Additional Modes for LWC Finalists
Technical Report, Version 1.0

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

13 June 2021

Abstract

The NIST Lightweight Cryptography Competition published a list of ten finalists in March 2021. The submissions to the competition took the form of an AEAD mode coupled with optional hashing or 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 technical report catalogues additional modes that can be deployed on top of the finalists.

Contents

List of Figures

List of Tables

1 Introduction

The NIST Lightweight Cryptography Competition published a list of ten finalists in March 2021 [3]. The submissions to the competition took the form of an AEAD mode coupled with optional hashing or 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 technical report catalogues additional modes that can be deployed on top of the competition finalists, preferably using proven designs from the literature. Sections marked with "experimental" contain speculative new designs that require further study and security analysis.

Finalists with a hashing mode can easily implement keyed message authentication, key derivation, and password hashing using existing designs such as HMAC [16], KMAC [15], HKDF [17], and PBKDF2 [19].

Pseudo-random number generation can be implemented with either the SpongePRNG [5] or CTR_DRBG [2] constructions depending upon whether the finalist is permutation based or block cipher based.

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) [10] modes can be deployed to wrap sensitive key material in trusted or untrusted storage. Some of the round 2 submissions to the competition had SIV modes, which we use as a guide for defining similar modes for the finalists. Of the final round submissions, only Romulus has a native SIV mode, Romulus-M.

2 Keyed message authentication

2.1 AEAD modes as authenticators

AEAD modes are by their nature keyed message authenticators already, generating a tag to authenticate the associated data and plaintext. A simple keyed authentication mode can be constructed by setting the plaintext to empty and authenticating the message as associated data under a key and nonce.

Some finalists use a version of the permutation with less rounds (TinyJAMBU) or a larger rate (Xoodyak) to process associated data. This does not employ the full strength of the AEAD mode. It is possible to work around this by encrypting the message and discarding all of the ciphertext except for the tag. The receiver would re-encrypt the message and check that the tags are identical.

The main drawback with using the AEAD mode for authentication is the small size of the tags; 64-bit or 128-bit in most cases. This is an insufficient security margin for scenarios where HMAC or KMAC modes have been used in the past.

2.2 HMAC

ASCON-HASH, PHOTON-Beetle-Hash, Romulus-H, ESCH256, and XOODYAK-Hash all have 256-bit digest outputs. ESCH384 has a 384-bit digest output.

HMAC [16] is a widely-implemented standard for creating keyed message authenticators from simple digest algorithms. While permutation-based sponge modes 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, 16 bytes for XOODYAK-Hash, and so on).

For backwards compatibility, we use SHA-256's block size of 64 bytes for algorithms with a 256-bit output, and SHA-384's block size of 128 bytes for algorithms with a 384-bit output.

HMAC requires that the key be injected at both the start and end of the processing. Also, keys longer than the block size are first hashed down to a shorter digest. These features can require a 64 or 128 byte temporary buffer to hold the formatted key block. This may be a concern in memory-constrained embedded environments.

Our implementation avoids the need for a full B-sized block of memory to format the key by requiring that it be provided twice at the start and the end of the processing. The key is formatted on the fly each time. This avoids the memory penalty but incurs a performance cost. It also prevents online operation where the key is only available at the start of the processing; the calling application must make provision to preserve the key value for later.

HMAC is implemented in our work for ASCON, PHOTON-Beetle, Romulus, SPARKLE, and XOODYAK.

2.3 KMAC

NIST SP 800-185 [15] defines an extension to the XOF modes SHAKE128 and SHAKE256 of SHA-3 to provide keyed message authentication (KMAC).

Finalists with a defined XOF mode can implement KMAC almost as-is, replacing SHAKE with the algorithm-specific XOF. This applies for ASCON-XOF, XOESCH256, XOESCH384, and XOODYAK-Hash.

The algorithm update document for the final round version of Romulus [12] suggests using the mask generation function MGF1 [13] to build an XOF mode out of Romulus-H. This suggestion doesn't appear in the final round specification [9] but seems a reasonable approach. We implemented the suggested XOF mode for Romulus as part of this work.

