

Speeding up Sumcheck using Algebraic Geometry

Rahul Dalal¹, Benedikt Bünz², Tamir Hemo¹, and Ron D. Rothblum¹

¹Succinct*

²NYU, Espresso Systems[†]

October 15, 2026

Abstract

The sumcheck protocol, a ubiquitous component of modern proof-systems, enables proving the sum of evaluations of a low-degree multivariate polynomial. In theory, sumcheck has been generalized from polynomials to (tensors of) any *multiplication code*, a code whose pointwise products of codewords lie in a related product code. All practical systems nonetheless specifically use polynomials. For proof-systems over binary fields that target *Boolean circuits*, this choice has a real cost: the product of two binary messages under Reed–Solomon is no longer binary. As a result, the prover’s work in the first, and most expensive, round of sumcheck (the so-called *univariate skip*) grows by a logarithmic factor.

We present a new concrete multiplication code based on algebraic geometry that is *binary-preserving*: the product of two encoded binary messages is again a binary message. Specifically, we construct an explicit multiplication code over $\mathbb{F}_{2^{128}}$ that carries 64-bit messages to binary product messages of length 222, and whose product code has relative distance greater than $1 - 2^{-119}$. The part of the product message that the prover actually sends in sumcheck is more than $3\times$ smaller than with Reed–Solomon, as used in practice. The code comes from a genus-95 curve over $\mathbb{F}_2$ with 64 rational points, built as a $(\mathbb{Z}/2)^3$ Artin–Schreier cover of a genus-4 curve.

We implemented the resulting protocol, which we call the *AG skip*, inside the Flock proof-system. For 2^{32} constraints, it speeds up the skip round by $2.36\times$ and the whole zerocheck, which was the prover’s most expensive component, by $1.65\times$.

*Emails: dalal.rahul@gmail.com, tamirhmo@gmail.com, rothblum@gmail.com

[†]Email: bb@nyu.edu

Contents

1	Introduction	3
1.1	Multiplication codes	5
1.2	Our contribution	6
1.3	On AI usage	7
1.4	Related work	7
1.5	Organization	8
2	The sumcheck protocol: A coding-theoretic perspective	8
2.1	The inner-product sumcheck	8
2.2	Coping with binary data	11
2.3	From inner products to the zerocheck	11
3	Evaluation codes from explicit equations	12
3.1	Preliminaries: function fields and equation models	13
3.2	Evaluation codes	15
3.3	Minimum distance	16
3.4	The size of the evaluation set	20
3.5	Efficient sampling	20
3.6	Summary: algebraic evaluation codes	20
4	Searching for algebraic evaluation codes	21
4.1	Curves, points at infinity, and divisors	21
4.2	Local coordinates and jet interpolation	22
4.3	Dimension, distance, and point counts	23
4.4	Abstract curve-and-divisor construction	23
4.5	Summary: the curve-and-divisor construction	25
5	AI search for explicit parameters	25
5.1	Prompting	25
5.2	Validation	26
A	The explicit instance	29
A.1	Curve and divisor data	29
A.2	Code data and maps	31
A.3	Machine-readable representation	32
A.4	Sampler for the explicit model	33
B	Independent validation of the explicit code	33
B.1	Checking the evaluation-code hypotheses	33
B.2	Norm and distance	34
C	Comparison with the asymptotic behavior	35

1 Introduction

The goal of probabilistic proof-systems, such as *probabilistically checkable proofs (PCPs)* or *interactive oracle proofs (IOPs)*, is to encode a computation so that any deviation from an honest execution can be easily detected by a verifier. Error-correcting codes solve the analogous problem one level down, protecting *information* rather than computation. In that light, it is perhaps not surprising that error-correcting codes play an extremely central role in proof-system design.

Polynomials (equivalently, the Reed–Solomon code) are the classical instrument. The standard paradigm encodes the computation transcript as a low-degree polynomial and hands the verifier its evaluations (e.g., as an oracle). The distance property of the Reed–Solomon code says that two distinct low-degree polynomials agree on only a few points. The encoding of an invalid transcript is therefore far from the encoding of every valid one and, glossing over a great amount of detail, a handful of random spot-checks suffices to expose it.

Once the role of the encoding is put this way, it is natural to ask whether other codes can take the place of the Reed–Solomon code and, more ambitiously, whether a better code yields a better proof-system. A line of theoretical work [Mei13, BKK⁺16, RR24, BCG20, HR22, RR25, BHI⁺24, ARR25], as well as more applied work [GLS⁺23, XZS22, BCF⁺25, NST24, BMMS25a, NA25, BMMS25b, BFRW25, GNR26, QGZ26], has given a positive answer to both questions.

So far, in the applied setting, this substitution has been made mostly in one place. It is standard to assemble a proof-system out of two largely independent components: an information-theoretic protocol, which reduces the statement being proven to a handful of claims about the prover’s data, and a *polynomial commitment scheme* (PCS), which lets the prover commit to that data and later answer those claims. The practical works cited above all change the error-correcting code inside the commitment layer. This is one concrete instance of the general theme: there, for example, a code that is faster to encode translates directly into a faster prover.

This raises a natural question: can changing the code used by the information-theoretic protocol itself, specifically by the sumcheck protocol and the zerocheck built on top of it, also lead to concrete efficiency gains? In this work we show that the answer is yes, for proof-systems over binary fields: we replace the Reed–Solomon code in the first, and most expensive, round of sumcheck with a new code based on algebraic geometry. The code multiplies two 64-bit messages into a 222-bit *binary* product message, of which the prover actually sends only 158 bits, more than $3\times$ fewer than with Reed–Solomon, as used in practice. Implemented inside the Flock proof-system [BRW26], this speeds up that round by $2.36\times$ and the entire zerocheck by $1.65\times$.

The sumcheck protocol. The sumcheck protocol [LFKN92] is among the most celebrated and heavily used components in proof-system design. One formulation of it will be particularly convenient. The prover holds two low-degree m -variate polynomials $f, g : \mathbb{F}^m \rightarrow \mathbb{F}$ over a finite field $\mathbb{F}$, together with a scalar $v \in \mathbb{F}$, and claims that

$$\sum_{x \in \{0,1\}^m} f(x) \cdot g(x) = v.$$

The protocol reduces this claim, which ranges over exponentially many terms, to claims about the values of f and g at a randomly chosen point. A related variant, called *zerocheck*, allows one to check that the pointwise product of f and g is all zero (on the Boolean hypercube).

Sumcheck can be presented more abstractly from a coding-theoretic perspective. Suppose that a and b are two vectors. The protocol reduces a claim about their *inner product* $\sum_i a(i) \cdot b(i) = v$ to *separate* claims about the individual encodings of a and of b under some linear error-correcting code. A code that supports such a procedure is loosely referred to as *sumcheckable* [RR24, RR25].

Multiplication codes. Which codes, then, support sumcheck? Meir [Mei13] identified the property that the sumcheck protocol actually depends on, and subsequent work [RR24, RR25] developed it: sumcheck works for (the tensor product of) essentially any *multiplication code*: a linear code in which the pointwise product of any two codewords lands in a somewhat larger linear *product code*. Reed–Solomon is the canonical example:

the product of two polynomials of degree d has degree $2d$, so the product code is again a Reed–Solomon code (of somewhat larger degree). See Section 1.1 for the formal treatment.

How could a different multiplication code help in the sumcheck protocol? After all, within the sumcheck protocol itself the prover does not explicitly perform a full encoding of a large message (e.g., an FFT in the case of Reed–Solomon). Indeed, the main computational task that the prover performs in sumcheck (repeatedly) is, given descriptions of two very small polynomials, to find the polynomial corresponding to their product. In the generalization to multiplication codes, which will be described in detail below, the task is to compute the message corresponding to the product of two codewords. How could we improve on Reed–Solomon in this setting? It turns out that a major saving can be achieved in the context of proof-systems over *binary fields*, on which we elaborate next.

The binary bottleneck. Binary fields (i.e., extensions of the two-element field $\mathbb{F}_2$) have long been used in theoretical works on proof-systems. More recently, a line of *practical* works [DP25, Sou, BRW26] has used binary fields to implement highly efficient proof-systems geared toward proving the correctness of *Boolean circuits*. Indeed, for such circuits, it is natural to express the computation transcript in bits, and, skipping some details, the sumchecks that arise there have messages over *bits*, that is, over $\mathbb{F}_2$.

