ARX-KW, a family of key wrapping constructions using SipHash and ChaCha

Satō Shinichi*

January 20, 2020

Abstract

ARX-KW is a family of key wrapping construction based on add-rotate-xor primitives: the pseudo-random function SipHash for authentication and the stream cipher ChaCha for confidentiality. This paper presents ARX-KW, proposes a specific instantiation of ARX-KW and details the design decisions that were made.

1 Necessity for a new key wrapping construction

Key wrapping algorithms intend to protect cryptographic keys by encrypting the keys with a key encryption key ("KEK"). There is no specific requirement to use a key wrapping algorithm; authenticated encryption works just as well.[5] The rest of this section explains why existing constructions using the ChaCha stream cipher for authenticated encryption are not practical for key wrapping.

Unlike a block cipher operating in ECB mode, which only requires a key, the ciphers in the ChaCha family always require a nonce (number used once) and a block counter. Reusing the same pair of nonce and block counter with the same key leads a loss of confidentiality: If two ciphertexts are xored from the same ChaCha block, the xor of the plaintexts is revealed to an attacker that has both of the ciphertexts, which allows recovering the plaintexts. It is therefore vital that a ChaCha block is never repeated for the same key.

Given this, it is necessary to manage nonces and block counters. Typically, the block counter for every new encryption or decryption operations is set to 0, reducing the issue to a nonce management problem instead.

ChaCha is specified in multiple variants. Daniel J. Bernstein's original specification made use of a 64-bit nonce with a 64-bit block counter.[3] The variant of ChaCha20 specified by the IETF in RFC 8439 specifies a 96-bit nonce with a 32-bit block counter.[8] Finally, there is a variant called XChaCha20, which uses a 192-bit nonce to derive a new key and nonce with one extra invocation of ChaCha20;[1] it uses the same method as XSalsa20 by Daniel J. Bernstein to extend the nonce of Salsa20.[4]

*The author is not currently affiliated with any company or academic institution. Contact: sato@airmail.cc

Of those, only the 192-bit nonce of XChaCha20 can be chosen at random for many invocations. Due to the birthday paradox, a 96-bit nonce will be reused with 50 % probability at $2^{96/2}$ or 2^{48} invocations. In the context of DRM systems such as Valve's Steam platform that must handle up to several thousands of keys, this may be an unacceptable probability.

Given that choosing a nonce at random is not an option, the wrapping process must be stateful to avoid nonce reuse. This is not an issue when unwrapping because the nonce can be stored along with the ciphertext. Alternatively, the nonce and counter could be treated as a 128-bit number and then that could be chosen at random for a 50 % collision probability of only after $2^{128/2}$ or 2^{64} invocations; for the purpose of this paper, a nonce and counter pair shall be called the *lower quarter block* or *LQB* for short because the last four words of a 16-word ChaCha matrix are used for the nonce and counter.

The LQB must therefore either be stateful or chosen at random. If it is stateful, then the key wrapping process is necessarily stateful; the key unwrapping process may be stateless if the statefully generated nonce is stored along with the wrapped key (leading to a size overhead) or the key unwrapping process may be stateful as well (leading to implementation complexity). If it is chosen at random, then the key wrapping process and the key unwrapping process may be stateless, but the nonce is necessarily larger and must be stored along with the wrapped key in all cases.

Assume that the authentication tag is 128 bits in length and that a single 256-bit key is being wrapped, this leads to a base overhead of 384 bits, plus 64 to 192 bits of nonce if the unwrapping process must be stateless. To reduce the storage overhead while also allowing operation without keeping state for the nonce, ARX-KW is proposed.

2 Prior work

Before proposing ARX-KW, a brief overview of existing techniques is given. The explanations are deliberately simplified down to the actual core ideas of each technique.

The **NIST key wrapping algorithm** KW-AE requires a block cipher that operates on 128-bit blocks.[5] It works by splitting the input into half-blocks of 64-bits starting with a static 64-bit initialization vector (IV). The IV is concatenated with the first half-block of input and processed as an input block to the block cipher. The first half-block of the ciphertext is stored. Then the second half-block of the ciphertext is concatenated with the second half-block of the ciphertext xor a round number. The first half-block of the ciphertext is stored. Repeat this until there are no more plaintext half-blocks. On decryption, this process is reversed; the plaintext is considered authenticated if the first plaintext half-block equals the static IV.