PHOTON-Beetle-Hash could also be used but the authors have not yet defined an official XOF mode for their submission at the time of writing. Like ASCON and XOODYAK-Hash, XOF output could be created by continuing to generate output blocks after the basic 256-bit hash.

The KMAC specification encodes the desired output length at the end of the input data that is absorbed into the sponge. ASCON has its own method to encode the desired output length into the 64-bit domain separation value in the initialization vector (where 0 means unlimited):

$$IV = \texttt{0x00400c0000000000} + output\_length\_in\_bits$$

We don't use this method at present, preferring to build ASCON-KMAC directly on top of ASCON-XOF. This keeps the API consistent across different underlying XOF algorithms.

NIST SP 800-185 also defines the modes cSHAKE, TupleHash, and ParallelHash, using the same underlying components as KMAC. We have not provided these modes in this work but they could be built in a similar way.

3 Key derivation functions (KDF's)

3.1 Key stretching for protocols

Secure communications protocols like TLS [23] and Noise [22] 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 [17] is the usual method, but KDF's based on KMAC could also be used. We provide HKDF implementations for ASCON, PHOTON-Beetle, Romulus, SPARKLE, and XOODYAK.

3.2 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 1686-2013 [11].

PBKDF2 [19] 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 [21] and the finalists to the Password Hashing Competition (PHC) [1].

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.

The winner and finalists of the Password Hashing Competition are still being investigated to determine if reasonable parameter choices can be made to make them suitable for devices with a small to medium amount of memory.

More research is required to identify a password hashing scheme that has reasonable performance and low memory requirements on small systems, but which quickly degenerates to memory-hard on cracking systems.

We provide PBKDF2 implementations for ASCON, PHOTON-Beetle, Romulus, SPARKLE, and XOODYAK.

4 Pseudo-random number generation

Embedded CPU's are increasingly fitted with 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 CPU's may "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.

4.1 Unique device identification

The PRNG has a hashed pool of static identification information h which is used to set up up a PRNG state for generating output. The application mixes serial numbers, MAC addresses, and other unique identifiers into the hashed pool at startup to ensure that every device generates a unique sequence of random bytes even if the the output from the system TRNG is accidentally identical. The **ADD_IDENT** function is used for this purpose:

ADD_IDENT(*data*) Adds *data* to the hashed pool of static identification information with h = **HASH**($h \parallel data$). Here, **HASH** is the digest hash algorithm or equivalent for the finalist.

4.2 PRNG API

Each PRNG implementation provides the following API with different details depending upon the underlying block operation that is used:

INIT(*s*) Initializes the PRNG state s with h and then calls **RESEED**(s).

RESEED(*s*) Explicitly re-seeds and re-keys the PRNG state s using more data from the system TRNG. The system TRNG is assumed to provide 256 bits of random data every time it is called.

FEED(*s*, *data*) Feeds *data* into the PRNG state s and then performs a re-key. This is intended for mixing data from other entropy sources into s.

***output* = FETCH(*s*, *n*)** Fetches n bytes of *output* from the PRNG state s. Will immediately re-key s after the n bytes have been generated to enforce forward security. The state s will be re-seeded every 16K bytes of output.

FREE(*s*) Frees the PRNG state s when the application is finished using it. All sensitive state will be destroyed.

***output* = GENERATE(*n*)** Fetches 256 bits of data from the system TRNG and uses it to seed a temporary PRNG state and generate n bytes of random *output*. This is for quick one-off generation of random byte sequences such as nonces, password salts, and ephemeral keys.

The 16K re-seeding interval can be adjusted by the application. It is intended to provide a trade-off between performance and periodic forced refresh of the entropy in s from the system TRNG.

The PRNG implementation could keep track of how long it has been since the previous re-seed operation to ensure that the seed is fresh if some time has passed since the last fetch operation. We haven't implemented this yet. We assume that the application will re-seed explicitly if it knows that some time has elapsed between requests. Or it will free the PRNG state and re-initialize it again later.

