# Nonce-encrypting AEAD Modes with Farfalle

Seth Hoffert

**Abstract.** Nonces are a fact of life for achieving semantic security. Generating a uniformly random nonce can be costly and may not always be feasible. Using anything other than uniformly random bits can result in information leakage; e.g., a timestamp can deanonymize a communication and a counter can leak the quantity of transmitted messages. Ideally, we would like to be able to efficiently encrypt the nonce to 1) avoid needing uniformly random bits and 2) avoid information leakage. This paper presents two new authenticated encryption modes built on top of Farfalle [2] that perfectly achieve these goals.

**Keywords:** farfalle, deck functions, authenticated encryption, wide block cipher, modes of use, encrypted nonce, onion AE

## 1 Introduction

A typical implementation of an AEAD mode tends to use some combination of a timestamp, a counter and uniform randomness when deriving a nonce. Using uniform randomness for nonce generation can be expensive on platforms that have a limited entropy pool, and using anything other than uniform randomness will inevitably leak information. For example, an observed timestamp can be enough for an adversary to deanonymize a communication if the sender's clock is known to be inexact. Using a counter can also leak information, because it allows an adversary to observe only two messages at different points in time and yet still be able to estimate the number of messages that were sent in between observations.

It would be nice if the nonce could somehow be contained within the plaintext. At first, this seems like a chicken-and-egg problem: how does one encrypt a nonce without using another nonce? The answer lies in the Feistel construction. Remarkably, it is possible to construct an efficient inverse-free mode that is very similar to Deck-PLAIN [3] and yet encrypts the nonce and redundancy with negligible additional overhead. We showcase a RUP-resistant variant as well, geared towards onion AE protocols.

### 1.1 Contributions

Our main contribution is constructing two efficient authenticated encryption modes on top of Farfalle [2] that feature nonce and redundancy encryption. The first mode can be viewed as the encrypted nonce analog of Deck-PLAIN [3], and the second mode can be viewed as a constrained version of Deck-JAMBOREE [3] that requires a nonce but still supports RUP, similar to GCM-RUP [1]. Because

both of our modes are built entirely on top of Farfalle [2], no block ciphers are involved and the inverse direction of the underlying permutation remains unused. Besides the obvious benefit of elegance from needing only one type of primitive, this provides a hardware space advantage in ASIC and FPGA designs.

### 1.2 Conventions

The length in bits of the string $X$ is denoted $|X|$. The concatenation of two strings $X, Y$ is denoted as $X\|Y$ and their bitwise addition as $X \oplus Y$. Bit string values are noted with a typewriter font, such as `01101`. The repetition of a bit is noted in exponent, e.g., $\mathtt{0}^3 = \mathtt{000}$. Finally, $\varnothing$ is the empty set and $\perp$ denotes an error code.

## 2 Constrained wide block cipher

We first present a constrained wide block cipher in algorithm 1.1, parameterized in terms of the Farfalle instance $\mathcal{F}$. A concrete nonce- and redundancy-encrypting AEAD mode is then presented in algorithm 1.2.

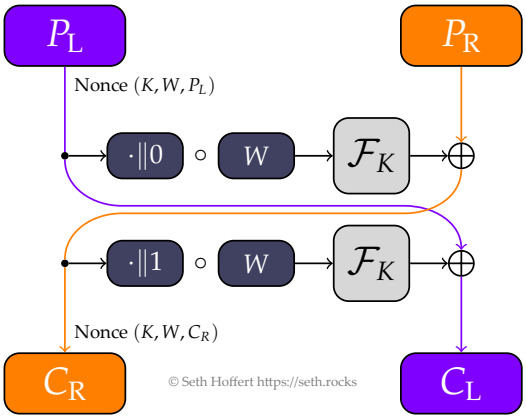


Fig. 1: Constrained wide block cipher

### 2.1 Details

We call the wide block cipher in algorithm 1.1 *constrained* because it is effectively a four-round Feistel network but with the outer two rounds removed. By removing the outer rounds, a nonce requirement is imposed in both $(K, W, P_L)$ and $(K, W, C_R)$. At first, these nonce constraints may seem unreasonable, as they would seem to require keeping track of all inputs that have been seen. However,