Another algorithm proposed for key wrapping is the **SIV block cipher mode of operation** by *Rogaway* and *Shrimpton*.[9] It generates a counter for the CTR block cipher mode of operation using a function called S2V. S2V takes an arbitrary amount of inputs, which form additional authenticated data, except for the last, which is the plaintext; each input is processed using CMAC. The resulting MACs are xored into an accumulator, which is doubled and reduced modulo 2^128 between each step; the final MAC is also xored with the last n

bits of plaintext, where n is the size of a block cipher block (and the size of the accumulator). The output of S2V is used as a counter value for the actual encryption operation.

The **GCM-SIV block cipher mode of operation** proposed by *Gueron* and *Lindell* works similarly.[6] It generates a counter for the CTR block cipher mode of operation using GHASH – a keyed universal hash function – over the additional data and plaintext (plus a block that encodes the lengths of the two inputs) and encrypting that with a subkey. Some of the lower bits of the counter are cleared, assuming no overlap in the higher bits, to avoid overlapping counter values. AES-GCM-SIV sacrifices another bit in the counter to encrypt the generated counter using the same key as used for the actual encryption.

Various aspects of these algorithms are unsuitable for a construction around the ChaCha family of ciphers, however: KW-AE is tied to the properties of a block cipher; ChaCha being a family of stream ciphers cannot be adapted to it whatsoever. SIV relies on CMAC and takes extra steps to be able to handle arbitrary amounts of pieces of additional data; CMAC cannot be replicated with stream ciphers, and while the measures taken to accommodate multiple pieces of additional data are not an issue per se, simplifications can be made if it is known that there will be no additional data. GCM-SIV is oriented around the CTR block cipher mode of operation, which makes it effectively oriented around stream ciphers and thus suitable; in fact, the AES-GCM-SIV construction can be trivially instantiated with Poly1305 and ChaCha instead of GHASH and AES-CTR, respectively. However, the effective resulting 95-bit nonce may be uncomfortably small.

The core idea from the SIV constructions, however, is clear: Generate a counter for the CTR block cipher mode of operation given a block cipher by using a message authenticator over the plaintext. As the CTR block cipher mode of operation effectively turns a block cipher into a stream cipher, this core idea can be reused. In particular, *Gueron* and *Lindell* have shown that the Universal-SIV scheme is safe to use;[6] the Universal-SIV scheme requires an ϵ-XOR universal hash function processed by a pseudo-random function (PRF).

3 Generating the nonce

A randomly chosen nonce and counter may be used to process a message larger than one block; each block causes the counter to be incremented by one, so that the probability of a collision increases with the total size of all processed messages. Unlike general-purpose encryption, key wrapping involves *short*, high-entropy inputs; this issue is therefore intentionally ignored in this paper. A ChaCha block is 64 bytes in length, thus allowing encryption of up to $512\tilde{}$bits of key material, or two 256-bit ChaCha keys.

The base nonce generation works the same for the two base variants of ARX-KW. The nonce is generated using an instantiation of SipHash. SipHash is a family of pseudo-random functions; as all PRFs are also MACs, it is suitable as the nonce generation function for ARX-KW, combining the element an ϵ-XOR universal hash and the subsequent invocation of a PRF as required by *Gueron* and *Lindell* into the single SipHash operation. Additionally, SipHash is optimized for short inputs,[2] which is especially useful in the context of key wrapping. Extended-nonce variants for use with XChaCha prepend a static

value to the tag generated by SipHash to obtain the nonce.

Unlike GCM-SIV, the output of SipHash can be used without first being encrypted because of its properties as a PRF. Similarly unlike AES-SIV, ARX-KW does not need to take special measures to preserve boundaries of inputs – such as doubling an accumulator – as there is no additional data. This allows ARX-KW to have the most simple possible way of using the output of the authenticator: The output can be used without any changes to itself.

SipHash is always used with 128-bit output. While a 64-bit authentication tag would be sufficient for the key (as demonstrated by the 64-bit tag in NIST KW-AE), a 128-bit nonce is required for a comfortable security margin as noted in section 1.

