THE MARVIN MESSAGE AUTHENTICATION CODE AND THE LETTERSOUP AUTHENTICATED ENCRYPTION SCHEME

MARCOS A. SIMPLICIO JR, PEDRO D'AQUINO F. F. S. BARBUDA, PAULO S. L. M. BARRETO†,
TEREZA C. M. B. CARVALHO, AND CINTIA B. MARGI

ABSTRACT. We present MARVIN, a new parallelizable message authentication code based on the ALRED family. The new algorithm is designed with resource-constrained platforms in mind and explores the structure of an underlying block cipher to provide security at a small cost in terms of memory needs. Also, we show how MARVIN can be used as an authentication-only function or else in an Authenticated Encryption with Associated Data (AEAD) scheme. We define a new AEAD proposal called LETTERSOUP, which is based on the LFSRC mode of operation. Finally, we analyze the security and performance of the resulting schemes.
Keywords: cryptographic algorithms, message authentication codes, authenticated encryption with associated data.

1. INTRODUCTION

Consider the problem of computing a message authentication code (MAC) for a message M under a k-bit key K. For convenience, assume that M is partitioned in t blocks $M_1 \ldots M_t$, where all blocks M_i except possibly M_t are n bits long. If an n-bit block cipher E is available, a commonplace and economical strategy is to construct a MAC based on E.

Conventional block cipher based MAC schemes invoke the underlying block cipher $t+\varepsilon$ times, where ε stands for a small fixed number. Typically, the ancillary processing in such schemes is much simpler and computationally less expensive than the cost of the block cipher invocations. This is important for space-constrained platforms like smart cards or cheap dedicated hardware, since the extra code storage or circuit area needed for the MAC algorithm is small, say, within 10% of the space needed by E itself. Examples of such schemes are CMAC [15, 21] and PMAC [6].

In contrast, Carter-Wegman [30] schemes potentially reduce the number of block cipher invocations to ε. This happens when the Carter-Wegman structure can be implemented more efficiently than one E call per message block, which is the case on platforms with large storage spaces. The overhead per message block is then typically around 10%–20% of a block cipher call. Besides, Carter-Wegman is fully parallelizable. However, space-constrained platforms must resort to space-saving but much slower implementation techniques, which nevertheless increase code storage or circuit area requirements as compared to conventional schemes and thus constitute an overall efficiency degradation.

The ALRED construction [11] provides a trade-off for iterated block ciphers that process data blocks in chunks of fixed bitlength, notably (but not exclusively) the ciphers of the SQUARE [8] or SHARK [22] family, which includes AES [20]. The cost of processing a message block is typically 25%–40% of a block cipher call. This is slower than the Carter-Wegman approach when storage is abundant, but since the ancillary processing is directly

†Supported by the Brazilian National Council for Scientific and Technological Development (CNPq) under grant 312005/2006-7. ‡Special thanks to Ericsson Research for the valuable support on this work.

based on the block cipher components, the extra code storage or circuit area is comparable to conventional schemes and thus far smaller. The ALRED construction is thus a good midway option to either conventional or Carter-Wegman MAC schemes, combining reasonable efficiency with platform flexibility. However, it has the disadvantage of being strictly sequential. This is particularly undesirable when we consider a situation where cheap dedicated hardware is available, leading to a scenario combining the limitations of code/area constrained devices with the large processing power of a highly parallel system.

Our main contributions in this paper are the definition of a new, fully parallelizable MAC called Marvin, based on the ALRED family and designed with resource-constrained platforms in mind, as well as a new Authenticated Encryption with Associated Data (AEAD) scheme based on Marvin, called LetterSoup, which is based on the LFSRC mode of operation and shows a high performance according to our benchmarks. We also present an initial work concerning the security analysis of the resulting schemes, giving the basis for a formal security proof.

This paper is organized as follows. Section 2 defines the mathematical background and notation used herein. We define Marvin structure and justify the design decisions on section 3. Section 4 discuss how the algorithm can be used in an AEAD scheme and describe LetterSoup. We present our security analysis for both algorithms on sections 5 and 6. Section 7 briefly covers the overall performance of the algorithm, while section 8 complements the discussion by showing our benchmark results. Finally, we conclude in section 9.

2. Preliminaries and notation

In what follows, $E : \{0,1\}^k \times (\{0,1\}^w)^b \to (\{0,1\}^w)^b$ stands for an iterated block cipher structured as a substitution-permutation network (SPN) with a key size of k bits and a block size of $n = bw$ bits, and $E_K(B)$ denotes the encryption of block B under key K. Here we assume that the n-bit data blocks are organized and processed as b bundles of w bits each (typically $w = 4$ or $w = 8$, meaning that the data is organized in nybbles or bytes, respectively); we say that E is a b-bundle block cipher.

A substitution table or S-box is a nonlinear mapping $S : \{0,1\}^w \to \{0,1\}^w$. S-boxes are the usual way to introduce nonlinearity in iterated block ciphers.

We represent the finite field with 2^n elements as $\mathrm{GF}(2^n) \equiv \mathrm{GF}(2)[x]/p(x)$, for a irreducible polynomial $p(x)$ of degree n such that x^w is a generator modulo $p(x)$.

We write $\mathbf{bin}(\ell)$ for the binary representation of the integer ℓ if $\ell > 0$, or the empty string if $\ell = 0$. The length of a bitstring B is denoted $|B|$. If $|B| \leqslant n$, then $\mathbf{rpad}(B) \equiv B \,||\, 0^*$ (resp. $\mathbf{lpad}(B) \equiv 0^* \,||\, B$) is the bitstring consisting of B left-padded (resp. right-padded) with as many binary zeros as needed (possibly none) to produce an n-bit string. Also, $B[\tau]$ is the string consisting of the leftmost τ bits of B. Finally, if B is a bitstring of fixed length w, the expressions $B \ll m$, $B \gg m$, and B **rotl** m stand respectively for B left-shifted, right-shifted, and left-rotated by m bits (with its length preserved, and padded with zero bits where needed).

2.1. Square-complete transforms.

Definition 1. *Let E be a b-bundle iterated block cipher. A* square-complete transform *(*SCT *for short)* $\blacksquare : (\{0,1\}^w)^b \to (\{0,1\}^w)^b$ *is the shortest sequence of unkeyed rounds of E such that any differential characteristic of E spanning that number of rounds contains at least b active S-boxes.*

SCTs are the kernel of ALRED message authentication codes. Ciphers of the Square [8] family have block size $b = (d-1)^2$ bundles where d is the distance of a certain linear error correcting code, and one can show [10] that any 4-round differential characteristic of such a cipher contains at least d^2 active S-boxes, even though there are 3-round differential characteristics containing no more than $2d - 1$ active S-boxes. This is the reason why Pelican adopts 4-round SCTs derived from these ciphers[1]. Similarly, ciphers of the Shark [22] family have block size $b = d - 1$ and any 2-round differential characteristic of such a cipher contains at least d active S-boxes, hence one could define ALRED schemes based on 2-round SCTs derived from these ciphers.

Other ciphers are handled analogously, although determining the number of rounds of an SCT may not be straightforward. For instance, Present [7] is a 16-bundle cipher known to contain at least 10 active S-boxes at each 5 rounds, so an SCT necessarily spans between 6 and 10 rounds, but computing the exact number (namely, 7 rounds) is somewhat involved. The same holds for Noekeon [9], which is a 32-bundle cipher containing at least 20 active S-boxes at each 4 rounds; here an SCT would span between 5 and 8 rounds but the exact number is not currently known.

3. The Marvin MAC structure

The Marvin structure closely follows the randomize-then-combine paradigm, introduced by Bellare and Micciancio in the context of incremental hash function design [4]. The advantage of this approach is twofold: the resulting MAC function is both incremental and parallelizable. The concept of incrementality can be better explained through an example: suppose one computes the MAC of a message M, resulting in the tag T, and later this message is modified, becoming M'. With an incremental MAC function, the new tag M' can be computed from T at a cost proportional the amount of modification made in M to get M', thus avoiding the need of computing T' from the scratch. This is particularly interesting when we consider huge messages (an entire hard disk, for example).

Marvin adopts a variant of Krawczyk's $h_p(M)$ cryptographic CRC [16] to generate secret offsets, and a square-complete transform to mix the offsets with message blocks. These operations are applied to each message block in parallel and accumulated. The encrypted (and then possibly truncated) accumulation result defines the MAC tag. Marvin uses a constant c left-padded to n bits to generate the offset seed. Algorithm 1 describes the Marvin message authentication code, written as $\textsc{Marvin}(M, K, \tau)$. It is also depicted on Figure 1.

3.1. **On the choice of c.** To ensure that the offsets O_i are non-zero it is necessary that R itself be non-zero. Assuming that the block cipher E has no weak key $\tilde{K}$ such that $E_{\tilde{K}}(B) = B$ for any block B, or that finding such a key is infeasible for any fixed block B, it is clear that $E_K(B) \oplus B \neq 0^n$. All things being equal, the simple choice $c = 0$ would be suitable. Indeed, many MAC constructions and other modes of operation for block ciphers adopt the encryption of a block of null bits, $E_K(0^n)$, as the seed for offset generation.

However, it is essential that R remain secret, yet disclosure of $E_K(0^n)$ for key confirmation purposes is widespread if not recommended in many scenarios, especially financial applications. It is therefore advisable to choose $c \neq 0$ for the sake of robust deployment in real-world systems. Any non-zero constant would seems to do equally well, a small (say, 1-byte) value being preferred not to impart efficiency on constrained processors. The actual value is $c = \texttt{0x2A}$, which was chosen largely at random, although it might be better justified in certain contexts [1].

