Twisted Polynomials and Forgery Attacks on GCM

Mohamed Ahmed Abdelraheem, Peter Beelen, Andrey Bogdanov, and Elmar Tischhauser *

Department of Mathematics and Computer Science
Technical University of Denmark
{mohab,pabe,anbog,ewti}@dtu.dk

Abstract. Polynomial hashing as an instantiation of universal hashing is a widely employed method for the construction of MACs and authenticated encryption (AE) schemes, the ubiquitous GCM being a prominent example. It is also used in recent AE proposals within the CAESAR competition which aim at providing nonce misuse resistance, such as POET. The algebraic structure of polynomial hashing has given rise to security concerns: At CRYPTO 2008, Handschuh and Preneel describe key recovery attacks, and at FSE 2013, Procter and Cid provide a comprehensive framework for forgery attacks. Both approaches rely heavily on the ability to construct *forgery polynomials* having disjoint sets of roots, with many roots ("weak keys") each. Constructing such polynomials beyond naïve approaches is crucial for these attacks, but still an open problem.
In this paper, we comprehensively address this issue. We propose to use *twisted polynomials* from Ore rings as forgery polynomials. We show how to construct sparse forgery polynomials with full control over the sets of roots. We also achieve complete and explicit disjoint coverage of the key space by these polynomials. We furthermore leverage this new construction in an improved key recovery algorithm.
As cryptanalytic applications of our twisted polynomials, we develop the first universal forgery attacks on GCM in the weak-key model that do not require nonce reuse. Moreover, we present universal weak-key forgery attacks for the recently proposed nonce-misuse resistant AE schemes POET, Julius, and COBRA.

Keywords: Authenticated encryption, polynomial hashing, twisted polynomial ring (Ore ring), weak keys, GCM, POET, Julius, COBRA

1 Introduction

Authenticated encryption (AE) schemes are symmetric cryptographic primitives combining the security goals of confidentiality and integrity.

Providing both ciphertext and an authentication tag on input of a plaintext message, they allow two parties sharing a secret key to exchange messages in privacy and with the assurance that they have not been tampered with.

Approaches to construct AE schemes range from generic composition of a symmetric block or stream cipher for confidentiality and a message authentication code (MAC) for integrity to dedicated designs. An important method for constructing both stand-alone MACs and the authentication tag generation part of dedicated AE algorithms is based on universal hash functions, typically following the Carter-Wegman paradigm [21]. This construction enjoys information-theoretic security and is usually instantiated by *polynomial hashing*, that is, the evaluation of a polynomial in H (the authentication key) over a finite field with the message blocks as coefficients.

One of the most widely adopted AE schemes is the Galois Counter Mode (GCM) [6], which has been integrated into important protocols such as TLS, SSH and IPsec; and furthermore has been standardized by among others NIST and ISO/IEC. It combines a 128-bit block cipher in CTR mode of operation for encryption with a polynomial hash in $\mathbb{F}_2^{128}$ over the ciphertexts to generate an authentication tag. The security of GCM relies crucially on the uniqueness of its nonce parameter [7, 10, 11].

As a field, authenticated encryption has recently become a major focus of the cryptographic community due to the ongoing CAESAR competition for a portfolio of recommended AE algorithms [1]. A large number of diverse designs has been submitted to this competition, and a number of the submissions feature polynomial hashing as part of their authentication functionality. Among these, the new AE schemes POET [2], Julius [5] and COBRA [4] feature stronger security claims about preserving confidentiality and/or integrity under nonce reuse (so-called *nonce misuse resistance* [12]).

Background. The usual method to build a MAC or the authentication component of an AE scheme from universal hash functions is to use *polynomial hashing*, in other words, to evaluate a polynomial in the authentication key with the message or ciphertext blocks as coefficients:

Definition 1 (Polynomial-based Authentication Scheme). *A polynomial hash-based authentication scheme processes an input consisting of a key H and plaintext/ciphertext $M = (M_1||M_2||\cdots||M_l)$, where each*

$M_i \in \mathbb{F}_2^n$, *by evaluating the polynomial*

$$h_H(M) := \sum_{i=1}^{l} M_i H^i \in \mathbb{F}_2^n.$$

To produce an authentication tag, the value $h_H(M)$ is often processed further, for example by encryption, or additive combination with another pseudorandom function. For a survey of existing constructions, we refer the reader to [10]. Out of these schemes, GCM [6, 13] is by far the most important and widespread algorithm. We therefore recapitulate existing security results about polynomial hashing at the example of GCM.

The Galois Counter Mode. GCM is defined as follows. It takes as input the plaintext $M = M_1||M_2||\cdots||M_l$, a key k and a nonce N. It outputs corresponding ciphertext $C = C_1||C_2||\cdots||C_l$ and an authentication tag T. The ciphertext blocks are generated using a block cipher E_k (usually AES) in counter mode: $C_i = E_k(J_i) \oplus M_i$, with J_0 an initial counter value derived from N, and the $J_1, J_2, \ldots$ successive increments of J_0. The ciphertexts are then processed with polynomial hashing to generate the tag

$$T = E_k(J_0) \oplus h_H(C)$$

with $H = E_k(0)$ as the authentication (hash) key. GCM is typically instantiated with a 128-bit block cipher, uses 128-bit keys and 96-bit nonces and produces 128-bit tags.

Joux' "forbidden" attack. Soon after the proposal of GCM, Joux [11] pointed out that the security of GCM breaks down completely if nonces are re-used with the same key. Since GCM is built upon the assumption of nonce uniqueness, his attack is referred to as the "forbidden" attack against GCM. It recovers the hashing key H using pairs of different messages M and M' that are authenticated using the same nonce N. This leads to the following equation in one unknown H:

$$T \oplus T' = h_H(C) \oplus E_K(N) \oplus h_H(C') \oplus E_K(N) = h_H(C \oplus C'),$$

where C/C' and T/T' are the ciphertext/tag of M/M'. This is equivalent to saying that the polynomial $T \oplus T' \oplus h_H(C \oplus C')$ has a root at H. By using multiple message pairs and computing the GCD of the arising polynomials, H can be uniquely identified. This attack does not apply to the nonce-respecting adversarial model.

Ferguson's Short Tag attacks. While Joux' attack establishes GCM's sensitivity to nonce reuse, Ferguson [7] demonstrated that truncation of its output to shorter tags of $s < 128$ bits not only (generically) limits its authentication security level to $s/2$ bits, but also allows a key recovery attack with little more than $2^{s/2}$ queries, which especially does not require a collision on the full 128-bit polynomial hash. Ferguson's attacks make use of so-called *error polynomials*

$$\sum_{i=1}^{l}(C_i - C'_i)H^i,$$

with the C_i the original and the C'_i the modified ciphertext blocks. Since GCM operates in a field or characteristic two, squaring is a linear operation, and this allows Ferguson to consider *linearized error polynomials*, i.e. where only the coefficients of H^{2^i} are nonzero. The effect of these modifications on the first bits of the (truncated) authentication tag is then a linear function of H and the coefficients. Using linear algebra, the coefficients of the error polynomial are then computed such that the first $s/2$ bits of the shortened tag will not change. The attack then exploits a generic birthday-type collision on the remaining $s/2$ tag bits to obtain a complete collision on the short tag. A small number of further forgeries then yield enough linear relations about bits of H to allow its complete recovery. Note that this attack does not require nonce reuse.

Handschuh and Preneel's Key Recovery Attacks. Handschuh and Preneel [10] propose various methods for recovering the hash key of polynomial hashing-based MACs, among them GCM. The main idea is to obtain a valid ciphertext-tag pair C, T and then to attempt verification with a different message C' but the same tag; here C' is chosen such that $C - C'$ has many distinct roots. If verification is not successful, another C'' is used which is chosen such that $C - C''$ has no roots in common with $C - C'$, and so on. Once a verification succeeds, this indicates that the authentication key is among the roots of this polynomial. Further queries can then be made to subsequently reduce the search space until the key is identified. When using polynomials of degree d in each step, the total number of verification queries needed is $2^n/d$. Knowing the authentication key then allows the adversary to produce forgeries for any given combination of nonce and corresponding ciphertext blocks. The attack of [10] does not require nonce reuse, however is limited to ciphertexts as it does not allow the adversary to create universal forgeries for any desired *plaintext message*.

Handschuh and Preneel further identify the key $H = 0$ as a trivially weak key for GCM-like authentication schemes. They further provide a formalization of the concept of weak keys, namely a class D of keys is called *weak* if membership in this class requires less than $|D|$ key tests and verification queries.

Saarinen's Cycling Weak Key Forgery Attacks. This concept of weak keys for polynomial authentication was taken a step further by Saarinen in [20], where a forgery attack for GCM is described for the case where the order of the hash key H in $\mathbb{F}^{\times}_{2^{128}}$ is small. If the hash key belongs to a cyclic subgroup of order t, i.e. $H^{t+1} = H$, then the attacker can create a blind forgery by simply swapping any two ciphertext blocks C_i and C_{i+jt}. Such hash keys with short cycles (small value of t) can be labelled as *weak keys*. In other words, Saarinen identifies all elements with less than maximal order in $\mathbb{F}^{\times}_{2^{128}}$ as weak keys. Since constructing a corresponding forgery requires a message length of at least 2^t blocks, and GCM limits the message to 2^{32} blocks, this means that all keys with order less than 2^{32} are weak keys for GCM. We finally note that cycling attacks depend on the factorisation of $2^n - 1$, since any subgroup order is a divisor of the order of $\mathbb{F}^{\times}_{2^{128}}$.

Procter and Cid's General Weak-Key Forgery Framework. The idea behind cycling attacks was extended and formalized by Procter and Cid [15] by introducing the notion of so-called *forgery polynomials*: Let H be the (unknown) hash key. A polynomial $q(X) = \sum_{i=1}^{l} q_i X^i$ is then called a forgery polynomial if it has H as a root, i.e. $q(H) = 0$. This designation is explained by noting that for $C = (C_1||C_2||\cdots||C_l)$ and writing $Q = q_1||\cdots||q_l$, we have

$$h_H(C) = h_H(C + Q),$$

that is, adding the coefficients of q yields the same authentication tag, i.e. a forgery.[1] More concretely, for GCM, we have that $(N, C+Q, T)$ is a forgery for (N, C, T) whenever $q(H) = 0$. This also means that all roots of q can be considered weak keys in the sense of [10]. In order to obtain forgeries with high probability, Procter and Cid note that a concrete choice for q should have a high degree and preferably no repeated roots.

Since any choice of q is a forgery polynomial for its roots as the key, Procter and Cid establish the interesting fact that *any* set of keys in

[1] Note that forgery polynomials are conceptually different from Ferguson's error polynomials, since the authentication key H typically is not a root of an error polynomial, while this is the defining property for forgery polynomials.

polynomial hashing can be considered weak: membership to a weak key class D can namely be tested by one or two verification queries using the forgery polynomial $q(X) = \prod_{d \in D}(X - d)$ regardless of the size of D. They also note that such a forgery polynomial can be combined with the key recovery technique of [10], namely by using the polynomial $q(X) = \prod_{H \in \mathbb{F}_2^n, H_n = 0}(X - H)$ and then subsequently fixing more bits of H according to the results of the verification queries. This only requires two queries for a first forgery, and at most $n + 1$ for complete key recvoery. Note however that this requires messages lengths up to 2^n blocks, which is clearly infeasible for GCM (where $n = 128$).

We also note that all previously described attacks can be seen as special cases of Procter and Cid's general forgery framework [15, 16].

Our problem. We start by noting that besides the attacks of Joux and Ferguson, which apply to the special cases where the nonce is reused or tags are truncated, only Saarinen's cycling attack gives a concrete security result on GCM and similar authentication schemes. In the formalism of [15], it uses the forgery polynomials $X^t - X$ with $t < 2^{32}$ the subgroup order. To the best of our knowledge, no other explicit forgery polynomials have been devised. In [15], two generic classes of forgery polynomials are discussed: random polynomials of degree d in $\mathbb{F}_{2^n}[X]$ or naïve multiplication of linear factors $(x - H_1) \cdots (x - H_d)$. The latter construction requires d multiplications already for the *construction* of the forgery polynomial, which quickly becomes impractical. We also note that in both cases, the coefficients will be "dense", i.e. almost all of them will be nonzero. This means that all of the ciphertext blocks have to be modified by the adversary to submit each verification query. In the same sense, the observation of [15] that any key is weak is essentially a certificational result only since $|D|$ multiplications are needed to produce q for a weak key class of size $|D|$. The construction of explicit forgery polynomials is left as an important open problem in [15].

Similarly, the key recovery technique of [10] does not deal with the important question of how to construct new polynomials of degree d having distinct roots from all previously chosen ones, especially without the need to store all d roots from each of the $2^n/d$ iterations. These observations lead to the following questions:

> *Can we efficiently construct* explicit *forgery polynomials having prescribed sets of roots, ideally having few nonzero coefficients? Moreover, can we disjointly cover the entire key space using these explicit forgery polynomials?*

Answers to these questions would essentially solve the open problem mentioned in [15], and also make the observation concrete that *any* key in polynomial hashing can be considered weak. It would also improve the key recovery algorithm of Handschuh and Preneel [10]. On the application side, we ask whether *plaintext-universal forgeries* for GCM can be constructed in the *nonce-respecting adversarial model*.