4 Specification of the ARX-KW constructions

ARX-KW is a cipher for deterministic authenticated encryption without additional data. It is specified in four base variants:

E ARX-KW-*R*-*c*-*d*-E[1] takes a 386-bit key. The first 128 bits are used as the key for SipHash *c*-*d*, the remaining 256 bits are used as the key for ChaCha*R*.

G ARX-KW-*R*-*c*-*d*-G[2] takes a 256-bit key. The 256-bit key is used to encrypt 384 zero bits with ChaCha*R*, of which the first 128 bits are used as the key for SipHash *c*-*d*, the remaining 256 bits are used as the key for ChaCha*R*.

EX ARX-KW-*R*-*c*-*d*-EX[3] works like the E variant, but uses XChaCha*R* instead of ChaCha*R*. The nonce is generated by prepending the string "arbitrEX" (encoded in ASCII) to the SipHash output.

GX ARX-KW-*R*-*c*-*d*-GX works like the G variant, but uses XChaCha*R* instead of ChaCha*R*. The nonce is generated by prepending the string "arbitrGX" (encoded in ASCII) to the SipHash output.

The values 2, 4 and 8 are recommended for the c, d and R parameters, respectively; following *Aumasson*, R has been reduced from the common 20 rounds of ChaCha to only 8;[7] XChaCha is used with reduced rounds here as well, both for the derivation of the new key and the new encryption.

The rest of this paper will assume this particular parameter choice to provide concrete algorithms. If in doubt, the GX variant should be chosen for any given application.

The G and E variants are unspecified for plaintexts and ciphertexts longer than the size of a ChaCha block (64 bytes). The GX and EX variants are intended for those scenarios.

It should be noted that ARX-KW is deterministic encryption. As such, it will leak by design whether two plaintexts were identical. This is not considered to be an issue because key wrapping involves high-entropy input, so plaintexts are not expected to be identical unless they refer to the same key. This may, however, still be an issue to be aware of in certain contexts, namely when a

[1] The E stands for **e**xtended key.
[2] The G stands for **g**enerated key.
[3] The X stands for the X in XChaCha.

key that is known to have been compromised is reused in other contexts and an attacker may be able to observe the wrapped key in both contexts.

Deterministic encryption was an intentional choice to avoid storage overhead for keys at rest. All variants of ARX-KW take no nonce arguments and produce a static, 128-bit overhead for the ciphertext tag, i. e. an overhead of only 50 % of a ChaCha key.

4.1 The ARX-KW-8-2-4-E variant

The ARX-KW-8-2-4-E encryption procedure takes a 386-bit key K and a plaintext P to produce a ciphertext C and an authentication tag T. The first 128 bits of K are used as the subkey K_1 for SipHash-2-4, the remaining 256 bits are used as the subkey K_2 key for ChaCha8. This variant trades off extending K by 128 bits to save one invocation of ChaCha8 during the encryption and decryption processes.

First, SipHash-2-4 is run over P to obtain T. T is then used for the nonce and counter values of ChaCha8; it does not matter whether the original variant of ChaCha or the IETF variant of ChaCha is used – it just moves where application code interfacing with a library must split T. ChaCha mandates that both nonce and counter are all parsed as 32-bit, little-endian units. ChaCha8 with the LQB set to T is then used to encrypt P into C. Finally, C and T are returned.

The decryption procedure takes C and T. T is used to populate the LQB as for encryption. It then runs ChaCha8 over C to obtain a plaintext candidate P'. It then runs SipHash-2-4 over P' to obtain a tag T'. If T is equal to T',[4] return P' as plaintext, else return $\perp$.

Algorithm 1: ARX-KW-8-2-4-E encryption

Data: key K of 384 bit length
plaintext P no longer than 512 bits
Result: ciphertext C
128-bit authentication tag T

$K_1 \leftarrow K^{0\ldots128}$
$K_2 \leftarrow K^{128\ldots384}$
$T \leftarrow$ SipHash-2-4(key=K_1, msg=P)
$C \leftarrow$ ChaCha8(key=K_2, counter=$T^{0\ldots64}$, nonce=$T^{64\ldots128}$, msg=P)
return C, T

[4] This comparison is to be done using a constant-time comparison function.