[1]The Alpha-MAC scheme is somewhat different in that it adopts 1-round transforms, applying them to partial blocks of size $d - 1$.

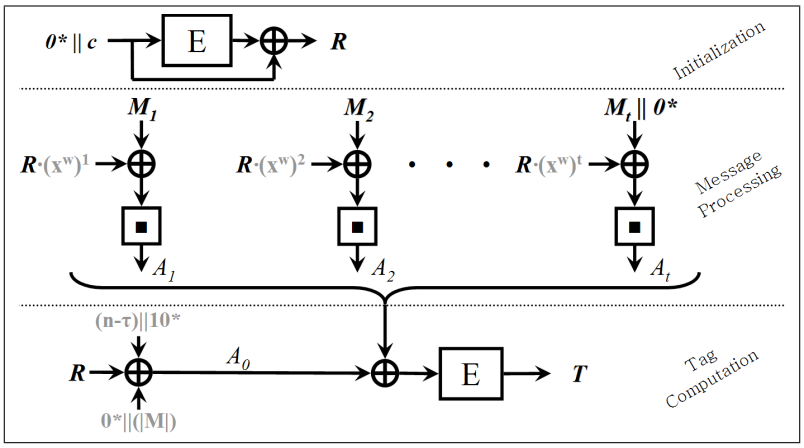

FIGURE 1. The MARVIN message authentication code

Algorithm 1 The MARVIN message authentication code

INPUT: M ▷ message to authenticate, with $|M| < 2^{\lfloor n/2 \rfloor}$.
INPUT: K ▷ MAC key.
INPUT: τ ▷ desired MAC length ($\tau \leqslant n$).
OUTPUT: T ▷ MAC of message M under key K, truncated to τ bits.

1: Partition the message as $M = M_1 \| \ldots \| M_t$, where $|M_i| = n$ for $i = 1, \ldots, t-1$, and $0 \leqslant |M_t| \leqslant n$ with $|M_t| = 0$ iff $|M| = 0$.
2: $R \leftarrow E_K(\mathbf{lpad}(c)) \oplus \mathbf{lpad}(c)$
3: **for** $i \leftarrow 1$ **to** t **do** ▷ incrementally or in parallel
4: $\quad O_i \leftarrow R \cdot (x^w)^i \quad ; \quad A_i \leftarrow \blacksquare(\mathbf{rpad}(M_i) \oplus O_i)$
5: **end for**
6: $A_0 \leftarrow R \oplus \mathbf{rpad}(\mathbf{bin}(n-\tau) \| 1) \oplus \mathbf{lpad}(\mathbf{bin}(|M|))$
7: $A \leftarrow \bigoplus_{i=0}^{t} A_i$ ▷ incrementally or in parallel
8: $T \leftarrow E_K(A)[\tau]$
9: **return** T

3.2. **Offset generation.** Computing offsets involves multiplication in a finite field GF(2^n), and hence has to be designed carefully to avoid unnecessary burden on a variety of platforms, particularly those where MAC computation must be carried out incrementally instead of in parallel.

The original Krawczyk proposal involves multiplication by the polynomial x^n and hence a shift by the full block length n followed by an n-step reduction, which is likely to be overkill. The variant employed in CS mode [26] and PMAC1 [24] consists of multiplication by the polynomial x and hence a shift by a single bit; although the ensuing reduction is very simple, this choice destroys bundle alignment and incurs costly operations on many platforms, particularly 8-bit smart cards and SSE2 processors. Such drawbacks are clearly undesirable.

MARVIN adopts the middle way: since the underlying block cipher organizes the data in w-bit bundles (supposedly fit for efficient implementation on the target platform), the most natural shift distance is w: it preserves bundle alignment, and as a consequence the offset generation involves multiplication by x^w. The polynomial reduction is potentially more

complex than that incurred by 1-bit shift; however, this can be remedied by using a small table, as described in appendix 12 for the special case where the block cipher is the AES.

3.3. **Message padding.** Perhaps the most popular padding method used in MAC schemes consists of replacing the last block M_t by $\mathbf{rpad}(M_t \| 1)$ when $|M_t| < n$, and adding a new block when $|M_t| = b$. This method ensures that no two distinct messages are padded to the same string, which would trivially lead to a MAC collision. A related method avoids adding an entire new block when $|M_t| = n$ by handling the last block in a different manner, for instance by using a different offset.

Marvin adopts a different approach. The last block is simply completed with zero bits; however, the message length and also the truncation length are separately included in the MAC tag computation. This avoids both the block addition and the need for a different block treatment.

4. Authenticated encryption with associated data

An authenticated-encryption (AE) scheme is a symmetric-key mechanism that provides both confidentiality (through the encryption of M) and authentication (through its MAC calculation). Certain modes of operation can authenticate messages consisting of both plaintext and ciphertext. This is the case when one desires to authenticate not only the encrypted message but also its packet header (a TCP/IP header, for example). A scheme of this kind is called an authenticated encryption mode with associated data (AEAD), where the "associated data" in the previous example is the packet header.

4.1. **Using Marvin in an authenticated encryption mode with associated data.** To date, many AE schemes have been proposed and analyzed, such as EAX [5], GCM [18] and AEM [24]. However, one problem with EAX is that it is not entirely parallelizable due to the use of CMAC. GCM is parallelizable in principle, but the definition of its internal MAC, GHASH, makes it necessary to either implement finite field exponentiation to combine the authentication tags of the plaintext and ciphertext parts computed in parallel, or else to impose that the full plaintext part be authenticated before processing of the ciphertext part begins, thus preventing complete parallelizability. AEM is fully parallelizable without the need of expensive arithmetic, but authentication costs one full cipher invocation per block. CS mode is almost ideal in this regard, but it needs both encryption and decryption block cipher functionalities to be available (all other modes considered here only need the encryption direction) and, according to its authors, it is not recommended to use with involutional block ciphers (a restriction not present in the other modes).

It turns out that one can avoid all these drawbacks by simply substituting Marvin for CMAC in the definition of EAX. Another possibility is to use Marvin in a new AEAD construction, discussed in the following.

4.2. **A new AEAD-mode proposal: LetterSoup.** It is interesting to notice that the linear feedback shift register counter (LFSRC) mode for block ciphers [25] could also be based on, and benefit from, the Marvin choice of offsets. This way, it is possible to maintain a single counter and use it both for the encryption and authentication sub-processes. The LFSRC mode is not only length-preserving ($|C| = |M|$) but also involutional: applying it twice recovers the original plaintext (cf. Algorithm 8).

Algorithm 2 describes the resulting LFSRC-Marvin authenticated-encryption mode of operation that, for the sake of pronunciation, we call LetterSoup(N, M, H, K, τ). For convenience, we write $\blacksquare^*(R, C, \tau)$ as the value A accumulated by steps 3 to 7 of the Marvin algorithm for (encrypted) message C and tag length τ. Also, LFSRC(N, M, K) denote the

LFSRC mode of operation, as described in Algorithm 8 and depicted in Figure 2. It is important to notice that the specific way the nonce is preprocessed ensures that the offset seeds R and L passed as parameters to $\blacksquare^*$ and LFSRC are nonzero, as long as the underlying block cipher does not have weak keys as discussed in section 3.1. Also, the nonce $N = 0^n$ is not allowed, since it is reserved for the computation of L.

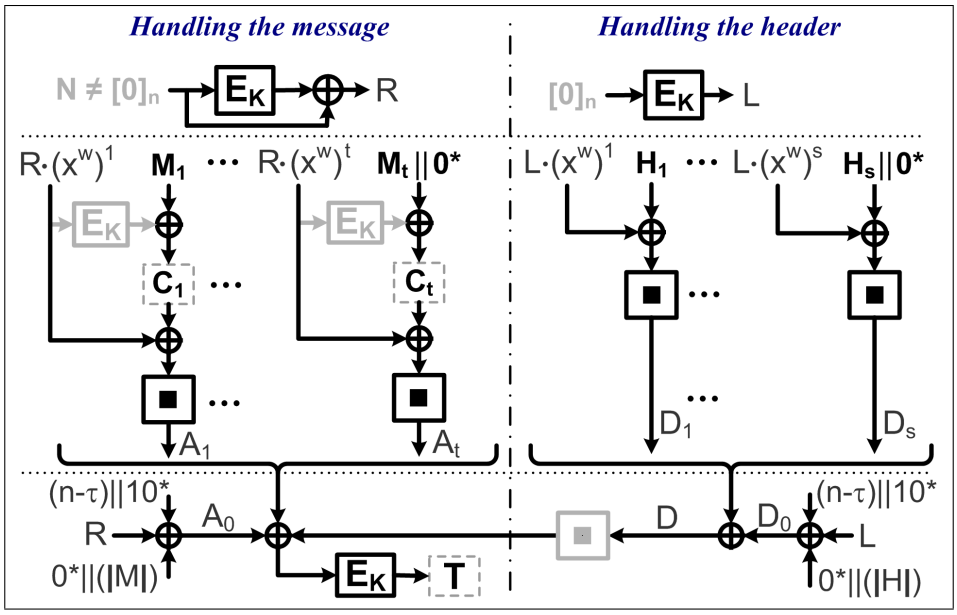

FIGURE 2. LETTERSOUP: AEAD mode

Algorithm 2 LETTERSOUP: AEAD mode

INPUT: N ▷ nonce, an integer value in range $0 < N < 2^n$.
INPUT: M ▷ message to encrypt and authenticate, with $|M| < 2^{\lfloor n/2 \rfloor}$.
INPUT: H ▷ associated data, with $|H| + |M| < 2^{\lfloor n/2 \rfloor}$.
INPUT: K ▷ cipher key.
INPUT: τ ▷ desired MAC length ($\tau \leqslant n$).
OUTPUT: C ▷ ciphertext of M under K.
OUTPUT: T ▷ MAC of the message, truncated to τ bits.

