Additional Modes for ASCON
Version 1.0

Rhys Weatherley
Southern Storm Software, Pty Ltd.
rhys.weatherley@gmail.com
https://github.com/rweather/ascon-suite

March 2023

Abstract

NIST selected the ASCON family of cryptographic primitives for standardization in February 2023 as the final step in the Lightweight Cryptography Competition. The ASCON submission to the competition provided Authenticated Encryption with Associated Data (AEAD), hashing, and Extensible Output Function (XOF) modes. Real world cryptography systems often need more than packet encryption and simple hashing. Keyed message authentication, key derivation, cryptographically secure pseudo-random number generation (CSPRNG), password hashing, and encryption of sensitive values in memory are also important. This paper defines additional modes that can be deployed on top of ASCON based on proven designs from the literature.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

NIST selected the ASCON family of cryptographic primitives for standardization in February 2023 as the final step in the Lightweight Cryptography Competition [3]. The ASCON submission to the competition provided Authenticated Encryption with Associated Data (AEAD), hashing, and Extensible Output Function (XOF) modes.

AEAD modes and hashing in isolation can be sufficient for specialized single-purpose lightweight devices such as key fobs, data collection devices, security access cards, and so on.

More complex embedded devices will be implementing higher-level communications protocols like TLS and link-layer encryption schemes like those in Bluetooth and Wi-Fi. It is to be expected that these standards bodies may adopt the NIST lightweight encryption scheme as an option in the future.

Complex systems often need more than packet encryption and simple hashing. Keyed message authentication, key derivation functions (KDF's), cryptographically secure pseudo-random number generation (CSPRNG), password hashing, and encryption of sensitive values in memory are also important.

This paper defines additional modes that can be deployed on top of ASCON, mostly using proven designs from the literature. Sections marked with "experimental" contain speculative new designs that require further study and security analysis.

Because ASCON has hashing support, it can easily implement keyed message authentication, key derivation, and password hashing using existing designs such as HMAC [17], KMAC [16], HKDF [18], and PBKDF2 [20]. Pseudo-random number generation can be implemented with the SpongePRNG [6] construction.

Devices that embed an asymmetric key pair for higher-level protocols often require a method to encrypt the key pair under a pass phrase or device-specific secret. Synthetic Initialization Vector (SIV) [13] modes can be deployed to wrap sensitive key material in trusted or untrusted storage. Some of the round 2 and final round submissions to the competition had SIV modes, which we use as a guide for defining a similar mode for ASCON.

A key contribution of this paper is a formal definition in section 4 for the customizable hashing mode ASCON-CXOF which we suggest for inclusion in the final NIST lightweight cryptography standard.

All of the modes described in this paper have been implemented by the author in the ASCON-SUITE repository to verify feasibility and performance.

This work is a continuation of previous work where the author explored additional modes for all of the NIST Lightweight Cryptography Competition finalists [27].

2 Definitions

$p^a(S)$
: Applies a rounds of the ASCON permutation to the 320-bit state S, where a is between 1 and 12.

$A \| B$
: Concatenates the bit strings A and B.

$\lfloor A \rfloor_\ell$
: Extracts the first ℓ bits of bit string A.

$\lceil A \rceil_\ell$
: Extracts the last ℓ bits of bit string A.

$A \oplus B$
: Computes the exclusive-OR of A and B.

0^c A bit string consisting of c zero bits.

0^{*r} A bit string consisting of enough zero bits to pad to a multiple of the rate r.

$len(A)$
Length of the bit string A in bits.

$encode(N, b)$
Encodes the nunber N as a b-bit string in big endian byte order.

$pad(A, b)$
Pads the bit string A to b bits with zeroes.

"" An empty bit string.

$dbl(A)$
Doubles the 128-bit value A in the GF(128) field.

$trng\_get()$
Gets 256 bits of seed data from the system True Random Number Generator (TRNG) or entropy source.

3 Modes from NIST submissions

The following modes were defined as part of the NIST submission for the ASCON family:

1. ASCON-128, AEAD mode with a 128-bit key, 128-bit nonce, and a 128-bit authentication tag. Uses an 8-byte rate.
2. ASCON-128A, AEAD mode with a 128-bit key, 128-bit nonce, and a 128-bit authentication tag. Uses a 16-byte rate for better performance.
3. ASCON-80PQ, AEAD mode with a 160-bit key, 128-bit nonce, and a 128-bit authentication tag. Uses an 8-byte rate. Intended to improve security against post-quantum threats.
4. ASCON-HASH, hashing mode with a 256-bit digest output and an 8-byte rate.
5. ASCON-HASHA, hashing mode with a 256-bit digest output and an 8-byte rate. Reduced-round version that is intended as a companion to ASCON-128A.
6. ASCON-XOF, extensible output function with arbitrary-length output and an 8-byte rate.
7. ASCON-XOFA, extensible output function with arbitrary-length output and an 8-byte rate. Reduced-round version that is intended as a companion to ASCON-128A.