Our results. In this paper, we answer the above-mentioned questions in the affirmative. We comprehensively address the issue of polynomial construction and selection in forgery and key recovery attacks on authentication and AE schemes based on polynomial hashing. In detail, the contributions of this paper are as follows.

Explicit construction of sparse forgery polynomials. In contrast to the existing generic methods to construct forgery polynomials, we propose a construction based on so-called twisted polynomial rings that allows us to explicitly describe polynomials of degree 2^d in any finite field $\mathbb{F}_2^n$ which have as roots precisely the elements of an arbitrary d-dimensional subspace of $\mathbb{F}_2^n$, independent of n or the factorisation of $2^n - 1$. While achieving this, our polynomials are very sparse, having at most $d + 1$ nonzero coefficients.

Complete disjoint coverage of the key space by forgery polynomials. In order to recover the authentication key (as opposed to blind forgeries), the attacks of Handschuh and Preneel [10] and Procter and Cid [15] need to construct polynomials having a certain set of roots, being disjoint from the roots of all previous polynomials. We propose an explicit algebraic construction achieving the partitioning of the whole key space $\mathbb{F}_2^n$ into roots of structured and sparse polynomials. This substantiates the certificational observation of [15] that any key is weak, in a concrete way. We give an informal overview of our construction of twisted forgery polynomials in the following proposition.

Proposition (informal). *Let $q = r^e$ and let V be a subspace of $\mathbb{F}_q$ of over the field $\mathbb{F}_r$ of dimension d. Then there exists a twisted polynomial ϕ from the Ore ring $\mathbb{F}_q\{\tau\}$ with the following properties:*

1. *ϕ can be written as $\phi(X) = c_0 + \sum_{i=1}^{d} c_i X^{2^i}$, i.e. ϕ has at most $d+1$ nonzero coefficients;*
2. *For any $a \in \mathbb{F}_q$, the polynomial $\phi(X) - \phi(a)$ has exactly $a + V$ as set of roots;*

3. *The sets of roots of the polynomials $\phi(X) - b$ with $b \in \operatorname{Im}\phi$ partition $\mathbb{F}_q$.*

Improved key recovery algorithm. We then leverage the construction of sparse forgery polynomials from the twisted polynomial ring to propose an improved key recovery algorithm, which exploits the particular structure of the root spaces of our forgery polynomials. In contrast to the key recovery techniques of [10] or [15], it only requires the modification of a logarithmic number of message blocks in each iteration (i.e., d blocks for a 2^d-block message). It also allows arbitrary trade-offs between message lengths and number of queries.

New universal forgery attacks on GCM. Turning to applications, we develop the first universal forgery attacks on GCM in the weak-key model that do not require nonce reuse. We first use tailored twisted forgery polynomials to recover the authentication key. Depending on the length of the nonce, we then either use a sliding technique on the counter encryptions or exploit an interaction between the processing of different nonce lengths to obtain valid ciphertext-tag pairs for any given combination of nonce and plaintext.

Analysis of POET, Julius, and COBRA. Using our framework, we finally present further universal forgery attacks in the weak-key model also for the recently proposed nonce-misuse resistant AE schemes POET, Julius, and COBRA.

Our results on POET prompted the designers to formally withdraw the variant with finite field multiplications as universal hashing from the CAESAR competition. Previously, an error in an earlier specification of POET had been exploited for constant-time blind forgeries [9]. This attack however does not apply to the corrected specification of POET. Likewise, for COBRA, a previous efficient attack by Nandi [14] does not yield universal forgeries.

Organization. The remainder of the paper is organized as follows. We introduce some common notation in Sect. 2. In Sect. 3, we describe our method to construct explicit and sparse forgery polynomials. Sect. 4 proposes two approaches to construct a set of explicit forgery polynomials whose roots partition the whole finite field $\mathbb{F}_2^{128}$. In Sect. 5, we describe our improved key recovery algorithm. In Sect. 6, two universal weak-key forgery attacks against GCM are presented. In Sect. 7, we present several universal forgery attacks on POET under the weak-key assumption. For

the attacks on Julius and COBRA, we refer to Appendix B and Appendix C respectively. We conclude in Sect. 8.

2 Preliminaries

Throughout the paper, we denote by $\mathbb{F}_{p^n}$ the finite field of order p^n and characteristic p, and write $\mathbb{F}_p^n$ for the corresponding n-dimensional vector space over $\mathbb{F}_p$. We use $+$ and $\oplus$ interchangeably to denote addition in $\mathbb{F}_{2^n}$ and $\mathbb{F}_2^n$.

Forgery polynomials. We formally define forgery polynomials [15] as polynomials $q(X) = \sum_{i=1}^{r} q_i X^i$ with the property that that $q(H) = 0$ for the authentication key H. Assume that $M = (M_1||M_2||\cdots||M_l)$ and that $l \leq r$. Then

$$h_H(M) = \sum_{i=1}^{r} M_i H^i = \sum_{i=1}^{l} M_i H^i + \sum_{i=1}^{r} q_i H^i = \sum_{i=1}^{r} (M_i + q_i) H^i = h_H(M+Q)$$

where $Q = q_1||\cdots||q_r$. If $l < r$, we simply pad M with zeros. Throughout the paper, we will refer to Q as the binary coefficient string of a forgery polynomial $q(X)$.

Using q as a forgery polynomial in a blind forgery gives a success probability $p = \frac{\#\text{roots of } q(X)}{2^n}$. Therefore, in order to have a forgery using the polynomial $q(X)$ with high probability, $q(X)$ should have a high degree and preferably no repeated roots.

In the next section, we will present methods to construct explicit sparse forgery polynomials $q(X)$ with distinct roots and high forgery probability.

3 Explicit construction of twisted forgery polynomials

When applying either the key recovery attack of [10] or any of the forgery or key recovery attacks of [15], a crucial issue lies in the selection of polynomials that have a certain number ℓ of roots in $\mathbb{F}_2^n$, and additionally being able to select each polynomial to have no common roots with the previous ones. Ideally, these polynomials should both be described by explicit constructive formulas, and they should be sparse, i.e. have few nonzero coefficients.

As noted in [15], the direct way to do this is to choose distinct elements $\alpha_1, \ldots, \alpha_\ell \in \mathbb{F}_{2^n}$ and to work out the product $(X-\alpha_1)\cdots(X-\alpha_\ell)$, which

quickly gets impractical for typical values of ℓ and will not result in sparse polynomials. The second suggestion described in [15] is to select them at random, which is efficient, but also does not produce sparse polynomials. Moreover, as noted in [16], subsequently chosen random polynomials will likely have common roots, which rules out the key recovery attacks of both [10] and [16].

The only proposed explicit construction of forgery polynomials so far are the polynomials $X^t - 1$ with $t|(2^{128} - 1)$, due to Saarinen [20]. Their roots correspond precisely to the cyclic subgroups of $\mathbb{F}_2^{128}$, which also limits their usefulness in the key recovery attacks.

In this section, we propose a new method which yields explicit constructions for polynomials with the desired number of roots. At the same time, the resulting polynomials are sparse in the sense that a polynomial with 2^d roots will have at most $d + 1$ nonzero coefficients.

For this, we use the fact that $\mathbb{F}_{2^{128}}$ can be seen as a vector space (of dimension 128) over $\mathbb{F}_2$. More precisely, given a subvector space V of $\mathbb{F}_{2^{128}}$ of dimension d with basis $\{b_1, \ldots, b_d\}$, we describe a fast procedure to find a polynomial $p_V(X) \in \mathbb{F}_{2^{128}}[X]$ whose roots are exactly all elements of V. Note that this implies that $\deg P_V(X) = 2^d$. We will also see that the $p_V(X)$ is sparse, more precisely that the only coefficients of $p_V(X)$ that may be distinct from zero are the coefficients of the monomials X^{2^i} with $0 \leq i \leq d$. In particular this will imply that $p_V(X)$ has at most $d + 1$ non-zero coefficients despite the fact that it has degree 2^d.

To explain the above, we introduce the concept of a twisted polynomial ring, also called an Ore ring.

Definition 2. *Let $\mathbb{F}_q$ be a field of characteristic p. The* twisted polynomial *or* Ore *ring $\mathbb{F}_q\{\tau\}$ is defined as the set of polynomials in the indeterminate τ having coefficients in $\mathbb{F}_q$ with the usual addition, but with multiplication defined by the relation $\tau\alpha = \alpha^p\tau$ for all $\alpha \in \mathbb{F}_q$.*

The precise ring we will need is the ring $\mathbb{F}_{2^{128}}\{\tau\}$. In other words, two polynomials in τ can be multiplied as usual, but when multiplying the indeterminate with a constant, the given relation applies. This makes the ring a non-commutative ring (see [8] for an overview of some of its properties). One of the reasons to study this ring is that it gives a convenient way to study linear maps from $\mathbb{F}_{2^{128}}$ to itself, when viewed as a vector space over $\mathbb{F}_2$. A constant $\alpha \in \mathbb{F}_{2^{128}}\{\tau\}$ then corresponds to the linear map sending $x \in \mathbb{F}_{2^{128}}$ to $\alpha \cdot x$, while the indeterminate τ corresponds to the linear map sending $x \in \mathbb{F}_{2^{128}}$ to x^2. Addition in the Ore ring corresponds to the usual addition of linear maps, while multiplication corresponds to

composition of linear maps. This explains the relation $\tau \cdot \alpha = \alpha^2 \cdot \tau$, since both expressions on the left and right of the equality sign correspond to the linear map sending x to $\alpha^2 x^2$. To any element ϕ from the Ore ring, we can associate a polynomial $\phi(X)$, by replacing τ^i with X^{2^i}. The resulting polynomials have possibly non-zero coefficients from $\mathbb{F}_{2^{128}}$ only for those monomials X^e, such that e is a power of 2. Such polynomials are called linearized and are just yet another way to describe linear maps from $\mathbb{F}_{2^{128}}$ to itself. The advantage of this description is that the null space of a linear map represented by a linearized polynomial $p(X)$ just consists of the roots of $p(X)$ in $\mathbb{F}_{2^{128}}$.

Now we describe how to find a polynomial $p_V(X)$ having precisely the elements of a subspace V of $\mathbb{F}_{2^{128}}$ as roots. The idea is to construct a linear map from $\mathbb{F}_{2^{128}}$ to itself having V as null space recursively. We will assume that we are given a basis $\{\beta_1, \ldots, \beta_d\}$ of V. For convenience we define V_i to be the subspace generated by $\{\beta_1, \ldots, \beta_i\}$. Note that $V_0 = \emptyset$ and $V_d = V$. Then we proceed recursively for $0 \leq i \leq d$ by constructing a linear map ϕ_i (expressed as an element of the Ore ring) with null space equal to V_i. For $i = 0$ we define $\phi_0 := 1$, while for $i > 0$ we define $\phi_i := (\tau + \phi_{i-1}(\beta_i))\phi_{i-1}$. For $d = 2$, we obtain for example

$$\phi_0 = 1, \ \phi_1 = \tau + \beta_1$$

and

$$\phi_2 = (\tau + (\beta_2^2 + \beta_1\beta_2))(\tau + \beta_1) = \tau^2 + (\beta_2^2 + \beta_1\beta_2 + \beta_1^2)\tau + \beta_1\beta_2^2 + \beta_1^2\beta_2.$$

The null spaces of these linear maps are the roots of the polynomials

$$X, \ X^2 + \beta_1 X$$

and

$$X^4 + (\beta_2^2 + \beta_1\beta_2 + \beta_1^2)X^2 + (\beta_1\beta_2^2 + \beta_1^2\beta_2)X.$$

It is easy to see directly that the null spaces of ϕ_0, ϕ_1, ϕ_2 have respective bases $\emptyset, \{\beta_1\}$ and $\{\beta_1, \beta_2\}$. More general, a basis for the null space of ϕ_i is given by $\{\beta_1, \ldots, \beta_i\}$: indeed, since $\phi_i := (\tau + \phi_{i-1}(\beta_i))\phi_{i-1}$, it is clear that the null space of ϕ_{i-1} is contained in that of ϕ_i. Moreover, evaluating ϕ_i in β_i, we find that

$$\phi_i(\beta_i) = (\tau + \phi_{i-1}(\beta_i))(\phi_{i-1}(\beta_i)) = \phi_{i-1}(\beta_i)^2 + \phi_{i-1}(\beta_i)\phi_{i-1}(\beta_i) = 0.$$

This means that the null space of ϕ_i at least contains V_i (and therefore at least 2^i elements). On the other hand, the null space of ϕ_i can be

expressed as the set of roots of the linearized polynomial $\phi_i(X)$, which is a polynomial of degree 2^i. Therefore the null space of ϕ_i equals V_i. For $i = d$, we obtain that the null space of ϕ_d is V. In other words: the desired polynomial $p_V(X)$ is just the linearized polynomial $\phi_d(X)$. The above claim about the sparseness of $p_V(X)$ now also follows. It is not hard to convert the above recursive description to compute $p_V(X)$ into an algorithm (see Alg. 5.1). In a step of the recursion, the multiplication $(\tau + \phi_{i-1}(\beta_i))\phi_{i-1}$ needs to be carried out in the Ore ring. Since the left term has degree one in τ, this is easy to do. To compute the coefficients in ϕ_i of all powers of τ one needs the commutation relation $\tau\alpha = \alpha^2\tau$ for $\alpha \in \mathbb{F}_{2^{128}}$. Computing a coefficient of a power of τ in a step of the recursion, therefore takes one multiplication, one squaring and one addition. The computation of ϕ_d can therefore be carried out without further optimization in quadratic complexity in d. A straightforward implementation can therefore be used to compute examples. Two examples are given in Appendix A with $d = 31$ and $d = 61$ needed for attacking GCM and POET.