1: Partition the message as $M = M_1 \| \ldots \| M_t$, where $|M_i| = n$ for $i = 1, \ldots, t-1$, and $0 \leqslant |M_t| \leqslant n$ with $|M_t| = 0$ iff $|M| = 0$
2: $R \leftarrow E_K(\mathbf{lpad}(\mathbf{bin}(N))) \oplus \mathbf{lpad}(\mathbf{bin}(N))$
3: $C \leftarrow \text{LFSRC}(R, M, K)$; $A \leftarrow \blacksquare^*(R, C, \tau)$
4: **if** ($H \neq \varepsilon$ and $|H| > 0$) **then**
5: Partition the header as $H = H_1 \| \ldots \| H_s$, where $|H_i| = n$ for $i = 1, \ldots, s-1$, and $0 < |H_s| \leqslant n$
6: $L \leftarrow E_K(\mathbf{bin}(0^n))$; $D \leftarrow \blacksquare^*(L, H, \tau)$; $A \leftarrow (A \oplus \blacksquare(D))$
7: **end if**
8: $T \leftarrow E_K(A)[\tau]$
9: **return** $C \| T$

It is also possible to further extend this idea to allow for associated data. Suppose the authentication tag for some confidential message M under key K and nonce N is $A \leftarrow \textsc{LetterSoup}(N, M, \varepsilon, K, \tau)$, and the authentication tag for some associated (cleartext) data H is $D \leftarrow \textsc{LetterSoup}(0^n, H, \varepsilon, K, \tau)$. The simplest way to combine them would be setting $T = A \oplus D$, as is the case with for the AEM mode [24]. However, if the same key K is used to authenticate H in a context *distinct* from that where K is used to encrypt and authenticate M (so that actually M and H are semantically unrelated to each other), an attacker can semantically link M and H by intercepting the authenticated encrypted message (N, C, ε, A) corresponding to M and forging the authenticated encrypted message with associated data $(N, C, H, A \oplus D)$ without knowing the key.

This undesirable situation could be prevented by using one-time keys, but the whole point of using nonces is to avoid the need to change keys too often. Another possible solution would be similar to the one adopted by EAX, which prepends a different constant t to the OMAC algorithms (OMAC^t) used for authenticating H and M, but this could bring a negative impact to performance. We decided to adopt a more elegant e efficient way to thwart the problem, combining the accumulated tags before the final encryption using the $\blacksquare$ operation either on A or D, the tags computed from M and from H, respectively.

5. On the Security of Marvin

We evaluate the Security of Marvin considering both key recovery attacks and message forgery. Most attacks are based in the notion of *accumulation collisions*, which generalize internal collisions as defined in [11, section 2.2] for parallelizable MAC schemes. Two messages $M = M_1 \dots M_r$ and $M' = M'_1 \dots M'_s$ are said to cause an accumulation collision if there exists a subset of indices $I \subseteq \{1, \dots, \min(r, s)\}$ such that $M_k = M'_k$ for all $k \in I$ and $\bigoplus_{i \notin I} A_i = \bigoplus_{j \notin I} A'_j$, where A_i (resp. A'_j) denotes the internal state of the MAC algorithm corresponding to the processing of message block M_i (resp. M'_j). Clearly such messages have the same MAC value, and hence given the tag of either of them one can forge the tag of the other. Finding the collision on the A_i and A'_j internal state is equivalent to solving the balance problem [4] in the additive group of a binary field *without* knowing the group elements, since these states depend on a secret key.

This phenomenon is unavoidable regardless of the details of the (parallelizable) MAC: if the same key is used to authenticate a large enough number of messages, namely, $O(2^{n/2})$ messages for a b-bit underlying block cipher, the birthday paradox [28] makes the condition $\bigoplus_{i \notin I} A_i = \bigoplus_{j \notin I} A'_j$ likely to occur, opening the way to forgery. Therefore, the number of messages authenticated under any particular key must be much smaller than $2^{\lfloor n/2 \rfloor}$.

As stated in the ALRED original security analysis, two general approaches to exploit the iteration function must be considered in order to generate Accumulation Collisions: *Extinguishing Differentials* and *Fixed Points*. The use of Extinguishing Collisions is similar to Differential Cryptanalysis of Block Ciphers. It consists in creating pairs of messages of equal length having a difference (with respect to some group operation at the choice of the attacker) that, with a high probability, results in a zero difference in a state following the difference injection. As the use of SCTs prevents differential trails with high probability, it assures a high resistance of the ALRED construction to this kind of attack. On the other hand, Fixed Points refer to words that can be inserted in the message sequence without impacting the state value after the insertion point. To avoid the viability of finding such a word using the MAC function as an oracle, the number of times a message can be authenticated under the same key is upper-bounded by $2^{\lfloor l_c/2 \rfloor}$, where the parameter l_c is named the

capacity. Preliminary studies of the ALRED design show that $l_c = b - 8$ is usually a good approximation for the capacity, with b the block size of the underlying cipher.

Given an *adversary* (a program with access to one or more oracles), the following theorems generalize the provable security properties satisfied by strictly sequential instances of the ALRED construction [11, section 3.3] in a straightforward fashion.

Theorem 1. *Every key recovery attack on* Marvin *requiring q (adaptively) chosen messages can be converted to a key recovery attack on the underlying block cipher requiring $q + 1$ adaptively chosen plaintexts.*

Proof. Let $\mathcal{E}$ be an encryption oracle that yields the ciphertext of a given message under the same (unknown) key K, and let $\mathcal{A}$ be a MAC adversary that, given the authentication tags (of length τ) corresponding to q adaptively chosen messages $M^{(j)}$, yields the MAC key. The key recovery attack on the underlying block cipher proceeds as follows.

(1) Request $c' = E_K(\mathbf{lpad}(c))$ from the encryption oracle $\mathcal{E}$ and set $R = c' \oplus \mathbf{lpad}(c)$.
(2) For $j = 1$ to q:
    (a) Request the adaptively chosen message $M^{(j)}$ from $\mathcal{A}$.
    (b) Compute $A^{(j)} = \blacksquare^*(R, M^{(j)}, \tau)$.
    (c) Request $T^{(j)} = E_K(A^{(j)})$ from $\mathcal{E}$ and input $T^{(j)}[\tau]$ to $\mathcal{A}$.
(3) Request the MAC key (which is also the cipher key) K from $\mathcal{A}$.

□

Theorem 2. *Every forgery attack on* Marvin *not involving accumulation collisions and requiring q (adaptively) chosen messages can be converted to a (known plaintext) ciphertext guessing attack on the underlying block cipher requiring $q+1$ adaptively chosen plaintexts.*

Proof. Let $\mathcal{E}$ be an encryption oracle that yields the ciphertext of a given message under the same (unknown) key K, and let $\mathcal{A}$ be a MAC adversary that, without involving accumulation collisions, given a message M and the authentication tags of length τ corresponding to q adaptively chosen messages $M^{(j)}$, yields the authentication tag of length τ corresponding to M. The ciphertext guessing (fake encryption) attack on the underlying block cipher proceeds as follows.

(1) Request $c' = E_K(\mathbf{lpad}(c))$ from the encryption oracle $\mathcal{E}$ and set $R = c' \oplus \mathbf{lpad}(c)$.
(2) Input the message M and the length τ to $\mathcal{A}$.
(3) For $j = 1$ to q:
    (a) Request the adaptively chosen message $M^{(j)}$ from $\mathcal{A}$.
    (b) Compute $A^{(j)} = \blacksquare^*(R, M^{(j)}, \tau)$.
    (c) Request $T^{(j)} = E_K(A^{(j)})$ from $\mathcal{E}$ and input $T^{(j)}[\tau]$ to $\mathcal{A}$.
(4) Request from $\mathcal{A}$ the authentication tag T of length τ corresponding to M.
(5) Compute $A = \blacksquare^*(R, M, \tau)$.
(6) If there is an index j for which $A = A^{(j)}$, then $\mathcal{A}$ has generated an accumulation collision, which conflicts with the assumption that the attack perpetrated by $\mathcal{A}$ does not involve any such circumstance. Otherwise the tag T is the ciphertext (truncated to length τ) corresponding to the known plaintext A.

□

Loosely speaking, Theorems 1 and 2 state that Marvin is no less secure than the underlying block cipher, since an attacker capable of breaking the MAC (by either recovering the key or simply forging a tag) using the cipher as a black box is necessarily also capable of breaking the cipher itself (by either recovering the key or faking a ciphertext) with roughly the same effort.

6. On the Security of LetterSoup

In this section we introduce formal security definitions and build our security analysis of LetterSoup. We follow mainly [5] for the set of definitions presented here.