The sumchecks that are utilized in these works use the Reed–Solomon code (i.e., univariate polynomials). This introduces a cost in terms of the alphabet size: overall the code needs to have relative distance close to 1 (for soundness) which requires a large field, so the codewords sent in the sumcheck protocol live over a large extension field $\mathbb{F}$ of $\mathbb{F}_2$. By itself, this is cheap, since the verifier inspects only a handful of codeword symbols. The real cost sits one level down, in the messages that the prover computes and sends. Glossing over details, in the coding-theoretic view of sumcheck, the prover’s messages are *product messages*: messages whose encodings are pointwise products of codewords. For Reed–Solomon, the product message of two binary messages is not binary—if the messages consist of k bits, the product message requires entries in a field of size at least $\Omega(k)$. This means that the prover’s bulk arithmetic and communication expand by a factor of roughly $\log_2(k)$ compared to the underlying binary data.

This overhead is concentrated in a single round. The traditional presentation of sumcheck assumes that the verifier can access the *multilinear extensions* of a and b : the multilinear polynomials that agree with a and b on the Boolean hypercube. Focusing on the case in which a and b take values in a base field (our case of $\mathbb{F}_2$ is an extreme form of that), Gruen [Gru24] showed a more efficient variant in which the verifier accesses a different polynomial encoding of a and b : a multivariate polynomial that has a somewhat large degree in its first variable and degree 1 in each of the rest. This technique, popularly known as the *univariate skip*, collapses a block of variables into a single first round whose prover message is a Reed–Solomon product message. That round is by far the most expensive round of the sumcheck, and it is exactly where the $\log_2(k)$ blowup is paid.

Binary-preserving codes and algebraic geometry. Given the coding-theoretic perspective of sumcheck, it is therefore natural to ask whether we could use a different multiplication code whose product messages remain over bits. Loosely speaking, we call a multiplication code *binary-preserving* if the message underlying the product of two encoded binary messages is itself binary and *product-systematic* if pointwise multiplication on the codeword side can be related to pointwise multiplication on the message side (see Section 1.1 for formal definitions). For Reed–Solomon, the product code has essentially the smallest dimension one could hope for, but there is no encoding map where the code is binary-preserving and product-systematic concurrently. We want all properties at once: a product code of small dimension, binary preservation, product-systematicity, and relative distance close to 1.

Good multiplication codes are rare; essentially all known examples are algebraic. In particular, for a generic linear code of dimension k , the products of codewords likely span a space of dimension about $k^2/2$, which is exceedingly large. For the product code to have dimension close to $2k$, the codewords must be closed under multiplication for a structural reason, and essentially the only known source of such structure is *algebraic geometry*: take the codewords to be evaluations of functions drawn from a space $L(D)$ of functions on an algebraic curve with prescribed poles; their pointwise products are automatically evaluations of functions

in $L(2D)$. Reed–Solomon is the genus-zero instance, with $L(D)$ the polynomials of bounded degree.

To obtain both binary preservation and product-systematicity, our construction interpolates a message x as a function $f_x \in L(D)$ whose values at k distinct $\mathbb{F}_2$ -rational points P_i are x_i . With an invertible interpolation map defined over $\mathbb{F}_2$, binary messages give functions defined over $\mathbb{F}_2$, and their products have binary coordinates in $L(2D)$. Moreover, $(f_x f_y)(P_i) = x_i y_i$, so we can choose the first k product coordinates to be these values. The codewords themselves are obtained by evaluating at many points over the large extension field $\mathbb{F}$, which provides the distance needed for soundness.

This direct approach needs k distinct $\mathbb{F}_2$ -rational points, whereas $\mathbb{P}^1(\mathbb{F}_2)$ has only three points, so it cannot accommodate $k = 64$.¹ Curves of higher genus g can have many more $\mathbb{F}_2$ -rational points. With a suitable non-special interpolation divisor, Riemann–Roch gives $k_* = 2k + g - 1$: the extra rational points come at the cost of g additional product coordinates. This is the tradeoff that our construction exploits.

Remark 1.1. *Curves enter here only as a source of functions from which to build a linear code, never as a source of computational hardness. All of our proof-systems are information-theoretic; in particular, nothing relies on the discrete logarithm problem (e.g., over elliptic curves) that Shor’s algorithm solves.*

1.1 Multiplication codes

Let $\mathbb{F}$ be a finite field. A linear error-correcting code $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$ is a *multiplication code* if there exists a linear code $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$ such that, for every pair of codewords x, y of C , the Hadamard product $x \circ y$ (i.e., their pointwise multiplication) is a codeword of C_* . We refer to C as the *base code* and C_* as the *product code* of C . A key property is the *multiplication rate*, defined as k/k_* .

We seek multiplication codes with distance close to 1 and with multiplication rate as close as possible to $1/2$. For example, the Reed–Solomon code of degree- d polynomials is a multiplication code, whose product code is the Reed–Solomon code of degree- $2d$ polynomials; its multiplication rate is $\frac{d+1}{2d+1} \approx 1/2$.

For $x, y \in \mathbb{F}^k$, we define their message product:

$$x \times y := C_*^{-1}(C(x) \circ C(y)),$$

as a k_* -dimensional vector. We say that the multiplication code is *product-systematic* if the first k coordinates of $x \times y$ are equal to $x \circ y$, and in that case we refer to the remaining $k_* - k$ coordinates as the *suffix* of $x \times y$. This is only a property of the message product; neither code is required to be systematic as an encoding.

We say that the multiplication code has *product complexity* c if, given $x, y \in \mathbb{F}^k$, the product message $x \times y$ can be generated in time c .

Binary-preserving multiplication codes. In this work we focus on *binary extension fields* and on multiplication codes that are *binary-preserving*: if the original messages $x, y \in \mathbb{F}_2^k$ are bits, then the entire k_* -dimensional vector $x \times y$ is over bits.

As discussed above, the Reed–Solomon code is *not* binary-preserving, even with a systematic encoding: the product polynomial’s values off the systematic set generically lie outside $\mathbb{F}_2$. In contrast, Reed–Muller codes consisting of multivariate polynomials of low total degree over $\mathbb{F}_2$ are binary-preserving, but have rather poor rate and product rate.

We summarize all properties needed by the code in the following definition:

Definition 1.2. A multiplication code C over $\mathbb{F}$ with product code C_* and multiplication map $\times$ is *sumcheck-friendly*, with respect to security parameter λ and product complexity c , if:

- It is binary-preserving.
- It is product-systematic.
- C_* has relative minimum distance $\delta_* > 1 - 2^{-\lambda}$.
- C has product complexity c .

¹This restriction concerns direct interpolation of one bit per rational point. Higher-degree places combined with RMFEs give a different route on $\mathbb{P}^1$, discussed in Section 1.4.

1.2 Our contribution

Our main result is a new algebraic-geometry-based multiplication code with surprisingly² good parameters. We emphasize that this is a specific concrete code found on a particular algebraic curve: we construct an explicit binary-preserving multiplication code over $\mathbb{F} = \mathbb{F}_{2^{128}}$ carrying 64-bit messages to product messages of length 222. The *suffix* of the product message, i.e., the coordinates beyond the prefix $x \circ y$, which are what the prover actually sends in sumcheck, is $222 - 64 = 158$ bits. The product code has relative distance greater than $1 - 2^{-119}$, but establishing this relies on the geometric construction. A more elementary analysis of the same code, which uses only its explicit equations and not the geometric construction, yields a weaker, but still likely sufficient, bound of $1 - 2^{-116}$. The exact extension matrix and the product-code evaluation data are recorded in Section A.3.

Our code comes from a particular algebraic curve X over $\mathbb{F}_2$ with at least 64 $\mathbb{F}_2$ -points, together with a divisor D on X , through the construction of Theorem 4.1; its parameters can be computed through Riemann–Roch. Specifically, when D satisfies a technical property called being non-special (as in our case), a message of length 64 yields a product message of length

$$k_* = 127 + g_X,$$

where g_X is the genus of X . For comparison, Reed–Solomon attains $k_* = 2 \cdot 64 - 1 = 127$, though, as discussed above, not over bits. Thus, the entire gap between our product message and that of Reed–Solomon is exactly the genus, and minimizing the overhead is precisely the problem of finding a curve over $\mathbb{F}_2$ with many rational points and small genus. Our realization uses a genus-95 curve with 64 rational points, presented as a $(\mathbb{Z}/2)^3$ Artin–Schreier cover of a genus-4 curve, giving $k_* = 127 + 95 = 222$. The Serre–Oesterlé bound puts the genus of any such curve at 89 or more, so this route cannot bring k_* below $127 + 89 = 216$; we do not know whether genus 95 can be beaten at all.