ISAP is a family of nonce-based authenticated ciphers with associated data (AEAD) designed with a focus on robustness against passive side-channel attacks [9]:

8. ISAP-A-128, AEAD mode with a 128-bit key, 128-bit nonce, and a 128-bit authentication tag.
9. ISAP-A-128A, AEAD mode with a 128-bit key, 128-bit nonce, and a 128-bit authentication tag. Reduced-round version that is intended as a companion to ASCON-128A.
10. ISAP-A-80PQ, AEAD mode with a 160-bit key, 128-bit nonce, and a 128-bit authentication tag. This variant is presently unique to ASCON-SUITE, extending ISAP-A-128 to 160-bit keys in the same manner as how ASCON-80PQ extends ASCON-128.

4 Customizable hashing

NIST SP 800-185 [16] defines an extension to SHAKE128 and SHAKE256 of SHA-3 to provide customizable XOF modes called cSHAKE128 and cSHAKE256. cSHAKE differs from SHAKE in that it specifies four input parameters in place of SHAKE's single input data stream:

1. X is the main input string of any length allowed by the underying XOF mode, including zero-length strings.

2. L is an integer representing the requested output length in bits, or zero for arbitrary-length output.
3. N is a function-name string, used by NIST to define functions based on cSHAKE such as "KMAC", "TupleHash", and "ParallelHash". When N is empty, the behaviour is identical to basic SHAKE.
4. C is a customization string, selected by the user to define an application-specific variant of the function. When no customization is desired, C is set to the empty string.

The ASCON hashing mode already supports the L parameter in the low 32 bits of the initialization vector (IV) of the first block. ASCON-HASH[A] is the special case where L is 256 bits and ASCON-XOF[A] is the special case where L is 0. See Table 1.

Table 1: IV values for ASCON hashing modes

	IV
ASCON-HASH	0x00400c00**00000100**
ASCON-HASHA	0x00400c04**00000100**
ASCON-XOF	0x00400c00**00000000**
ASCON-XOFA	0x00400c04**00000000**

NIST SP 800-185 is slightly more flexible in that L can be provided dynamically after the X input string has been absorbed, in case its value is unknown ahead of time. However, it is unusual for an application to be unaware of the desired output length before hashing starts, except in the special case of arbitrary-length output (0).

The remaining 256 bits of the initial block for ASCON hashing are set to zero in the standard version. We propose that this space be used to encode the function-name string N, as demonstrated in Figure 1 for N = "KMAC".

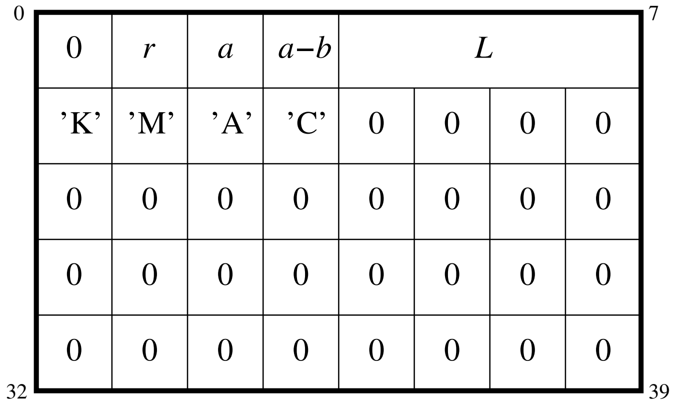

Figure 1: First block for customizable hashing

If N is less than or equal to 256 bits in length, then pad it with zero bits and populate the spare space in the initial block with the padded string. If N is greater than 256 bits in length, then compute ASCON-HASH(N) and place the digest value into the spare space instead.

The initial block is hashed with the ASCON permutation and then C and X are absorbed. The hashed version of the initial block can be precomputed by the application for known constant values of L and N.

The customization string C is absorbed before X. After absorbing C, the last bit of the state is inverted to provide domain separation between the C and X blocks. If C is empty, then no additional blocks are absorbed before X.