6.1. **Definitions.** AEAD Schemes. A *set of keys* is a nonempty set of strings having a distribution (the uniform distribution when the set is finite). A (nonce-based) *authenticated-encryption with associated data* (AEAD) scheme is a pair of algorithms $\mathbf{\Pi} = (\mathbf{D}, \mathbf{E})$ where $\mathbf{E}$ is a deterministic *encryption* algorithm $\mathbf{E} : \mathsf{Key} \times \mathsf{Nonce} \times \mathsf{Header} \times \mathsf{Plaintext} \rightarrow \mathsf{Ciphertext}$ and $\mathbf{D}$ is a deterministic *decryption* algorithm $\mathbf{D} : \mathsf{Key} \times \mathsf{Nonce} \times \mathsf{Header} \times \mathsf{Ciphertext} \rightarrow \mathsf{Plaintext} \bigcup \textsc{Invalid}$. The *Key space* Key is a set of keys while the *nonce space* Nonce and the *header space* Header (also called the space of *associated data*) are non-empty sets of strings. We will write $\mathbf{E}_K^{N\ H}(M)$ for $\mathbf{E}(K, N, H, M)$ and $\mathbf{D}_K^{N\ H}(CT)$ for $\mathbf{D}(K, N, H, CT)$, with $K \in \mathsf{Key}$, $N \in \mathsf{Nonce}$, $H \in \mathsf{Header}$, $M \in \mathsf{Plaintext}$, and $CT \in \mathsf{Ciphertext}$. We require that $\mathbf{D}_K^{N\ H}(\mathbf{E}_K^{N\ H}(M)) = M$ for all $M \in \mathsf{Plaintext}$. For notational simplicity, we assume along this document that Nonce, Header, Plaintext and Ciphertext are all $\{0,1\}^*$ and that $|\mathbf{E}_K^{N\ H}(M)| = |M|$.

Nonce-Respecting. Suppose $\mathcal{A}$ is an adversary with access to an *encryption oracle* $\mathbf{E}_K^{\cdot\ \cdot}(\cdot)$. This oracle, on input (N, H, M), returns $\mathbf{E}_K^{N\ H}(M)$. Let the tuples $(N_1, H_1, M_1), \ldots, (N_q, H_q, M_q)$ denote its oracle queries. Adversary $\mathcal{A}$ is said to be *nonce-respecting* if it never repeats N, i.e. if $N_1, \ldots, N_q$ are always distinct, regardless of oracle responses and of $\mathcal{A}$'s internal coins. Adversaries for AE and AEAD schemes are always assumed to be nonce-respecting [23]. We write $\mathcal{A}^{e(\cdot)}$ to indicate that $\mathcal{A}$ uses oracle $e(\cdot)$.

Privacy of AEAD Schemes. We consider nonce-respecting adversaries with access to an encryption oracle $\mathbf{E}_K^{\cdot\ \cdot}(\cdot)$. The advantage of such an adversary $\mathcal{A}$ in violating the privacy of an AEAD scheme $\mathbf{\Pi} = (\mathbf{D}, \mathbf{E})$ having key space Key is:

$$\mathbf{Adv}_{\Pi}^{PRIV}(\mathcal{A}) = Pr[K \xleftarrow{\$} \mathsf{Key} : \mathcal{A}^{\mathbf{E}_K^{\cdot\ \cdot}(\cdot)} = 1] - Pr[K \xleftarrow{\$} \mathsf{Key} : \mathcal{A}^{\$^{\cdot\ \cdot}(\cdot)} = 1].$$

where $\$^{\cdot\ \cdot}(\cdot)$ denotes an oracle that, on input (N, H, M), returns a random string of length $|M|$.

Authenticity of AEAD Schemes. We provide the adversary $\mathcal{A}$ with two oracles: an encryption oracle $\mathbf{E}_K^{\cdot\ \cdot}(\cdot)$ as the one above and a *verification oracle* $\hat{\mathbf{D}}_K^{\cdot\ \cdot}(\cdot)$. The latter oracle takes as input (N, H, CT), returning 1 if $\mathbf{D}_K^{N\ H}(CT) \in \mathsf{Plaintext}$ and 0 if $\mathbf{D}_K^{N\ H}(CT) = \textsc{Invalid}$. The adversary is assumed to satisfy the following three conditions, which must hold regardless of the response to the its oracle queries or $\mathcal{A}$'s internal coins:

- Adversary $\mathcal{A}$ must be nonce-respecting concerning its encryption oracle; a nonce used in an encryption oracle query may be used in a query to its verification oracle, though.
- Adversary $\mathcal{A}$ may never make a verification oracle query (N, H, CT) such that the encryption oracle previously returned CT in response to a query (N, H, M).
- Adversary $\mathcal{A}$ must call its verification oracle exactly once, and may not subsequently call its encryption oracle. (That is, it makes a sequence of encryption oracle queries, then a verification oracle query, and then halts).

We say that an adversary forges if its verification oracle returns 1 in response to the single query made to it. The advantage of such an adversary $\mathcal{A}$ in violating the authenticity of an AEAD scheme $\mathbf{\Pi} = (\mathbf{D}, \mathbf{E})$ having key space Key is:

$$\mathbf{Adv}_{\Pi}^{AUTH}(\mathcal{A}) = Pr[K \xleftarrow{\$} \mathsf{Key} : \mathcal{A}^{\mathbf{E}_K^{\cdot\ \cdot}(\cdot), \hat{\mathbf{D}}_K^{\cdot\ \cdot}(\cdot)} forges].$$

IV-based encryption. An *IV-based encryption scheme* (IVE scheme) is a pair of algorithms $\Pi = (\mathcal{E}, \mathcal{D})$, where $\mathcal{E} : \mathsf{Key} \times \mathsf{IV} \times \mathsf{Plaintext} \to \mathsf{Ciphertext}$ is a deterministic *encryption* algorithm and $\mathcal{D} : \mathsf{Key} \times \mathsf{IV} \times \mathsf{Ciphertext} \to \mathsf{Plaintext} \bigcup \textsc{Invalid}$ is a deterministic *decryption* algorithm. The *key space* Key is a set of keys, while the *plaintext space* Plaintext, the *ciphertext space* Ciphertext and the *IV space* IV are all nonempty sets of strings. We write $\mathcal{E}_K^R(M)$ for $\mathcal{E}(K, R, M)$ and $\mathcal{D}_K^R(C)$ for $\mathcal{D}(K, R, C)$. We require that $\mathcal{D}_K^R(\mathcal{E}_K^R(M)) = M$ for all $K \in \mathsf{Key}$ and $R \in \mathsf{IV}$ and $M \in \mathsf{Plaintext}$. We assume, as before, that $\mathsf{Plaintext} = \mathsf{Ciphertext} = \{0,1\}^*$ and that $|\mathcal{E}_K^R(M)| = |M|$. Also, we assume that $\mathsf{IV} = \{0,1\}^n$ for some $n \geqslant 1$ called the *IV length*.

Pseudorandom functions. A *family of functions*, or a *pseudorandom function* (PRF) is a map $F : \mathsf{Key} \times D \to \{0,1\}^n$ where Key is a set of keys and D is a nonempty set of strings. We call n the *output length* of F. We write F_K for the function $F(K, \cdot)$ and we write $f \xleftarrow{\$} F$ to mean $K \xleftarrow{\$} \mathsf{Key}$; $f \leftarrow F_K$. We denote by $\mathcal{R}_n^\alpha$ the set of all functions with domain $\{0,1\}^\alpha$ and range $\{0,1\}^n$. We identify a function with its key, making $\mathcal{R}_n^\alpha$ pseudorandom functions. The advantage of adversary $\mathcal{A}$ in violating the pseudorandomness of the PRF $F : \mathsf{Key} \times \{0,1\}^\alpha \to \{0,1\}^n$ is given by

$$\mathbf{Adv}_F^{prf}(\mathcal{A}) = Pr[K \xleftarrow{\$} \mathsf{Key} : \mathcal{A}^{F_K(\cdot)} = 1] - Pr[\rho \xleftarrow{\$} \mathcal{R}_n^\alpha : \mathcal{A}^{\rho(\cdot)} = 1].$$

A family of functions $E : \mathsf{Key} \times D \to \{0,1\}^n$ is a *block cipher* if $D = \{0,1\}^n$ and each E_K is a permutation. We let $\mathcal{P}_n$ denote all the permutations on $\{0,1\}^n$ and define:

$$\mathbf{Adv}_E^{prp}(\mathcal{A}) = Pr[K \xleftarrow{\$} \mathsf{Key} : \mathcal{A}^{\mathcal{E}_K(\cdot)} = 1] - Pr[\pi \xleftarrow{\$} \mathcal{P}_n : \mathcal{A}^{\pi(\cdot)} = 1].$$

Resources. If **xxx** is an advantage notion for which $\mathbf{Adv}_{\mathbf{\Pi}}^{\mathbf{xxx}}(\mathcal{A})$ has been defined, we write $\mathbf{Adv}_{\mathbf{\Pi}}^{\mathbf{xxx}}(R)$ for the maximal value of $\mathbf{Adv}_{\mathbf{\Pi}}^{\mathbf{xxx}}(\mathcal{A})$ over all adversaries $\mathcal{A}$ that use resources at most R. When counting resource usage of the adversary, one maximizes over all possible oracle responses, including those that could not be returned by any experiment we have specified for adversarial advantage. Resources of interest are the ones named: t – the running time; q – the total number of oracle queries; σ – the aggregate length of these queries; $\hat{\sigma}$ – the length of the longest query. The running time t of an algorithm is its actual running time (relative to some fixed RAM model of computation) plus its description size (relative to some standard encoding of algorithms). The data complexity σ is defined as the sum of the lengths of all strings encoded in the adversary's oracle queries. In this document, the length of the strings will be measured in n-bit blocks, for some understood value n. The number of blocks in a string M is defined as $\|M\|_n = max\{1, \lceil |M|/n \rceil\}$, so that the empty string counts as one block. When the big-O notation is used, it is understood that the constants hidden by the notation are absolute constants.