It is useful to compare this new multiplication code to what could be obtained by relying on Reed–Solomon. Encoding a 64-bit message with Reed–Solomon means that at the very least the product message will live in the 6-bit field. The product of two such polynomials is represented by 127 coefficients, and so the suffix of the product message is $63 \times 6 = 378$ bits. In practice [DP25, BRW26] it is far more convenient to use 8-bit field elements (as it allows storing elements as bytes), leading to a suffix of 64 bytes, i.e., 512 bits. Thus, the size of the suffix (which, as we shall see, is the main bottleneck in applications) is more than $3 \times$ smaller using our AG code.

The AG skip. We use our code as a drop-in replacement for Reed–Solomon in the univariate skip, and refer to the resulting variant as the *AG skip*. Thus, the verifier accesses a particular encoding of a and b : a tensor of our AG code with the multilinear extension code.

To measure what this buys, we implemented both skips inside Flock [BRW26] and timed the prover-side phases of its zerocheck, with the $\mathbb{F}_2$ witnesses packed at one bit per coefficient. Both implementations skip the same 6 variables, so the skipped block is exactly the $2^6 = 64$ coordinates that our code encodes, and both hand the same number of multilinear variables to the same tail code. The benchmarks were run on an Apple M4 Max (10 performance cores) using multiple threads.

The univariate skip that we benchmarked is based on an optimized variant of the univariate skip introduced in [DP25] and improved in [BRW26]. This optimization evaluates the product message over a small subfield, which keeps the symbols relatively small, specifically in $\mathbb{F}_{2^8}$.

Over 2^{32} constraints, the round in which the substitution takes place drops from 102.9 to 43.6 milliseconds, a factor of $2.36 \times$. The zerocheck as a whole drops from 162.8 to 98.5 milliseconds, a factor of $1.65 \times$. Before the AG skip, the zerocheck was the most expensive part of the Flock prover.

Remark 1.3. *Loosely speaking, the product complexity (i.e., cost of generating the product message) is quadratic.³ At first glance, this seems much worse than the quasi-linear product complexity of the Reed–*

²Surprising in the sense that the parameters are *better* than what the asymptotic theory of algebraic-geometry codes predicts; see Section C.

³Since the message is of a constant size, the statement that the product complexity is quadratic is meaningless. What we mean here is that producing the product message involves a generic matrix-vector multiplication.

Solomon code (via the FFT). On second glance, though, since the messages involved are tiny (64 bits) even in the case of Reed-Solomon the naive “quadratic” encoding is preferable (and indeed used, e.g., by Binius64 [DP25] and Flock [BRW26]). Thus, we do not actually suffer from our code having quadratic product complexity, as compared to Reed-Solomon.

1.3 On AI usage

The code presented in this work was found with heavy use of AI assistance, and we believe the project would have been infeasible without modern LLMs. In short, the curve and divisor underlying our code were produced by an LLM-driven search: the models proposed candidate families based on the research literature and wrote and ran Sage scripts to test them. Importantly, the correctness of our results does not rest on any claim made by an AI model: what we rely on is that the final code satisfies the properties of Definition 1.2, and these are validated from the explicit equations of the code, without relying on the geometric construction used to discover it (see Section B). A full account of the methodology appears in Section 5.

AI was also used to proofread this writeup, help with formatting, implement notational refactors, and do section reorganizations.

1.4 Related work

Algebraic geometry in proof-systems. Algebraic-geometry codes (see [CR20] for a survey) have a long history in the theory of probabilistic proofs, where they are used as asymptotically good codes, and in particular as multiplication codes, over small alphabets. Examples include the constant-rate PCPs of [BKK⁺16] and the dot-product proofs of [BHI⁺24]. In these works, AG codes serve as an asymptotic tool, whereas we use a specific, concrete code. In the more applied setting, AG codes have mostly been proposed for the *commitment* layer, as a replacement for Reed–Solomon proximity testing. Bordage, Lhotel, Nardi, and Randriam [BLNR22] gave IOPs of proximity for AG codes, generalizing FRI. The ECFFT line of work [BSCKL21, BSCKL22] uses elliptic curves to obtain FFT-like algorithms, and hence efficient proof-systems, over fields that lack suitable multiplicative or additive structure, and Circle STARKs [HLP24] use the circle curve for a similar purpose over Mersenne-prime fields. In all of these, the curve serves to provide a large, structured evaluation domain. In contrast, we use a curve *inside the sumcheck protocol*, and for a different property: binary preservation, which comes from the many rational points of the curve over $\mathbb{F}_2$.

Reverse multiplication-friendly embeddings. *Reverse multiplication-friendly embeddings* (RMFEs) [CCXY18] embed $\mathbb{F}_2^s$ into an extension field $\mathbb{F}_{2^m}$ so that coordinate-wise products of bit vectors can be recovered, via a linear map, from a single multiplication in $\mathbb{F}_{2^m}$. They are commonly constructed from algebraic curves, and are used to amortize $\mathbb{F}_2$ -arithmetic in multiparty computation and in proof-systems. Like our construction, they rely on the multiplicative structure $L(D) \cdot L(D) \subseteq L(2D)$ of functions on a curve. However, an RMFE represents a product by its value at a single place of the curve, and has no distance requirement, whereas our code represents it by its coordinates over $\mathbb{F}_2$ and obtains its distance by evaluating at many points over the large field $\mathbb{F}$.

Concurrent work. Concurrently and independently, Soukhanov [Sou26b, Sou26a] proposed *RMFE-skip*, a variant of the univariate skip for Boolean data. Bits are encoded into low-degree polynomials over $\mathbb{F}_2$ by gluing together small RMFEs via the Chinese remainder theorem, products are computed as products of binary polynomials, and the pointwise product of the bits is recovered by a linear projection. In our terminology, RMFE-skip is also a binary-preserving multiplication code: the recovery map is $\mathbb{F}_2$ -linear, so after an $\mathbb{F}_2$ -change of basis of the product messages, it too is product-systematic. Geometrically, it corresponds to the genus-zero curve, with the data placed on higher-degree places (the irreducible factors used in the CRT) rather than on rational points. Its implementation in [Sou26a] encodes 96 bits into polynomials of degree less than 192, so that products have at most 383 bits. In prefix form, the suffix of the product message therefore has at most $383 - 96 = 287$ bits, or about 2.99 bits per data bit, compared to $158/64 \approx 2.47$ for our code. (The protocol as described in [Sou26b] sends the full product.)

An interesting open problem is to explore whether his techniques can be combined with the ones in this work.

1.5 Organization

Section 2 gives the coding-theoretic view of sumcheck and locates the binary overhead. Section 3 develops the equation-level evaluation framework and gives the concrete validation criteria. Section 4 introduces the curve-and-divisor vocabulary and states the abstract construction without mixing in the explicit realization. Section 5 describes the AI-assisted search. Finally, Sections A and B record the explicit instance and validate it using only the explicit equations, and Section C compares the parameters of our code with the asymptotic behavior of algebraic-geometry codes.

2 The sumcheck protocol: A coding-theoretic perspective

The sumcheck protocol [LFKN92] is a fundamental component in both theoretical and practical proof-system design. We show that the code introduced in this work can significantly reduce its cost in certain contexts.

Following [Mei13, RR24, RR25], we begin in Section 2.1 by giving an abstract coding-theoretic perspective on the sumcheck protocol. In a nutshell, these works showed that one can work with tensors of arbitrary multiplication codes rather than specifically with polynomials.