**Algorithm 1.1:** Constrained wide block cipher

**Definition :** Farfalle instance $\mathcal{F}$

**function** encrypt(key $K$, tweak $W$, plaintext $(P_L, P_R)$): ciphertext or $\perp$
  **if** $(K, W, P_L)$ is not a nonce WRT $P_R$ **then return** $\perp$
  $C_R \leftarrow P_R \oplus \mathcal{F}_K(P_L \| \mathtt{0} \circ W)$
  $C_L \leftarrow P_L \oplus \mathcal{F}_K(C_R \| \mathtt{1} \circ W)$
  **if** $(K, W, C_R)$ is not a nonce WRT $C_L$ **then return** $\perp$
  **return** $(C_L, C_R)$

**function** decrypt(key $K$, tweak $W$, ciphertext $(C_L, C_R)$): plaintext or $\perp$
  **if** $(K, W, C_R)$ is not a nonce WRT $C_L$ **then return** $\perp$
  $P_L \leftarrow C_L \oplus \mathcal{F}_K(C_R \| \mathtt{1} \circ W)$
  $P_R \leftarrow C_R \oplus \mathcal{F}_K(P_L \| \mathtt{0} \circ W)$
  **if** $(K, W, P_L)$ is not a nonce WRT $P_R$ **then return** $\perp$
  **return** $(P_L, P_R)$

**Algorithm 1.2:** Nonce- and redundancy-encrypting AEAD mode

**Definition :** Farfalle instance $\mathcal{F}$

ctr $\leftarrow \mathtt{0}^{64}$

**function** encrypt(key $K$, metadata $A$, plaintext $P$): ciphertext
  time $\leftarrow$ current 64-bit timestamp
  $P_L \leftarrow$ time$\|$ctr
  $P_R \leftarrow \mathtt{0}^{128} \| \text{pad}(P)$ such that $|P_R| \geq 256$
  $C_R \leftarrow P_R \oplus \mathcal{F}_K(P_L \| \mathtt{0} \circ A)$
  $C_L \leftarrow P_L \oplus \mathcal{F}_K(C_R \| \mathtt{1} \circ A)$
  ctr $\leftarrow$ ctr $+ 1$
  **return** $C_L \| C_R$

**function** decrypt(key $K$, metadata $A$, ciphertext $C$): plaintext or $\perp$
  **if** $|C| < 384$ **then return** $\perp$
  $C_L \| C_R \leftarrow C$ such that $|C_L| = 128$
  $P_L \leftarrow C_L \oplus \mathcal{F}_K(C_R \| \mathtt{1} \circ A)$
  $P_R \leftarrow C_R \oplus \mathcal{F}_K(P_L \| \mathtt{0} \circ A)$
  $T \| P \leftarrow P_R$ such that $|T| = 128$
  **if** $T \neq \mathtt{0}^{128}$ **then return** $\perp$
  time$\|$ctr$' \leftarrow P_L$ such that $|\text{time}| = 64$
  **return** $(\text{time}, \text{ctr}', \text{unpad}(P))$

note that these conditions can be satisfied in a practical way by providing a nonce in $(K, W, P_L)$ and validating redundancy in $P_R$. WLOG, we build one such concrete mode in algorithm 1.2.

This mode is effectively a phase-shifted (and stateless) version of Deck-PLAIN [3]. In the encryption oracle, note that $C \sim \mathrm{PRF}_K(A, P_L)$. By requiring $(K, A, P_L)$ to be a nonce, we ensure that $C$ is indistinguishable from random. Likewise, in the decryption oracle, note that $P_R \sim \mathrm{PRF}_K(A, C)$. By requiring verifiable redundancy in $P_R$, we ensure that any tampering of $(A, C)$ is detected. Additionally, note that $P_L$ is allowed to contain arbitrary plaintext, as long as $(K, A, P_L)$ is a nonce. Once $P_R$ reaches the minimum length requirement, an implementation may choose to fill $P_L$ to an entire block to achieve optimal performance.