6.2. **Security Results.** In the following, we analyze the pseudorandomness of LetterSoup, which is essential to show why we can use a single key when using Marvin and LFSRC together. We do not provide all elements for a complete security proof, but rather the ones on which a formal security analysis can be based upon.

Theorem 3. ***Pseudorandomness of* LetterSoup *with a random pseudorandom function.*** *Let $\mathcal{A}$ be a nonce-respecting attacker using resources at most (q, σ) aiming to distinguish* LetterSoup *from a random function. Then we have*

$$\mathbf{Adv}_{\textsc{LetterSoup}[\mathcal{R}_n^n]}^{prf}(q, \sigma) \leqslant 7q^2(\hat{\sigma}^2 + 1)/2^{n+1}$$

Proof. Let $\mathcal{A}$ be a nonce-respecting attacker trying to distinguish $\textsc{LetterSoup}[\mathcal{R}_n^n]$ from a random function. Assume that $\mathcal{A}$ uses resources σ and makes no repeated queries. We

simulate the behavior of LetterSoup$[\mathcal{R}_n^n]$ oracle in game **Q1**, depicted in Algorithm 3. As a standard, **Q1** avoids choosing $\rho \xleftarrow{\$} \mathcal{R}_n^n$ at the initialization and instead fills in values incrementally. We write $Domain(\rho)$ for the set of all $X \in \{0,1\}^n$ such that $\rho(X)$ has been set previously. Any time we need a $\rho(X)$ value that has not yet been defined, we choose a random value from $\{0,1\}^n$ and make this to be $\rho(X)$. When we need a $\rho(X)$ that has already been defined, we use that old value. In the latter case, we also set a flag to *bad*, which is not visible to an adversary but is of central importance for the security analysis. Let **Q1** be another game, equivalent to **R1**, where we dropped the statements that immediately follow the setting of the flag *bad* and we record the domain of ρ in a variable $\mathcal{R}$ instead of using ρ itself. This is a standard way to deal with security analysis using the game-playing technique. With these definitions, we have that $\mathbf{Adv}^{prf}_{\text{LetterSoup}[\mathcal{R}_n^n]}(\mathcal{A}) = \mathbf{Adv}^{dist}_{\mathbf{Q1},\mathbf{R1}}(\mathcal{A}) \leqslant Pr[A^{\mathbf{R1}} \text{ sets } bad]$. Thus, all we need is to understand the adversary's chance of setting *bad* in game **R1**.

Algorithm 3 Game **Q1**, which simulates LetterSoup$_\rho$, with ρ a random function from $\mathcal{R}_n^n$

1: $L \xleftarrow{\$} \{0,1\}^n \quad ; \quad \rho(0^n) \leftarrow \{L\} \quad ; \quad bad \leftarrow false$
2: On query (N, H, M), where $M = M_1 \| \ldots \| M_t$ and $N = N_1 \| \ldots \| N_u$, do the following
3: $R \xleftarrow{\$} \{0,1\}^n$
4: **if** $N \in Domain(\rho)$ **then**
5: $\quad bad \leftarrow true \quad ; \quad R \leftarrow \rho(N) \quad ; \quad \rho(N) \leftarrow R$ ▷ Due to Nonce
6: **end if**
7: **for** $i \leftarrow 1$ **to** t **do**
8: $\quad S_i \xleftarrow{\$} \{0,1\}^n \quad ; \quad O_i \leftarrow (R \oplus N) \cdot (x^w)^i$
9: $\quad$ **if** $O_i \in Domain(\rho)$ **then**
10: $\quad\quad bad \leftarrow true \quad ; \quad S_i \leftarrow \rho(O_i) \quad ; \quad \rho(O_i) \leftarrow S_i$ ▷ Due to Offset
11: $\quad$ **end if**
12: $\quad A_i \leftarrow \blacksquare(M_i \oplus O_i)$
13: **end for**
14: $A_0 \leftarrow R \oplus \mathbf{rpad}(\mathbf{bin}(n-\tau) \| 1) \oplus \mathbf{lpad}(\mathbf{bin}(|M|))$
15: $A \leftarrow \bigoplus_{i=0}^{t} A_i$
16: **for** $i \leftarrow 1$ **to** u **do**
17: $\quad O_i \leftarrow L \cdot (x^w)^i \quad ; \quad D_i \leftarrow \blacksquare(H_i \oplus O_i)$
18: **end for**
19: $D_0 \leftarrow L \oplus \mathbf{rpad}(\mathbf{bin}(n-\tau) \| 1) \oplus \mathbf{lpad}(\mathbf{bin}(|H|))$
20: $D \leftarrow \bigoplus_{i=0}^{t} D_i \quad ; \quad A \leftarrow (A \oplus \blacksquare(D))$
21: $T \xleftarrow{\$} \{0,1\}^n$
22: **if** $A \in Domain(\rho)$ **then**
23: $\quad bad \leftarrow true \quad ; \quad T \leftarrow \rho(A) \quad ; \quad \rho(A) \leftarrow T$ ▷ Due to Accumulation Collision
24: **end if**
25: **return** $T \| S_0 \ldots S_t$

There are three ways for $\mathcal{A}$ to set *bad* in game **R1**: using a Nonce N that happens to be in $\mathcal{R}$ (step 4), generate a value A_i that leads to an Accumulation Collision (step 22) or to generate an offset O_i that is already defined in $\mathcal{R}$ (step 9). In the first case, $\mathcal{A}$ knows which value has been added, but this information is not particularly useful because $\mathcal{A}$ is not able to ask another query using N (which would certainly set *bad*) and, thus, $\mathcal{A}$ has to use another of the two methods that involves variables A_i and O_i which $\mathcal{A}$ can not manipulate

Algorithm 4 Game **R1**, which simulates LetterSoup$_\rho$, with ρ a random function from $\mathcal{R}_n^n$

$L \xleftarrow{\$} \{0,1\}^n$; $\mathcal{R} \leftarrow \{L\}$; $bad \leftarrow false$
On query (N, H, M), where $M = M_1 \| \ldots \| M_t$ and $N = N_1 \| \ldots \| N_u$, do the following:
$R \xleftarrow{\$} \{0,1\}^n$
if $N \in \mathcal{R}$ **then** $bad \leftarrow true$ ▷ Due to Nonce
    $\mathcal{R} \leftarrow \mathcal{R} \cup N$
end if
for $i \leftarrow 1$ **to** t **do**
    $S_i \xleftarrow{\$} \{0,1\}^n$; $O_i \leftarrow (R \oplus N) \cdot (x^w)^i$
    if $O_i \in \mathcal{R}$ **then** $bad \leftarrow true$ ▷ Offset already defined
    end if
    $A_i \leftarrow \blacksquare(M_i \oplus O_i)$
end for
$\mathcal{R} \leftarrow \mathcal{R} \cup \{O_0, \ldots O_t\}$
$A_0 \leftarrow R \oplus \mathbf{rpad}(\mathbf{bin}(n-\tau) \| 1) \oplus \mathbf{lpad}(\mathbf{bin}(|M|))$
$A \leftarrow \bigoplus_{i=0}^{t} A_i$
for $i \leftarrow 1$ **to** u **do**
    $O'_i \leftarrow L \cdot (x^w)^i$; $D_i \leftarrow \blacksquare(H_i \oplus O'_i)$
end for
$D_0 \leftarrow L \oplus \mathbf{rpad}(\mathbf{bin}(n-\tau) \| 1) \oplus \mathbf{lpad}(\mathbf{bin}(|H|))$
$D \leftarrow \bigoplus_{i=0}^{t} D_i$; $A \leftarrow (A \oplus \blacksquare(D))$
$T \xleftarrow{\$} \{0,1\}^n$
if $A \in \mathcal{R}$ **then** $bad \leftarrow true$ ▷ Accumulation Collision
end if
$\mathcal{R} \leftarrow \mathcal{R} \cup \{A\}$
return $T \| S_0 \ldots S_t$

directly. In fact, O_i is dependent of R, which is randomly defined on each query, while A_i also depends on R and uses the $\blacksquare$ transform exactly to prevent manipulations of H or M that could lead to an accumulation collision. With these remarks in mind, we can calculate the probability associated to each method in a query q as shown in Table 1.