2.1 The inner-product sumcheck

Let $a, b \in \mathbb{F}^k$ and $v \in \mathbb{F}$. We focus on the following simple form of sumcheck: reducing an inner-product claim of the form

$$\sum_{i \in [k]} a(i) \cdot b(i) = v,$$

to separate linear claims on a and b . For example, the classical sumcheck protocol of [LFKN92], which relies on multivariate polynomials, reduces such an inner-product claim to multilinear evaluation claims on a and b . Later (see Section 2.2), we will focus on the case in which the vectors a and b contain Boolean values, but for the moment we consider general $a, b \in \mathbb{F}^k$.

We show that the same task can be solved using any multiplication code, without relying specifically on polynomials.

2.1.1 Warmup: A base protocol

We start with a simple multiplication-code-based protocol that is quite inefficient. Jumping ahead, this protocol will be the base case of a recursive approach described later.

As mentioned above, the key component we need is a multiplication code, as defined in Section 1.1. Let $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$ be a linear multiplication code with product code $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$, and assume that C is product-systematic. Recall that we defined the product message as $a \times b = C_*^{-1}(C(a) \circ C(b))$, whereas $a \circ b$ is simply the pointwise product. Thus the k -long prefix of $a \times b$ is $a \circ b$.

Using these codes, we give a protocol that reduces the inner product claim $v = \sum_i a_i \cdot b_i$ to separate linear claims about a and b :

1. The prover, given as input $a, b \in \mathbb{F}^k$, generates and sends $g = a \times b$ to the verifier.
2. The verifier checks the prefix sum $\sum_{i \in [k]} g(i) = v$.
3. The verifier samples a uniform point $\rho \in [n]$ and sends it to the prover.
4. The prover computes and sends to the verifier $\alpha = C(a)[\rho]$ and $\beta = C(b)[\rho]$.
5. The verifier computes $C_*(g)[\rho]$ and checks that it is equal to $\alpha \cdot \beta$.

6. If the verifier's checks pass, then it outputs the linear claims " $C(a)[\rho] = \alpha$ " and " $C(b)[\rho] = \beta$ ".

Completeness. If $\sum_i a_i \cdot b_i = v$ and the prover sends $g = a \times b$, then, since $a \circ b$ is the k -long prefix of $a \times b$, the check in Step 2 passes. The prover then sends $\alpha = C(a)[\rho]$ and $\beta = C(b)[\rho]$. By the multiplicative property,

$$\alpha \cdot \beta = (C(a)[\rho]) \cdot (C(b)[\rho]) = C_*(a \times b)[\rho] = C_*(g)[\rho]$$

so the verifier's test in Step 5 passes. Since α and β are computed correctly, the resulting linear claims are also valid.

Soundness. Suppose $\sum_i a_i \cdot b_i \neq v$. The honest message $g = a \times b$ has prefix sum $\sum_{i \in [k]} g(i) \neq v$, so to pass the verifier's check in Item 2, the prover must send some $\tilde{g} \neq g$. Assume that this is indeed the case.

Thus, $C_*(\tilde{g})$ and $C_*(g)$ are distinct codewords of C_* and hence agree on at most a $1 - \delta_*$ fraction of their coordinates, where δ_* is the distance of C_* . Hence, with probability at least δ_* over ρ , we have $C_*(\tilde{g})[\rho] \neq C_*(g)[\rho]$. We show that when such a ρ is selected, the verifier always rejects or outputs a false linear claim.

Suppose the prover honestly sends $\alpha = C(a)[\rho]$ and $\beta = C(b)[\rho]$. Then:

$$\alpha \cdot \beta = (C(a)[\rho]) \cdot (C(b)[\rho]) = C_*(a \times b)[\rho] \neq C_*(\tilde{g})[\rho],$$

and so the verifier rejects in Step 5. Thus, the prover must send false linear claims: either $\alpha \neq C(a)[\rho]$ or $\beta \neq C(b)[\rho]$. Thus, the reduction has soundness error $1 - \delta_*$.

Remark 2.1. Note that this protocol imposes two efficiency requirements on our codes:

- On the prover's side, we need to be able to compute the product message $a \times b$ given a and b .
- On the verifier's side, we need to be able to compute the "encoding at a point" $C_*(g)[\rho]$ given g .

Importantly, for the protocol to have small soundness error, we need the code to have distance very close to 1 and therefore the blocklength n to be very large, and so we would like both tasks to take time proportional to k rather than to n . Neither is automatic: computing $a \times b$ naively passes through length- n encodings. The second task can be solved if one can efficiently generate a row of the generator matrix of C_* .

The downside of the base protocol is that it requires sending g , which is very large — indeed, larger than a and b . Next, we show how to use tensors to reduce the proof size.

2.1.2 Sumcheck for a tensor of multiplication codes

Let $C' : \mathbb{F}^{k'} \rightarrow \mathbb{F}^{n'}$ and $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$ be linear codes. Recall that the tensor code $C' \otimes C : \mathbb{F}^{k' \cdot k} \rightarrow \mathbb{F}^{n' \cdot n}$ is defined by viewing the $(k' \cdot k)$ -size message as a $k' \times k$ matrix, encoding each column by C' and then encoding each of the resulting n' rows by C .

We show that if C' and C are product-systematic multiplication codes, then the code $C' \otimes C$ has an efficient sumcheck protocol. In what follows, it is useful to think of the tensor as being skewed $k' \gg k$.

Recursive sumcheck. Let $a, b \in \mathbb{F}^{k' \times k}$ and $v \in \mathbb{F}$. In what follows, we denote by $a(i', \cdot) \in \mathbb{F}^k$ the i' -th row of a and by $a(\cdot, i) \in \mathbb{F}^{k'}$ the i -th column, and similarly for b .

Recall that the goal is to reduce a claim of the form $\sum_{i' \in [k'], i \in [k]} a(i', i) \cdot b(i', i) = v$ to separate linear claims on messages $a, b \in \mathbb{F}^{k' \times k}$.

1. The prover computes and sends the message:

$$g = \sum_{i' \in [k']} a(i', \cdot) \times b(i', \cdot).$$

2. The verifier checks the prefix sum $\sum_{i \in [k]} g(i) = v$. It then samples a uniform $\rho \in [n]$, and sends it to the prover.
3. The prover computes $a', b' \in \mathbb{F}^{k'}$ defined as $a'(i') = C(a(i', \cdot))[\rho]$ and $b'(i') = C(b(i', \cdot))[\rho]$.
4. The parties recurse, now with the code C' , on the claim $\sum_{i' \in [k']} a'(i') \cdot b'(i') = C_*(g)[\rho]$.

Observe that the entire prover-to-verifier communication consists of the single message g , plus the cost of the recursion. The recursion generates linear claims on a' and b' of the form $C'(a')[\rho'] = \alpha$ and $C'(b')[\rho'] = \beta$, for $\rho' \in [n']$ and $\alpha, \beta \in \mathbb{F}$. Equivalently, these can be viewed as $(C' \otimes C)(a)[\rho', \rho] = \alpha$ and $(C' \otimes C)(b)[\rho', \rho] = \beta$.

Completeness. Suppose $\sum_{i' \in [k'], i \in [k]} a(i', i) \cdot b(i', i) = v$ and that the prover sends the honest message $g = \sum_{i' \in [k']} a(i', \cdot) \times b(i', \cdot)$. Since C is product-systematic, the k -long prefix of each $a(i', \cdot) \times b(i', \cdot)$ equals $a(i', \cdot) \circ b(i', \cdot)$, and hence

$$\sum_{i \in [k]} g(i) = \sum_{i \in [k]} \sum_{i' \in [k']} a(i', i) \cdot b(i', i) = v,$$

so the check in Step 2 passes. For each $i' \in [k']$,

$$\begin{aligned} a'(i') \cdot b'(i') &= C(a(i', \cdot))[\rho] \cdot C(b(i', \cdot))[\rho] \\ &= (C(a(i', \cdot)) \circ C(b(i', \cdot)))[\rho] \\ &= C_*(a(i', \cdot) \times b(i', \cdot))[\rho], \end{aligned}$$

using in turn the definitions of $a'(i')$ and $b'(i')$, the fact that the ρ -th coordinate of a pointwise product is the product of the ρ -th coordinates, and the multiplication property $C(a(i', \cdot)) \circ C(b(i', \cdot)) = C_*(a(i', \cdot) \times b(i', \cdot))$ of C . Summing over $i' \in [k']$ and using the linearity of C_* ,

$$\sum_{i' \in [k']} a'(i') \cdot b'(i') = \sum_{i' \in [k']} C_*(a(i', \cdot) \times b(i', \cdot))[\rho] = C_*(g)[\rho]. \quad (2.1)$$

Thus the recursive claim holds, and completeness follows from that of the recursion in Step 4, applied to a' and b' .