Remarkably, the early rejection feature of Deck-PLAIN is retained. By keeping $P_L$ short and placing the entirety of the redundancy within the first block of $P_R$, the decryption oracle can expand just enough bits to decrypt the redundancy and validate it. If valid, then it can expand the remaining bits to decrypt the rest of $C_R$. Asymptotically, authentication can be performed after a single read pass, just like in Deck-PLAIN. Because of this feature, the risk of leaking unverified plaintext is eliminated. Another advantage of keeping $P_L$ short is to allow the encryption oracle to be online: the asymptotic majority of the ciphertext bits are produced immediately after compressing $P_L$.

Because $P_R \sim \mathrm{PRF}_K(A, C)$, this implies that if the corresponding redundancy bits of $C$ are tampered with, then every bit of decrypted redundancy is flipped with 50% probability. This allows for the use of a non-constant-time equality function. Due to the fact that the redundancy is non-malleable, nothing is gleaned from the timing of the comparison during redundancy validation. Note that if this feature is not needed (i.e., a constant-time equality function is used), then the compression of the redundancy portion of $C_R$ can be skipped on lines 7 and 13 of algorithm 1.2. This optimization brings the mode even closer to parity with Deck-PLAIN. For brevity, the modified algorithm is deferred to a future paper.

Metadata string $A$ is optional and can be safely omitted from compression thanks to the frame bits on the plaintext. Note, however, that metadata-only input is not supported. The algorithm can be modified to support the metadata-only case, but this circumvents the nonce- and redundancy-encrypting goals of the mode. If a message authentication code is desired, then we recommend supplying the nonce and redundancy as part of the plaintext for consistency.

Algorithms 1.1 and 1.2 are stateless by design. Refer to section A for the treatment of sessions.

### 2.2 Application: virtual private network protocol

Consider a virtual private network (VPN) protocol. In such a protocol, many small packets are exchanged at a high frequency. It would be very costly to generate a uniformly random nonce per packet for the following reasons:

– **Space**: to achieve 128 bits of security, we would need a 256-bit nonce due to birthday bound collisions
– **Time**: because of the high-frequency nature of the application, the OS entropy pool could become exhausted and block the application until more can be gathered

Deck-PLAIN allows for a single nonce to be specified per session. However, because packets can be lost and arrive out-of-order, we cannot use a session-based mode. Instead, algorithm 1.2 satisfies the requirements perfectly.

We can use a timestamp and counter as the nonce. Because the nonce is part of the plaintext, no semantic information is leaked. Additionally, such a nonce requires only half the amount of storage contrasted against a uniformly random nonce. The timestamp and counter could be trimmed to further reduce the space requirements.

### 2.3 Features

– **Encrypted nonce and redundancy**: both the nonce and redundancy are encrypted, allowing non-random nonces to be used without risk of information leak
– **Performance**: asymptotically performs only a single read- and write-pass over the plaintext
– **Online encryption**: the encryption oracle produces the majority of the ciphertext bits immediately after compressing the nonce
– **Early rejection**: by placing the verifiable redundancy at the beginning of $P_R$, early rejection can be realized
– **Timing-insensitive authentication**: because the redundancy is non-malleable, a non-constant-time equality function can be used to validate redundancy
– **Optional/static metadata**: the metadata string compression can be skipped if empty; additionally, static metadata can be factored out and reused across invocations

### 2.4 Security proof

We defer the security proof to a future revision.

## 3 Constrained wide block cipher with RUP resistance

We now present the RUP-resistant constrained wide block cipher in algorithm 1.3, along with onion AE algorithms 1.4 and 1.5. The wide block cipher is parameterized in terms of the Farfalle instance $\mathcal{F}$ and target security level $s$. We recommend a target security level of $s = 128$.