During the calculations we assumed that the values of the offsets O_i generated inside a query are always different, which should hold true since we consider that x^w is a generator for GF(2^n). To facilitate the calculations, we will use $\hat{\sigma} = max_i\{\sigma_i\}$ instead of each σ^i. Thus, we can write Pr(Nonce)$[q, \sigma] \leqslant (2q^2 + q\hat{\sigma}(q-1))/2^{n+1}$ and Pr(Offset)$[q, \sigma] \leqslant \hat{\sigma}(2q(1+q) + q\hat{\sigma}(q-1))/2^{n+1}$ and Pr(Ac. Collision)$[q, \sigma] \leqslant q(1+q)(2+\hat{\sigma})/2^{n+1}$, which leads to the conclusion of the proof:

$$\begin{aligned}
&\mathbf{Adv}^{prf}_{\text{LetterSoup}[\mathcal{R}_n^n]}(\mathcal{A}) = \mathbf{Adv}^{dist}_{\mathbf{Q1},\mathbf{R1}}(\mathcal{A}) \\
&\quad \leqslant Pr[A^{\mathbf{R1}} \text{ sets } bad] \\
&\quad = (4q^2 + 4q^2\hat{\sigma} + 2q\hat{\sigma} + q^2\hat{\sigma}^2 - q\hat{\sigma}^2 + 2q)/2^{n+1} \\
&\quad \leqslant 7q^2(\hat{\sigma}^2 + 1)/2^{n+1}.
\end{aligned}$$

□

This result is of fundamental importance to evaluate the security of the final LetterSoup scheme. For that, we describe attacker $\mathcal{P}$, (in Algorithm 5), which has an oracle z that

responds to queries (N, M, H) with a string $R\ S_0 S_1 \cdots S_t$ (where $t = |M|$) and simulates LetterSoup$[R_n^n]$. With this attacker, we are able achieve the following results:

$$\mathbf{Adv}^{PRIV}_{\text{LetterSoup}[\mathcal{R}_n^n,\tau]}(q,\sigma) \leqslant 7q^2(\hat{\sigma}^2+1)/2^{n+1}$$
$$\mathbf{Adv}^{AUTH}_{\text{LetterSoup}[\mathcal{R}_n^n,\tau]}(q,\sigma) \leqslant 7q^2(\hat{\sigma}^2+1)/2^{n+1} + \sigma^2/2^n + 1/2^\tau$$

Algorithm 5 Adversary $\mathcal{P}^z$ for attacking LetterSoup$[\mathcal{R}_n^n, \tau]$

Run $\mathcal{A}$
for $i \leftarrow 1$ **to** q **do**
    When $\mathcal{A}$ makes oracle query $Q_i(N, H, M)$, answer the query as follows:
    $T \,\|\, S_0 S_1 \cdots S_t \leftarrow z(N, H, M)$ ▷ with $|T| = n$
    $C \leftarrow M_i \oplus (S_0 S_1 \ldots S_t)$
    return $CT \leftarrow (C \,\|\, T[\tau])$ as the oracle response
end for
When $\mathcal{A}$ halts and outputs a bit b, **return** b
When $\mathcal{A}$ makes a forgery attempt (N, H, C, T), and halts, do the following:
$Tag \leftarrow z(N, H, M)$ ▷ with $|Tag| = n$
if $Tag[\tau] \equiv T$ **and** $(N, H, C\|T) \neq (N_i, H_i, C_i\|T_i)$ for all $1 \leqslant i \leqslant q$ **then**
    return 1
else
    return 0
end if

We omit the proof, that can be constructed using $\mathcal{P}$ like in EAX [5, Theorem 5] when one considers the LFSRC mode instead of CTR as IVE scheme. Moving to the usage of a block cipher $E : \mathsf{Key} \times D \to \{0,1\}^n$ instead of R_n^n, we can achieve the following result in a completely standard way ($t' = t + O(\sigma)$):

$$\mathbf{Adv}^{PRIV}_{\text{LetterSoup}[E,\tau]}(q,\sigma) \leqslant \mathbf{Adv}^{prp}_{E}(t',\sigma) + (\sigma^2 + 7q^2(\hat{\sigma}^2+1))/2^{n+1}$$
$$\mathbf{Adv}^{AUTH}_{\text{LetterSoup}[E,\tau]}(q,\sigma) \leqslant \mathbf{Adv}^{prp}_{E}(t',\sigma) + (\sigma^2 + 7q^2(\hat{\sigma}^2+1))/2^{n+1} + \sigma^2/2^n + 1/2^\tau$$

7. Efficiency considerations

Ignoring setup time, the computational cost incurred by a Galois-Carter-Wegman MAC like GHASH to process one message block can be as small as that needed to compute one round of the block cipher. This is achieved by using w-bit lookup tables (LUTs), which may be very large: $O(2^w b)$ n-bit blocks of extra storage, or about 64 KiB per key for a 128-bit underlying block cipher and 8-bit lookup indices. In storage-constrained environments the cost of a plain GHASH implementation is likely to be comparable to a full encryption. In contrast, Marvin typically consists of 2 rounds of a Shark-like cipher or 4 rounds of a Square-like cipher, and hence it is 2 to 4 times slower than GHASH. However, Marvin does not require any extra storage space since it simply reuses part of the already available block cipher implementation itself. Thus on storage-constrained platforms Marvin can easily match Galois-Carter-Wegman schemes.

On the other hand, Marvin is typically 2.5–4 times faster than MAC constructions that need a full cipher invocation per message block. Furthermore, contrary to strictly sequential MAC constructions like CMAC or Pelican [12], Marvin is fully parallelizable.

Table 2 summarizes the features and efficiency considerations for the authentication schemes discussed here, while Table 3 does the same for all the AEAD modes considered.

8. Benchmark results

In this section, we provide a preliminary benchmark of our Marvin implementations, comparing it to CMAC, EAX and GCM both in constrained and powerful platforms. The underlying block cipher chosen for the following tests is AES with 128-bit keys, for which we have $R = 4$ rounds instead of the usual 10 rounds.

8.1. **32-bit Platform: x86.** The AES, EAX and GCM implementations used for the tests are the ones developed by Gladman [13], while Marvin, LetterSoup and CMAC were developed according to specification. All the implementation are written in C++ and compiled using Visual Studio. The results of our tests, performed on a Intel Core 2 Processor, are depicted in Figure 3.

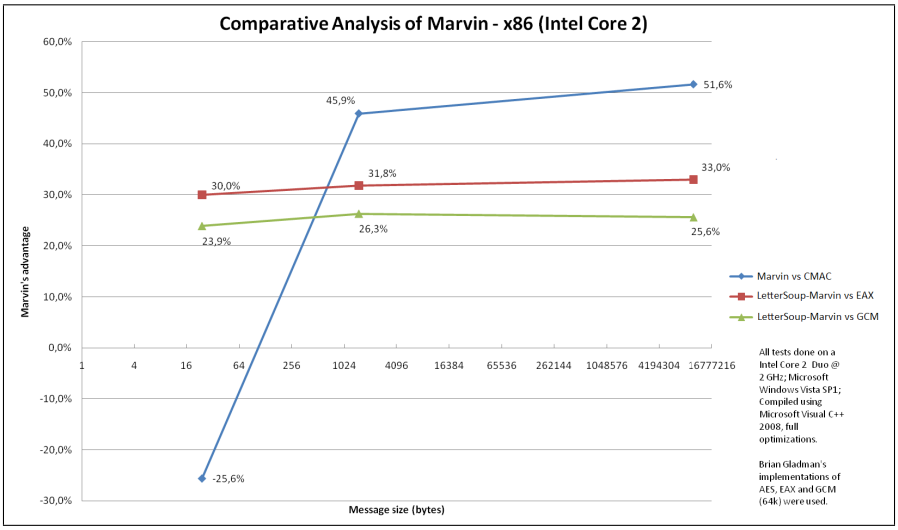

Figure 3. Marvin performance on 32-bit platform

We note that Marvin is slower than CMAC for very short messages, which is due to Marvin's offset multiplication overhead: for short messages, e.g. 24 bytes, Marvin performs 2 full encryptions and 2 square-complete transforms, which adds up to 28 unkeyed AES rounds. On the other hand, CMAC performs 3 full encryptions, or 30 full rounds. In this scenario, the overhead introduced by the offset multiplication (which enable it to be parallelizable) become more visible. As shown in Figure 4, the break-even point for the Marvin vs. CMAC comparison is achieved around 80 bytes. As the difference is at most a few milliseconds, it should not be considered a problem in most 32-bit platforms. We note two performance peaks, for 80 and 160-byte messages, which are explained by the lack of padding in the measured points, since they are multiple of the cipher block size.

On the other hand, Marvin's performance in 32-bit platforms is of much higher interest when we consider larger messages. On a preliminary calculation, where only the number of rounds of the underlying cipher is taken into account, Marvin's advantage over CMAC should be $10/10 - 4/10 = 60\%$ as presented in Table 2. In practice, we have achieved

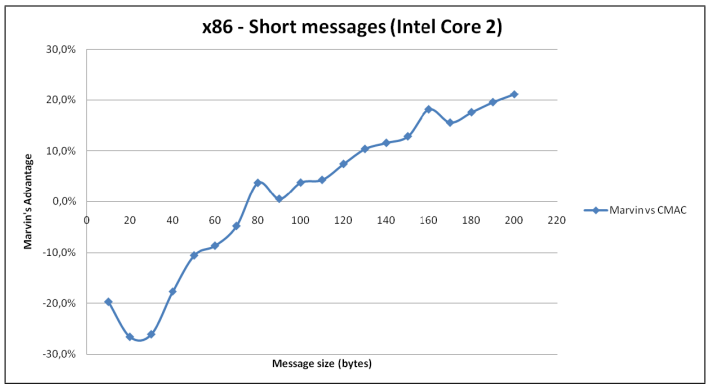

FIGURE 4. Detail of MARVIN vs. CMAC on 32-bit platform, for short messages

52% in our tests. Furthermore, using LETTERSOUP we were able to achieve a substantial advantage compared to both EAX and GCM (using 64KiB tables), for all message sizes.