Algorithm 1 provides a formal definition for ASCON-CXOF. The standard ASCON hashing modes can be defined in terms of ASCON-CXOF as follows:

- ASCON-HASH(X) = ASCON-CXOF(X, 256, "", "", 12, 12, 64)
- ASCON-HASHA(X) = ASCON-CXOF(X, 256, "", "", 12, 8, 64)
- ASCON-XOF(X) = ASCON-CXOF(X, 0, "", "", 12, 12, 64)

- ASCON-XOFA(X) = ASCON-CXOF(X, 0, "", "", 12, 8, 64)

The security of ASCON-CXOF flows directly from the security of the XOF and AEAD modes in [3]:

- The formatting of the first block is similar to that used by the the standard ASCON AEAD modes that place the IV, key, and nonce in the first block. Here we put the function name N in the same place as the key and nonce.
- The domain separation between C and X is almost identical to the domain separation between associated data and plaintext in the AEAD modes.
- The AEAD modes apply domain separation even if the associated data is empty, whereas ASCON-CXOF omits domain separation if C is empty. This was done to maintain compatibility with the standard definitions for ASCON-HASH, ASCON-HASHA, ASCON-XOF, and ASCON-XOFA.
- Output lengths other than 0 and 256 are allowed.
- Everything else is identical to the description of hashing in section 2.5 of [3].

5 Squeezing then absorbing

The hashing and XOF modes of ASCON specify an absorb phase followed by a squeeze phase. The implication is that squeezing is only useful after all of the input has been absorbed.

This is incorect. Pseudorandom number generators with entropy collection intermix absorbing new entropy into the state with squeezing random output.

The Noise protocol [23] absorbs the outputs of Diffie-Hellman operations into the session state and then squeezes out new encryption keys for subsequent traffic. Standard Noise does this with repeated applications of HKDF [18], re-initializing HKDF every time it is invoked.

Efficiency would be improved if the same state could continue, alternating between absorbing and squeezing as the session progresses. The Strobe protocol framework [1] and Noise's own Stateful Hash Objects extension [24] are examples of this.

There is also an API safety aspect. Incremental XOF API's tend to offer separate `init`, `absorb`, and `squeeze` methods. What is the behaviour when the user's application invokes `absorb` after `squeeze`? The behaviour might vary from doing something useful, misusing the internal state in insecure ways, to invalid state exceptions.

It would be useful to have a standard method to switch from squeezing back to absorbing so that each application or cryptography library doesn't invent their own incompatible methods.

ASCON-SUITE invokes p^a on the permuation state when it was previously squeezing and the application invokes `absorb`. It then proceeds to absorb the supplied data in the usual way. This mixes the state to separate the squeeze and absorb phases. It also avoids XOR'ing new input data with previously-squeezed output in the permutation state.

6 Keyed message authentication

6.1 ASCON-MAC

While not part of the official NIST submission, the ASCON designers have defined a Message Authentication Code (MAC) [11].

The recommended instance of ASCON-MAC takes a 128-bit key and an arbitrary amount of input, and produces a 128-bit authentication tag. The design is similar to the authentication part of ASCON-128 without encryption.

The small size of the key and authentication tag may be an insufficient security margin for some applications. HMAC or KMAC modes may provide greater confidence.

Algorithm 1 Customizable extensible output function ASCON-CXOF

Input: $X \in \{0,1\}^*$, $0 \leq L \leq 2^{32}-1$, $N \in \{0,1\}^*$, $C \in \{0,1\}^*$, $1 \leq a \leq 12$, $1 \leq b \leq a$, $r \in \{64,128\}$
Output: $H \in \{0,1\}^*$

```
function ASCON-CXOF($X, L, N, C, a, b, r$)
    $IV \leftarrow (0^8 \| encode(r,8) \| encode(a,8) \| encode(a-b,8) \| encode(L,32))$    ▷ Initialization
    if $len(N) \leq 256$ then
        $N' \leftarrow pad(N, 256)$
    else
        $N' \leftarrow$ ASCON-CXOF($N$, 256, "", "", $a, b, r$)
    end if
    $S \leftarrow p^a(IV \| N')$
    if $len(C) > 0$ then    ▷ Customization
        $C_1, \ldots, C_c \leftarrow C \| 1 \| 0^{*r}$
        for $i = 1, \ldots, c$ do
            $S \leftarrow p^b((\lfloor S \rfloor_r \oplus C_i) \| (\lceil S \rceil_{320-r}))$
        end for
        $S \leftarrow S \oplus (0^{319} \| 1)$    ▷ Domain separation
    end if
    $X_1, \ldots, X_x \leftarrow X \| 1 \| 0^{*r}$    ▷ Absorbing
    for $i = 1, \ldots, x-1$ do
        $S \leftarrow p^b((\lfloor S \rfloor_r \oplus X_i) \| \lceil S \rceil_{320-r})$
    end for
    $S \leftarrow (\lfloor S \rfloor_r \oplus X_x) \| \lceil S \rceil_{320-r}$
    if $L = 0$ then    ▷ Squeezing
        $L = max$
    end if
    $S \leftarrow p^a(S)$
    for $i = 1, \ldots, t = \lceil L/r \rceil$ do
        $H_i = \lfloor S \rfloor_r$
        $S \leftarrow p^b(S)$
    end for
    return $\lfloor H_1 \| \ldots \| H_t \rfloor_L$
end function
```

6.2 HMAC

HMAC [17] is a widely-implemented standard for creating keyed message authenticators from simple digest algorithms. While ASCON-CXOF can provide better options, HMAC has the desirable feature that existing protocols that use HMAC can be easily retrofitted to use a new digest algorithm.