Algorithm 2: ARX-KW-8-2-4-E decryption

Data: key K of 384 bit length
ciphertext C no longer than 512 bits
authentication tag T
Result: ciphertext C
authentication tag T

$K_1 \leftarrow K^{0\ldots128}$
$K_2 \leftarrow K^{128\ldots384}$
$P' \leftarrow$ ChaCha8(key=K_2, counter=$T^{0\ldots64}$, nonce=$T^{64\ldots128}$, msg=C)
$T' \leftarrow$ SipHash-2-4(key=K_1, msg=P')
if $T = T'$ **then**
  return P'
else
  return $\perp$

4.2 The ARX-KW-8-2-4-G variant

The ARX-KW-8-2-4-G encryption procedure takes a 256-bit key K and a plaintext P to produce a ciphertext C and an authentication tag T. The only difference between this and ARX-KW-8-2-4-E is how the subkeys K_1 and K_2 are generated, hence this description is abbreviated.

First, the 256-bit key K is used to encrypt a 386-bit all-zero message using ChaCha8 with an all-zero LQB to obtain a 128-bit key K_1 for SipHash-2-4 and a 256-bit key K_2 for ChaCha8. Then, SipHash-2-4 is run over P to obtain T. T is then used for the nonce and counter values of ChaCha8. ChaCha8 with the LQB set to T is then used to encrypt P into C. Finally, C and T are returned.

The decryption procedure takes C and T. T is used to populate the LQB as for encryption. It then runs ChaCha8 over C to obtain a plaintext candidate P'. It then runs SipHash-2-4 over P' to obtain a tag T'. If T is equal to T', return P' as plaintext, else return $\perp$.

Algorithm 3: ARX-KW-8-2-4-G encryption

Data: key K of 256 bit length
plaintext P no longer than 512 bits
Result: ciphertext C
128-bit authentication tag T

$G \leftarrow$ ChaCha8(key=K, counter=0, nonce=0, msg=0^{384})
$K_1 \leftarrow G^{0\ldots128}$
$K_2 \leftarrow G^{128\ldots384}$
$T \leftarrow$ SipHash-2-4(key=K_1, msg=P)
$C \leftarrow$ ChaCha8(key=K_2, counter=$T^{0\ldots64}$, nonce=$T^{64\ldots128}$, msg=P)
return C, T

Algorithm 4: ARX-KW-8-2-4-G decryption

Data: key K of 256 bits length
plaintext P no longer than 512 bits
authentication tag T
Result: ciphertext C
authentication tag T

$G \leftarrow$ ChaCha8(key=K, counter=0, nonce=0, msg=0^{384})
$K_1 \leftarrow G^{0\ldots128}$
$K_2 \leftarrow G^{128\ldots384}$
$P' \leftarrow$ ChaCha8(key=K_2, counter=$T^{0\ldots64}$, nonce=$T^{64\ldots128}$, msg=C)
$T' \leftarrow$ SipHash-2-4(key=K_1, msg=P')
if $T = T'$ **then**
  return P'
else
  return $\perp$

4.3 The ARX-KW-8-2-4-EX variant

The ARX-KW-8-2-4-EX encryption procedure takes a 386-bit key K and a plaintext P to produce a ciphertext C and an authentication tag T. The first 128 bits of K are used as the subkey K_1 for SipHash-2-4, the remaining 256 bits are used as the subkey K_2 key for XChaCha8.

First, SipHash-2-4 is run over P to obtain T. The string `61 72 62 69 74 72 45 58` is concatenated with T, which is then used for the nonce value of XChaCha8. XChaCha8 is then used to encrypt P into C. Finally, C and T are returned.

The decryption procedure takes C and T. The string `61 72 62 69 74 72 45 58` is concatenated with $'T$, which is then used for the nonce value of XChaCha8. It then runs XChaCha8 over C to obtain a plaintext candidate P'. It then runs SipHash-2-4 over P' to obtain a tag T'. If T is equal to T', return P' as plaintext, else return $\perp$.