Soundness. Suppose $\sum_{i' \in [k'], i \in [k]} a(i', i) \cdot b(i', i) \neq v$. The honest message g has prefix sum $\sum_{i \in [k]} g(i) \neq v$, so to pass the check in Step 2, the prover must send some $\tilde{g} \neq g$; assume that this is indeed the case. Then $C_*(\tilde{g})$ and $C_*(g)$ are distinct codewords of C_* and hence agree on at most a $1 - \delta_*$ fraction of their coordinates, where δ_* is the distance of C_* . Hence, with probability at least δ_* over ρ , we have $C_*(\tilde{g})[\rho] \neq C_*(g)[\rho]$; assume that such a ρ is selected.

By (2.1), the honestly folded vectors of Step 3 satisfy $\sum_{i' \in [k']} a'(i') \cdot b'(i') = C_*(g)[\rho]$, whereas the verifier takes the target of the recursion in Step 4 to be $C_*(\tilde{g})[\rho]$. The recursive claim $\sum_{i' \in [k']} a'(i') \cdot b'(i') = C_*(\tilde{g})[\rho]$ is therefore false, so by the soundness of the recursion the reduction rejects or outputs a false linear claim on a' and b' — equivalently, on a and b . The soundness error contributed by this round is $1 - \delta_*$.

Unfolding the recursion and accounting. Consider a t -fold tensor code $C^{(1)} \otimes \dots \otimes C^{(t)}$.

Recurring the above protocol $t - 1$ times and then executing the base case, we obtain a t -round protocol in which each round folds one of the factors. In the round folding $C^{(j)}$, the prover sends a single $C_*^{(j)}$ -message g_j and receives a challenge ρ_j ; the final round additionally returns the two scalars α, β . The total prover communication is thus $2 + \sum_{j=1}^t \dim C_*^{(j)}$ field elements. For soundness, each round is the one-round reduction above, applied to a single factor, and has soundness error $1 - \delta_*^{(j)}$, so a union bound over the t factors gives overall error $\sum_{j=1}^t (1 - \delta_*^{(j)})$.⁴

⁴This is a *round-by-round soundness* argument in the sense of [CCH⁺19].

Example 2.2 (Multilinear extension). Let $\text{Lin} : \mathbb{F}^2 \rightarrow \mathbb{F}^{\mathbb{F}}$ be the code of affine functions. It maps a message (a_0, a_1) to the codeword whose coordinate indexed by $X \in \mathbb{F}$ is $a_0 \cdot (1 - X) + a_1 \cdot X$.

Note that Lin is systematic and multiplicative, with the product code corresponding to quadratics (polynomials of degree ≤ 2). To make the code product-systematic, we represent a quadratic q by the message $(q(0), q(1), q_2)$, where q_2 is its X^2 -coefficient; that is, $\text{Lin}_*(q_0, q_1, q_2)$ is the codeword whose coordinate indexed by X is $q_0 \cdot (1 - X) + q_1 \cdot X + q_2 \cdot X(X - 1)$. In particular, the prover sends 3 field elements per round, as in the classical protocol. The m -fold tensor of Lin is the classical multilinear extension code, and the protocol above, specialized to this code, recovers the classical polynomial sumcheck protocol.

2.2 Coping with binary data

In proof-systems geared toward Boolean computations, we will be interested in proving statements of the form $v = \sum a_i \cdot b_i$, where $a, b \in \{0, 1\}^k$ are vectors of bits and the arithmetic is over the binary field $\mathbb{F}_2$. Although the data consists of bits, we cannot simply assume that our codewords do as well. Codes over small alphabets cannot have large relative distance (see the *Plotkin bound*, [GRS25, Theorem 4.4.1]). Thus, we use a large extension field $\mathbb{F}$ of $\mathbb{F}_2$, and the codes are over $\mathbb{F}$. The key difficulty is the large overhead in viewing elements of $\mathbb{F}_2$ as elements in the large field $\mathbb{F}$.

The first place where this overhead materializes in the sumcheck protocol, as presented abstractly in Section 2.1.2, is in the prover's computation of $a(i', \cdot) \times b(i', \cdot)$ for every $i' \in [k']$. Although the individual vectors $a(i', \cdot)$ and $b(i', \cdot)$ are known to be over bits, there is a priori no such guarantee on the result. In particular, if both describe evaluations of some degree- $(k - 1)$ polynomial on some particular evaluation set S , there is no reason to expect the result to remain in $\mathbb{F}_2$ when expanding beyond S . For instance, for the Reed-Solomon code (i.e., the univariate skip), the product has degree $2k - 2$, so its values outside S must be taken at points of a subfield of size at least $2k - 1$. Each of these entries therefore takes about $\log_2(2k)$ bits rather than one (8 bits in practice). As we explain at the end of Section 2.3, this is precisely what the prover pays for.

This is where we rely on a multiplication code that is *binary-preserving* (see Section 1.1), which guarantees that the product message is indeed over $\mathbb{F}_2$.

The second place where the overhead materializes is when the prover computes a' and b' —the foldings of a and b at the point ρ . The bit vectors $a, b \in \{0, 1\}^{k' \cdot k}$ are now reduced to vectors $a', b' \in \mathbb{F}^{k'}$. If $k \leq \log(|\mathbb{F}|)$, then the new vectors are actually *larger* than the original ones! For example, for the multilinear extension code, $k = 2$, and the new vectors are larger by a factor of $\log(|\mathbb{F}|)/2$ in the recursive call.

Through this lens, Gruen's [Gru24] univariate skip is the observation that, by using a larger code in the first round—increasing k from $k = 2$ in plain sumcheck to a moderate size, though still $k \ll k'$ —we can drastically reduce this overhead.

2.3 From inner products to the zerocheck

In practice, sumcheck is often used as a subroutine of a *zerocheck protocol*, described next.

Given bit-vectors a, b , the goal of the zerocheck protocol is to reduce the claim “ $a_i \cdot b_i = 0, \forall i$ ” to separate linear claims on a and b .

Let $k \in \mathbb{N}$ and $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$ be a multiplication code with product code $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$. We view $a, b : \{0, 1\}^m \times [k] \rightarrow \{0, 1\}$. The verifier samples a random point $r \in \mathbb{F}^m$. Define

$$P := \sum_{x \in \{0, 1\}^m} \text{eq}(x, r) \cdot (a(x, \cdot) \times b(x, \cdot)).$$

The message P lies in $\mathbb{F}^{k_*}$; it is exactly the tensor-sumcheck message of Section 2.1.2, now eq-weighted, for the tensor $\text{Lin}^{\otimes m} \otimes C$ (the m Boolean variables against the multiplication code C). Its first k coordinates are $\sum_{x \in \{0, 1\}^m} \text{eq}(x, r) \cdot (a(x, \cdot) \circ b(x, \cdot))$, whose j -th coordinate

$$\sum_{x \in \{0, 1\}^m} \text{eq}(x, r) \cdot a(x, j) b(x, j)$$

is the multilinear extension of the j th column of $a \cdot b$, evaluated at r . Hence, if the claim is true (i.e., $a \cdot b \equiv 0$), the prefix of P vanishes. By contrast, if $a(x, j) b(x, j) \neq 0$ for some (x, j) , then the j th column of $a \cdot b$ is a nonzero function on $\{0, 1\}^m$. With probability at least $1 - m/|\mathbb{F}|$ over r , its multilinear extension therefore does not vanish at r —that is, the j th prefix coordinate of P is nonzero.