Note that the above theory can easily be generalized to the setting of a finite field $\mathbb{F}_{r^e}$ and $\mathbb{F}_r$-subspaces V over the field $\mathbb{F}_{r^e}$. In the corresponding Ore ring $\mathbb{F}_{r^e}\{\tau\}$ the commutation relation is $\tau\alpha = \alpha^r\tau$. Similarly as above, for any subspace of a given dimension d one can find a polynomial $p_V(X)$ of degree r^d having as set of roots precisely the elements of V. It may have non-zero coefficients only for monomials of the form X^{r^i}. In the program given in Appendix A, r and e can be chosen freely. See [8] for a more detailed overview of properties of linearized polynomials and the associated Ore ring.

4 Disjoint coverage of the key space with roots of structured polynomials

The purpose of this section is to describe how one can cover the elements of a finite field $\mathbb{F}_q$ by sets of roots of families of explicitly given polynomials. We will focus our attention to the case that $q = 2^{128}$, but the given constructions can directly be generalized to other values of $q = r^e$. We denote by γ a primitive element of $\mathbb{F}_q$. Two approaches will be described. The first one exploits the multiplicative structure of $\mathbb{F}_q\backslash\{0\}$, while the second one exploits the additive structure of $\mathbb{F}_q$ seen as a vector space over $\mathbb{F}_2$. We will in fact describe a way to partition the elements of $\mathbb{F}_q$ as sets of roots of explicit polynomials, that is to say that two sets of roots of distinct polynomials will have no elements in common. In both cases

the algebraic fact that will be used is the following: Let G be a group with group operation $*$ and let $H \subset G$ be a subgroup. Then two cosets $g * H$ and $f * H$ are either identical or disjoint. Moreover the set of cosets gives rise to a partition of G into disjoint subsets.

4.1 Using the multiplicative structure

We first consider the group $G = \mathbb{F}_q \backslash \{0\}$ with group operation $*$ the multiplication in $\mathbb{F}_q$. For any factorization $q-1 = n \cdot m$ we find a subgroup $H_m := \{\gamma^{nj} \mid 0 \leq j \leq m-1\}$ consisting of m elements. This gives rise to the following proposition:

Proposition 1. *Let γ be a primitive element of the field $\mathbb{F}_q$ and suppose that $q-1 = n \cdot m$ for positive integers n and m. For i between 0 and $n-1$ define*

$$A_i := \{\gamma^{i+nj} \mid 0 \leq j \leq m-1\}.$$

Then the sets $A_0, \ldots, A_{n-1}$ partition $\mathbb{F}_q \backslash \{0\}$. Moreover, the set A_i consists exactly of the roots of the polynomial $X^m - \gamma^{im}$.

Proof. As mentioned we work in the multiplicative group $\mathbb{F}_q \backslash \{0\}$ and let H_m be the subgroup of G of order m. Note that $A_0 = H_m$ and that H_m is the kernel of the group homomorphism $\phi : G \to G$ sending x to x^m. In particular, H_m is precisely the set of roots of the polynomial $X^m - 1$. Any element from the coset gH_m is sent by ϕ to g^m. This means that gH_m is precisely the set of roots of the polynomial $X^m - g^m$. Note that $g^m = \gamma^{im}$ for some i between 0 and $n-1$, so that the set of roots of $X^m - g^m$ equals $\gamma^i H_m = A_i$ for some i between 0 and $n-1$. Varying i we obtain all cosets of H_m, so the result follows.

If $q = r^e$ for some prime power r, one can choose $n = r - 1$ and $m = r^{e-1} + \cdots + r + 1$. For any element $\alpha \in \mathbb{F}_q$ we then have $\alpha^m \in \mathbb{F}_r$, since α^m is just the so-called $\mathbb{F}_q/\mathbb{F}_r$-norm of α. Therefore the family of polynomials in the above lemma in this case take the particularly simple form $x^m - a$, with $a \in \mathbb{F}_r \backslash \{0\}$. In case $q = 2^{128}$, Proposition 1 gives rise to a family of polynomials whose roots partition $\mathbb{F}_{2^{128}} \backslash \{0\}$. For more details about the explicit form of these polynomials, we refer the reader to Appendix F.

4.2 Using the additive structure

Now we use a completely different approach to partition the elements from $\mathbb{F}_q$ in disjoint sets where we exploit the additive structure. Suppose

again that $q = r^e$, then we can view $\mathbb{F}_q$ as a vector space over $\mathbb{F}_r$. Now let $V \subset \mathbb{F}_q$ be any linear subspace (still over the field $\mathbb{F}_r$). If V has dimension d, then the number of elements in V equals r^d. For any $a \in \mathbb{F}_q$, we define $a + V$, the translate of V by a, as

$$a + V := \{a + v \mid v \in V\}.$$

Of course $a + V$ can also be seen as a coset of the subgroup $V \subset \mathbb{F}_q$ with addition as group operation. Any translate $a + V$ has r^d elements and moreover, it holds that two translates $a + V$ and $b + V$ are either disjoint or the same. This means that one can choose $n := r^e/r^d = r^{e-d}$ values of a, say $a_1, \ldots, a_n$ such that the sets $a_1 + V, \ldots, a_n + V$ partition $\mathbb{F}_q$.

The next task is to describe for a given subspace V of dimension d, the $n := r^{d-e}$ polynomials with $a_1 + V, \ldots, a_n + V$ as sets of roots. As a first step, we can just as before, construct an $\mathbb{F}_r$-linear map ϕ from F_q to itself, that can be described using a linearized polynomial of the form $p_V(X) = X^{r^d} + c_{d-1}X^{r^{d-1}} + \cdots + c_1X^r + c_0X$. The linear map ϕ then simply sends x to $p_V(x)$ and has as image

$$W := \{p_V(x) \mid x \in \mathbb{F}_q\}.$$

A coset $a + V$ of V is then sent to the element $p_V(a)$ by ϕ. This means that any coset of V can be described as the set of roots of the polynomial $p_V(X) - p_V(a)$, that is to say of the form $p_V(X) - b$ with $b \in W$ (the image of the map ϕ). Combining this, we obtain that we can partition the elements of $\mathbb{F}_q$ as sets of roots of polynomials of the form $p_V(X) - b$ with $b \in W$. Note that these polynomials still are very structured: just a constant term is added to the already very sparse polynomial $p_V(X)$. Note that $p_V(X) - p_V(a) = p_V(X - a)$, since $p_V(X)$ is a linearized polynomial. This makes it easy to confirm that indeed the set of roots of a polynomial of the form $p_V(X) - p_V(a)$ is just the coset $a + V$. The number of elements in W is easily calculated: since it is the image of the linear map ϕ and the dimension of the null space of ϕ is d (the dimension of V), the dimension of its image is $e - d$. This implies that W contains r^{e-d} elements. We collect some of this in the following proposition:

Proposition 2. *Let $q = r^e$ and let V be a linear subspace of $\mathbb{F}_q$ of over the field $\mathbb{F}_r$ of dimension d. Moreover denote by $p_V(x)$ be the linearized polynomial associated to V and define $W := \{p_V(x) \mid x \in \mathbb{F}_q\}$.*

Then for any $a \in \mathbb{F}_q$, the polynomial $p_V(x) - p_V(a)$ has as sets of roots exactly $a + V$. Moreover, the sets of roots of the polynomials $p_V(x) - b$ with $b \in W$ partition $\mathbb{F}_q$.

A possible description of a basis of W can be obtained in a fairly straightforward way. If $\{\beta_1, \ldots, \beta_d\}$ is a basis of V, one can extend this to a basis of $\mathbb{F}_q$, say by adding the elements $\beta_{d+1}, \ldots, \beta_e$. Then a basis of the image W of ϕ is simply given by the set $\{p_V(\beta_{d+1}), \ldots, p_V(\beta_e)\}$ (note that $\phi(\beta_i) = p_V(\beta_i) = 0$ for $1 \leq i \leq d$). This means that the r^{e-d} polynomials whose roots partition $\mathbb{F}_q$ are given by

$$p_V(X) + \sum_{i=d+1}^{e} a_i p_V(\beta_i), \quad \text{with } a_i \in \mathbb{F}_r.$$

The set of roots of a polynomial of this form is given by $\sum_{i=d+1}^{e} a_i\beta_i + V$. In the appendix, we give examples for $r^e = 2^{128}$ and $d = 31$ or $d = 61$.

5 Improved key recovery algorithm

Suppose that we have observed a polynomial hash collision for some forgery polynomial $p_V(X)$ of degree d, i.e. some observed message M and $M + p_V$ have the same image under h_H with the unknown authentication key H. This means that H must be among the roots of $p_V(X)$, and we can submit further verification queries using specially chosen forgery polynomials to recover the key.

5.1 An explicit key recovery algorithm using twisted polynomials

Being constructed in a twisted polynomial ring, our polynomials $p_V(X)$ are linearized polynomials, so that all roots are contained in a d-dimensional linear space $V \subset \mathbb{F}_2^n$. This enables an explicit and particularly efficient key recovery algorithm which recovers the key H by writing it as $H = \sum_{i=1}^{d} b_i\beta_i$ with respect to (w.r.t.) a basis $\mathcal{B} = \{\beta_1, \ldots, \beta_d\}$ for V over $\mathbb{F}_2$ and determining its d binary coordinates w.r.t. $\mathcal{B}$ one by one. Shortening the basis by the last element, we can test if $b_d = 0$ by using the forgery polynomial corresponding to $V' = \text{span}\{\beta_1, \ldots, \beta_{d-1}\}$. If this query was not successful, we deduce $b_d = 1$. We then proceed recursively for the next bit.

Unless all $b_i = 0$, the search space will be restricted to an affine instead of a linear subspace at some point. It is easy to see, however, that the corresponding polynomial for $A = V + a$ with V a linear subspace, can always be determined as $p_A(X) = p_V(X - a) = p_V(X) - p_V(a)$ since the $p_V(X)$ are linearized polynomials.

Algorithm 5.1 Construction of twisted polynomials

Input: basis $\mathcal{B} = \{\beta_1, \ldots, \beta_d\}$ of $V \subset \mathbb{F}_2^n$
Output: polynomials $p_{V^{(i)}}(X)$ having $\mathrm{span}\{\beta_1, \ldots, \beta_i\}$ as set of roots

Set $a_1 \leftarrow 1$
Set $a_i \leftarrow 0$ for $2 \leq i \leq d+1$
for $i = 1$ to d **do**
  $v \leftarrow \sum_{k=1}^{d} a_k \beta_i^{2^k}$
  $c_1 \leftarrow v \cdot a_1$
  for $j = 2$ to $d+1$ **do**
    $c_j \leftarrow a_{j-1}^2 + v \cdot a_j$
  end for
  $p_{V^{(i)}} \leftarrow \sum_{k=1}^{d+1} c_k X^{2^{k-1}}$
end for
return polynomials $p_{V^{(1)}}(X), \ldots, p_{V^{(d)}}(X)$

Algorithm 5.2 Key recovery using twisted polynomials

Input: message M, polynomial $p_V(X)$ s.t. $h_H(M) = h_H(M + P_V)$,basis $\mathcal{B} = \{\beta_1, \ldots, \beta_d\}$ of d-dimensional linear subspace $V \subset \mathbb{F}_2^n$.
Output: authentication key H.

$b_i \leftarrow 0, \quad 1 \leq i \leq d$
Call Alg. 5.1 on V, obtain $p_{V^{(1)}}, \ldots, p_{V^{(d)}}$
for $i = d$ downto 1 **do**
  Denote $U^{(i)} = \mathrm{span}\{\beta_1, \ldots, \beta_{i-1}\}$, so that $p_{U^{(i)}} = p_{V^{(i-1)}}$
  $\alpha \leftarrow p_{U^{(i)}}(\sum_{j=i}^{d} b_j \beta_j)$
  if $h_H(M) = h_H(M + P_{U^{(i)}} + \alpha)$ **then**
    $b_i \leftarrow 0$
  else
    $b_i \leftarrow 1$
  end if
end for
return key $H = \sum_{i=1}^{d} b_i \beta_i$

The complexity of Algorithm 5.2 for a polynomial of degree d (corresponding to $|V| = 2^d$) is given by d verification queries and one invocation of the polynomial construction algorithm 5.1, which in turn takes $O(d^2)$ finite field operations. Note that typically, $d < 64$. The total length of all verification queries is limited by 2^{d+1} blocks. Since the polynomials $p_{U^{(i)}}(X)$ have at most $d+1$ nonzero coefficients, they are very sparse and only very few additions to M are required to compute the message $M + P_{U^{(i)}}$ for the forgery attempt.

We emphasize that this algorithm can be readily generalized to deal with input polynomials $p_A(X)$ having affine root spaces $A = V + a$ by operating with the corresponding linear space V and adding $p_{V^{(i)}}(a)$ to all verification queries. This especially allows to combine this algorithm with the key space covering strategy of Sect. 4.2.