4.3 Preserving entropy across a system restart

Some system TRNG's may produce poor output or have very little entropy available at startup. If the device has non-volatile storage, then the application may wish to preserve some of the hard-won entropy from one system run to another.

The simplest approach is to call **FETCH** or **GENERATE** periodically to generate at least 32 bytes to be saved. Upon a system restart, the saved data is passed to **ADD_IDENT** to be hashed into h.

It is recommended that the saved data be overwritten at system startup with newly generated data in case the device loses power before another entropy save can occur. This will hopefully prevent the system from restarting in the same state repeatedly.

The time period between entropy saves is a trade-off between system performance and wear on the non-volatile storage medium. Once a day should be sufficient for long-running devices and will provide 27 years of operation on flash memory that is rated for 10,000 erase/write cycles.

4.4 Sponge-based PRNG modes

SpongePRNG [5] is a technique for converting a permutation-based sponge function into a PRNG. API's are provided to absorb seed material as it becomes available and to generate 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 unfeasible to reverse the PRNG's state without guessing the rate bits before they were zeroed. This is the re-keying process for sponge-based PRNG's.

SpongePRNG is a natural fit for building PRNG's for ASCON, Elephant, PHOTON-Beetle, SPARKLE, and XOODYAK. The SpongePRNG paper provides an example using Keccak-f[200], which is also used by the Elephant variant Delirium.

We try to operate the permutation for each finalist in the same manner as the corresponding hashing mode, with the same rate and number of rounds. In the case of Elephant which does not have a hashing mode, the permutation is operated as recommended by the SpongePRNG paper.

Reusing the same rate and number of rounds may make it possible to use the same hardware as the regular modes on devices that accelerate the permutation in hardware. It also allows us to inherit the security strength claims of the finalist's hashing mode. Table 1 summarizes the parameter choices.

Table 1: Parameters for sponge-based PRNG modes

Finalist	*Permutation*	*Rate (bits)*	*Rounds*	*State Size (bits)*
ASCON	ASCON	64	12	320
Elephant	Keccak-f[200]	64	18	200
PHOTON-Beetle	PHOTON_{256}	32	12	256
SPARKLE	SPARKLE384	128	7 / 11	384
XOODYAK	XOODOO	128	12	384

The SPARKLE hashing mode ESCH256 uses 7 rounds to absorb all input blocks except the last, for which 11 rounds is used. SPARKLE also defines a function $\mathcal{M}_3$ for transforming input data before it is injected into the state. We try to preserve these behaviours for consistency with ECSH256.

4.5 Block cipher based PRNG modes

NIST-800-90A [2] defines a PRNG mode called CTR_DRBG which is based on a block cipher and the CTR block mode. The state consists of a *Key* and a rolling state value V. The seed length, *seedlen*, is recommended to be the same size as concatenating *Key* and V.

CTR_DRBG_Instantiate initializes the PRNG with an *entropy_input* and a *personalization_string* which correspond to the TRNG seed and *h* in our case.

CTR_DRBG_Update and **CTR_DRBG_Reseed** feed data into the PRNG by XOR'ing it with some CTR output to generate new *Key* and *V* values. The data may be preprocessed with a derivation function if it is not exactly *seedlen* in size.

CTR_DRBG_Generate generates the output data in CTR mode and then generates a few more CTR blocks for new *Key* and *V* values.

GIFT-128 is a natural fit for this design. With a 128-bit key, NIST recommends a *seedlen* of 256 bits, which is what we get from our TRNG.

Skinny-128-384+ can be used in the same way, although its *seedlen* would need to be 512 bits if we follow the NIST design. To address that, we use a hash-based derivation function based on Romulus-H to expand the 256-bit TRNG seed to 512 bits during **CTR_DRBG_Instantiate**.

An alternate process is used if re-seeding is required during **FETCH**: XOR the TRNG seed with the TK2 and TK3 parts of the tweakey and then generate four blocks of output for a new *Key* and *V*. This was more stack efficient than hashing the TRNG seed and calling **CTR_DRBG_Update**.