The parties now proceed as in the tensor sumcheck of Section 2.1.2, except that the check “ $\sum_i g(i) = v$ ” is replaced by the requirement that the prefix of P be zero:

1. The prover sends P . As its first k coordinates vanish for an honest prover, only the remaining $k_* - k$ coordinates need be transmitted; the verifier fills the prefix with 0.
2. The verifier samples a uniform point $\rho \in [n]$, and the parties reduce to the claim

$$\sum_{x \in \{0, 1\}^m} \text{eq}(x, r) \cdot a'(x) b'(x) = C_*(P)[\rho], \quad a'(x) := C(a(x, \cdot))[\rho], \quad b'(x) := C(b(x, \cdot))[\rho].$$

Completeness and soundness follow as in Section 2.1.2: by the multiplicative property, the honest folded vectors satisfy $\sum_x \text{eq}(x, r) a'(x) b'(x) = C_*(P)[\rho]$, whereas if the prover sends $\tilde{P} \neq P$, then $C_*(\tilde{P})$ and $C_*(P)$ are distinct codewords, which disagree at a random ρ with probability at least the distance δ_* of C_* , making the reduced claim false. Overall, the soundness error of this round is at most $m/|\mathbb{F}| + (1 - \delta_*)$, and the soundness error of the remaining sumcheck is added to it.

In this single round, the message P has collapsed the entire multiplication-code factor C — the “skip” — leaving an eq-weighted inner-product claim on the folded vectors $a', b' \in \mathbb{F}^{2^m}$. Since $\text{eq}(\cdot, r)$ is itself multilinear, this is a sumcheck over a product of three multilinear functions, in which each round polynomial has degree 3; using Gruen’s factoring of eq [Gru24], the prover need only send a degree-2 polynomial per round. The parties continue with this standard sumcheck.

The saving here is exactly the accounting of Section 2.1.2. An ordinary multilinear sumcheck over the $[k]$ block would spend $\log_2 k$ rounds, folding the Boolean data two coordinates at a time and—as discussed in Section 2.2—inflating it into $\mathbb{F}$ by a factor of about $\log(|\mathbb{F}|)/2$. The skip instead collapses all of $[k]$ into the single message P : the prover transmits its final $k_* - k$ coordinates and folds a, b just once, into $a', b' \in \mathbb{F}^{2^m}$, an inflation of only $\log(|\mathbb{F}|)/k$.

Cost of computing P , and where binary preservation helps. The saving above is shared by every skip, whatever the multiplication code. The choice of the code matters in the dominant prover cost of the skip round: computing

$$P = \sum_{x \in \{0, 1\}^m} \text{eq}(x, r) \cdot (a(x, \cdot) \times b(x, \cdot)).$$

Only the suffix of P is needed, so for each $x \in \{0, 1\}^m$ the prover computes the suffix of $a(x, \cdot) \times b(x, \cdot)$, multiplies it by the scalar $\text{eq}(x, r) \in \mathbb{F}$, and adds the result to an accumulator in $\mathbb{F}^{k_* - k}$. If C is binary-preserving, this suffix is a $(k_* - k)$ -bit vector, so the step is a subset sum: $\text{eq}(x, r)$ is added to the accumulators indexed by the nonzero bits. This takes about $k_* - k$ bit-by-field operations per x . For the Reed–Solomon code, the suffix instead consists of about k elements of a subfield $\mathbb{F}_{2^s}$ with $2^s \geq 2k - 1$ (in practice, $s = 8$), and multiplying each of them by $\text{eq}(x, r)$ costs about s bit-by-field operations, for a total of about sk . To first order, the cost of the skip round is therefore proportional to the bit-size of the suffix. For $k = 64$, the suffix has 158 bits with our code, versus 512 bits with Reed–Solomon as used in practice. This first-order model ignores implementation techniques such as packing (e.g., the two-layer packing of [BRW26]), which change the constants; the benchmarks in Section 1 measure the actual effect.

3 Evaluation codes from explicit equations

Motivated by the need for binary-preserving codes described in Section 2.2, we turn to codes arising from algebraic geometry. These codes give us large evaluation sets over $\mathbb{F}_2$ in the form of rational points, i.e.,

sets of solutions to equations. In particular, our code can be understood as a special case of a *differential Goppa code* as studied in [GCV26]. However, the construction described here differs somewhat from the usual presentation, partly because of the process we used to find the code through an AI-assisted search.

Traditionally, algebraic codes are constructed from various algebraic structures on curves, and the properties of the code follow from the geometric properties of the algebraic variety in question. Here, however, completely verifying all the claims made by the AI agent is mathematically and computationally complicated, so we used simpler evaluation criteria for the codes we find. This motivates the presentation here, which separates the search space and evaluation criteria from the process of finding mathematical structures that satisfy them, in which most of the algebraic geometry enters. Namely, the geometry mainly enters in the form of a “prompt” helping the agent find good places to look.

We begin by describing the general structure of the codes of interest and identifying precise evaluation criteria that are relatively easy to verify. This is the only source of trust in the code’s properties. The next section describes the ideas from algebraic codes that motivate the specific realization we have found. Readers with a background in algebraic geometry might want to jump directly to Theorem 4.1.

3.1 Preliminaries: function fields and equation models

Consider the set of polynomials $\mathbb{F}_2[x]$ in a single variable.⁵ Formally adding fractions produces the field of rational functions, denoted $\mathbb{F}_2(x)$, whose elements can be thought of as fractions $\frac{f(x)}{g(x)}$ with $f(x), g(x) \in \mathbb{F}_2[x]$ and nonzero $g(x)$. Two such elements, $\frac{f(x)}{g(x)}$ and $\frac{d(x)}{e(x)}$, are considered equal if

$$f(x) \cdot e(x) = d(x) \cdot g(x)$$

as polynomials. We can view elements of $\mathbb{F}_2(x)$ as partially defined functions $\mathbb{F} \rightarrow \mathbb{F}$ for any extension $\mathbb{F}$ of $\mathbb{F}_2$. Namely, every element $\frac{f(x)}{g(x)}$ defines a function on the set of points $U \subset \mathbb{F}$ where g is nonzero. For $u \in U$, we define:

$$\frac{f(x)}{g(x)}(u) = \frac{f(u)}{g(u)}.$$

Let $h_0, \dots, h_{k_*-1} \in \mathbb{F}_2(x)$, and let $S \subset \mathbb{F}$ be a subset on which they are all defined. Denote the size of S by n . We can identify the linear space $\mathbb{F}^n$ with the set of functions $S \rightarrow \mathbb{F}$. Each element $h_i \in \mathbb{F}_2(x)$ then defines an element $H_i \in \mathbb{F}^n$ via:

$$H_i(s) = h_i(s).$$

We can consider the corresponding *evaluation code* $C_S : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$ sending the basis elements to H_i .

This is the simplest example of a more general construction. Instead of considering only functions defined on subsets of $\mathbb{F}$, we can consider more complicated sets that arise from solutions to polynomial equations. These are known as *algebraic varieties*; of particular interest to us are *algebraic curves*. To avoid overburdening this exposition with unnecessary background, we will not discuss the exact definition of algebraic curves. Instead, we will approach them by defining the notion of a function on them and using it to define evaluation codes.

Definition 3.1. An *algebraic function field* in one variable is a finite extension of the field $\mathbb{F}_2(x)$ of rational functions.

This gives the dimension check that will matter below: an equation system of this form has one free parameter x , and hence is one-dimensional.

As the name suggests, algebraic function fields can be realized as “fields of rational functions” on algebraic sets. We will try to give an informal explanation of the relation.

Given a finite extension K_X of $\mathbb{F}_2(x)$, choose generators $y_1, \dots, y_t$. Since the extension is finite, there are polynomials $P_j \in \mathbb{F}_2[x, y_1, \dots, y_t]$ ⁶ for $j \in \{0, \dots, c_{\text{eq}} - 1\}$ such that there is an identification

$$K_X \simeq \mathbb{F}_2(x)[y_1, \dots, y_t] / (P_0, \dots, P_{c_{\text{eq}}-1}).$$

⁵At this point there is no degree bound; one will be imposed later through a divisor.

⁶A priori, the relations have coefficients in $\mathbb{F}_2(x)$; clearing denominators makes them polynomial.

Here t is the number of auxiliary generators, c_{eq} is the number of displayed relations, and

$$\nu := [K_X : \mathbb{F}_2(x)]$$

is the degree of the field extension. Neither t nor c_{eq} is the dimension of the equation system; the dimension is one because K_X is finite over $\mathbb{F}_2(x)$. We can think of K_X as a field of functions defined on the algebraic set consisting of the solutions $(x, y_1, \dots, y_t)$ to

$$P_j(x, y_1, \dots, y_t) = 0, \quad \forall j \in \{0, \dots, c_{\text{eq}} - 1\}. \quad (3.1)$$

Conversely, starting with a set of equations as in Eq. (3.1), we can form the quotient

$$K_{\text{eq}} := \mathbb{F}_2(x)[y_1, \dots, y_t] / (P_0, \dots, P_{c_{\text{eq}}-1}).$$

If this quotient is a field, then by definition it is an algebraic function field; for the equations chosen from K_X above, it is isomorphic to K_X . We formalize how the equations represent an abstract curve in the next definition.