We point out that, in the present benchmark, we did not take advantage of MARVIN's intrinsic parallelism. For short messages this approach would most probably not be the best choice, since the overhead involved in creating a thread would overshadow the performance improvement from the concurrent processing; nonetheless, for large messages (e.g. 10MiB), a significant performance enhancement could be achieved.

8.2. **8-bit platform: Avrora Sensor Simulator.** As testbed, we adopted the Avrora Simulator version 1.6.0 - Beta [29], simulating a microcontroller from the ATmega128 [2] series; as recommended by Avrora documentation, we adopted *avr-objdump* and *avr-gcc* (both GNU utilities) as compilation tools. We also developed CMAC, AEX and MARVIN/LETTERSOUP implementations, aiming at investigating the effect of size- and speed-optimizations in the comparison. The underlying AES code is the C implementation originally developed by Martins and Barreto [17], which is tailored to constrained platforms.

The results of our tests are presented on Table 4, which shows that MARVIN has a slight advantage over CMAC in terms of performance, but has a considerably bigger code size. Based on these results, our recommendation is for an implementation which does not use tables and is speed-optimized.

We have also tested LETTERSOUP and compared its results with EAX. Due to the reduced memory available on the platform, all tests were performed without using precomputed tables. The results, presented in Table 5, show that LETTERSOUP outperforms EAX-CMAC by 29-28% in all tested scenarios. Thus, our recommendation is for a size-optimized implementation.

9. CONCLUSIONS

We have presented MARVIN, a new parallelizable message authentication code based on the ALRED family, and LETTERSOUP, an AEAD scheme that explores some interesting properties of LFSRC block-cipher mode of operation. Both algorithms are interesting when one needs platform flexibility: on the one hand, they need about 25%–40% of a block cipher call per processed block and may be implemented using a reduced space in memory, interesting features in constrained scenarios such as mobile and sensor networks; on the other hand, they can take advantage of scenarios where higly parallel systems are available and can be further optimized using extra memory.

Also, we analyzed the security and provided a benchmark comparing their performance with other important algorithms both on limited and resourceful platforms. Our security analysis is not completely developed, but presents the essential elements that support the security of the proposed schemes, providing the basis for a formal security proof. Our preliminary benchmark shows that MARVIN is an attractive alternative to CMAC when one needs to authenticate fairly big messages in 32-bit platforms; for more constrained platforms, MARVIN has a bigger code size than CMAC, but is an advantageous algorithm in terms of processing speed for all message sizes. Finally, LETTERSOUP benchmark shows that the algorithm is an interesting option both in resource-constrained and in powerful platforms.

10. FUTURE AND ONGOING WORK

We are currently evaluating the performance of MARVIN and LETTERSOUP with other underlying ciphers, especially those of the CURUPIRA [3, 27] family, which were developed with constrained platforms in mind. In fact, the adoption of the CURUPIRA-2 [27] should considerably decrease the amount of code needed by the algorithms proposed here, as they present similar instructions in their structure.

REFERENCES

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Completely Unexpected Productions Ltd, 1979.

[2] Atmel. *AVR 8-Bit RISC processor - ATmega128 e ATmega128L*, 2007.

[3] P. Barreto and M. Simplicio. CURUPIRA, a block cipher for constrained platforms. In *Anais do 25º Simpsio Brasileiro de Redes de Computadores e Sistemas Distribudos - SBRC 2007*, volume 1, pages 61–74, Belm, Brazil, 2007. SBC.

[4] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology – Eurocrypt'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192, Heidelberg, Germany, 1997. Springer.

[5] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation: A two-pass authenticated-encryption scheme optimized for simplicity and efficiency. In *Fast Software Encryption 2004*, pages 389–407, February 2004.

[6] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology – Eurocrypt'2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397, Heidelberg, Germany, 2002. Springer.

[7] J. Black and P. Rogaway. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems – CHES'2007*, Lecture Notes in Computer Science, Heidelberg, Germany, 2007. Springer. To appear.

[8] J. Daemen, L. R. Knudsen, and V. Rijmen. The block cipher SQUARE. In *Fast Software Encryption – FSE'97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165, Heidelberg, Germany, 1997. Springer.

[9] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. The NOEKEON block cipher. In *First open NESSIE Workshop*, pages 384–397, Leuven, Belgium, November 2000. NESSIE Consortium.

[10] J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, Heidelberg, Germany, 2002.

[11] J. Daemen and V. Rijmen. A new MAC construction Alred and a specific instance Alpha-MAC. In *Fast Software Encryption – FSE'2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17, Heidelberg, Germany, 2005. Springer.

[12] J. Daemen and V. Rijmen. The Pelican MAC function. Cryptology ePrint Archive, Report 2005/088, 2005. `http://eprint.iacr.org/2005/088`.

[13] B. Gladman. Aes and combined encryption/authentication modes. http://fp.gladman.plus.com/AES/, 2008.

[14] IEEE. *Standard Specifications for Public-Key Cryptography – IEEE Std 1363:2000*. IEEE P1363 Working Group, 2000.

[15] T. Iwata and K. Kurosawa. OMAC: One-key CBC MAC. In *Fast Software Encryption – FSE'2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153, Heidelberg, Germany, 2003. Springer.

[16] H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology – Crypto'94*, volume 839 of *Lecture Notes in Computer Science*, pages 129–139, Heidelberg, Germany, 1994. Springer.

[17] G. Y. Martins. Projeto de um dispositivo de autenticao e assinatura. Master's thesis, Escola Politcnica da Universidade de So Paulo, 2007.
[18] D. McGrew and J. Viega. The galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process, January 2004.
[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, USA, 1999.
[20] NIST. *Federal Information Processing Standard (FIPS 197) – Advanced Encryption Standard (AES)*. National Institute of Standards and Technology – NIST, November 2001.
[21] NIST. *Special Publication 800-38B – Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. National Institute of Standards and Technology, May 2005. `http://csrc.nist.gov/CryptoToolkit/modes/800-38_Series_Publications/SP800-38B.pdf`.
[22] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The cipher Shark. In *Fast Software Encryption – FSE'96*, volume 1039 of *Lecture Notes in Computer Science*, pages 99–111, Heidelberg, Germany, 1996. Springer.
[23] P. Rogaway. Authenticated-encryption with associated-data. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107, 2002.
[24] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology – Asiacrypt'2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31, Heidelberg, Germany, 2004. Springer.
[25] R. Schroeppel. Comment for AES cipher selection. Email Comments on Block Cipher Modes of Operation (NIST document), 2000. `http://csrc.nist.gov/CryptoToolkit/modes/workshop1/email-comments.pdf`.
[26] R. Schroeppel, W. E. Anderson, C. L. Beaver, T. J. Draelos, and M. D. Torgerson. Cipher-state (CS) mode of operation for AES. Submission to NIST's Computer Security Resource Center (CSRC) forum on modes of operation for symmetric key block ciphers, 2004. `csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/cs/cs-spec.pdf`.
[27] M. Simplicio, P. Barreto, T. Carvalho, C. Margi, and M. Näslund. The Curupira-2 block cipher for constrained platforms: Specification and benchmarking. In *Proceedings of the 1st International Workshop on Privacy in Location-Based Applications - 13th European Symposium on Research in Computer Security (ESORICS'2008)*, volume 397, Malaga, Spain, 2008. CEUR-WS.
[28] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC Press, Boca Raton, USA, 2nd edition, 2002.
[29] B. Titzer, D. Lee, and J. Palsberg. Avrora scalable simulation of sensor networks with precise timing. Center for Embedded Network Sensing Posters - Paper 93, 2004.
[30] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

11. Tables

Table 1. Probability to set *bad* in game **R1**. The arrows indicate how many items are added to the set $\mathcal{R}$

#query	$\lvert\mathcal{R}\rvert$	Pr(Nonce) $\rightarrow +1$	Pr(Offset) $\rightarrow +\sigma_i$	Pr(Ac. Collision) $\rightarrow +1$
$i = 1$	1	$1/(2^n)$	$(2\sigma_1)/(2^n)$	$(2+\sigma_1)/(2^n)$
$i = 2$	$\sigma_1 + 3$	$(\sigma_1+3)/(2^n)$	$(\sigma_1+4)\sigma_2/(2^n)$	$(\sigma_1+4+\sigma_2)/(2^n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i = q$	$1+\sum_{j=1}^{q-1}\sigma_j+2$	$(\lvert\mathcal{R}\rvert)/(2^n)$	$(\lvert\mathcal{R}\rvert+1)\sigma_i/(2^n)$	$(\lvert\mathcal{R}\rvert+1+\sigma_i)/(2^n)$

12. Appendix: Incremental generation of offsets

The Marvin offsets are incrementally defined as $O_0 \leftarrow R$, $O_{i+1} \leftarrow O_i \cdot x^w$. Let $\mathrm{GF}(2^n) = \mathrm{GF}(2)/p(x)$ where $p(x)$ is a primitive pentanomial over GF(2) such that x^w is a primitive

TABLE 2. Efficiency comparison between some MACs

MAC	Parallelizable?	Encryptions/block	Storage (blocks)
CMAC	No	1	$O(1)$
PELICAN	No	≈ 0.25–0.4	$O(1)$
GHASH (LUTs)	Yes	≈ 0.1–0.25	$O(2^w b)$
GHASH (plain)	Yes	≈ 1	$O(1)$
MARVIN	Yes	≈ 0.25–0.4	$O(1)$