It was considered to instead tweak 128-bit SipHash to return a 192-bit value to use directly as a nonce. However, doing so would have hampered implementation simplicity because this would have meant existing SipHash implementations could not be leveraged without changes.

Algorithm 5: ARX-KW-8-2-4-EX encryption

Data: key K of 384 bit length
plaintext P
Result: ciphertext C
128-bit authentication tag T

$K_1 \leftarrow K^{0...128}$
$K_2 \leftarrow K^{128...384}$
$T \leftarrow$ SipHash-2-4(key=K_1, msg=P)
$N \leftarrow$ `61 72 62 69 74 72 45 58` $||\ T$
$C \leftarrow$ XChaCha8(key=K_2, counter=0, nonce=N, msg=P)
return C, T

Algorithm 6: ARX-KW-8-2-4-EX decryption

Data: key K of 384 bit length
ciphertext C
authentication tag T
Result: ciphertext C
authentication tag T

$K_1 \leftarrow K^{0...128}$
$K_2 \leftarrow K^{128...384}$
$N \leftarrow$ `61 72 62 69 74 72 45 58` $||\ T$
$P' \leftarrow$ XChaCha8(key=K_2, counter=0, nonce=N, msg=C)
$T' \leftarrow$ SipHash-2-4(key=K_1, msg=P')
if $T = T'$ **then**
  return P'
else
  return $\perp$

4.4 The ARX-KW-8-2-4-GX variant

The ARX-KW-8-2-4-GX encryption procedure takes a 256-bit key K and a plaintext P to produce a ciphertext C and an authentication tag T. The only difference between this and ARX-KW-8-2-4-E is how the subkeys K_1 and K_2 are generated.

First, the 256-bit key K is used to encrypt a 386-bit all-zero message using ChaCha8 with an all-zero LQB to obtain a 128-bit key K_1 for SipHash-2-4 and a 256-bit key K_2 for XChaCha8. Then, SipHash-2-4 is run over P to obtain T. The string `61 72 62 69 74 72 47 58` is concatenated with T, which is then used for the nonce value of XChaCha8. XChaCha8 is then used to encrypt P into C. Finally, C and T are returned.

The decryption procedure takes C and T. The string `61 72 62 69 74 72 47 58` is concatenated with $'T$, which is then used for the nonce value of XChaCha8. It then runs XChaCha8 over C to obtain a plaintext candidate P'. It then runs SipHash-2-4 over P' to obtain a tag T'. If T is equal to T' return P' as plaintext, else return $\perp$.

Algorithm 7: ARX-KW-8-2-4-GX encryption

Data: key K of 256 bit length
plaintext P
Result: ciphertext C
128-bit authentication tag T

$G \leftarrow$ ChaCha8(key=K, counter=0, nonce=0, msg=0^{384})
$K_1 \leftarrow G^{0\ldots128}$
$K_2 \leftarrow G^{128\ldots384}$
$N \leftarrow$ `61 72 62 69 74 72 47 58` $\|\| T$
$C \leftarrow$ XChaCha8(key=K_2, counter=0, nonce=0, msg=P)
return C, T

Algorithm 8: ARX-KW-8-2-4-GX decryption

Data: key K of 256 bits length
plaintext P no longer than 512 bits
authentication tag T
Result: ciphertext C
authentication tag T

$G \leftarrow$ ChaCha8(key=K, counter=0, nonce=0, msg=0^{384})
$K_1 \leftarrow G^{0\ldots128}$
$K_2 \leftarrow G^{128\ldots384}$
$N \leftarrow$ `61 72 62 69 74 72 47 58` $\|\| T$
$P' \leftarrow$ XChaCha8(key=K_2, counter=0, nonce=N, msg=C)
$T' \leftarrow$ SipHash-2-4(key=K_1, msg=P')
if $T = T'$ **then**
  return P'
else
  return $\perp$

5 Speed

The following speeds were measured using a Sandy Bridge processor that was released in early 2011. The executable was compiled from the unoptimized reference implementation written in C using Clang 8.0.1 with these flags: `-O3 -flto -march=native`.