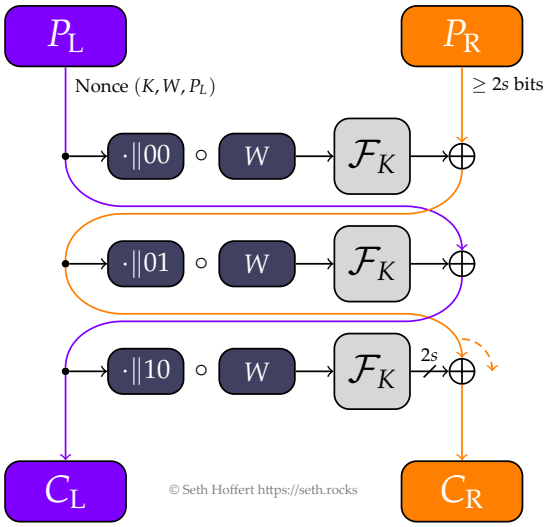


Fig. 2: Constrained wide block cipher with RUP resistance

**Algorithm 1.3:** Constrained wide block cipher with RUP resistance

**Definition :** Farfalle instance $\mathcal{F}$
**Definition :** Security level $s = 128$ bits

**function** encrypt(key $K$, tweak $W$, plaintext $(P_L, P_R)$): ciphertext or $\perp$
  **if** $|P_R| < 2s$ **then return** $\perp$
  **if** $(K, W, P_L)$ is not a nonce WRT $P_R$ **then return** $\perp$
  $C_R \leftarrow P_R \oplus \mathcal{F}_K(P_L \| \texttt{00} \circ W)$
  $C_L \leftarrow P_L \oplus \mathcal{F}_K(C_R \| \texttt{01} \circ W)$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{F}_K(C_L \| \texttt{10} \circ W)$
  **return** $(C_L, C_R)$

**function** decrypt(key $K$, tweak $W$, ciphertext $(C_L, C_R)$): plaintext or $\perp$
  **if** $|C_R| < 2s$ **then return** $\perp$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{F}_K(C_L \| \texttt{10} \circ W)$
  $P_L \leftarrow C_L \oplus \mathcal{F}_K(C_R \| \texttt{01} \circ W)$
  $P_R \leftarrow C_R \oplus \mathcal{F}_K(P_L \| \texttt{00} \circ W)$
  **if** $(K, W, P_L)$ is not a nonce WRT $P_R$ **then return** $\perp$
  **return** $(P_L, P_R)$

**Algorithm 1.4:** Onion AE mode (client)

**Definition :** Farfalle instance $\mathcal{F}$

$\text{ctr}_*^{\downarrow} \leftarrow \mathtt{0}^{64}$
$\text{ctr}_*^{\uparrow} \leftarrow \mathtt{0}^{64}$

**function** init(client keys $K_*$)
  $\mathcal{G} \leftarrow \mathcal{F}_K$

**function** wrap(target $i$, plaintext $P$): ciphertext
  $P_L \| P_R \leftarrow \mathtt{0}^{128} \| \text{pad}(P)$ such that $|P_L| = 256, |P_R| \geq 256$
  **for** $j = i$ through $0$ **do**
    $C_R \leftarrow P_R \oplus \mathcal{G}_j(P_L \| \mathtt{000} \circ \text{ctr}_j^{\downarrow})$
    $C_L \leftarrow P_L \oplus \mathcal{G}_j(C_R \| \mathtt{001} \circ \text{ctr}_j^{\downarrow})$
    $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}_j(C_L \| \mathtt{010} \circ \text{ctr}_j^{\downarrow})$
    $\text{ctr}_j^{\downarrow} \leftarrow \text{ctr}_j^{\downarrow} + 1$
    $(P_L, P_R) \leftarrow (C_L, C_R)$
  **return** $C_L \| C_R$

**function** unwrap(ciphertext $C$): (source, plaintext) or $\perp$
  **if** $|C| < 512$ **then return** $\perp$
  $C_L \| C_R \leftarrow C$ such that $|C_L| = 256$
  **for** $j = 0$ through $|\mathcal{G}| - 1$ **do**
    $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}_j(C_L \| \mathtt{110} \circ \text{ctr}_j^{\uparrow})$
    $P_L \leftarrow C_L \oplus \mathcal{G}_j(C_R \| \mathtt{101} \circ \text{ctr}_j^{\uparrow})$
    $P_R \leftarrow C_R \oplus \mathcal{G}_j(P_L \| \mathtt{100} \circ \text{ctr}_j^{\uparrow})$
    $\text{ctr}_j^{\uparrow} \leftarrow \text{ctr}_j^{\uparrow} + 1$
    $T \| P' \leftarrow P_L \| P_R$ such that $|T| = 128$
    **if** $T = \mathtt{0}^{128}$ **then return** $(j, \text{unpad}(P'))$
    $(C_L, C_R) \leftarrow (P_L, P_R)$
  **return** $\perp$

