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

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, suitable for use in e.g., onion routing 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., $\texttt{0}^3 = \texttt{000}$. Finally, $\varnothing$ is the empty set and $\perp$ denotes an error code.

2 Nonce- and redundancy-encrypting AEAD mode

The nonce- and redundancy-encrypting AEAD mode is now presented in its most general form in algorithm 1.1. The mode is parameterized in terms of the Farfalle instance $\mathcal{F}$ and target security level s. We recommend a target security level of $s = 128$ bits. The *valid* function called by the decryption oracle is expected to validate the redundancy that is present in P_R so that the plaintext can be safely released.

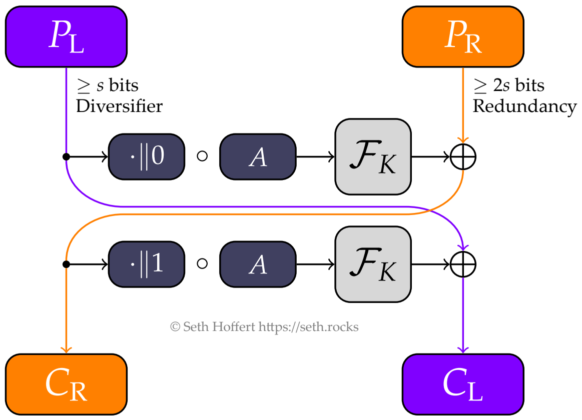

Fig. 1: Nonce- and redundancy-encrypting AEAD mode

2.1 Details

This mode is effectively a phase-shifted (and stateless) version of Deck-PLAIN [3]. In the encryption oracle, note that $C \sim \text{PRF}_K(A, P_L)$. By requiring a nonce in

Algorithm 1.1: Nonce- and redundancy-encrypting AEAD mode

```
   Definition : Farfalle instance $\mathcal{F}$
   Definition : Security level $s = 128$ bits
1  function encrypt(key $K$, metadata $A$, plaintext $(P_L, P_R)$): ciphertext
2      assert $(K, A, P_L)$ is a nonce
3      assert $|P_L| \geq s$
4      assert $|P_R| \geq 2s$ and $P_R$ contains valid redundancy
5      $C_R \leftarrow P_R \oplus \mathcal{F}_K(P_L \| 0 \circ A)$
6      $C_L \leftarrow P_L \oplus \mathcal{F}_K(C_R \| 1 \circ A)$
7      return $(C_L, C_R)$
8  function decrypt(key $K$, metadata $A$, ciphertext $(C_L, C_R)$): plaintext or $\perp$
9      if $|C_L| < s$ or $|C_R| < 2s$ then return $\perp$
10     $P_L \leftarrow C_L \oplus \mathcal{F}_K(C_R \| 1 \circ A)$
11     $P_R \leftarrow C_R \oplus \mathcal{F}_K(P_L \| 0 \circ A)$
12     if not valid$(P_R)$ then return $\perp$
13     return $(P_L, P_R)$
```

(K, A, P_L), 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) satisfies the nonce requirement. 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. Note that a bulk of the ciphertext bits are produced immediately after compressing P_L. This leads to another advantage of keeping P_L short: it allows the encryption oracle to be online.

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 line 6 of algorithm 1.1. This optimization brings the mode even closer in parity to Deck-PLAIN. For brevity, the modified algorithm is deferred to a future paper.

2.2 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 in function *valid*
- **Optional/static metadata**: the metadata string compression can be skipped if empty; additionally, static metadata can be factored out and reused across invocations

2.3 Security proof

We defer the security proof to a future revision.

2.4 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 1.1 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.

3 Nonce- and redundancy-encrypting AEAD mode with RUP resistance

We now present the RUP-resistant mode in algorithm 1.2. Similar to algorithm 1.1, the mode is parameterized in terms of the Farfalle instance $\mathcal{F}$ and target security level s.