Encryption

Algorithm	Message length	Cycles	Cycles per Byte
ARX-KW-8-2-4-E	32	597	19
ARX-KW-8-2-4-G	32	811	25
ARX-KW-8-2-4-EX	32	873	27
ARX-KW-8-2-4-GX	32	968	30

Decryption

Algorithm	Message length	Cycles	Cycles per Byte
ARX-KW-8-2-4-E	32	505	16
ARX-KW-8-2-4-G	32	817	26
ARX-KW-8-2-4-EX	32	809	25
ARX-KW-8-2-4-GX	32	1023	32

6 Intellectual Property

As far as I am aware to the best of my knowledge, ARX-KW is not affected by any patents; I do not intend to hold or apply for any patent which may affect the ARX-KW construction, its reference implementation or any optimized implementation thereof.

The reference implementation[5] has been dedicated to the public domain through the Creative Commons CC0 waiver. The ChaCha8 part of the reference implementation was built on Monocypher, which is also dedicated to the public domain through CC0; the SipHash part of the reference implementation was taken without modification from the SipHash reference implementation, which is also dedicated to the public domain through CC0. No optimized implementation is provided.

[5]The reference implementation is available from `https://gitlab.com/SATOshinichi/arxkw`.

7 Test vectors

This section provides the final values for a set of inputs. All values are specified in hexadecimal notation and specified as a sequence of bytes without explicit endianness; the underlying primitives consume and produce sequences of bytes.

7.1 ARX-KW-8-2-4-E

```
K: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b
   1c1d1e1f202122232425262728292a2b2c2d2e2f
P: deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
   deadbeef
T: c4f21d3b4dbcc566c3a73bbc59790f2f
C: e6457d24abaf7c2ebdb91416a18366d31a66db61a4e45c9f42a119c3
   53bb1eb1
```

7.2 ARX-KW-8-2-4-G

```
K: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b
   1c1d1e1f
P: deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
   deadbeef
T: 016325cf6a3c4b2e3b039675e1ccbc65
C: f63830f5148a039b6aacc4b9b6bc281d7704d906e4b5d91e045a62cd
   fc25eb10
```

7.3 ARX-KW-8-2-4-EX

```
K: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b
   1c1d1e1f202122232425262728292a2b2c2d2e2f
P: deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
   deadbeef
N: 6172626974724558c4f21d3b4dbcc566c3a73bbc59790f2f
T: c4f21d3b4dbcc566c3a73bbc59790f2f
C: 02a55ab1d7f549db160e8ecb33e1c6d65a05d0ebaba54dc071228578
   7c8a62db
N: 6172626974724558c4f21d3b4dbcc566c3a73bbc59790f2f
```

7.4 ARX-KW-8-2-4-GX

```
K: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b
   1c1d1e1f
P: deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
   deadbeef
N: 617262697472475801632 5cf6a3c4b2e3b039675e1ccbc65
T: 016325cf6a3c4b2e3b039675e1ccbc65
C: 2f83f391c97f3606ccd5709c6ee15d66cd7e65a2aeb7dc3066636e8f
   6b0d39c3
N: 617262697472475801632 5cf6a3c4b2e3b039675e1ccbc65
```

References

[1] Scott Arciszewski. XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Internet-Draft draft-irtf-cfrg-xchacha-01, Internet Engineering Task Force (IETF) Secretariat, July 2019. URL: `https://www.ietf.org/internet-drafts/draft-irtf-cfrg-xchacha-01.txt`.

[2] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology – INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508, 2012.

[3] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, pages 273–278, 2008.

[4] Daniel J. Bernstein. Extending the Salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop 2011*, 2011. URL: `http://skew2011.mat.dtu.dk/proceedings/Extending%20the%20Salsa20%20nonce.pdf`.

[5] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping. Special Publication 800-38F, National Institute of Standards and Technology (NIST), December 2012.

[6] Shay Gueron and Yehuda Lindell. GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 109–119, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Jean-Philippe Aumasson. Too Much Crypto. Cryptology ePrint Archive, Report 2019/1492, 2019. Version 20200103111400; URL: `https://eprint.iacr.org/2019/1492`.

[8] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, Internet Engineering Task Force (IETF), June 2018.

[9] Phillip Rogaway and Thomas Shrimpton. Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Key-Wrap Problem. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390, 2006.