In the context of authenticated encryption, M will typically correspond to ciphertexts instead of plaintexts, so also in this case, only calls to the verification oracle are required. It is also straightforward to adapt Algorithm 5.2 to cases where a polynomial hash collision cannot directly be observed, but instead propagates into some other property visible from

ciphertext and tag. This is for example used in our attacks on the COBRA authenticated encryption scheme (see Appendix C).

5.2 Comparison to previous work

The idea of using a binary search-type algorithm to recover authentication keys has previously been applied to various universal hashing-based MAC constructions by Handschuh and Preneel [10]. Their attack algorithm however does not deal with the (important) questions of determining new polynomials having distinct roots from all previously used ones, and also requires the calculation and storage of the 2^d roots during the key search phase. Also, the required polynomials will not be sparse and require up to 2^d nonzero coefficients. By contrast, our algorithm leverages the twisted polynomial ring to explicitly construct sparse polynomials with exactly the necessary roots for restricting the search space in each iteration.

A different approach for binary-search type key recovery is given in Sect. 7.3 of [15], suggesting the use of forgery polynomial $q(X) = \prod_{H \in \mathbb{F}_2^n, H_n=0}(X - H)$ and then subsequently fixing more bits of H according to the results of the verification queries. While this is clearly optimal with respect to the number of queries (which is n), the resulting messages are up to 2^n blocks long, which typically exceeds the limits imposed by the specifications. Additionally, the polynomials will have almost no zero coefficients, which requires up to 2^{n+1} additions for the verification queries. By contrast, when combined with the keyspace covering strategy outlined in Sect. 4.2, our algorithm requires $2^{n/d} \cdot d$ queries, each of them being maximally 2^d blocks long. This not only allows staying within the specified limits, but also allows choosing any desired trade-off between the number and length of the queries. Our explicit polynomials also have a maximum of $d + 1$ nonzero coefficients each, which limits the number of additions to $2^{n/d} \cdot (d + 1)$.

6 Nonce-respecting universal forgeries for GCM

In this section, we describe two nonce-respecting universal forgery attacks against GCM [6] under weak keys. Before describing the attacks we describe the GCM authenticated encryption scheme and the GCM counter values generation procedure as defined in the NIST standard [6].

6.1 More details on GCM

We recall the GCM ciphertext/tag generation:

$$T = E_k(J_0) \oplus h_H(C),$$

with T denoting the tag, with $M = M_1||M_2||\cdots||M_l$ the plaintext and $C = C_1||C_2||\cdots||C_l$ the ciphertext blocks produced using a block cipher E_k in counter mode, i.e. $C_i = E_K(J_i) \oplus M_i$. The J_i's are successive counters with the initial J_0 generated from the nonce N; furthermore $H = E_k(0)$ with k the secret key.

We now focus on the detailed generation of the counter values in GCM. We have

$$J_0 = \begin{cases} N||0^{31}||1 & \text{if } |N| = 96, \\ h_H(N||0^{s+64}||[|N|]_{64}) & \text{if } |N| \neq 96, \end{cases}$$

where $J_i = \text{inc}_{32}(J_{i-1})$, where $s = 128\lceil |N|/128 \rceil - |N|$, $[X]_{64}$ is the 64-bit binary representation of X and $\text{inc}_{32}(X)$ increments the right-most 32 bits of the binary string X modulo 2^{32}; the other left-most $|X| - 32$ bits remain unchanged.

6.2 Universal Forgery Attacks on GCM

Our universal forgery attacks are possible if the hash key H is weak. Therefore, our attack starts by detecting whether the hashing key H is weak or not using our forgery polynomial $q(X) = p_V(X)$ of degree 2^{31} explicitly described in Appendix A.1. In other words, we make a blind forgery for an observed ciphertext/tag pair $(C; T)$ by asking for the verification of the forged ciphertext $(C + Q); T$ where $Q = q_1||...||q_l$. Now if H is a weak key according to our forgery polynomial – is a root of $q(X) = p_V(X)$ – then the verification succeeds and the GCM scheme outputs a random plaintext.

Once we know that H is a weak-key, then we can recover it using Algorithm 5.2 over the roots of $q(X) = p_V(X)$ (see Appendix A.1) where at each query we can choose different nonces.

Now, the only hurdle for generating a nonce-respecting forgery is computing the value of $E_K(J_0)$ since we do not know the secret key K (we have only recovered $H = E_K(0)$). However, since GCM is using a counter mode encryption where the successive counter values J_i, are generated from the nonce, we can easily get the encryption of the counter values $E_K(J_i)$ by simply xoring the corresponding plaintext and ciphertext blocks (Note

that in NIST GCM, the right-most 32 bits of the counter values are successive modulo 2^{32} as shown below). In the sequel, we show how to use the encryption of the counter values in order to construct universal forgeries.

Slide universal forgeries using chosen nonce N with $|N| \neq 96$ Suppose that we have observed an l-block plaintext/ciphertext with tag T, $M = M_1||\cdots||M_l$ and $C = C_1||\cdots||C_l$, where $C_i = M_i \oplus E_K(J_i)$, $J_i = \text{inc}_{32}(J_{i-1})$ and $T = E_K(J_0) \oplus h_H(C)$. Our goal now is to generate a valid ciphertext/tag for a different message M' using a different chosen nonce N' where $|N'| \neq 96$.

As mentioned above, the counter mode of operation enables us to find the encryption of the counter values

$$E_K(J_0), E_K(J_1), \cdots, E_K(J_v), \cdots, E_K(J_l).$$

The idea of the attack is to slide these encrypted counter values v positions to the left in order to re-use the $(l-v)$ encrypted counter values $E_K(J_v), \cdots, E_K(J_l)$ to generate valid ciphertext/tag for any new message M' with a new chosen nonce N' that gives us an initial counter value $J'_0 = J_v$. This will enable us to make slide universal forgeries for an $(l-v)$-block message. See Fig. 1.

One can see that using J_v, $v > 0$, it is possible to choose a nonce N' that gives $J'_0 = J_v$ by solving the following equation for N'

$$J'_0 = J_v = h_H(N'||0^{s+64}||[|N'|]_{64})$$

Note that when $|N'| = 128$ (i.e. $s = 0$), we have only one solution for N' and more than one solution for $|N'| > 128$. However, when $|N'| < 128$ we might have no solution. Therefore we assume that $|N'| \geq 128$.

Once we find the nonce N' that yields $J'_0 = J_v$, then one can see that we have the following 'slid' identities:

$$E_K(J'_0) = E_K(J_v), E_K(J'_1) = E_K(J_{v+1}), \cdots, E_K(J'_{l-v}) = E_K(J_l)$$

Consequently, we are able to compute $C'_i = M'_i \oplus E_K(J'_i)$ for $1 \leq i \leq l-v$ and $T' = E_K(J'_0) \oplus h_H(C')$. Thus observing the encryption of an l-block message and setting $J'_0 = J_v$ as shown above enable us to generate a valid ciphertext/tag (C'/T') for an $(l-v)$-block message M' under the nonce-respecting setting.

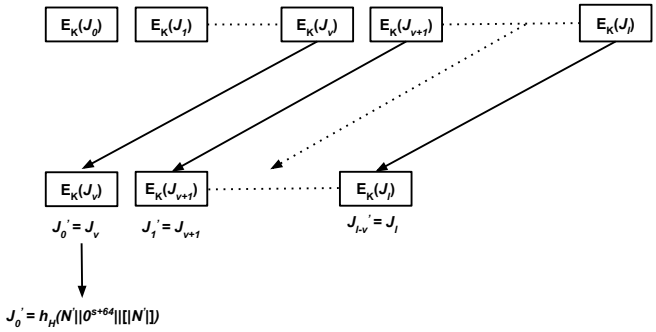


Fig. 1: Forgeries for GCM via sliding the counter encryptions

Universal forgeries using arbitrary nonces N with $|N| = 96$ Assume that we are using a GCM implementation that supports variable nonce lengths. For example, the implementation of GCM in the latest version of OpenSSL [17,18] makes the choice of the nonce length optional, i.e. one can use different nonce sizes under the same secret key. Now, suppose that using such a GCM oracle with the secret key K, we need to find the ciphertext/tag of a message $M = M_1||\cdots||M_l$ with a nonce N where $|N| = 96$, so $J_0 = N||0^{31}||1$. In order to generate the ciphertext/tag we need to find $E_K(J_i)$ where $J_i = \text{inc}_{32}(J_{i-1})$. We do not know the secret key K. However, since we know the secret hash key H, we can solve for N' the following equation

$$J_0 = h_H(N'||0^{s+64}||[|N'|]_{64}) \quad \text{where} \quad |N'| \neq 96$$

Note that we assume that $|N'| \geq 128$ as otherwise we might not get a solution. After finding N', we can query the same GCM oracle (that has been queried for encrypting M with the nonce N where $|N| = 96$) with a new nonce N' that has a different size $|N'| \geq 128$ [2] for the encryption of some plaintext $M = M_1'||\cdots||M_l'$. Now, $|N'| \neq 96$ means that the initial counter value $J_0' = h_H(N'||0^{s+64}||[|N'|]_{64}) = J_0$. Therefore, from the corresponding ciphertext blocks $C_1', \cdots, C_l'$, we find $E_K(J_i) = E_K(J_i') = M_i' \oplus C_i'$. Consequently the corresponding ith ciphertext block of M_i is $C_i = E_K(J_i') \oplus M_i$ and the corresponding tag is $T = E_K(J_0') \oplus h_H(C)$. It

[2] Two of the test vectors (Test Case 3 and Test Case 6, see Appendix D) for the GCM implementation in the latest release of OpenSSL share the same secret key (and therefore the same hash key) but they use different nonce sizes, Test Case 3 uses a nonce with length 96 while Test Case 6 uses a nonce with length 480 [18]. This suggests that it is conceivable to have different IV sizes under the same secret key.

is worthy to note, that this interaction possibility between two different nonce lengths on GCM had been listed in [19] as one of the undesirable characteristics of GCM. Fig. 2 demonstrates the interaction attack.

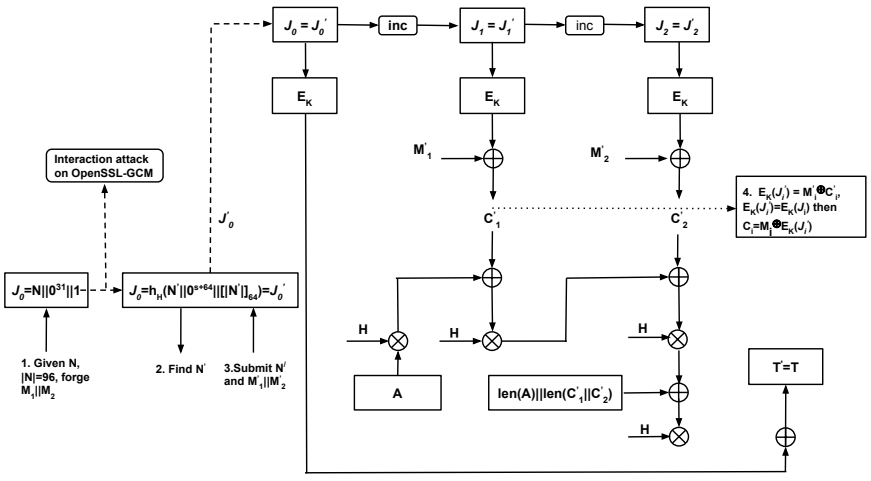


Fig. 2: Forgeries for GCM via cross-nonce interaction

7 Analysis of POET

In this section, we present a detailed weak key analysis of the online authentication cipher POET when instantiated with Galois-Field multiplication. More specifically, we create universal forgery attacks once we recover the hashing weak key. Before this we give a brief description of POET.

7.1 Description of POET

A schematic description of POET [2] is given in Fig. 3a. Five keys $L, K, L_{\text{top}}, L_{\text{bot}}$ and L_T are derived from a user key as encryptions of the constants $1, \ldots, 5$. K denotes the block cipher key, L is used as the mask in the AD processing, and L_T is used as a mask for computing the tag. Associated data (AD) and the nonce are processed using the secret value L in a PMAC-like fashion (see [2] for details) to produce a value τ which is then used as the initial chaining value for both top and bottom mask layers, as well as for generating the authentication tag T. The "header"

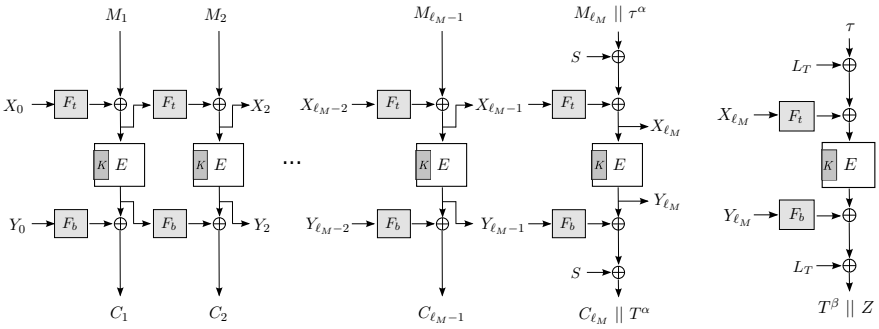
H encompasses the associated data (if present) and includes the nonce in its last block. S denotes the encryption of the bit length of the message M, i.e. $S = E_K(|M|)$. The inputs and outputs of the i-th block cipher call during message processing are denoted by X_i and Y_i, respectively.