The main wrinkle is the block size B from RFC 2104, which is the basic unit of absorption for the underlying digest algorithm. B needs to be at least as big as the digest output, ideally twice the size, to format the key prior to hashing it. But sponge hashing modes use a "rate" that is smaller than the digest output (8 bytes for ASCON-HASH). For backwards compatibility, we recommend SHA-256's block size of 64 bytes.

6.3 KMAC

NIST SP 800-185 [16] defines instances of cSHAKE to provide provide keyed message authentication (KMAC). We can do the same using ASCON-CXOF:

- ASCON-KMAC(K, L, C, X) = ASCON-CXOF(K || X, L, "KMAC", C, 12, 12, 64)
- ASCON-KMACA(K, L, C, X) = ASCON-CXOF(K || X, L, "KMAC", C, 12, 8, 64)

We recommend that the key K be a 256-bit value to avoid the need for complex padding and domain separation between K and X. Outside of contrived test vectors and special cases like PBKDF2, it is rare to find a use of keyed message authentication that doesn't have a standard-size key. Usually the key is generated by a key derivation function in the standard size.

If a particular application needs a key size other than 256 bits, then it can do one of the following:

- Hash the key value with ASCON-HASH or ASCON-HASHA to produce a 256-bit K value.
- Modify the ASCON-CXOF function name to include the key length in bits; for example, "KMAC-192".

We deal with PBKDF2 separately in section 8 below.

7 Key derivation functions

Secure communications protocols like TLS [25] and Noise [23] generate a shared secret as part of the session handshake, and then stretch that secret into as much key material as is required to secure the session. HKDF [18] is the usual method.

The ASCON designers have defined a Pseudorandom Function (PRF) [11] with a 128-bit key and a fixed-length or arbitrary-length output.

ASCON-CXOF can be parameterized for use as a KDF mode:

- ASCON-KDF(K, L, C) = ASCON-CXOF(K, L, "KDF", C, 12, 12, 64)
- ASCON-KDFA(K, L, C) = ASCON-CXOF(K, L, "KDF", C, 12, 8, 64)

8 Password-based authentication

Often a device needs to convert a PIN or pass phrase into an encryption key for data that is stored on the device, such as asymmetric key pairs. Or passwords are used for user accounts on the device as part of a role-based access control (RBAC) scheme such as that described in IEEE Std 1686-2022 [14].

PBKDF2 [20] can be implemented with a very small amount of RAM for temporary intermediate values, which makes it attractive in small devices compared with memory-hard password hashing options like scrypt [22], Balloon [7], or the finalists to the Password Hashing Competition (PHC) [2].

PBKDF2 is defined in terms of a pseudorandom function PRF, which is usually HMAC with some specific choice of digest algorithm, such as SHA-256. ASCON-CXOF can be paramterized for use as the PRF instead:

- PRF(P, X) = ASCON-CXOF(X, 256, "PBKDF2", P, 12, 12, 64)

It is well known that PBKDF2 is easy to parallelize on password cracking systems. PBKDF2 includes an iteration counter to address this. But counter values high enough to pose a challenge to crackers will have prohibitive performance on small systems.

Research is needed into memory-hard password hashing schemes for embedded devices that are hard enough to deter crackers with access to significant resources, but not so hard that the memory requirements are prohibitively expensive on embedded devices.

9 Pseudo-random number generation

Embedded microcontrollers are increasingly fitted with True Random Number Generator (TRNG) peripherals as standard. The quality of the random data from these peripherals can vary. The random data may be directly from an on-chip entropy source and so the randomness may not be uniformly distributed throughout the output bits. The output data rate may also be very slow.

Some microcontrollers "whiten" the output with a hash algorithm or block cipher inside the TRNG peripheral, but the behaviour and strength of this whitening process is usually opaque. Users may also have reason to distrust the data from a built-in TRNG as it could have been watermarked or backdoored in some fashion by a nation state actor.

Unfortunately most embedded application writers will have no choice but to use the built-in TRNG as there is no other source of random-seeming data available. Desktop operating system entropy sources that measure the jitter in keystroke, disk access, or network timings are usually not available.

If the device is sampling an analog input source, then thermal noise in the low bits of the digitized samples may help. But the analog inputs may be under the control of an attacker who can then affect the generated output by injecting chosen signals. It is also complicated to remove bias from such entropy sources.