**Algorithm 1.5:** Onion AE mode (node)

**Definition :** Farfalle instance $\mathcal{F}$

$\text{ctr}^{\downarrow} \leftarrow \mathtt{0}^{64}$
$\text{ctr}^{\uparrow} \leftarrow \mathtt{0}^{64}$

**function** init(ephemeral key $K$)
  $\mathcal{G} \leftarrow \mathcal{F}_K$

**function** wrap(from-me, plaintext $P$): ciphertext or $\perp$
  **if** from-me **then**
    $P_L \| P_R \leftarrow \mathtt{0}^{128} \| \text{pad}(P)$ such that $|P_L| = 256, |P_R| \geq 256$
  **else**
    **if** $|P| < 512$ **then return** $\perp$
    $P_L \| P_R \leftarrow P$ such that $|P_L| = 256$
  $C_R \leftarrow P_R \oplus \mathcal{G}(P_L \| \mathtt{100} \circ \text{ctr}^{\uparrow})$
  $C_L \leftarrow P_L \oplus \mathcal{G}(C_R \| \mathtt{101} \circ \text{ctr}^{\uparrow})$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}(C_L \| \mathtt{110} \circ \text{ctr}^{\uparrow})$
  $\text{ctr}^{\uparrow} \leftarrow \text{ctr}^{\uparrow} + 1$
  **return** $C_L \| C_R$

**function** unwrap(ciphertext $C$): (to-me, plaintext) or $\perp$
  **if** $|C| < 512$ **then return** $\perp$
  $C_L \| C_R \leftarrow C$ such that $|C_L| = 256$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}(C_L \| \mathtt{010} \circ \text{ctr}^{\downarrow})$
  $P_L \leftarrow C_L \oplus \mathcal{G}(C_R \| \mathtt{001} \circ \text{ctr}^{\downarrow})$
  $P_R \leftarrow C_R \oplus \mathcal{G}(P_L \| \mathtt{000} \circ \text{ctr}^{\downarrow})$
  $\text{ctr}^{\downarrow} \leftarrow \text{ctr}^{\downarrow} + 1$
  $T \| P' \leftarrow P_L \| P_R$ such that $|T| = 128$
  **if** $T = \mathtt{0}^{128}$ **then**
    **return** $(\text{true}, \text{unpad}(P'))$
  **else**
    **return** $(\text{false}, P_L \| P_R)$

### 3.1 Details

Similar to algorithm 1.1, we call the wide block cipher in algorithm 1.3 *constrained*. In this wide block cipher mode, we are stripping away only the first round from a four-round Feistel network in order to retain RUP resistance. As a result, the $(K, W, C_R)$ nonce constraint is relaxed, opening the doors for specialized modes that require RUP (e.g., onion AE).

In algorithm 1.1, note of the decryption oracle that $P_L$ is malleable. To achieve RUP resistance, we need only communicate a digest of $C_L$ into $C_R$ to ensure that $P \sim \mathrm{PRF}_K(A, C)$. In spite of this additional round, note that a single read- and write-pass over the plaintext is still achieved by keeping $P_L$ short (i.e., one block maximum).

Because algorithm 1.3 allows for RUP, we could trivially build a RUP-resistant AEAD mode. However, we believe such a mode is of limited benefit thanks to the early rejection feature. Instead, we feel that the application of onion AE is a much more interesting and useful mode for this constrained wide block cipher. We present such a mode in algorithms 1.4 and 1.5.

Algorithms 1.3, 1.4 and 1.5 are stateless by design. Refer to section A for the treatment of sessions.

### 3.2 Application: onion AE protocol

Consider an onion AE protocol. At a high level, such a protocol recursively applies the cryptographic algorithm to a payload that is passed between nodes. Such a protocol demands two important properties:

- **RUP resistance**: because intermediate nodes decrypt the payload and potentially pass along the results to the next node, the cryptographic algorithm needs to be resistant to RUP
- **Length preservation**: because the cryptographic algorithm is applied recursively, the payload length must be preserved to avoid leaking semantic information about number of layers

Deck-PLAIN is ruled out immediately since it incurs a per-encryption expansion due to explicit redundancy. Algorithm 1.2 does not quite fit the bill either because of its lack of RUP resistance. Algorithms 1.4 and 1.5 are exactly what we need.