Other options are possible with Skinny-128-384+. For example, put a 256-bit *Key* in TK2/TK3, and use TK1 as a LFSR block counter rather than incrementing *V* each block. This reduces the *seedlen* to 384 and makes the PRNG closer to Romulus-N in its structure. This may allow some of the AEAD code or hardware to be reused.

For now we have implemented CTR_DRBG for Skinny-128-384+ with a 512-bit internal state and the initialization sequence modified as described above. Romulus-H is used as a hash-based derivation function when re-seeding or feeding data into the PRNG state. Note that while the *Key* is 384 bits in length, the design of Skinny means that the overall PRNG is equivalent in strength to 128 bits.

NIST allows the counter for CTR mode to be less than the full block size of the cipher. We have chosen to use 16-bit counters, which allows up to 1M of output before re-seeding is forced. Since we plan to reseed every 16K anyway, this limitation isn't a problem for us.

Due to the size of the key schedules, our CTR_DRBG implementations are quite memory hungry compared to the sponge-based PRNG's. In a practical implementation, it may be better to use versions of the block cipher that expand the key schedule on the fly during encryption.

4.6 AEAD based PRNG modes (experimental)

Grain-128AEAD and TinyJAMBU do not lend themselves to either SpongePRNG or CTR_DRBG. We therefore use a variation on the AEAD modes to generate random numbers instead.

RESEED fetches data from the system TRNG, XOR's it with the existing key to preserve any existing entropy in the PRNG, and then re-keys the PRNG. **FEED** absorbs data into the AEAD state in the same manner as associated data and then re-keys. **FETCH** operations encrypt a plaintext sequence of zeroes with the ciphertext being the random output.

Re-keying generates 256 bits of output which is used as a new key to re-initialize the cipher. The AEAD nonce value is set to the number of times that re-keying has been done so far. The static pool *h* is absorbed as associated data or as part of the nonce.

We specified above that the system TRNG returns 256 bits each time it is called. That makes TinyJAMBU-256 a natural fit for random number generation.

ADD_IDENT hashes the existing static pool and the new data to create the hash value for the new static pool. Grain-128AEAD and TinyJAMBU do not have hashing modes, but we can operate the PRNG itself as a hash:

1. Initialize the PRNG with an all-zeroes key and nonce.
2. Absorb *h* and the new *data* in the same manner as **FEED**.
3. Call **FETCH** to generate the new *h*.

Grain-128AEAD needs some special handling to turn the AEAD mode into a PRNG. First, it uses a 128-bit key instead of our preferred 256-bit seed. For simplicity, we use the first half of the 256-bit TRNG seed as the AEAD key and absorb the rest as associated data.

Grain-128AEAD absorbs associated data into a separate shift register instead of the main LFSR/NFSR state. But we need the result of **RESEED** and **FEED** operations to be absorbed into the main state. We handle this by feeding the associated data into the LFSR instead, 32 bits at a time.

Normally Grain-128AEAD encrypts with even bits from the keystream and authenticates with odd bits from the keystream. We don't need authentication, so we use all bits as random output from **FETCH**.

5 Key wrapping (SIV mode)

5.1 Competition candidates with SIV modes

Romulus-M was designed as a SIV mode, so it can be used directly for key wrapping.

SUNDAE-GIFT [4] was a sister algorithm to GIFT-COFB during round 2 of the competition that supported a SIV mode natively. SUNDAE-GIFT could be used to provide key wrapping on devices that implement the GIFT-128 block cipher. Alternatively, the SIV key wrapping mode from RFC 5297 [10] could be used with GIFT-128 in place of AES.

5.2 SIV modes for non-SIV finalists (experimental)

SIV modes are typically 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 sponge operated in a simple CTR or OFB encryption mode.

We provide an example using ASCON. It is simple to restructure the ASCON AEAD modes into a two-pass form without changing too much of the original design from [8]; see Figure 1. 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).