One of the variants of POET instantiates the functions F_t and F_b by $F_t(x) = L_{\text{top}} \cdot x$ and $F_b(x) = L_{\text{bot}} \cdot x$, with the multiplication taken in $\mathbb{F}_2^{128}$. This is also the variant that we consider in this paper. The top AXU hash chain then corresponds to the evaluation of a polynomial hash in $\mathbb{F}_2^{128}$:

$$g_t(X) = \tau {L_{\text{top}}}^m + \sum_{i=1}^{m} X_i {L_{\text{top}}}^{m-i},$$

with g_t being evaluated at $X = M_1, \ldots, M_{m-1}, M_m \oplus S$.

For integral messages (*i.e.*, with a length a multiple of the block size), the authentication tag T then generated as $T = T^\beta$ with empty Z, as shown in Fig. 3b. Otherwise, the tag T is the concatenation of the two parts T^α and T^β, see Fig. 3a and 3b.



(a) First-part tag generation in POET [2]

(b) Second-part tag generation in POET [2]

Fig. 3: Schematic description of POET

7.2 Universal weak-key forgeries for POET

We start by the following observations.

Observation 1 (Collisions in g_t imply tag collisions). *Let $M = M_1, \ldots, M_m$ and $M' = M'_1, \ldots, M'_m$ be two distinct messages of m blocks length such that $g_t(M) = g_t(M')$ or $g_t(M_1, \ldots, M_\ell) = g_t(M'_1, \ldots, M'_\ell)$*

with $\ell < m$ and $M_i = M'_i$ for $i > \ell$. This implies a collision on POET's internal state X_i, Y_i for $i = m$ or $i = \ell$ respectively, and therefore equal tags for M and M'.

We note that such a collision also allows the recovery of L_{top} by Algorithm 5.2.

Observation 2 (Knowing L_{top} implies knowing L_{bot}). *Once the first hash key L_{top} is known, the second hash key L_{bot} can be determined with only two 2-block queries: Choose arbitrary M_1, M_2, ∇_1 with $\nabla_1 \neq 0$ and obtain the encryptions of the two 2-block messages M_1, M_2 and M'_1, M'_2 with $M'_1 = M_1 \oplus \nabla_1, M'_2 = M_2 \oplus \nabla_1 \cdot L_{top}$. Denote $\Delta_i = C_i \oplus C'_i$. Then we have the relation $\Delta_1 \cdot L_{bot} = \Delta_2$, so $L_{bot} = \Delta_1^{-1} \cdot \Delta_2$.*

It is worth noting that this procedure works for arbitrary L_{bot}, and is in particular not limited to L_{bot} being another root of the polynomial q.

A generic forgery. In the setting of [15], consider an arbitrarily chosen polynomial $q(X) = \sum_{i=1}^{m-1} q_i X^i = p_V(X)$ of degree $m-1$ and some message $M = M_1 \| \cdots \| M_{m-1} \| M_m$. Write $Q = q_1 \| \cdots \| q_{m-1}$ and define $M' \stackrel{\text{def}}{=} M + Q$ with Q zero-padded as necessary. For a constant nonce (1-block header) H, denote ciphertext and tag corresponding to M by $C = C_1, \ldots, C_m$ and T, and ciphertext and tag corresponding to $M' = M + Q$ by $C' = C'_1, \ldots, C'_m$ and T', respectively.

If some root of q is used as the key L_{top}, we have a collision between M and $M' = M + Q$ in the polynomial hash evaluation after $m-1$ blocks:

$$\tau {L_{\text{top}}}^m + \sum_{i=1}^{m-1} M_i {L_{\text{top}}}^{m-i} = \tau' {L_{\text{top}}}^m + \sum_{i=1}^{m-1} M'_i {L_{\text{top}}}^{m-i}$$

This implies $X_{m-1} = X'_{m-1}$ and therefore $Y_{m-1} = Y'_{m-1}$. Since the messages are of equal length, $S = S'$ and we also have a collision in X_m and Y_m. It follows that $C_m = C'_m$. Furthermore, since $\tau = \tau'$, the tag T is colliding as well. Since then M and $M + Q$ have the same tag, $M + Q$ is a valid forgery whenever some root of $q(X) = p_V(X)$ is used as L_{top}. Note that both M and the forged message will be m blocks long.

Using the class of weak keys represented by the roots of the forgery polynomial $q(X) = p_V(X)$ explicitly described in Appendix A and Appendix A.2, we discuss the implication of having one such key as the universal hash key L_{top}. Since POET allows nonce-reuse, we consider nonce-repeating adversaries, i.e. for our purposes, the nonce will be fixed to some constant value for all encryption and verification queries. However,

once we recovered τ, we will be able to recover the secret value L and consequently we can make forgeries without nonce-reuse.

More specifically, we show that weak keys enable universal forgeries for POET under the condition that the order of the weak key is smaller than the maximal message length in blocks. For obtaining universal forgeries, we first use the polynomial hash collision described above to recover the weak keys L_{top} and L_{bot}, and then recover τ, which is equal to the initial states X_0 and Y_0, under the weak key assumption.

Recovering $\boldsymbol{\tau}$ Suppose that we have recovered the weak keys $L_{\text{top}}$and L_{bot}. Now our goal is to recover the secret $X_0 = Y_0 = \tau$. We know that $X_i = \tau L_{\text{top}}^i + M_1 L_{\text{top}}^{i-1} + M_2 L_{\text{top}}^{i-2} + \cdots + M_i$ and $X_{i+j} = \tau L_{\text{top}}^{i+j} + M_1 L_{\text{top}}^{i+j-1} + M_2 L_{\text{top}}^{i+j-2} + \cdots + M_{i+j}$.

Now if L_{top} has order j , i.e. $L_{\text{top}}^j =$ Identity, then we get $X_i = X_{i+j}$ by constructing $M_{i+1}, \cdots, M_{i+j}$ such that $M_{i+1} L_{\text{top}}^{j-1} + M_{i+2} L_{\text{top}}^{j-2} + ... + M_{i+j} = 0$. The easiest choice is to set $M_{i+1} = M_{i+2} = \cdots = M_{i+j} = 0$. This gives us $Y_i = Y_{i+j}$. Now equating the following two equations and assuming that $L_{\text{bot}}^j \neq$ Identity, $Y_i = \tau L_{\text{bot}}^i + C_1 L_{\text{bot}}^{i-1} + C_2 L_{\text{bot}}^{i-2} + \cdots + C_i$ and $Y_{i+j} = \tau L_{\text{bot}}^{i+j} + C_1 L_{\text{bot}}^{i+j-1} + C_2 L_{\text{bot}}^{i+j-2} + \cdots + C_{i+j}$. We get

$$\tau = \left(C_1 L_{\text{bot}}^{i-1} + C_2 L_{\text{bot}}^{i-2} + \cdots + C_i + C_1 L_{\text{bot}}^{i+j-1} + C_2 L_{\text{bot}}^{i+j-2} + \cdots + C_{i+j}\right) \cdot (L_{\text{bot}}^i + L_{\text{bot}}^{i+j})^{-1},$$

which means that we now know the initial values of the cipher state.

Querying POET's block cipher $\boldsymbol{E_K}$. One can see from Fig. 3a that once we know L_{top}, L_{bot} and τ, we can directly query POET's internal block cipher without knowing its secret key K. internal block cipher, i.e. we want to compute $E_K(x)$. Now from Fig. 3a, we see that the following equation holds: $E_K(\tau L_{\text{top}} \oplus M_1) = C_1 \oplus \tau L_{\text{bot}}$, therefore $E_K(x) = C_1 \oplus \tau L_{\text{bot}}$. If M_1 was the last message block, however, we would need the encryption $S = E_K(|M|)$. Therefore we have to extend the auxiliary message for the block cipher queries by one block, yielding the following:

Observation 3 (Querying POET's block cipher). *Knowing L_{top}, L_{bot} and τ enables us to query POET's internal block cipher without the knowledge of its secret key K. To compute $E_K(x)$ for arbitrary x, we form a two-block auxiliary message $M_1' = (x \oplus \tau L_{top}, M_2')$ for arbitrary M_2' and obtain its POET encryption as C_1', C_2'. Computing $E_K(x) := C_1' \oplus \tau L_{bot}$ then yields the required block cipher output.*

This means that we can produce valid ciphertext blocks $C_1, \ldots, C_{\ell_M}$ and (if necessary) partial tags T^α for any desired messages, by simply following the POET encryption algorithm using the knowledge of

$L_{\text{top}}, L_{\text{bot}}, \tau$ and querying POET with the appropriate auxiliary messages whenever we need to execute an encryption E_K. Note that this also includes the computation of $S = E_K(|M|)$.

Generating the final tag. In order to generate the second part of the tag T^β (see Fig. 3b), which is the full tag T for integral messages, we use the following procedure.

We know the value of X_{ℓ_M} for our target message M from the computation of C_{ℓ_M}. If we query the tag for an auxiliary message M' with the same $X'_{\ell_{M'}}$, the tag for M' will be the valid tag for M as well, since having $X'_{\ell_{M'}} = X_{\ell_M}$ means that $Y'_{\ell_{M'}} = Y_{\ell_M}$ and consequently $T^{\beta\prime} = T^\beta$.

Therefore, we construct an auxiliary one-block message $M' = (X_{\ell_M} \oplus E_K(|M'|) \oplus \tau L_{\text{top}}$ and obtain its tag as T' (computing the encryption of the one-block message length by querying E_K as above). By construction $X'_1 = X_{\ell_M}$, so T' is the correct tag for our target message M as well.

By this, we have computed valid ciphertext blocks and tag for an arbitrary message M by only querying some one- or two-block auxiliary messages. This constitutes a universal forgery.

We finish by noting that in case a one- or two-block universal forgery is requested, we artificially extend our auxiliary messages in either the final tag generation (for one-block targets) or the block cipher queries (for two-block messages) with one arbitrary block to avoid having queried the target message as one of our auxiliary message queries.

7.3 Further forgery strategies

Since the universal forgery of the previous section relies on having a weak key L_{top} with an order smaller than the maximum message length for recovering τ, we describe two further forgery strategies that are valid for any weak key, regardless of its order. We also show how the knowledge of τ enables us to recover the secret value L. This will enable us to make universal forgeries on POET within the nonce-respecting adversary model. In other words, recovering the secret value L means that we will be able to process the header (associated data and nonce) and generate a new τ and consequently have a total control over the POET scheme. Due to the space limitation, all these further forgery attacks are given in Appendix E.

8 Conclusion

Polynomial hashing is used in a large number of MAC and AE schemes to derive authentication tags, including the widely deployed and standardized GCM, and recent nonce misuse-resistant proposals such as POET, Julius, and COBRA. While a substantial number of works has pointed out weaknesses stemming from its algebraic structure [10, 15, 20], a crucial part of the proposed attacks, the construction of appropriate forgery polynomials, had not been satisfactorily addressed.

In this paper, we deal with this open problem of polynomial construction and selection in forgery and key recovery attacks on such schemes. We describe explicit constructions of such forgery polynomials with controlled sets of roots that have the additional advantage of being very sparse. Based upon this, we propose two strategies to achieve complete disjoint coverage of the key space by means of such polynomials, again in an explicit and efficiently computable construction. We also saw that this yields an improved strategy for key recovery in such attacks.

We then apply our framework to GCM in the weak-key model and describe, to the best of our knowledge, the first universal forgeries without nonce reuse. We also describe such universal forgeries for the recent AE schemes POET, Julius, and COBRA.

References

1. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, March 2014. `http://competitions.cr.yp.to/caesar.html`.
2. Farzaneh Abed, Scott Fluhrer, John Foley, Christian Forler, Eik List, Stefan Lucks, David McGrew, and Jakob Wenzel. The POET Family of On-Line Authenticated Encryption Schemes. Submission to the CAESAR competition, 03 2014.
3. Elena Andreeva, Andrey Bogdanov, Martin M. Lauridsen, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. COBRA: A Parallelizable Authenticated Online Cipher Without Block Cipher Inverse. Submission to the CAESAR competition, 03 2014.
4. Elena Andreeva, Atul Luykx, Bart Mennink, and Kan Yasuda. COBRA: A Parallelizable Authenticated Online Cipher Without Block Cipher Inverse. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption, FSE 2014*, Lecture Notes in Computer Science, page 24. Springer-Verlag, 2014. to appear.
5. Lear Bahack. Julius: Secure Mode of Operation for Authenticated Encryption Based on ECB and Finite Field Multiplications. Submission to the CAESAR competition, 03 2014. `http://competitions.cr.yp.to/round1/juliusv10.pdf`.
6. Morris Doworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, November, 2007. `csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`.
7. Neils Ferguson. Authentication weaknesses in GCM. Comments submitted to NIST Modes of Operation Process, 2005.