In order to satisfy the nonce requirement, the client encryption oracle accepts only plaintexts that contain valid redundancy and its decryption oracle releases only plaintexts that contain valid redundancy. These validations prevent keystream harvesting while also allowing the non-client nodes to freely encrypt, decrypt and release payloads that are ultimately invalid.

Note specifically that no constraints are placed on the exit node's oracles: a malicious exit node can freely encrypt, decrypt and release invalid plaintext without causing any issues. Because of the RUP resistance, the decrypted plaintext is non-malleable and tagging attacks are not possible.

The benefit of having both the nonce and redundancy reside in non-malleable $P_L$ is that they are treated on equal footing. For example, if a timestamp comprises $P_L$, then an implementation can validate the timestamp against a fixed time window. With a 64-bit timestamp and an allowed time window of 256 seconds, $64 - \log_2 256 = 56$ bits of authenticity is already achieved. This saves space in the plaintext by reducing the need for a separate nonce and redundancy. Additionally, an application that uses ephemeral keys with a metadata-based counter (e.g., onion AE) need not place any additional nonce value in $P_L$. Instead, $P_L$ can be a block of arbitrary plaintext.

### 3.3 Features

- **Encrypted nonce and redundancy**: both the nonce and redundancy are encrypted, allowing non-random nonces to be used without risk of information leak
- **Performance**: asymptotically performs only a single read- and write-pass over the plaintext
- **Online encryption**: the encryption oracle produces the majority of the ciphertext bits immediately after compressing the nonce
- **Early rejection**: by placing the verifiable redundancy at the beginning of $P_L$, early rejection can be realized
- **Timing-insensitive authentication**: because the redundancy is non-malleable, a non-constant-time equality function can be used to validate redundancy
- **Optional/static metadata**: the metadata string compression can be skipped if empty; additionally, static metadata can be factored out and reused across invocations
- **RUP resistance**: this mode can be used in onion AE protocols thanks to its RUP resistance
- **Combined nonce and redundancy**: nonces can be used as verifiable redundancy, resulting in a space savings in the plaintext

### 3.4 Security proof

We defer the security proof to a future revision.

## 4 Conclusions

The presented modes are a testament to Farfalle's flexibility and power. Algorithm 1.1 can be viewed as an enhanced Deck-PLAIN [3], and algorithm 1.3 is ideal for situations where RUP is needed (e.g., onion AE). Both modes achieve the encrypted nonce goal and are efficient, asymptotically requiring only a single read- and write-pass over the plaintext.

## References


1. Ashur, T., Dunkelman, O., Luykx, A.: Boosting authenticated encryption robustness with minimal modifications. Cryptology ePrint Archive, Paper 2017/239 (2017), https://eprint.iacr.org/2017/239, https://eprint.iacr.org/2017/239
2. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: Farfalle: parallel permutation-based cryptography. IACR Trans. Symmetric Cryptol. 2017(4), 1–38 (2017)
3. Băcuieți, N., Daemen, J., Hoffert, S., Assche, G.V., Keer, R.V.: Jammin' on the deck. Cryptology ePrint Archive, Paper 2022/531 (2022), https://eprint.iacr.org/2022/531, https://eprint.iacr.org/2022/531


## A Session support

We now present session-based variants of algorithms 1.2, 1.4 and 1.5.

### A.1 Application: stream encryption

Consider wanting to build a stream encryption application for a platform that has a limited entropy pool, or a lack thereof. We could use algorithm 1.2 and encrypt the messages statelessly, using some combination of a timestamp and counter as the nonce. However, this is not so elegant due to the overall space consumption of the per-message nonces. It would be preferrable to use a session-based mode like Deck-PLAIN so that we need a nonce only at session initialization.

Because the platform has a limited entropy pool, we would like to avoid uniformly random nonces. Algorithm 1.6 fits the bill perfectly because it allows us to specify a single non-random nonce for the entirety of the session. This results in a space savings, ensures semantic security and obviates the need for per-session system entropy.

Algorithm 1.6 inherits all of the features of algorithm 1.2 plus session support. This mode can be seen as a hybrid of the nonce-encrypting mode and Deck-PLAIN, in that the first message is handled specially such that the nonce is encrypted. Subsequent messages are handled in a manner similar to Deck-PLAIN. Note that the redundancy-encrypting feature is not needed in the subsequent messages because such redundancy can be stripped out of the plaintext and reinserted on the unwrapping end, thereby offsetting the spatial cost of the tag.