Since the built-in TRNG is all most application writers will have, it is prudent to feed the output of the TRNG into a PRNG to distribute the entropy evenly, expand the amount of random data to arbitrary lengths, increase the data rate, incorporate unique device identifiers, absorb data from other sources of entropy known to the device, and destroy any potential watermarks.

9.1 Simple PRNG's

A simple approach is to expand an initial seed from the TRNG into an arbitrary amount of random material using ASCON-CXOF:

- ASCON-PRNG-SIMPLE(Seed, C) = ASCON-CXOF(Seed, 0, "SimplePRNG", C, 12, 12, 64)

The customization string C can be populated with device serial numbers, network MAC addresses, or other unique identifying information. This will ensure that different devices of the same type will generate different random sequences even if the starting seeds are identical due to TRNG hardware failure.

This approach does not permit additional TRNG seeds to be incorporated into the state later. The PRNG object must be destroyed and recreated to access fresh entropy from the TRNG. This may not matter for many applications as only a small amount of random material is required to establish a secure session, after which a KDF is used to generate keys for encrypting and authenticating session traffic.

9.2 Entropy-collecting PRNG's

SpongePRNG [6] is a technique for converting a permutation-based sponge function into a PRNG. API's are provided to "feed" in new seed material as it becomes available and to "fetch" a continuous sequence of random output at other times.

Forward security is provided by zeroing the rate part of the state and then iterating the permutation. It is not feasible to reverse the PRNG's state without guessing the rate bits before they were zeroed. This is the re-keying process for SpongePRNG. The SpongePRNG paper recommends zeroing the rate and iterating the permutation several times when re-keying.

SpongePRNG is a natural fit for building a PRNG for ASCON. Algorithm 2 provides a formal description of the API. It is assumed that the underlying ASCON-CXOF implementation can switch back and forth between squeezing and absorbing.

Algorithm 2 Entropy-collecting pseudorandom number generator

Input: Customization string $C \in \{0,1\}^*$, round count $a = 12$, rate $r = 64$, re-seeding $limit \in \mathbb{N}_0$
Output: Random output $R \in \{0,1\}^L$

```
class members
    $State \in \{0,1\}^{320}$
    $counter \in \mathbb{N}_0$
end class members

function ASCON-PRNG-INIT($C$)
    $State \leftarrow$ASCON-cXOF-INIT(0, "SpongePRNG", $C, a, a, r$)
    ASCON-PRNG-RESEED($State$)
end function

function ASCON-PRNG-FEED($State, Data$)                ▷ Feed in entropy data from the application
    ASCON-cXOF-ABSORB($State, (Data \| 0^{*r})$)
    ASCON-PRNG-REKEY($State$)
end function

function ASCON-PRNG-FETCH($State, L$)                  ▷ Fetch $L$ bits of random data
    if $counter \geq limit$ then
        ASCON-PRNG-RESEED($State$)
    end if
    $R \leftarrow$ ASCON-cXOF-SQUEEZE($State, L$)
    ASCON-PRNG-REKEY($State$)
    $counter \leftarrow counter + L$
    return $R$
end function

function ASCON-PRNG-RESEED($State$)                    ▷ Re-seed the PRNG from the system TRNG
    $Seed \leftarrow trng\_get()$
    ASCON-cXOF-ABSORB($State, (Seed \| 0^{*r})$)
    ASCON-PRNG-REKEY($State$)
    $counter \leftarrow 0$
end function

function ASCON-PRNG-REKEY($State$)                     ▷ Re-key the PRNG to provide forward security
    for $i = 1, \dots, \lceil (320 - r)/r \rceil$ do
        $State \leftarrow (0^r \| \lceil State \rceil_{320-r})$
        $State \leftarrow p^a(State)$
    end for
end function
```

10 Key wrapping

Synthethic Initialization Vector (SIV) is a special case of Authenticated Encryption with Associated Data (AEAD) that is optimized for wrapping symmetric keys, asymmetric key pairs, passwords, or other sensitive keying material.

ASCON-128, ASCON-128A, and ASCON-80PQ are not suitable for key wrapping as they are not secure if the same key and nonce combination is used to encrypt different plaintexts.

In the second round of the NIST Lightweight Cryptography Competition, Romulus-M [12] and SUNDAE-GIFT [4] provided SIV capabilities. Romulus-M made it into the third and final round of the competition.

Both candidates are built as two-pass algorithms. On the first pass, the key and nonce are used to hash the associated data and plaintext to produce the authentication tag. On the second pass, the authentication tag is used as a new nonce to encrypt the plaintext with the original key. Authentication of the plaintext is skipped on the second pass, with the underlying block cipher or permutation operated in a simple CTR or OFB encryption mode.