TABLE 3. Features and Efficiency of AEAD modes

Mode	Parallelizable	Input order	MAC cost	Extra storage	Encryption only
EAX-CMAC	No	Free	1	No	Yes
GCM	Yes	Partially	≈ 0.1–0.25	Yes	Yes
AEM	Yes	Free	1	No	Yes
CS	Yes	Free	≈ 0.1	No	No
EAX-MARVIN	Yes	Free	≈ 0.25–0.4	No	Yes
LETTERSOUP	Yes	Free	≈ 0.25–0.4	No	Yes

TABLE 4. MARVIN performance on the 8-bit platform

MARVIN	Cycles	Adv. CMAC	Code Size	Adv. CMAC
Using tables, speed-optimized	44414	4.6%	1412	−192.9%
No tables, speed-optimized	44433	4.5%	942	−95.4%
Using tables, size-optimized	45302	2.7%	1322	−174.3%
No tables, size-optimized	45334	2.6%	846	−75.5%

TABLE 5. LETTERSOUP performance on the 8-bit platform

LETTERSOUP	Cycles	Adv. EAX	Code Size	Adv. EAX
Speed-optimized	76907	29.2%	2098	−24.7%
Compromise	77972	28.2%	1662	1.2%
Size-optimized	78099	28.1%	1364	18.9%

root of $p(x)$. We choose a pentanomial basis representation because primitive pentanomials are available for all n of practical interest [14]; primitive trinomials would result in a better performance but are seldom available, and do not exist at all in the important case $8 \mid n$ [19]). The motivation for a multiplication by x^w instead of x is the result of the following theorem:

Theorem 4. *Let $p(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1$ be a primitive pentanomial of degree $n = bw$ over* GF(2) *such that $k_3 > k_2 > k_1$, $k_3 - k_1 \leqslant w$, and either $w \mid k_3$ or $w \mid k_1$. Then multiplication by x^w in $\mathrm{GF}(2^n) = \mathrm{GF}(2)[x]/p(x)$ can be implemented with no more than 5 XORs and 4 shifts on w-bit words. Moreover, if 2×2^w bytes of storage are available, the cost drops to no more than 2 XORs on w-bit words and 2 table lookups.*

Proof. For $u = \bigoplus_{d=0}^{n-1} u_d x^d \in \mathrm{GF}(2^n)$ let $U_i = u_{wi+w-1} x^{w-1} + \cdots + u_{wi}$, $i = 0, \ldots, b-1$, so that $u = U_{b-1} x^{w(b-1)} + U_{b-2} x^{w(b-2)} + \cdots + U_0$, which for brevity we write $u = (U_{b-1}, \ldots, U_0)$.

Then one can compute $u \cdot x^w$ as:

$$\begin{aligned}
&(U_{b-1}x^{w(b-1)} + U_{b-2}x^{w(b-2)} + \cdots + U_0) \cdot x^w = \\
&U_{b-1}x^n + U_{b-2}x^{w(b-1)} + \cdots + U_0 x^w = \\
&U_{b-2}x^{w(b-1)} + \cdots + U_0 x^w + U_{b-1}(x^{k_3} + x^{k_2} + x^{k_1} + 1) = \\
&(U_{b-2}, \ldots, U_0, U_{b-1}) \oplus U_{b-1}(x^{k_3} + x^{k_2} + x^{k_1}).
\end{aligned}$$

Assume that $k_1 = wk$ for some k; the case $w \mid k_3$ is handled analogously. Thus:

$$u \cdot x^w = (U_{b-2}, \ldots, U_0, U_{b-1}) \oplus U_{b-1}(x^{k_3-k_1} + x^{k_2-k_1} + 1)x^{wk}.$$

Since $\deg(U_{b-1}) \leqslant w - 1$ and $\deg(x^{k_3-k_1} + x^{k_2-k_1} + 1) \leqslant w$, their product is a polynomial of degree not exceeding $2w - 1$, and hence it fits two w-bit words for any value of U_{b-1}. Besides, multiplication of this value by x^{wk} corresponds to simply displacing it k words to the left. Let

$$\begin{aligned}
T_1[U] &\equiv (U \gg (w - (k_3 - k_1))) \oplus (U \gg (w - (k_2 - k_1))), \\
T_0[U] &\equiv (U \ll (k_3 - k_1)) \oplus (U \ll (k_2 - k_1)) \oplus U;
\end{aligned}$$

these values can be either computed on demand or else precomputed and stored in two 2^w-entry tables. Then $u \cdot x^w = (U_{b-2}, \ldots, U_k \oplus T_1[U_{b-1}], U_{k-1} \oplus T_0[U_{b-1}], \ldots, U_0, U_{b-1})$. One easily sees by direct inspection that the computational cost is that stated by the theorem. □

As an illustration of this method, the following polynomials are especially well suited for implementation according to the above theorem, tailored for commonplace block sizes and data organized in 8-bit bytes:

- $x^{64} + x^8 + x^7 + x^5 + 1$:

  $(U_7, \ldots, U_0) \cdot x^8 = (U_6, \ldots, U_1, U_0 \oplus T_1[U_7], T_0[U_7])$, where
  $T_1[U] \equiv U \oplus (U \gg 1) \oplus (U \gg 3)$,
  $T_0[U] \equiv U \oplus (U \ll 7) \oplus (U \ll 5)$.

- $x^{96} + x^{16} + x^{13} + x^{11} + 1$:

  $(U_{11}, \ldots, U_0) \cdot x^8 = (U_{10}, \ldots, U_1 \oplus T_1[U_{11}], U_0 \oplus T_0[U_{11}], U_{11})$, where
  $T_1[U] \equiv U \oplus (U \gg 3) \oplus (U \gg 5)$,
  $T_0[U] \equiv (U \ll 5) \oplus (U \ll 3)$.

- $x^{128} + x^{29} + x^{27} + x^{24} + 1$:

  $(U_{15}, \ldots, U_0) \cdot x^8 = (U_{14}, \ldots, U_3 \oplus T_1[U_{15}], U_2 \oplus T_0[U_{15}], U_1, U_0, U_{15})$, where
  $T_1[U] \equiv (U \gg 3) \oplus (U \gg 5)$,
  $T_0[U] \equiv (U \ll 5) \oplus (U \ll 3) \oplus U$.

In fact, the circular byte permutation does not need to be effectively implemented: the same effect can be achieved if we keep track of the index i_{msb} corresponding to the most significant byte of U. Thus, if we use the i_{msb}-th byte as the first byte of U in every calculation, it suffices to update i_{msb} after each multiplication by x^w. Using this strategy, word permutations within a block can be performed essentially for free.

We point out that these observations do not impair implementation on platforms that are not byte-oriented. For instance, if a block fits one machine word, then clearly multiplication by x^w incurs only 1 rotation, 1 AND, 3 shifts, and 3 XORs, as illustrated by algorithm 6. Then again, we can improve the computation performance by means of a single 2^w-word table $T_w[U] = (U \ll k_3) \oplus (U \ll k_2) \oplus (U \ll k_1)$

Algorithm 6 Wordwise multiplication by x^w in GF(2^n)

INPUT: V ▷ n-bit word to multiply by x^w.
OUTPUT: $V \cdot x^w$.
1: $V \leftarrow V$ **rotl** w ; $R \leftarrow V$ & **bin**$(2^w - 1)$
2: $V \leftarrow V \oplus (R \ll k_3) \oplus (R \ll k_2) \oplus (R \ll k_1)$
3: **return** V

For the sake of comparison, algorithm 7 describes bytewise multiplication by x in GF(2^n), assuming that $\deg(p(x) - x^n) < 8$ (so that the mask Z computed at the beginning of the algorithm fits one byte). One sees that the cost is b XORs, $2b - 1$ shifts, and one bit test. In other words, the cost of multiplying by x is proportional to the block size, while that of multiplying by x^w is constant.

Algorithm 7 Bytewise multiplication by x in GF(2^n)

INPUT: B ▷ n-bit word to multiply by x.
OUTPUT: $B \cdot x$.
1: **if** $testbit(B_{b-1}, 7) = 1$ **then**
2: $Z \leftarrow p(x) - x^n$
3: **else**
4: $Z \leftarrow 0$
5: **end if**
6: **for** $i \leftarrow b - 1$ **downto** 1 **do**
7: $B_i \leftarrow (B_i \ll 1) \oplus (B_{i-1} \gg 7)$
8: **end for**
9: $B_0 \leftarrow (B_0 \ll 1) \oplus Z$
10: **return** B

13. Appendix: Other Algorithms

Algorithm 8 Linear feedback shift register counter (LFSRC) encryption-only mode.

INPUT: N ▷ nonce, an integer value in range $0 < N < 2^n$.
INPUT: K ▷ cipher key.
INPUT: M ▷ message to encrypt.
OUTPUT: C ▷ ciphertext of M under K.
1: Partition the message as $M = M_1 \| \ldots \| M_t$, where $|M_i| = n$ for $i = 1, \ldots, t - 1$, and $0 \leqslant |M_t| \leqslant n$ with $|M_t| = 0$ iff $|M| = 0$.
2: **for** $i \leftarrow 1$ **to** t **do** ▷ incrementally or in parallel
3: $O_i \leftarrow N \cdot (x^w)^i$; $C_i \leftarrow M_i \oplus E_K(O_i)[|M_i|]$
4: **end for**
5: **return** $C \leftarrow C_1 \| \ldots \| C_t$

Departamento de Engenharia de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, Brazil. E-mail: {mjunior,paquino,pbarreto,carvalho,cbmargi}@larc.usp.br