The security of this construction should be equivalent to the regular AEAD mode, but this needs further study. The other finalists are still under investigation but it should be possible to convert many of them in a similar fashion.

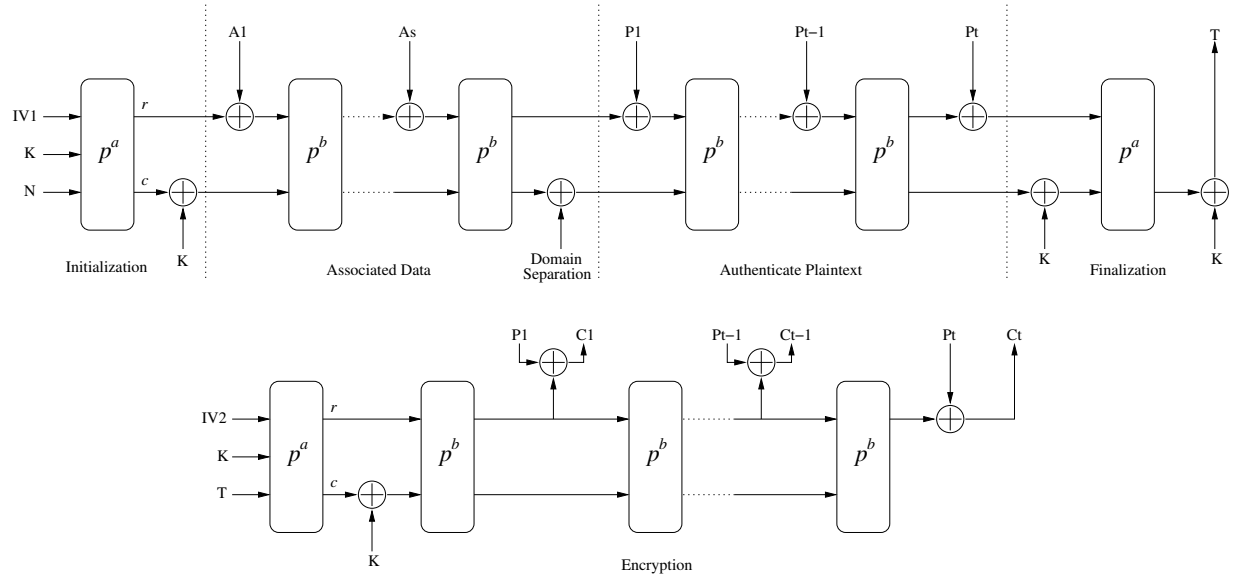

Figure 1: ASCON SIV encryption mode

Table 2: IV values for ASCON modes

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

6 Tweakable block ciphers (TBC's)

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. These applications require a tweakable block cipher or tweakable block cipher mode.

6.1 Skinny

Skinny-128-384+ from Romulus is already a tweakable block cipher so it is a natural fit for memory encryption and disk encryption.

For memory encryption, the memory address can be used as the tweak. For disk encryption, the disk block address and offset within the block can be combined to form the tweak.

6.2 GIFT-128

GIFT-128 is obviously a block cipher but it is not tweakable in its default formulation. This isn't a problem for the standard XEX [24] and XTS [14] modes, as they were originally designed for untweaked AES.

During previous rounds of the competition, ESTATE [6] modified GIFT-128 to include a 4-bit tweak value which was used to provide domain separation at different stages of the ESTATE processing. ESTATE combines an expanded version of the tweak value with the bit-sliced state word S_0 every 5 rounds. Four bits is insufficient for a memory address, but the technique could be extended to all 32 bits of S_0 (experimental).

6.3 Feistel networks (experimental)

It is possible to construct a block cipher out of any pseudo-random function and a Feistel network using the Luby-Rackoff construction [18] [20]. 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 $\mathcal{F}$ by concatenating R, the key K, the tweak T, and a round constant rc_i.
3. Compute $B' = \mathcal{F}(B)$ 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 [18] and [20] 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.