Definition 3.2. Let $\mathbb{F}_2(X)$ be an algebraic function field in one variable. An *affine equation model* U with auxiliary variables $y_1, \dots, y_t$ on X is a collection of multivariate polynomials $\{P_j\}_j$ in $\mathbb{F}_2[x, y_1, \dots, y_t]$ and an $\mathbb{F}_2$ -algebra map $\varphi_U : \mathbb{F}_2[x, y_1, \dots, y_t] \rightarrow \mathbb{F}_2(X)$ such that

$$\mathbb{F}_2(x)[y_1, \dots, y_t] / (P_0, \dots, P_{c_{\text{eq}}-1}) \simeq \mathbb{F}_2(X)$$

is the field isomorphism induced by φ_U .

We sometimes call the solution set of such an equation model an *affine variety*. We keep the three related objects notationally distinct: X is the abstract curve, U is its affine equation model, and $\mathbb{F}_2(X)$ is its function field.

Remark 3.3. A given function field has many different possible affine equation models. Any two systems that give rise to the same field are said to be *birationally equivalent*.

Example 3.4. The projective line $\mathbb{P}^1$ has the affine equation model $\mathbb{A}^1$ in a single variable x , without any equations. Its function field is the field $\mathbb{F}_2(x)$ of rational functions.

Example 3.5. Consider the variety $\mathbb{G}_m$ defined by the affine equation model in two variables x, y given by the equation $xy = 1$. We have an identification

$$\mathbb{F}_2(\mathbb{G}_m) \simeq \mathbb{F}_2(x)[y] / (xy - 1) \simeq \mathbb{F}_2(x, x^{-1}) \simeq \mathbb{F}_2(x).$$

Thus $\mathbb{G}_m$ is birationally equivalent to the affine line $\mathbb{A}^1$.

Example 3.6 (An equation model). In characteristic two, consider the equation

$$y^2 + y = x^3.$$

It rewrites every occurrence of y^2 as $y + x^3$, so every polynomial expression reduces to the form

$$a(x) + b(x)y.$$

Thus x is the one free parameter, with at most two choices of y above each value of x . This is the simplest version of the equation systems used in our explicit construction.

Given a function field $\mathbb{F}_2(X)$ and an affine equation model U as above, we can define the set of K -rational points $U(K) \subset K^{t+1}$ for every field extension K of $\mathbb{F}_2$ by

$$U(K) = \left\{ (x, y_1, \dots, y_t) \in K^{t+1} : P_i(x, y_1, \dots, y_t) = 0, \forall i \in \{0, \dots, c_{\text{eq}} - 1\} \right\}.$$

Thus a K -valued point of the affine equation model is simply a tuple solving its displayed equations.

Remark 3.7. One can formalize the affine equation model by assigning to every $\mathbb{F}_2$ -algebra R the solution set $U(R)$. This assignment is called the functor of points of U .

Given an algebraic function field $\mathbb{F}_2(X)$, an affine equation model U , and a field extension K of $\mathbb{F}_2$, we can view elements $f \in \mathbb{F}_2(X)$ as partially defined functions on $U(K)$. Namely, from the identification of

$$\mathbb{F}_2(X) \simeq \mathbb{F}_2(x)[y_1, \dots, y_t]/(P_0, \dots, P_{c_{\text{eq}}-1}),$$

we can think of f as a fraction $g(x, y_1, \dots, y_t)/h(x)$. It is therefore well defined outside the zero locus of h , which we denote by U_h :

$$U_h(K) = \{(x, y_1, \dots, y_t) \in U(K) : h(x) \neq 0\}.$$

In fact, any such nonzero locus is itself an affine equation model given by the coordinates $(x, y_1, \dots, y_t, z)$ with the extra equation

$$h(x)z - 1 = 0.$$

As in Example 3.5, this equation model gives rise to the same function field $\mathbb{F}_2(X)$. In this way, $\mathbb{F}_2(X)$ acts as a “field of rational functions” on the curve X . In our case, it suffices to understand how $\mathbb{F}_2(X)$ can give rise to functions on sets of solutions to polynomial equations defining an affine equation model for X .

Definition 3.8. Let $\mathbb{F}_2(X)$ be a function field and let U be an affine equation model on X . We say that an element $f \in \mathbb{F}_2(X)$ is *defined on U* if it is in the image of the map

$$\mathbb{F}_2[x, y_1, \dots, y_t] \rightarrow \mathbb{F}_2(X)$$

defined by U .

Remark 3.9. The definition above does not exactly match other definitions in the algebraic geometry literature, but we chose to give a simpler definition here that fits our purposes.

Remark 3.10. Intuitively, an element $f \in \mathbb{F}_2(X)$ is defined on U if its denominator does not vanish anywhere on U .

In particular, an element $f \in \mathbb{F}_2(X)$ defined on U gives rise to a function on $U(K)$ for any extension field K of $\mathbb{F}_2$.

Example 3.11. Consider the element $\frac{1}{x} \in \mathbb{F}_2(x)$. It is defined on the variety $\mathbb{G}_m$ from Example 3.5 as the image of y under the affine-model map $\mathbb{F}_2[x, y] \rightarrow \mathbb{F}_2(x)$ sending y to x^{-1} . However, it is not defined on the affine line since it is not in $\mathbb{F}_2[x]$. This aligns with the identification of $\mathbb{G}_m$ as the subset of $\mathbb{A}^1$ of all nonzero values.

3.2 Evaluation codes

We are now ready to define the special class of codes that we are interested in. We fix an algebraic function field $\mathbb{F}_2(X)$ and an affine equation model U defined by a set of equations

$$P_j(x, y_1, \dots, y_t) = 0, \quad \forall j \in \{0, \dots, c_{\text{eq}} - 1\},$$

so that we can identify

$$\mathbb{F}_2(X) \simeq \mathbb{F}_2(x)[y_1, \dots, y_t]/(P_0, \dots, P_{c_{\text{eq}}-1}).$$

Denote the number of points on $U(\mathbb{F})$ by n . Given an element $f \in \mathbb{F}_2(X)$ that is defined on U (in the sense of Definition 3.8), denote by $\text{Ev}_U(f)$ the evaluation function

$$\begin{aligned} \text{Ev}_U(f) : U(\mathbb{F}) &\rightarrow \mathbb{F} \\ s &\mapsto f(s). \end{aligned}$$

Definition 3.12. Fix integers $k_*, n \in \mathbb{N}$. An *algebraic evaluation code* is a tuple $(\mathbb{F}_2(X), H, U, e)$ consisting of

- A function field $\mathbb{F}_2(X)$ and an affine equation model U .
- A k_* -dimensional vector subspace $H \subseteq \mathbb{F}_2(X)$ with all of its elements defined on U .
- A choice of basis $\{h_i\}_{i=0}^{k_*-1}$ for H , viewed as a linear map $e : \mathbb{F}_2^{k_*} \rightarrow H$.

After extending scalars from $\mathbb{F}_2$ to $\mathbb{F}$, the same basis determines a linear map $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$ sending the i th standard basis vector to the function $\text{Ev}_U(h_i)$, considered as an element of $\mathbb{F}^n$.

We will often identify the algebraic evaluation code with its encoding map C_* . Later in the section, we will give criteria that bound its distance and show that its evaluation set is efficiently sampleable. Before that, we describe the multiplicative property.

The map C_* will play the role of the product code. The base code C will be obtained from a subspace of H that has the following multiplicative property.

Definition 3.13. Let $(\mathbb{F}_2(X), H, U, e)$ be an evaluation code in the sense of Definition 3.12, with encoding map C_* . A k -dimensional *multiplicative subspace* is a linear subspace $M \subseteq H$ of dimension k such that, for $f, g \in M$, we have $f \cdot g \in H$, where the multiplication takes place in $\mathbb{F}_2(X)$.

