208  0b b2 31 60 53 fa 76 99 19 55 eb d6 31 59 43 4e  ..1'S.v..U..1YCN
224  ce bb 4e 46 6d ae 5a 10 73 a6 72 76 27 09 7a 10  ..NFm.Z.s.rv'.z.
240  49 e6 17 d9 1d 36 10 94 fa 68 f0 ff 77 98 71 30  I....6...h..w.q0
256  30 5b ea ba 2e da 04 df 99 7b 71 4d 6c 6f 2c 29  0[.......{qMlo,)
272  a6 ad 5c b4 02 2b 02 70 9b 00 00 00 00 00 00 00  ..\..+.p........
288  0c 00 00 00 00 00 00 00 09 01 00 00 00 00 00 00  ................
```

We calculate the Poly1305 tag and find that it matches

```
Calculated Tag:
000  ee ad 9d 67 89 0c bb 22 39 23 36 fe a1 85 1f 38  ...g..."9#6....8
```

Finally, we decrypt the ciphertext

```
Plaintext::
000  49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 73 20  Internet-Drafts
016  61 72 65 20 64 72 61 66 74 20 64 6f 63 75 6d 65  are draft docume
032  6e 74 73 20 76 61 6c 69 64 20 66 6f 72 20 61 20  nts valid for a
048  6d 61 78 69 6d 75 6d 20 6f 66 20 73 69 78 20 6d  maximum of six m
064  6f 6e 74 68 73 20 61 6e 64 20 6d 61 79 20 62 65  onths and may be
080  20 75 70 64 61 74 65 64 2c 20 72 65 70 6c 61 63   updated, replac
096  65 64 2c 20 6f 72 20 6f 62 73 6f 6c 65 74 65 64  ed, or obsoleted
112  20 62 79 20 6f 74 68 65 72 20 64 6f 63 75 6d 65   by other docume
128  6e 74 73 20 61 74 20 61 6e 79 20 74 69 6d 65 2e  nts at any time.
144  20 49 74 20 69 73 20 69 6e 61 70 70 72 6f 70 72   It is inappropr
160  69 61 74 65 20 74 6f 20 75 73 65 20 49 6e 74 65  iate to use Inte
176  72 6e 65 74 2d 44 72 61 66 74 73 20 61 73 20 72  rnet-Drafts as r
192  65 66 65 72 65 6e 63 65 20 6d 61 74 65 72 69 61  eference materia
208  6c 20 6f 72 20 74 6f 20 63 69 74 65 20 74 68 65  l or to cite the
224  6d 20 6f 74 68 65 72 20 74 68 61 6e 20 61 73 20  m other than as
240  2f e2 80 9c 77 6f 72 6b 20 69 6e 20 70 72 6f 67  /...work in prog
256  72 65 73 73 2e 2f e2 80 9d                       ress./...
```

Appendix B. Performance Measurements of ChaCha20

The following measurements were made by Adam Langley for a blog post published on February 27th, 2014. The original blog post was available at the time of this writing at <https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html>.

Chip	AES-128-GCM	ChaCha20-Poly1305
OMAP 4460	24.1 MB/s	75.3 MB/s
Snapdragon S4 Pro	41.5 MB/s	130.9 MB/s
Sandy Bridge Xeon (AES-NI)	900 MB/s	500 MB/s

Table 1: Speed Comparison

Acknowledgements

ChaCha20 and Poly1305 were invented by Daniel J. Bernstein. The AEAD construction and the method of creating the one-time Poly1305 key were invented by Adam Langley.

Thanks to Robert Ransom, Watson Ladd, Stefan Buhler, Dan Harkins, and Kenny Paterson for their helpful comments and explanations. Thanks to Niels Moller for suggesting the more efficient AEAD construction in this document. Special thanks to Ilari Liusvaara for providing extra test vectors, helpful comments, and for being the first to attempt an implementation from this document. Thanks to Sean Parkinson for suggesting improvements to the examples and the pseudocode. Thanks to David Ireland for pointing out a bug in the pseudocode, and to Stephen Farrell and Alyssa Rowan for pointing out missing advise in the security considerations.

Special thanks goes to Gordon Procter for performing a security analysis of the composition and publishing [Procter].

Authors' Addresses

Yoav Nir
Check Point Software Technologies, Ltd.
5 Hasolelim St.
Tel Aviv 6789735
Israel

EMail: ynir.ietf@gmail.com

Adam Langley
Google, Inc.

EMail: agl@google.com