The performance of such a scheme may not be great compared with a dedicated block cipher design: it is between 2 and 5 times slower than the native AEAD or hashing mode with rate r (the rate of the above construction is $2r$). Reduced-round versions of the permutation could be used to offset the cost.

Variations are possible, such as precomputing key schedule elements $S_i = \mathcal{F}(K \parallel rc_i)$ and then computing $B' = \mathcal{F}(S_i \oplus (R \parallel T))$ during round i. Another variation might use the key setup procedure of ISAP [7] to expand the key schedule while providing some side channel protection to the key.

This approach could be useful as a last ditch method for memory and disk encryption on devices that only implement permutation-based cryptographic primitives.

Table 3 provides some possible parameter choices, but other choices are possible. The small size of the permutations for Elephant and PHOTON-Beetle probably makes this technique inadvisable for them.

Table 3: Parameters for sponge-based tweakable block ciphers

Finalist	*Permutation*	*Rate (bits)*	*Block (bits)*	*Key (bits)*	*Tweak (bits)*	*RC (bits)*
ASCON	ASCON	64	128	128	96	32
Elephant	Keccak-f[200]	32	64	128	32	8
PHOTON-Beetle	PHOTON_{256}	32	64	128	64	32
SPARKLE	SPARKLE384	128	256	128	96	32
SPARKLE	SPARKLE512	128	256	256	96	32
XOODYAK	XOODOO	128	256	128	96	32

6.4 TinyJAMBU (experimental)

It is possible to operate the TinyJAMBU permutation directly as a tweakable 128-bit block cipher. If the key and the full 128 bits of permutation output are known, then the permutation can be reversed.

For each 128-bit block of plaintext or ciphertext, first compute a block-specific key using a variation on TinyJAMBU's AEAD key/nonce setup procedure, as shown in Figure 2.

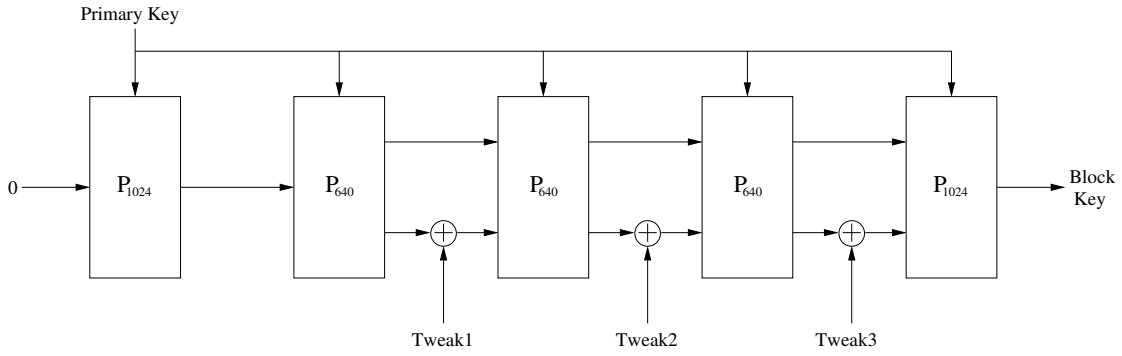

Figure 2: Generating a block key using TinyJAMBU

As for the TinyJAMBU AEAD nonce, the 96-bit tweak is split into three 32-bit chunks and absorbed into the permutation state with repeated applications of the permutation. A final permutation call is used to generate the block-specific key from the state. Encryption or decryption is performed as shown in Figure 3.

If the primary key and the leading 64 bits of the tweak are fixed, then it is possible to precompute the block key generation state to the point just after the third application of P_{640}. Then each encryption operation involves an XOR of the 32-bit memory offset Tweak3 into the state, followed by two applications of P_{1024} with the primary key and block key respectively. 192-bit and 256-bit keys can be used for the primary key by replacing P_{1024} with P_{1152} or P_{1280} respectively during generation of the block key.

This is a proposal only. Further study is required to determine if this construction is secure.