8. David Goss. *Basic structures of function field arithmetic*, volume 35 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)]*. Springer-Verlag, Berlin, 1996.
9. Jian Guo, Jrmy Jean, Thomas Peyrin, and Wang Lei. Breaking POET Authentication with a Single Query. Cryptology ePrint Archive, Report 2014/197, 2014. `http://eprint.iacr.org/`.
10. Helena Handschuh and Bart Preneel. Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2008.
11. Antoine Joux. Authentication Failures in NIST version of GCM. Comments submitted to NIST Modes of Operation Process, 2006.
12. David McGrew, Scott Fluhrer, Stefan Lucks, Christian Forler, Jakob Wenzel, Farzaneh Abed, and Eik List. Pipelineable On-Line Encryption. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption, FSE 2014*, Lecture Notes in Computer Science, page 24. Springer-Verlag, 2014. to appear.
13. David McGrew and John Viega. The galois/counter mode of operation (gcm). *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf*, 2004.
14. Mridul Nandi. Forging attacks on two authenticated encryptions cobra and poet. Cryptology ePrint Archive, Report 2014/363, 2014. `https://eprint.iacr.org/2014/363`.
15. Gordon Procter and Carlos Cid. On Weak Keys and Forgery Attacks against Polynomial-based MAC Schemes. In Shiho Moriai, editor, *Fast Software Encryption, FSE 2013*, Lecture Notes in Computer Science, page 14. Springer-Verlag, 2013. to appear.
16. Gordon Procter and Carlos Cid. On weak keys and forgery attacks against polynomial-based mac schemes. Cryptology ePrint Archive, Report 2013/144, 2013. `http://eprint.iacr.org/`.
17. OpenSSL Project. `https://www.openssl.org/`.
18. OpenSSL Project. GCM Implementation: crypto/modes/gcm128.c. `https://www.openssl.org/source/`, Latest release: 7 April 2014, openssl-1.0.1g.
19. Phillip Rogaway. Evaluation of some blockcipher modes of operation. *Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.
20. Markku-Juhani Olavi Saarinen. Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes. In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2012.
21. Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

A Appendix: Forgery polynomial suggestions for GCM and POET

In this appendix we give some examples of polynomials whose roots form a linear subspace V_d of $\mathbb{F}_{2^{128}}$ of dimension d for $d = 31$ and $d = 61$. As vector space V_d we have chosen the space spanned by the elements $1, \gamma, \cdots, \gamma^{d-1}$,

with γ a primitive elements of $\mathbb{F}_{2^{128}}$ satisfying $\gamma^{128} = \gamma^7 + \gamma^2 + \gamma + 1$. The following MAGMA-program produces such polynomials:

```
r:=2;
e:=128;
k<gamma>:=GF(r,e);
d:=61;
B:=[gamma^(i-1): i in [1..d]];
c:=[0*gamma: i in [1..(d+1)]];
c[1]:=gamma^0;
for b in B do
v:=0;
for i in [1..(d+1)] do
v:=v+c[i]*b^(r^(i-1));
end for;
cc:=c;
c[1]:=-v^(r-1)*cc[1];
for j in [2..(d+1)] do
c[j]:=cc[j-1]^r-v^(r-1)*cc[j];
end for;
end for;
P<X>:=PolynomialRing(k);
pV:=0;
for i in [1..(d+1)] do
pV:=pV+c[i]*X^(r^(i-1));
end for;
pV;
```

Fig. 4: Magma source code generating forgery polynomial of degree 2^d on $\mathbb{F}_2^{128}$ (here $d = 61$)

The calculated polynomial will have the form $c_{d+1}X^{2^d} + c_dX^{2^{d-1}} + \cdots + c_1X^{2^0}$ and it is sufficient to simply state the coefficients c_i, which can be expressed in the form a^{e_i} with $0 \leq e_i \leq 2^{128} - 2$. To save space we only list the exponents e_i for each polynomial in the following tables.

A.1 Forgery polynomial with degree 2^{31} for attacking GCM

For $d = 31$, one obtains the following coefficients:

i	e_i
1	576613647098987897394216259339443067 7
2	886405851238878607712823602816508493 69
3	228467699759147933517306066079059941262
4	608709206422113118601250588783762399 67
5	699813938596682643737860908514039195 97
6	255459844209463555435845538974500206397
7	263576500668765237830541241929740306586
8	371670151494514720087160030776564926 21
9	58043277378748107723324135119415484405
10	321767455835401530567257366419614234023
11	450338884514507376214297123948464446 57
12	258425985086309803122357832308421510564
13	105831989526232747717837668269825340779
14	267464360177071876386745024557199320756
15	280644372754658909872880662034708629284
16	105000326856250697615431403289357708609
17	458258183594606115422832253689081928 57
18	828459613081692598766012671274594169 89
19	442179899361942084725223538212208611 15
20	690629439605523090898429831294031742 17
21	268462019404836089359334939776220681511
22	300016489421132402121135552937497655 14
23	669737854382487997736546203881056449
24	127958856468256956044189872000451203235
25	277162238678239965835219683143318848400
26	134662498954166373112542807113066342554
27	219278415175240762588240883266619436470
28	216197476010311230105259534730909158682
29	281783005767613667130380044536264251829
30	181483131639777656403198412151415404929
31	383848366876114263330516022408845847 92
32	0

Table 1: The table shows the coefficients of the forgery polynomial $q(X) = p_V(X)$ for attacking GCM

A.2 Forgery polynomial with degree 2^{61} for attacking POET

Similarly for $d = 61$ one obtains the following coefficients:

i	e_i	i	e_i
1	205269631350267731197715294199912473 27	32	109604555581389038896555752982244394616
2	264546851691026540251722618719245777504	33	119482829110451460647031381779266776526
3	792797323058334749028936476772171594921	34	165259785861038013124994816644344468967
4	325712555585908542291537560181869632351	35	155444340258770748055544634836807134293
5	281140838798434203589324885475612499 13	36	869821844387300458212740258310619614 30
6	271147943451442547572675283203493325775	37	104870645496065737272877350967826010844
7	335255520823733252020392488407731432338	38	562812815790023183370379193561271053 69
8	671801688290763317086056756932989527 3	39	100068518982837928471870587740499831 41
9	255889065981883867903019621991013125435	40	936879200755548123588902448980883454 49
10	49457687721601463712640189217755474230	41	698326729003034322484017536582625335 06
11	311579005442569730277030755228683616807	42	246360754285298743574294101515912517720
12	227984510405461964893924913268809066393	43	895678936019042717674614594480764049 68
13	324660953045118328235538900161997992161	44	337681726780870315172220356080972321854
14	101370059745789285127519397790494215441	45	210317547004302372764274348440690947691
15	335840777837142047555650075244373419708	46	158574321133010145534802861165087620178
16	314588499802672014617473470717109075 23	47	291559826228649927512447763293001897434
17	339477818976914242962906054286547702007	48	156351243312442316097609527177914577 46
18	267056244491330957618685443721979120206	49	196562458398036090488379086660199368109
19	115274327651619347046091793992432007152	50	308779188958300135859037769338975723488
20	309606471838332610868454369483105904888	51	311961723579011854596575128443762996895
21	314728319634705433804935434967329763	52	153505386496968503239745640447605550270
22	191332595597193424626322329032056378009	53	266880473479137548264080346617303001989
23	189553913431309255614514163550670075672	54	325361660912502344542873376867973189476
24	224617322052671248319257827067474740867	55	756486261013747940931759163320432850 57
25	630412303067880329731111455333070515 62	56	122904035765598179315104311504496672627
26	221576606272152354153350739375040337239	57	240654849065616783877381099532333510366
27	291799903540006289220245045188573741192	58	717747464603164639815429745582806718 65
28	290489624437950764499707232619770186293	59	318833970371431372762935716012099244730
29	263754726506046639985479240660603777000	60	176351990917361872511208705771673004140
30	451608074361673079906891507920526707 07	61	227372417807158122619428517134408021585
31	336308819059966309252377016229504259 50	62	0

Table 2: The table shows the coefficients of the forgery polynomial $q(X) = p_V(X)$ for attacking POET

Let us denote the found polynomials by $p_d(X)$ (with $d = 31$ or $d = 61$). From $p_d(X)$, we can obtain a family of 2^{128-d} polynomials whose root sets partition $\mathbb{F}_2^{128}$. The polynomials have the form $p_d(X) + b$, with $b \in W_d := \{p_d(a) \mid a \in \mathbb{F}_2^{128}\}$. Since in the above examples V_d has basis $\{1, \gamma, \ldots, \gamma^{d-1}\}$ A basis of W_d is given by $\{p_d(\gamma^i) \mid d \leq i \leq 127\}$, making it straightforward to describe all possibilities for b.

B Universal forgeries for Julius under weak keys

Julius is a new authenticated encryption scheme submitted to the ongoing CAESAR competition [5]. Julius has four modes of operation: Julius-regular and Julius-compact, both offered in ECB and CTR modes. We only consider the regular versions of Julius-ECB and Julius-CTR, which are considered stronger and are the recommended versions.

All Julius modes use polynomial hashing to generate a *seed* which is then encrypted to produce a value $\mu = E_K(seed)$. This μ serves either directly as tag (CTR) or is encrypted once more (ECB) to form the tag. We exploit weak keys in polynomial hashing to describe universal forgery strategies for Julius.

B.1 Description of regular Julius-ECB and Julius-CTR

Both Julius-ECB and Julius-CTR use mode-specific padding rules. For simplicity, we describe the algorithms when the message length $|M|$ is a multiple of the block length $n = 128$ bits, and when no associated data A is used.

For Julius-CTR, a message M is padded as follows, where brackets denote full 128-bit blocks:

$$P = \overline{0 \cdots 01}\ \overline{IV}\ \overline{0\,|M|}\ M\ \overline{0 \cdots 0}$$

For Julius-ECB, the padded message P is formed as follows:

$$P = \overline{0 \cdots 01}\ \overline{IV}\ \overline{0\,|M|}\ \overline{0 \cdots 0}\ M \tag{1}$$

The padded message $P = P_1 || \cdots || P_l$ is then used to generate a seed using polynomial hashing with key $\delta \stackrel{\text{def}}{=} E_K(0)$:

$$seed = P_1\delta^{l-1} \oplus P_2\delta^{l-2} \oplus \cdots \oplus P_{l-1}\delta \oplus P_l.$$

Both Julius-CTR and Julius-ECB then encrypt the seed to produce a value $\mu = E_K(seed)$. In Julius-CTR, the i-th ciphertext block, $1 \leq i \leq l$, is given by $C_i = M_i \oplus E_K(\mu \oplus i)$, with $C_{l+1} = \mu$ serving as the authentication tag. In Julius-ECB, we have $C_1 = E_K(\mu)$ serving as the tag, and $C_{i+1} = E_K(\mu\delta^i \oplus M_i)$ for $1 \leq i \leq l$.

The Julius scheme is designed to provide resistance against nonce misuse [5].

B.2 Universal forgeries for Julius-ECB and CTR

We now describe how forge ciphertext and tags for arbitrary messages if the authentication key δ occurs as the root of an arbitrary forgery polynomial q. In this case, suppose we have obtained the encryption and tag for some message M. Then the polynomial hashing of M and $M+Q$ (following the notation of Sect. 1) will produce the same output *seed*, which by single (CTR) or double encryption (ECB) in turn leads to identical tags. Having observed this, we can recover the value of δ by the key search algorithm of Sect. 5.

Generating universal forgeries for Julius-CTR. Having recovered δ, we can calculate the value of *seed* for arbitrary messages. In the nonce-reuse model, we can therefore produce universal forgeries for any desired message $M = M_1, \ldots, M_l$ by calculating the *seed* for this message and then querying the Julius-CTR oracle on the auxiliary message $M' = M'_1, \ldots, M'_{l-1}, M'_l$ constructed by choosing arbitrary values for $M'_1, \ldots, M'_{l-1}$ and setting

$$M'_l := \mathit{seed} \oplus \delta^{l+3-1} \oplus \delta^{l+3-2} IV \oplus \delta^{l+3-3} |M'| \oplus \delta^{l+3-4} M'_1 \oplus \cdots \oplus \delta M'_{l-1},$$

which implies *seed'* = *seed* and hence $\mu' = \mu$. From this, we obtain a ciphertext $C'_1, \ldots, C'_l, C'_{l+1}$ with $C'_{l+1} = \mu' = \mu$, and since the CTR keystream is given by $E_K(\mu + i - 1) = C'_i \oplus M'_i$, we can construct $C_1, \ldots, C_{l+1}$ as

$$\begin{aligned} C_i &:= C'_i \oplus M'_i \oplus M_i, \quad 1 \le i \le l, \\ C_{l+1} &:= C'_{l+1} = \mu, \end{aligned}$$

which gives a valid ciphertext and tag for our target message M.

Generating universal forgeries for Julius-ECB. Even with knowledge of the authentication key δ, generating universal forgeries for Julius-ECB is more involved since *seed* is encrypted twice to form the tag, which means that the value of μ is not revealed to the adversary. The key observation here is that the same block cipher key K is used to derive $\delta = E_K(0)$ and $\mu = E_K(\mathit{seed})$. Since we know δ, we can carefully choose messages M' such that *seed'* = 0. This implies $\mu' = \delta$, which is known and allows queries to the internal block cipher calls of Julius-ECB (which are masked by powers of μ).

Denote by $p_\delta(M)$ the polynomial hash evaluation of the message M padded according to the rule from Eq. (1). Since the message is hashed last, for any n-bit block N we have $p_\delta(M||N) = p_\delta(M) \cdot \delta \oplus N$. To produce ciphertext and tag for an arbitrary message $M = M_1, \ldots, M_l$, we then proceed as follows:

1. Calculate the value of *seed* for M as $\mathit{seed} = p_\delta(M)$.
2. Query Julius-ECB on the auxiliary message $M' = (M'_1, M'_2)$ with

$$M'_1 := \delta^2 \oplus \mathit{seed}, \quad M'_2 := p_\delta(M'_1) \cdot \delta,$$

such that *seed'* = 0. Obtain the correct μ for M as $\mu := C'_2$.