It is simple to restructure the ASCON AEAD modes into a two-pass form without changing too much of the original design from [10]; see Figure 2. In fact, the first pass of SIV encryption is identical to the regular AEAD mode, but with the ciphertext discarded and replaced with the output from the second pass. We use IV1 and IV2 values that are distinct from the IV values for regular AEAD modes to provide domain separation (see Table 2).

If the same key, nonce, and associated data is used to encrypt a different plaintext, then the ciphertext will be completely different. If the same key, nonce, and associated data is used to encrypt a previous plaintext, then the SIV mode will leak that the same plaintext has been encrypted again but the plaintext itself will not be revealed.

The security of this construction should be equivalent to the regular AEAD mode, but this needs further study.

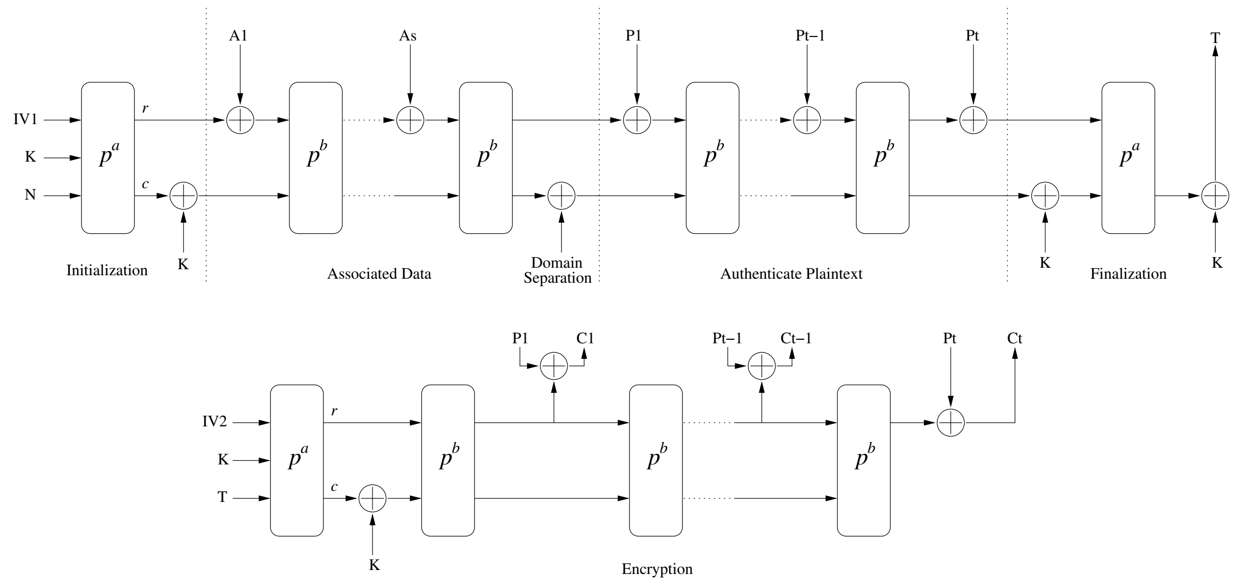

Figure 2: ASCON SIV encryption mode

Table 2: IV values for ASCON SIV modes

	AEAD IV	*SIV IV1*	*SIV IV2*
ASCON-128	0x80400c0600000000	0x81400c0600000000	0x82400c0600000000
ASCON-128a	0x80800c0800000000	0x81800c0800000000	0x82800c0800000000
ASCON-80pq	0xa0400c06	0xa1400c06	0xa2400c06

11 Tweakable block cipher (experimental)

There are two applications where a standard permutation-based encryption mode is not suitable because simply XOR'ing the output of a permutation with the plaintext is not secure: on-the-fly memory encryption and disk encryption. SIV modes are also not suitable because they increase the size of the plaintext to include the mandatory authentication tag.

Applications that perform memory or disk encryption require a tweakable block cipher. For memory encryption, the memory address is used as the tweak. For disk encryption, the disk block address and offset within the block are combined to form the tweak.

11.1 Block ciphers from other finalists

Skinny-128-384+ from Romulus [12] is a tweakable block cipher. It would have been a natural fit for memory and disk encryption if Romulus had been chosen as the competition winner.

GIFT-128 from the finalist GIFT-COFB [5] is a 128-bit block cipher but it is not tweakable in its default formulation. This isn't a problem for the standard XEX [26] and XTS [15] modes, as they were originally designed for untweaked AES.

During previous rounds of the competition, ESTATE [8] modified GIFT-128 to include a 4-bit tweak value which was used to provide domain separation at different stages of the ESTATE processing. This tweak size is insufficient for memory and disk encryption, so an alternate tweak formulation would be required.