### A.2 Application: onion AE protocol

In section 3.2 we describe a sessionless solution to the onion AE problem. The main disadvantage of the sessionless algorithm is that if a message is corrupted, then subsequent messages are unaffected. This problem can be solved by making use of a chaining value in the metadata input, but this requires additional operations. Instead, algorithms 1.7 and 1.8 solve this problem elegantly and avoids explicit chaining values and per-message nonces.

**Algorithm 1.6:** Nonce-encrypting session AEAD mode

**Definition :** Farfalle instance $\mathcal{F}$

$\text{ctr} \leftarrow \mathtt{0}^{64}$
$o \leftarrow 0$
**function** init-sender(key $K$, metadata $A$, plaintext $P$): ciphertext
    $\mathcal{G} \leftarrow \mathcal{F}_K$
    time $\leftarrow$ current 64-bit timestamp
    $P_L \leftarrow \text{time} \| \text{ctr}$
    $P_R \leftarrow \mathtt{0}^{128} \| \text{pad}(P)$ such that $|P_R| \geq 256$
    **if** $|A| \neq 0$ **then** history $\leftarrow A$
    $C_R \leftarrow P_R \oplus \mathcal{G}(P_L \| \mathtt{0} \circ \text{history})$
    $C_L \leftarrow P_L \oplus \mathcal{G}(C_R \| \mathtt{1} \circ \text{history})$
    history $\leftarrow C_L \circ C_R \| \mathtt{1} \circ \text{history}$
    ctr $\leftarrow$ ctr $+ 1$
    **return** $C_L \| C_R$
**function** init-receiver(key $K$, metadata $A$, ciphertext $C$): plaintext or $\perp$
    **if** $|C| < 384$ **then return** $\perp$
    $\mathcal{G} \leftarrow \mathcal{F}_K$
    $C_L \| C_R \leftarrow C$ such that $|C_L| = 128$
    **if** $|A| \neq 0$ **then** history$' \leftarrow A$
    $P_L \leftarrow C_L \oplus \mathcal{G}(C_R \| \mathtt{1} \circ \text{history}')$
    $P_R \leftarrow C_R \oplus \mathcal{G}(P_L \| \mathtt{0} \circ \text{history}')$
    $T \| P \leftarrow P_R$ such that $|T| = 128$
    **if** $T \neq \mathtt{0}^{128}$ **then return** $\perp$
    history $\leftarrow C_L \circ C_R \| \mathtt{1} \circ \text{history}'$
    time$\|$ctr$' \leftarrow P_L$ such that $|\text{time}| = 64$
    **return** $(\text{time}, \text{ctr}', \text{unpad}(P))$
**function** wrap(plaintext $P$): ciphertext
    **assert** history $\neq \varnothing$
    $C \leftarrow P \oplus \mathcal{G}(\text{history}) \ll o$
    $T \leftarrow \mathtt{0}^{128} \oplus \mathcal{G}(C \circ \text{history})$
    history $\leftarrow C \circ$ history
    $o \leftarrow 128$
    **return** $C \| T$
**function** unwrap(ciphertext $C$): plaintext or $\perp$
    **assert** history $\neq \varnothing$
    **if** $|C| < 128$ **then return** $\perp$
    $C' \| T' \leftarrow C$ such that $|T'| = 128$
    $T \leftarrow T' \oplus \mathcal{G}(C' \circ \text{history})$
    **if** $T \neq \mathtt{0}^{128}$ **then return** $\perp$
    $P \leftarrow C' \oplus \mathcal{G}(\text{history}) \ll o$
    history $\leftarrow C' \circ$ history
    $o \leftarrow 128$
    **return** $P$