3. Query Julius-ECB on the auxiliary message $M'' = (M''_1, \ldots, M''_l, M''_{l+1})$ with

$$M''_i := \delta^{i+1} \oplus \mu \cdot \delta^i \oplus M_i, \quad 1 \leq i \leq l,$$
$$M''_{l+1} := p_\delta(M''_1 || \cdots || M''_l) \cdot \delta,$$

such that *seed"* $= 0$. Obtain the target ciphertexts $C_i := C''_i$ for $2 \leq i \leq l$.

4. Query Julius-ECB on the auxiliary message $M''' = (M'''_1, M'''_2)$ with

$$M'''_1 := \delta^2 \oplus \mu, \quad M'''_2 := p_\delta(M'''_1) \cdot \delta,$$

such that *seed*$''' = 0$ and obtain the first target ciphertext (the tag) as $C_1 := C'''_2$.

Then $C_1, \ldots, C_{l+1}$ constitutes a valid ciphertext and tag for the message M.

C Weak-Key Analysis of COBRA

In this section, we briefly describe COBRA [3, 4]. We also note that we only describe the COBRA scheme defined for messages whose lengths are positive multiple of $2n$ since it is the scheme that we have analyzed in this paper.

C.1 Description of COBRA

COBRA is a Feistel network and misuse resistant online authentication cipher. It is a GCM-like authentication scheme – meaning that it uses one finite field multiplication plus one block cipher call per message block – with the additional feature of being secure under nonce repetition. One advantage of COBRA over the recent parallelizable authentication schemes is that it does not use the inverse block cipher during the decryption process since it employs a Feistel network. A schematic description of COBRA is given in Fig. 5 and Fig. 6.

As shown in the figures, COBRA uses a user key K during each block cipher call. It also uses the user key K for generating the secret values L and L' used in finite field multiplication during the encryption and authentication processes and the secret value J used in processing the associated data.

COBRA's encryption process takes as input: a message of length multiple to $2n$, a nonce N and associated data. It uses the secret value L to

perform polynomial hashing on the input message and also the user key K during each block cipher call to produce the corresponding ciphertext blocks. Before computing the tag, it computes the value U after performing polynomial hashing for the associated data using J as the key. For the purpose of our paper, let P_i denotes the hash polynomial value after the employment of ith Feistel network. Let S denote the polynomial hash value resulting from processing the associated data before generating U by calling the block cipher. The following formula gives the value of P_t – the polynomial hash value just before the employment of the last Feistel network – resulting during processing the $(2t+3)$-blocks message $M = M_1M_2||\cdots||M_{2t}M_{2t+1}||M_{2t+2}M_{2t+3}$.

$$P_t = L^{2t+2} + N \cdot L^{2t+1} + \sum_{i=0}^{2t-1} M_{i+1}L^{2t-i}$$

The following formula gives the value of S – the polynomial hash value resulting from processing the associated data before generating U by the block cipher.

$$S = J^4 \oplus A_1J^3 \oplus A_2J^2 \oplus A_410 * \oplus 2j$$

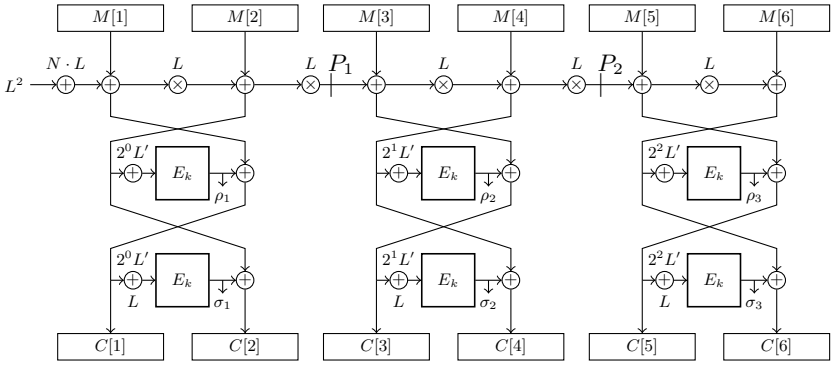


Fig. 5: Schematic description of COBRA's encryption process [4]

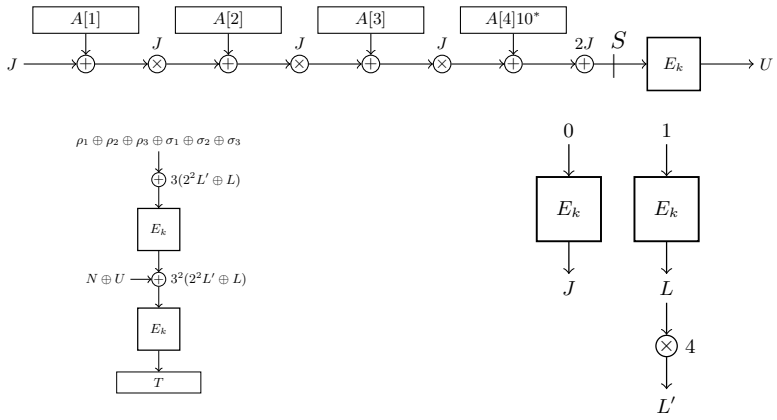


Fig. 6: Associated data processing, tag generation and secret keys generation in COBRA [4]

C.2 Two Weak Key Attacks on COBRA

In this section, we describe two weak key attacks on COBRA. Both of the attacks work under the assumption that the involved polynomial hash keys – only J in the first attack and only L in the second attack – are weak keys. Both of the attacks use the following observation to make a forgery.

Observation 4 (Forgery on COBRA).
Two different messages, M and M', have the same tag if $\sum_i \rho_i \oplus \sigma_i = \sum_i \rho'_i \oplus \sigma'_i$ and $N \oplus U = N' \oplus U'$.

The first attack performs a forgery attack by targeting the possible weakness of the secret polynomial hash key J during the process of the associated value and the generation of the value U which is a secret value since it is not in COBRA's output.

The second attack starts by firstly finding a distinguisher assuming that the secret polynomial hash key L is a weak key and secondly recovering the weak key L and consequently the key $L' = 4L$. Then the attack uses the following observation which will be explained later to make arbitrary forgeries for any message/ciphertext/tag and to generate ciphertext/tag for a new message with more than two blocks without knowing the block cipher's secret key K.

Observation 5 (Querying COBRA's Block Cipher). *Recovering the weak keys L and L' enables us to query COBRA's internal block cipher without the knowledge of its secret key K.*

First weak key attack Suppose that we are processing the same message with the same nonce but with two different associated data: the first is $A = A_1||\cdots||A_s$ and the other is $A + Q$, where $Q = q_1||\cdots||q_r$ and all the q_i's represent the coefficients of our forgery polynomial $q(X) = \sum_{i=1}^{r} q_i X^i$. So we have the following two inputs $I_1 = M||A||N$ and $I_2 = M||A \oplus Q||N$. Then we always get the same cipertext blocks but different tags.

However, assuming that the secret value J is weak according to our forgery polynomial we can get the same value S right before the block cipher call during processing the associated data and generating the value U for both inputs I_1 and I_2. This means that we will get the same U for both inputs I_1 and I_2 and consequently the same tags for two reasons: firstly because we are using the same plaintext which means that the difference in the input values for the first block cipher call $\sum_i \rho_i \oplus \sigma_i$ during the tag generation for both inputs I_1 and I_2, and secondly because we are using the same nonce, so the difference in the input values to the second block cipher call $N \oplus U$ during the tag generation will also be zero for both I_1 and I_2 and therefore the difference in the tags output for both inputs I_1 and I_2 will be zero which means that we get the same tag for both inputs I_1 and I_2.

To summarize, assuming the secret value J is weak according to our forgery polynomial $q(X)$ we can get endless number of forgeries by repeating two times the authentication of any message and any nonce of our choice using two different associated data yielding the same value U as described above.

Second weak key attack

Recovering the weak key L Suppose that the secret value L is a weak key according to our forgery polynomial $q(X) = \sum_{i=1}^{r} q_i X^i$, i.e. $q(L) = 0$, then two different messages $M = M_1 M_2||\cdots||M_{2t}M_{2t+1}||M_{2t+2}M_{2t+3}$ and $M' = M \oplus Q$ where $Q = q_1 q_2||\cdots||q_{2t}q_{2t+1}||q_{2t+2}q_{2t+3}$ might collide at P_t right before the employment of the last Feistel network. Now if the last two input blocks for both messages are equal, i.e. $q_{2t+2} = M_{2t+2}$ and $q_{2t+3} = M_{2t+3}$, then we will get the same ciphertext blocks C_{2t+2} and C_{2t+3} for both input messages M and M'. As a result, we will be assured that these two messages have already collided at P_t which means that the secret key L is weak key, i.e. it is one of the roots of our forgery polynomial $q(x)$. In the following we describe how to recover L and how to query the COBRA's internal block cipher.

Using a binary search algorithm, one can find L from the roots of the forgery polynomial $q(X)$ using only few queries to COBRA. Consequently, we can easily find the secret value L' since it is equal to $4L$. Now we know almost everything to produce the ciphertext blocks and the tag except the block cipher's secret key and the secret value J used during processing the associated data and the generation of U. However, if we are able to query COBRA's internal block cipher, then we can nicely produce the ciphertext blocks and the tag without knowing the block cipher's secret key. Next, we will explain exactly how to do this.

Querying COBRA's internal block cipher Suppose we want to find the encryption of the plaintext x using COBRA's internal block cipher, $E_K(x)$ where K is the secret key which is unknown to us. Now from Fig. 5, we see that the following equation holds:

$$E_K(L^3 \oplus N \cdot L^2 \oplus M_1 L \oplus M_2 \oplus 2^0 L') = \rho_1 = L^2 \oplus NL \oplus M_1 \oplus C_1$$

Since we know L and L', using the above equation we can choose the values N, M_1 and M_2 such that:

$$L^3 \oplus NL^2 \oplus M_1 L \oplus M_2 \oplus 2^0 L' = x$$

If the above equation holds, then

$$E_K(x) = \rho_1 = C_1 \oplus L^2 \oplus NL \oplus M_1$$

One can make the above equations simpler by setting $N = 0$ and $M_1 = 0$. Then we choose M_2 in order to get

$$x = L^3 + M_2 + 2^0 L'$$

Thus,

$$E_K(x) = \rho_1 = C_1 \oplus L^2$$

Now setting $x = 0$, gives us $J = E_K(0)$ and consequently we will be able to compute U.

Next, we will show how to use our ability to query COBRA's block cipher in order to construct universal forgeries.

Universal forgeries Suppose we want to find the ciphertext blocks and tag of the message $M = M_1||M_2||\cdots||M_{t-1}||M_t$. Then, we query COBRA several times for different messages $M' = M'_1 M'_2$ where M'_1 and M'_2 are chosen as follows.

To find the ciphertext block C_1, we need to find $E_K(L^3 \oplus NL^2 \oplus M_1L \oplus M_2 \oplus 2^0L')$ by querying COBRA's block cipher as outlined above. More specifically, we ask for the ciphertext blocks of $M' = M'_1||M'_2$ with nonce N', where

$$L^3 \oplus N'L^2 \oplus M'_1L \oplus M'_2 \oplus 2^0L' = L^3 \oplus NL^2 \oplus M_1L \oplus M_2 \oplus 2^0L'$$

Setting $M'_1 = 0, N' = 0$ in the above equation, we get

$$L^3 \oplus M'_2 \oplus 2^0L' = L^3 \oplus NL^2 \oplus M_1L \oplus M_2 \oplus 2^0L'$$

This gives us $M'_2 = NL^2 \oplus M_1L \oplus M_2$. Now we query COBRA for $M' = 0||NL^2 \oplus M_1L \oplus M_2$, we get $C'_1 = \rho'_1 \oplus L^2 \oplus N'L \oplus M'_1 = \rho'_1 \oplus L^2$. Note that $\rho_1 = \rho'_1$. Therefore $\rho_1 = C'_1 \oplus L^2$ and consequently

$$C_1 = \rho_1 \oplus L^2 \oplus NL \oplus M_1$$

Now using our knowledge of ρ_1, we can find C_2 by asking for $E_K(L^2 \oplus NL \oplus M_1 \oplus \rho_1 \oplus 2^0L' \oplus L)$ as described above, say we get σ_1. Then $C_2 = \sigma_1 \oplus L^3 \oplus NL^2 \oplus M_1L \oplus M_2$. Repeating this procedure for any two blocks $M_{2i-1}||M_{2i}$ we find all the ciphertext blocks of $M = M_1||M_2||\cdots||M_{t-1}||M_t$.