11.2 Building Feistel networks with ASCON

It is possible to construct a block cipher out of any pseudo-random function and a Feistel network using the Luby-Rackoff construction [19] [21]. The psuedo-random function does not need to be reversible. If the rate of a permutation-based sponge is r bits, then we can construct a $2r$-bit block cipher out of the permutation as follows:

1. Break the $2r$-bit plaintext block into two r-bit halves L and R.
2. Form an input block B for the permutation function p^a by concatenating R, the key K, the tweak T, and a round constant rc_i.
3. Compute $B' = p^a(B)$ and XOR the first r bits of B' with L.
4. Swap L and R on all rounds except the last.
5. Repeat the previous 3 steps for the remaining rounds.
6. Concatenate the final L and R values to produce the $2r$-bit ciphertext block.

The recommended number of rounds in [19] and [21] varies depending upon the type of attack that we wish to defend against: 4 rounds can protect against known plaintext attacks, 7 rounds against adaptive chosen plaintext attacks, and 10 rounds against adaptive chosen plaintext and chosen ciphertext attacks. Might as well use 10 for maximum coverage.

The performance of such a scheme is very dependent upon the number of permutation rounds a used in step 3 above; see Table 3. Reduced-round versions of the permutation can be used to offset the cost.

Variations are possible, such as precomputing key schedule elements $S_i = p^a(K \parallel rc_i)$ and then computing $B' = p^b(S_i \oplus (R \parallel T))$ during round i. Another variation might use the key setup procedure of ISAP [9] to expand the key schedule while providing some side channel protection to the key.

ASCON-SUITE contains an example implementation of ASCON-ECB, a block cipher with a 128-bit key, a 96-bit tweak, a 32-bit round constant, and $a = 6$. The cipher operates on 128-bit plaintext and ciphertext blocks. An alternative formulation could use a 160-bit key and a 64-bit tweak to provide more post quantum resistance.

ASCON-SUITE uses the first 10 round constants of SHA-256 as "nothing up my sleeve numbers" but almost any other method of choosing round constants should work. To avoid using the same key for every round, ASCON-SUITE doubles the key in the GF(128) field each round.

Algorithm 3 provides a formal definition of ASCON-ECB as it is implemented in ASCON-SUITE.

Table 3: Performance of ASCON-ECB for different a values

	Block size (bits)	*Permutation rounds per block*	*Rounds per byte*
ASCON-128A	128	8	0.500
ASCON-ECB, $a = 1$	128	10	0.625
ASCON-128	64	6	0.750
ASCON-HASHA	64	8	1.000
ASCON-ECB, $a = 2$	128	20	1.250
ASCON-HASH	64	12	1.500
ASCON-ECB, $a = 3$	128	30	1.875
ASCON-ECB, $a = 4$	128	40	2.500
ASCON-ECB, $a = 6$	128	60	3.750
ASCON-ECB, $a = 8$	128	80	5.000
ASCON-ECB, $a = 12$	128	120	7.500

This approach could be useful as a last ditch method for memory and disk encryption on devices that only implement ASCON. Obviously a lot of research is required to prove the security of using the ASCON permutation in this way, and to find the minimum a value that remains secure.

Algorithm 3 Tweakable block cipher ASCON-ECB

```
Input: $K \in \{0,1\}^{128}$, $T \in \{0,1\}^{96}$, $P \in \{0,1\}^{128}$, $1 \leq a \leq 12$
Output: $C \in \{0,1\}^{128}$

function ASCON-ECB-ENCRYPT($K, T, P, a$)
    $L \| R \leftarrow P$                                  ▷ Split plaintext into two halves
    for $i = 1, \ldots, 10$ do
        $S \leftarrow (R \| K \| T \| encode(rc_i, 32))$        ▷ Construct round input $S$
        $L \leftarrow L \oplus \lfloor p^a(S) \rfloor_{64}$            ▷ XOR $L$ with permuted $S$
        $K \leftarrow dbl(K)$                              ▷ Adjust $K$ for the next round
        $L, R \leftarrow R, L$                              ▷ Swap $L$ and $R$
    end for
    return $C \leftarrow R \| L$                            ▷ Return ciphertext
end function

function ASCON-ECB-DECRYPT($K, T, C, a$)
    $L \| R \leftarrow C$                                  ▷ Split ciphertext into two halves
    $K \leftarrow dbl^{10}(K)$                              ▷ Fast-forward the key schedule
    for $i = 10, \ldots, 1$ do
        $K \leftarrow dbl^{-1}(K)$                         ▷ Adjust $K$ for this round
        $S \leftarrow (R \| K \| T \| encode(rc_i, 32))$        ▷ Construct round input $S$
        $L \leftarrow L \oplus \lfloor p^a(S) \rfloor_{64}$            ▷ XOR $L$ with permuted $S$
        $L, R \leftarrow R, L$                              ▷ Swap $L$ and $R$
    end for
    return $P \leftarrow R \| L$                            ▷ Return plaintext
end function
```

12 Other modes

TupleHash and ParallelHash instances can be defined using ASCON-CXOF in a similar manner to how NIST SP 800-185 [16] uses cSHAKE. It is unclear whether ParallelHash would be useful on resource-constrained devices, although TupleHash might.