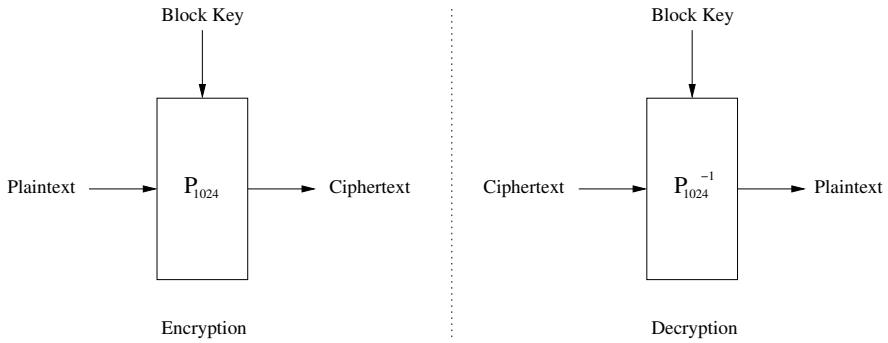

Figure 3: Encryption and decryption using TinyJAMBU as a block cipher

7 Other modes

7.1 ISAP

ISAP is a family of nonce-based authenticated ciphers with associated data (AEAD) designed with a focus on robustness against passive side-channel attacks [7]. ISAP can use either ASCON or Keccak-*p*[400] as the underlying permutation for the design. ASCON is the preferred permutation in the final round of the competition.

ISAP-A can be viewed as an additional mode for ASCON that provides protection against side-channel attacks. So we don't provide any more modes for ISAP in this work. All of our additional modes for ASCON can be applied on a device that uses ISAP-A for encryption.

In theory the ISAP design could be used as an additional mode for other permutations. Suitable parameters would need to be defined for ISAP's core elements in terms of the alternate permutations. We haven't attempted that here.

8 Summary

Table 4 summarizes the modes that were discussed above. Diamonds (♦) indicate modes that were part of the submissions to the final round of the competition. Bullet points (•) indicate modes that have been implemented as part of this work. Daggers (†) indicate modes that should be feasible as described in the text above but which have not been implemented yet. Questions marks (?) are still under investigation.

For the most part we prefer designs that have been proven in the literature. However, there are some experimental designs in the text above that are newly proposed by this work. Experimental designs are marked with a star (⋆) and should be treated with caution until a proper security analysis can be performed.

Table 4: Modes for LWC finalists

	AEAD	*HASH*	*XOF*	*HMAC*	*KMAC*	*HKDF*	*PBKDF2*	*PRNG*	*SIV*	*TBC*
ASCON	♦	♦	♦	•	•	•	•	•	•⋆	†⋆
Elephant	♦							•	?	
GIFT-COFB	♦							•	†	†
Grain-128AEAD	♦							•⋆	?	
ISAP	♦	Use ASCON's modes instead.								
PHOTON-Beetle	♦	♦	†	•	†	•	•	•	?	
Romulus	♦	♦	•	•	•	•	•	•	♦	♦
SPARKLE	♦	♦	♦	•	•	•	•	•	?	†⋆
TinyJAMBU	♦							•⋆	?	†⋆
XOODYAK	♦	♦	♦	•	•	•	•	•	?	†⋆

Revision History

1.0
: First version of this technical report.

References

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

[2] NIST Special Publication 800-90A, Recommendation for random number generation using deterministic random bit generators, 2015. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf.

[3] NIST: Lightweight Cryptography: Finalists, 2021. https://csrc.nist.gov/Projects/lightweight-cryptography/finalists.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] IEEE Power and Energy Society. IEEE Std 1686-2013 – IEEE Standard for Intelligent Electronic Devices Cyber Security Capabilities, 2013.

[12] T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus for round 3, 2020. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/status-update-sep2020/Romulus-for-Round-3.pdf.

[13] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specifications version 2.0. *RFC*, 2437:1–39, 1998.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] J. Patarin. Luby-rackoff: 7 rounds are enough for $2^{n(1\text{-epsilon})}$ 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.

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

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

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

[24] 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.