D OpenSSL GCM Implementation using different IV sizes under the same secret key

Here we explain that some users could use different IV sizes under the same secret key. From the figures below, one can see that Test Case 3 and Test Case 6 share the same secret key $K3 = K6$ and therefore share the same hash key but they use different IVs (which are nonces according to our description). Test Case 3 uses a nonce $IV3$ with $|IV3| = 96$-bit while Test Case 6 uses a nonce $IV6$ with $|IV6| = 480$-bit [18].

```
1614 /* Test Case 3 */
1615 #define A3 A2
1616 static const u8 K3[]=  {0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08},
1617                 P3[]=  {0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
1618                         0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
1619                         0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
1620                         0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,0xba,0x63,0x7b,0x39,0x1a,0xaf,0xd2,0x55},
1621                 IV3[]= {0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,0xde,0xca,0xf8,0x88},
1622                 C3[]=  {0x42,0x83,0x1e,0xc2,0x21,0x77,0x74,0x24,0x4b,0x72,0x21,0xb7,0x84,0xd0,0xd4,0x9c,
1623                         0xe3,0xaa,0x21,0x2f,0x2c,0x02,0xa4,0xe0,0x35,0xc1,0x7e,0x23,0x29,0xac,0xa1,0x2e,
1624                         0x21,0xd5,0x14,0xb2,0x54,0x66,0x93,0x1c,0x7d,0x8f,0x6a,0x5a,0xac,0x84,0xaa,0x05,
1625                         0x1b,0xa3,0x0b,0x39,0x6a,0x0a,0xac,0x97,0x3d,0x58,0xe0,0x91,0x47,0x3f,0x59,0x85},
1626                 T3[]=  {0x4d,0x5c,0x2a,0xf3,0x27,0xcd,0x64,0xa6,0x2c,0xf3,0x5a,0xbd,0x2b,0xa6,0xfa,0xb4};
```

Fig. 7: Test Case 3 [18]

```
1654 /* Test Case 6 */
1655 #define K6 K5
1656 #define P6 P5
1657 #define A6 A5
1658 static const u8 IV6[]= {0x93,0x13,0x22,0x5d,0xf8,0x84,0x06,0xe5,0x55,0x90,0x9c,0x5a,0xff,0x52,0x69,0xaa,
1659                         0x6a,0x7a,0x95,0x38,0x53,0x4f,0x7d,0xa1,0xe4,0xc3,0x03,0xd2,0xa3,0x18,0xa7,0x28,
1660                         0xc3,0xc0,0xc9,0x51,0x56,0x80,0x95,0x39,0xfc,0xf0,0xe2,0x42,0x9a,0x6b,0x52,0x54,
1661                         0x16,0xae,0xdb,0xf5,0xa0,0xde,0x6a,0x57,0xa6,0x37,0xb3,0x9b},
1662                 C6[]=  {0x8c,0xe2,0x49,0x98,0x62,0x56,0x15,0xb6,0x03,0xa0,0x33,0xac,0xa1,0x3f,0xb8,0x94,
1663                         0xbe,0x91,0x12,0xa5,0xc3,0xa2,0x11,0xa8,0xba,0x26,0x2a,0x3c,0xca,0x7e,0x2c,0xa7,
1664                         0x01,0xe4,0xa9,0xa4,0xfb,0xa4,0x3c,0x90,0xcc,0xdc,0xb2,0x81,0xd4,0x8c,0x7c,0x6f,
1665                         0xd6,0x28,0x75,0xd2,0xac,0xa4,0x17,0x03,0x4c,0x34,0xae,0xe5},
1666                 T6[]=  {0x61,0x9c,0xc5,0xae,0xff,0xfe,0x0b,0xfa,0x46,0x2a,0xf4,0x3c,0x16,0x99,0xd0,0x50};
```

Fig. 8: Test Case 6 [18]

```
1614 /* Test Case 3 */
1615 #define A3 A2
1616 static const u8 K3[]=  {0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08}

1628 /* Test Case 4 */
1629 #define K4 K3

1643 /* Test Case 5 */
1644 #define K5 K4

1654 /* Test Case 6 */
1655 #define K6 K5
```

Fig. 9: Test Case 3 and Test Case 6 share the same secret key $K3 = K6$ [18]

E Further Forgery Strategies

Constructing shorter (blind) forgeries Having generated a polynomial hash collision, and therefore recovered the universal hash keys L_{top} and L_{bot}, we can freely produce blind forgeries for any ciphertext-tag pair of at least 2 blocks length. Suppose we have a ciphertext $C = C_1, \ldots, C_m$ with corresponding tag T for $m \geq 2$. Then T is also a valid tag for $C' = (C_1 \oplus \Delta, C_2 \oplus \Delta \cdot L_{\text{bot}}, C_3, \ldots, C_m)$ and the same nonce, since during the decryption process, we have $Y_2' = C_2 \oplus \Delta \cdot L_{\text{bot}} \oplus (C_1 \oplus \Delta \oplus \tau \cdot L_{\text{bot}}) \cdot L_{\text{bot}} = C_2 \oplus (C_1 \oplus \tau \cdot L_{\text{bot}}) \cdot L_{\text{bot}} = Y_2$. Therefore $X_2' = X_2$ as well, and this collision is preserved by having $C_i' = C_i$ for $i > 2$.

Recovering the secret value L The secret value L is used during the generation of the intermediate tag τ. Recovering the secret value L means that we will be able to process the header (associated data and nonce) and generate a new τ and consequently have a total control over the POET scheme.

Using the blind forgery outlined above, we can decrypt the recovered secret value τ. Now assuming that the recovered τ was generated using an integral one block header H_1, we get $3L + \hat{\tau} = H_1$, where $\hat{\tau}$ as shown in Fig. 10 is the value right before the last block cipher call that generates τ. Therefore $L = 3^{-1}(\hat{\tau} \oplus H_1)$.

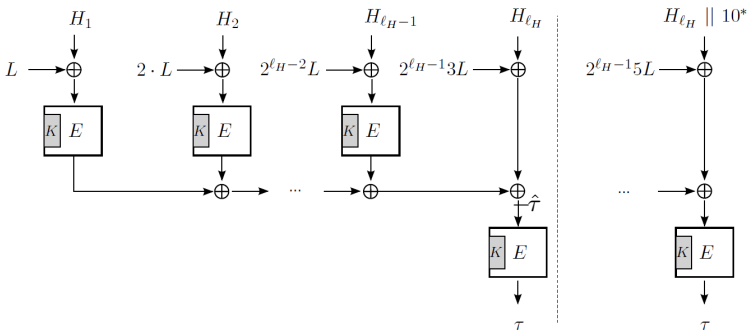


Fig. 10: Generation of τ in POET [2]

In the following, we describe how to find $\hat{\tau}$. Let $C = C_1||C_2||C_3$ be a valid ciphertext with tag T. Then as described above, the ciphertext

$$C' = (C'_1||C'_2||C'_3) = (C_1 \oplus \Delta||C_2 \oplus \Delta L_{\text{bot}}||C_3)$$

where Δ can be any difference, yields the same tag T. Now setting $\Delta = C_1 \oplus \tau \oplus \tau L_{\text{bot}}$, we get $Y'_1 = \tau$ since $Y'_1 = C'_1 \oplus \tau L_{\text{bot}}$. Now when querying for the decryption and authentication of C' with tag T, the POET scheme will return $M' = M'_1||M'_2||M'_3$. Now since $M'_1 = \tau L_{\text{top}} \oplus X'_1$ where $X'_1 = \hat{\tau}$ since X'_1 is the value before the block cipher that generates $Y'_1 = \tau$, then $\hat{\tau} = M'_1 \oplus \tau L_{\text{top}}$.

Once we get the secret value L, we can use our ability to query POET's internal block cipher (using the recovered τ) to generate a new τ and thus gain full control over the POET scheme.

Constructing meaningful (targeted) forgeries We can also leverage collisions in the polynomial hash to produce targeted forgeries with complete control over the differences in the first $m-2$ message blocks with a complexity of only two encryption queries per forgery. The length of these queries is one block longer or shorter than the length of the message we want to provide a forgery for, and can be as short as two blocks. Being able to produce forgeries for arbitrary messages with *chosen* differences in the first $m-2$ message blocks already comes close to a universal forgery.

We first describe the procedure for the case of m-block messages with $m \geq 3$ and deal with $m = 2$ later.

Let $m \geq 3$, $M = M_1, \ldots, M_{m-1}, M_m$ denote the target message, $(C_1, \ldots, C_m; T)$ its encryption and tag and $\nabla_1 \neq 0, \ldots, \nabla_{m-2} \neq 0$ the desired differences in $M_1, \ldots, M_{m-2}$. We then produce a valid ciphertext with equal tag T for $M_1 \oplus \nabla_1, M_2 \oplus \nabla_2, \ldots, M_{m-1} \oplus \nabla_{m-1}, M_m$, with uncontrollable ∇_{m-1}.

Step 1: Recovering L_{top}. We first note that the collisions in C_m and T from the generic forgery can be used to detect the collision in $g_t(X)$ and therefore whether a root of q was used as L_{top}. We can then use our binary search key recovery algorithm to recover the value of L_{top} with about $128 - \log_2(m) + 1$ verification queries.

Step 2: Querying for prefix. Once L_{top} is known, we can use this to query for a prefix of our forged message as follows. Define

$$\nabla_{m-1} \stackrel{\text{def}}{=} \begin{cases} \nabla_1 \cdot L_{\text{top}} & \text{if } m = 3 \\ \nabla_1 \cdot (L_{\text{top}})^{m-2} \oplus \cdots \oplus \nabla_{m-2} \cdot L_{\text{top}} & \text{if } m > 3. \end{cases}$$

Form $m-1$-block messages $M_1, \ldots, M_{m-1}$ and $M'_1, \ldots, M'_{m-1}$ with $M'_i \stackrel{\text{def}}{=} M_i \oplus \nabla_i$, and obtain their encryptions $C_1, \ldots, C_{m-1}$ and $C'_1, \ldots, C'_{m-1}$. Denote the ciphertext differences by $\Delta_i \stackrel{\text{def}}{=} C_i \oplus C'_i$. Note that ∇_{m-1} is chosen to eliminate the differences introduced by the previous message blocks, yielding $X_{m-1} = X'_{m-1}$ and therefore also $Y_{m-1} = Y'_{m-1}$, a collision on the internal state of POET. This situation is illustrated in Fig. 11.

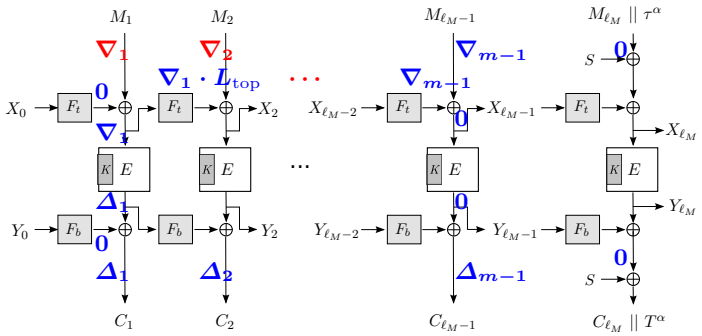


Fig. 11: Constructing targeted forgeries for POET. Freely chosen differences are indicated in red, uncontrolled differences in blue.

Step 3: Constructing the forgery. The knowledge of the "right pair" $(M_1, \ldots, M_{m-1})$ and $(M'_1, \ldots, M'_{m-1})$ for our internal state collision differential now enables us to construct the desired forgery. Query POET on the target message $M = (M_1, \ldots, M_{m-1}, M_m)$ and obtain ciphertext $C = (C_1, \ldots, C_m)$ and tag T. Then $(C_1 \oplus \Delta_1, \ldots, C_{m-1} \oplus \Delta_{m-1}, C_m; T)$ is a valid ciphertext-tag pair for $(M_1 \oplus \nabla_1, \ldots, M_{m-1} \oplus \nabla_{m-1}, M_m)$. Since this message was not queried before, this constitutes a valid forgery.

Constructing two-block forgeries. If the target message is two blocks long, we cannot use the above procedure since we need at least a two-block prefix query to achieve the internal state collision. For $m = 2$, we would then already have queried the message forged in Step 3 in Step 2. We can however follow an entirely analogous procedure by simply extending the queries in Step 2 by one arbitrary block Z. Let ∇_1 be the chosen difference for the first message block. Compute L_{top} as described in Step 1. In Step 2, we then obtain the encryption of (M_1, M_2, Z) as (C_1, C_2, C_Z) and $(M_1 \oplus \nabla_1, M_2 \oplus \nabla_1 \cdot L_{\text{top}}, Z)$ as (C'_1, C'_2, C'_Z), and then construct the forgery in Step 3 as (C'_1, C'_2).

F Forgery polynomials constructed using the multiplicative structure

In case $q = 2^{128}$, Proposition 1 gives rise to the following family of polynomials whose roots partition $\mathbb{F}_{2^{128}} \backslash \{0\}$:

Example 1. Let γ be a primitive element of $\mathbb{F}_{2^{128}}$ and n be a divisor of $2^{128} - 1$. Further define $m := (2^{128} - 1)/n$. Then the sets of roots of the polynomials $x^m - \gamma^{im}$ with $0 \leq i \leq n - 1$ partition $\mathbb{F}_{2^{128}} \backslash \{0\}$.

The number $2^{128} - 1$ is relatively easy to factor, since we can directly obtain the partial factorization

$$2^{128} - 1 = (2^{64} + 1)(2^{32} + 1)(2^{16} + 1)(2^8 + 1)(2^4 + 1)(2^2 + 1)(2^1 + 1).$$

The numbers $2^{2^k} + 1$ with $k \in \{0, 1, 2, 3, 4\}$ are prime (so-called Fermat primes), while $2^{32} + 1 = 641 \cdot 6700417$ and $2^{64} + 1 = 274177 \cdot 67280421310721$. This gives an easy way to determine all possibilities for n and m.