13 Summary

This paper describes many new modes for ASCON based on well-known designs in the literature: KMAC, KDF, PRNG, SIV, HMAC, HKDF, and PBKDF2.

A key contribution of this paper is a formal definition for the customizable hashing mode ASCON-CXOF which we suggest for inclusion in the final NIST lightweight cryptography standard.

The following areas are suggested for further research:

- A memory-hard password hashing mode for embedded devices that is hard enough to deter crackers with access to significant resources, but not hard enough for the memory requirements to be prohibitively expensive on embedded devices.
- The security of the Synthethic Initialization Vector (mode) described in section 10 should follow directly from the security of the standard AEAD modes, but this has not been formally proven.
- A tweakable block cipher mode based on the ASCON permutation that can be used for memory and disk encryption on embedded devices. An example using the Luby-Rackoff construction is provided in this paper, but a security analysis has not yet been attempted.

Revision History

1.0
  First version of this paper, March 2023.

References

[1] Strobe protocol framework. `https://strobe.sourceforge.io/`.

[2] Password Hashing Competition (PHC), 2013. `https://password-hashing.net/`.

[3] Lightweight Cryptography Standardization Process: NIST Selects Ascon, 2023. `https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon`.

[4] S. Banik, A. Bogdanov, T. Peyrin, Y. Sasaki, S. M. Sim, E. Tischhauser, and Y. Todo. SUNDAE-GIFT v1.0, 2019. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/SUNDAE-GIFT-spec-round2.pdf`.

[5] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. Meng Sim, and Y. Todo. GIFT-COFB v1.1, 2021. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf`.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2010.

[7] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks. Cryptology ePrint Archive, Paper 2016/027, 2016. `https://eprint.iacr.org/2016/027`.

[8] A. Chakraborti, N. Datta, A. Jha, C.M. Lopez, M. Nandi, and Y. Sasaki. ESTATE, 2019. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/estate-spec-round2.pdf`.

[9] C. Dobraunig, E. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. ISAP v2.0 submission to NIST, 2021. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/isap-spec-final.pdf`.

[10] C. Dobraunig, M. Eichlseder, F. Mendel, and M Schläffer. ASCON v1.2 Submission to NIST, 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf.

[11] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon PRF, MAC, and Short-Input MAC. Cryptology ePrint Archive, Paper 2021/1574, 2021. https://eprint.iacr.org/2021/1574.

[12] C. Guo, T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus v1.3, 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf.

[13] D. Harkins. Synthetic initialization vector (SIV) authenticated encryption using the advanced encryption standard (AES). *RFC*, 5297:1–26, 2008. https://tools.ietf.org/html/rfc5297.

[14] IEEE Power and Energy Society. IEEE Std 1686-2022 – IEEE Standard for Intelligent Electronic Devices Cybersecurity Capabilities, 2022.

[15] J. Kelsey, S. Chang, and R. Perlner. NIST Special Publication 800-38E, Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, 2010. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38e.pdf.

[16] J. Kelsey, S. Chang, and R. Perlner. NIST Special Publication 800-185, SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf.

[17] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: keyed-hashing for message authentication. *RFC*, 2104:1–11, 1997. https://tools.ietf.org/html/rfc2104.

[18] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (HKDF). *RFC*, 5869:1–14, 2010. https://tools.ietf.org/html/rfc5869.

[19] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.

[20] K. Moriarty, B. Kaliski, and A. Rusch. PKCS #5: Password-based cryptography specification version 2.1. *RFC*, 8018:1–40, 2017. https://tools.ietf.org/html/rfc8018.

[21] J. Patarin. Luby-rackoff: 7 rounds are enough for $2^{\text{n(1-epsilon)}}$ security. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 513–529. Springer, 2003.

[22] C. Percival and S. Josefsson. The scrypt password-based key derivation function. *RFC*, 7914:1–16, 2016.

[23] T. Perrin. The Noise protocol framework, revision 34, 2018. http://noiseprotocol.org/noise.html.

[24] T. Perrin. Stateful Hash Objects: API and Constructions, Revision 1, 2018. https://github.com/noiseprotocol/sho_spec/blob/master/output/sho.pdf.

[25] E. Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[26] P. Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC, 2004. Dept. Of Computer Science, University of California, Davis., https://www.cs.ucdavis.edu/~rogaway/papers/offsets.pdf.

[27] R. Weatherley. Additional modes for LWC finalists, 2021. https://github.com/rweather/lwc-finalists/blob/master/doc/lwc-modes-v1-0.pdf.