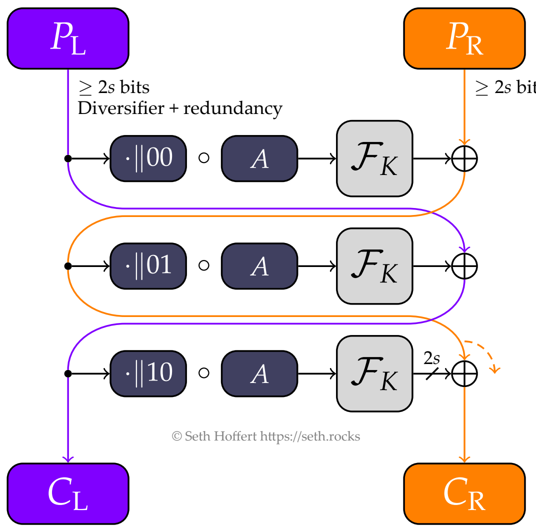

Fig. 2: Nonce- and redundancy-encrypting AEAD mode with RUP resistance

Algorithm 1.2: Nonce- and redundancy-encrypting AEAD mode with RUP resistance

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

**function** encrypt(key $K$, metadata $A$, plaintext $(P_L, P_R)$): ciphertext
    **assert** $(K, A, P_L)$ is a nonce
    **assert** $|P_L| \geq 2s$ and $P_L$ contains valid redundancy
    **assert** $|P_R| \geq 2s$
    $C_R \leftarrow P_R \oplus \mathcal{F}_K(P_L \| \texttt{00} \circ A)$
    $C_L \leftarrow P_L \oplus \mathcal{F}_K(C_R \| \texttt{01} \circ A)$
    $C_{R0} \leftarrow C_{R0} \oplus \mathcal{F}_K(C_L \| \texttt{10} \circ A)$
    **return** $(C_L, C_R)$

**function** decrypt(key $K$, metadata $A$, ciphertext $(C_L, C_R)$): plaintext or $\perp$
    **if** $|C_L| < 2s$ **or** $|C_R| < 2s$ **then return** $\perp$
    $C_{R0} \leftarrow C_{R0} \oplus \mathcal{F}_K(C_L \| \texttt{10} \circ A)$
    $P_L \leftarrow C_L \oplus \mathcal{F}_K(C_R \| \texttt{01} \circ A)$
    **if not** valid$(P_L)$ **then return** $\perp$
    $P_R \leftarrow C_R \oplus \mathcal{F}_K(P_L \| \texttt{00} \circ A)$
    **return** $(P_L, P_R)$
```

3.1 Details

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).

Note that it is important that the verifiable redundancy be placed in P_L. If no redundancy is present in P_L and the adversary is able to observe unverified plaintext from the decryption oracle, keystream can be harvested for the observed nonce in P_L. Although P_L is non-malleable, if the nonces are short then keystream harvesting could become feasible. By placing s bits of redundancy in P_L, the adversary would need to perform on average 2^{s-1} decryption queries to obtain a P_L that would even be considered valid by the encryption oracle.

A 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 the nonce, 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 separate redundancy. Additionally, an application that uses ephemeral keys with a metadata-based counter (e.g., onion routing) need not place any additional nonce value in P_L. Instead, P_L can be a block of plaintext.

3.2 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 in function *valid*
- **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 routing 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

3.3 Security proof

We defer the security proof to a future revision.

3.4 Application: onion routing protocol

Consider an onion routing 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 preserving**: 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 1.1 does not quite fit the bill either because of its lack of RUP resistance. Algorithm 1.2 is exactly what we need.

In order to prevent keystream harvesting, 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.

4 Session support

In the interest of conciseness of description, neither of the presented modes has native session support. One could implement session-based versions of both of the presented algorithms simply by keeping an incremental history. Implementing this feature is deferred to a future paper.

5 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 1.2 is ideal for situations when RUP is needed (e.g., onion routing). 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