**Algorithm 1.7:** Onion session AE mode (client)

**Definition :** Farfalle instance $\mathcal{F}$

**function** init(client keys $K_*$)
  $\mathcal{G}_* \leftarrow \mathcal{F}_{K_*}$

**function** wrap(target $i$, plaintext $P$): ciphertext
  $P_L \| P_R \leftarrow \mathtt{0}^{128} \| \text{pad}(P)$ such that $|P_L| = 256, |P_R| \geq 256$
  **for** $j = i$ through 0 **do**
    $C_R \leftarrow P_R \oplus \mathcal{G}_j(P_L \| \mathtt{000} \circ \text{history}_j^{\downarrow})$
    $C_L \leftarrow P_L \oplus \mathcal{G}_j(C_R \| \mathtt{001} \circ \text{history}_j^{\downarrow})$
    $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}_j(C_L \| \mathtt{010} \circ \text{history}_j^{\downarrow})$
    $\text{history}_j^{\downarrow} \leftarrow C_R \| \mathtt{001} \circ \text{history}_j^{\downarrow}$
    $(P_L, P_R) \leftarrow (C_L, C_R)$
  **return** $C_L \| C_R$

**function** unwrap(ciphertext $C$): (source, plaintext) or $\perp$
  **if** $|C| < 512$ **then return** $\perp$
  $C_L \| C_R \leftarrow C$ such that $|C_L| = 256$
  **for** $j = 0$ through $|\mathcal{G}| - 1$ **do**
    $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}_j(C_L \| \mathtt{110} \circ \text{history}_j^{\uparrow})$
    $P_L \leftarrow C_L \oplus \mathcal{G}_j(C_R \| \mathtt{101} \circ \text{history}_j^{\uparrow})$
    $P_R \leftarrow C_R \oplus \mathcal{G}_j(P_L \| \mathtt{100} \circ \text{history}_j^{\uparrow})$
    $\text{history}_j^{\uparrow} \leftarrow C_R \| \mathtt{101} \circ \text{history}_j^{\uparrow}$
    $T \| P' \leftarrow P_L \| P_R$ such that $|T| = 128$
    **if** $T = \mathtt{0}^{128}$ **then return** $(j, \text{unpad}(P'))$
    $(C_L, C_R) \leftarrow (P_L, P_R)$
  **return** $\perp$

**Algorithm 1.8:** Onion session AE mode (node)

**Definition :** Farfalle instance $\mathcal{F}$

**function** init(ephemeral key $K$)
  $\mathcal{G} \leftarrow \mathcal{F}_K$

**function** wrap(from-me, plaintext $P$): ciphertext or $\perp$
  **if** from-me **then**
    $P_L \| P_R \leftarrow 0^{128} \| \text{pad}(P)$ such that $|P_L| = 256, |P_R| \geq 256$
  **else**
    **if** $|P| < 512$ **then return** $\perp$
    $P_L \| P_R \leftarrow P$ such that $|P_L| = 256$
  $C_R \leftarrow P_R \oplus \mathcal{G}(P_L \| \texttt{100} \circ \text{history}^{\uparrow})$
  $C_L \leftarrow P_L \oplus \mathcal{G}(C_R \| \texttt{101} \circ \text{history}^{\uparrow})$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}(C_L \| \texttt{110} \circ \text{history}^{\uparrow})$
  $\text{history}^{\uparrow} \leftarrow C_R \| \texttt{101} \circ \text{history}^{\uparrow}$
  **return** $C_L \| C_R$

**function** unwrap(ciphertext $C$): (to-me, plaintext) or $\perp$
  **if** $|C| < 512$ **then return** $\perp$
  $C_L \| C_R \leftarrow C$ such that $|C_L| = 256$
  $\text{left}(C_R, 256) \leftarrow \text{left}(C_R, 256) \oplus \mathcal{G}(C_L \| \texttt{010} \circ \text{history}^{\downarrow})$
  $P_L \leftarrow C_L \oplus \mathcal{G}(C_R \| \texttt{001} \circ \text{history}^{\downarrow})$
  $P_R \leftarrow C_R \oplus \mathcal{G}(P_L \| \texttt{000} \circ \text{history}^{\downarrow})$
  $\text{history}^{\downarrow} \leftarrow C_R \| \texttt{001} \circ \text{history}^{\downarrow}$
  $T \| P' \leftarrow P_L \| P_R$ such that $|T| = 128$
  **if** $T = 0^{128}$ **then**
    **return** $(\text{true}, \text{unpad}(P'))$
  **else**
    **return** $(\text{false}, P_L \| P_R)$