Remark 3.14. By bilinearity, to check that $M \subseteq H$ is multiplicative, it is enough to establish $m_i \cdot m_j \in H$ for all elements of any basis $\{m_i\}$ of M .

We can identify a k -dimensional subspace with an injective linear map $R : \mathbb{F}^k \rightarrow \mathbb{F}^{k_*}$. The composition with C_* gives a map $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$. The following proposition follows easily from the definition above.

Proposition 3.15. *Given an algebraic evaluation code $(\mathbb{F}_2(X), H, U, e)$ whose encoding map C_* is injective and a multiplicative subspace M of dimension k , the linear code $C : \mathbb{F}^k \rightarrow \mathbb{F}^n$ is a multiplication code with product code $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$. Moreover, C is binary-preserving.*

Proof. The codewords of C_* are spanned by the evaluations of the basis functions h_i . For $f, g \in M$ and $u \in U(\mathbb{F})$, evaluation respects multiplication:

$$f(u) \cdot g(u) = (f \cdot g)(u)$$

and the right-hand side is a coordinate of a codeword of C_* because $f \cdot g \in H$.

If the two input messages lie in $\mathbb{F}_2^k$, their corresponding functions lie in M with coefficients in $\mathbb{F}_2$. Their product therefore has coefficients in $\mathbb{F}_2$ when expressed in the basis $\{h_i\}$ of H , so the product message lies in $\mathbb{F}_2^{k_*} \subseteq \mathbb{F}^{k_*}$. $\square$

Thus an algebraic evaluation code gives a multiplication code once one verifies the finite condition $m_i m_j \in H$ on a basis of M . The less trivial part, in which more algebraic geometry enters, is finding such subspaces. However, once a subspace is found, the multiplicative condition itself is easy to verify.

A useful evaluation code also needs good distance and an efficiently sampleable evaluation set. We address these two properties next.

3.3 Minimum distance

Fix an algebraic evaluation code $(\mathbb{F}_2(X), H, U, e)$, and write C_* for its encoding map. The affine equation model U defines an identification

$$\mathbb{F}_2(X) \simeq \mathbb{F}_2(x)[y_1, \dots, y_t] / (P_0, \dots, P_{c_{\text{eq}}-1}),$$

together with a choice of basis $\{h_i\}_{i=0}^{k_*-1}$ for H . The goal of this section is to give a useful bound on the minimum distance of the code C_* that depends on the algebraic data alone. The results below are standard, but we give a self-contained presentation for the convenience of the reader.

Since the code C_* is linear, its minimum distance is determined by the minimum Hamming weight of a nonzero codeword, or equivalently by the maximum number of zeros of a nonzero element of H on $U(\mathbb{F})$. Thus, we want to bound the number of zeros using explicit information about the functions in H .

To do so, we need to introduce another algebraic construction known as the *field norm*. Recall that $\nu = [\mathbb{F}_2(X) : \mathbb{F}_2(x)]$. Thus we can think of $\mathbb{F}_2(X)$ as a ν -dimensional vector space over $\mathbb{F}_2(x)$. Every element $f \in \mathbb{F}_2(X)$ defines an $\mathbb{F}_2(x)$ -linear map

$$\begin{aligned} m_f : \mathbb{F}_2(X) &\rightarrow \mathbb{F}_2(X), \\ s &\mapsto f \cdot s. \end{aligned}$$

As a map of finite-dimensional vector spaces, it has a well-defined determinant $\det(m_f) \in \mathbb{F}_2(x)$. The assignment

$$\begin{aligned} N_X : \mathbb{F}_2(X) &\rightarrow \mathbb{F}_2(x) \\ f &\mapsto \det(m_f) \end{aligned}$$

is called the *field norm*. The following properties of the field norm follow easily from the definition:

- For all $f, g \in \mathbb{F}_2(X)$, we have $N_X(f \cdot g) = N_X(f)N_X(g)$.
- For all $f \in \mathbb{F}_2(X)$ and $g \in \mathbb{F}_2(x)$, we have $N_X(g \cdot f) = g^\nu \cdot N_X(f)$.

The same determinant construction works after extending constants from $\mathbb{F}_2$ to $\mathbb{F}$. When a function has coefficients in $\mathbb{F}$, we therefore interpret $N_X(f)$ as an element of $\mathbb{F}(x)$. We will use the norm to bound the number of zeros of elements on $U(\mathbb{F})$. To do so, we restrict our attention to the class of functions defined on U . These are exactly the functions in $\mathbb{F}_2(X)$ coming from the map

$$\mathbb{F}_2[x, y_1, \dots, y_t] \rightarrow \mathbb{F}_2(X).$$

We denote this image by $\mathcal{O}_U$. We will bound the zeros of elements of H using information about their norms. First, we need to understand the image of the norms of elements of $\mathcal{O}_U$.

For each $1 \leq i \leq t$, the element $y_i \in \mathbb{F}_2(X)$ has a minimal polynomial over $\mathbb{F}_2(x)$. Let $\sigma(x) \in \mathbb{F}_2[x]$ be a common denominator for the coefficients of these minimal polynomials. Adjoining the reciprocal $1/\sigma$ gives the rings

$$\mathbb{F}_2[x, 1/\sigma] \subseteq \mathbb{F}_2(x) \quad \text{and} \quad \mathcal{O}_U[1/\sigma] \subseteq \mathbb{F}_2(X).$$

Their elements have the form f/σ^j , with f in $\mathbb{F}_2[x]$ or $\mathcal{O}_U$, respectively. The second ring is a finitely generated module over the first. Since $\mathbb{F}_2[x, 1/\sigma]$ is a principal ideal domain, this module is free of rank ν . In particular, $\mathbb{F}_2(X)$ has a basis over $\mathbb{F}_2(x)$ consisting of elements of $\mathcal{O}_U[1/\sigma]$.

We also require that the map induced by φ_U identifies

$$\mathbb{F}_2[x, 1/\sigma, y_1, \dots, y_t]/(P_0, \dots, P_{c_{\text{eq}}-1}) \simeq \mathcal{O}_U[1/\sigma] = \mathcal{O}_{U_\sigma}. \quad (3.2)$$

This can be arranged by multiplying σ by additional nonzero polynomials in x , since the identification already holds over $\mathbb{F}_2(x)$.

Without loss of generality, we can assume $1/\sigma$ is defined on U . Otherwise we replace U by the nonzero locus of σ . Thus, from now on, $1/\sigma \in \mathcal{O}_U$.

Using this basis, we see that the norm of an element of $\mathcal{O}_U$ lies in $\mathbb{F}_2[x, 1/\sigma]$. Consider the map $\pi_U : U(\mathbb{F}) \rightarrow \mathbb{F}$ sending a point $(x, y_1, \dots, y_t)$ to its first coordinate. We can establish a relation between zeros of the norm and zeros of elements of $\mathcal{O}_U$. For that, the following lemma is useful.

Lemma 3.16. *For all $x_0 \in \mathbb{F}$, the size of the preimage $\pi_U^{-1}(x_0)$ is at most ν , the degree of the extension $\mathbb{F}_2(X)$ over $\mathbb{F}_2(x)$.*

Proof. Since $1/\sigma$ is defined on U , the image of the projection excludes the zero set of σ . Thus the fiber is empty when $\sigma(x_0) = 0$, and we may assume that $\sigma(x_0) \neq 0$.

By definition, the preimage $\pi_U^{-1}(x_0)$ consists of the tuples $(x_0, y_1, \dots, y_t)$ such that

$$P_i(x_0, y_1, \dots, y_t) = 0, \quad \forall i \in \{0, \dots, c_{\text{eq}} - 1\}.$$

Each point p gives an evaluation map $i_p : \mathbb{F}[x, y_1, \dots, y_t] \rightarrow \mathbb{F}$. It belongs to $U(\mathbb{F})$ exactly when i_p sends every P_i to zero. The points in the fiber above x_0 are therefore in bijection with the $\mathbb{F}$ -algebra homomorphisms

$$\mathcal{O}_{U, x_0} := \mathcal{O}_U \otimes_{\mathbb{F}_2[x, 1/\sigma]} \mathbb{F} \longrightarrow \mathbb{F},$$

where $\mathbb{F}$ is an $\mathbb{F}_2[x, 1/\sigma]$ -algebra via $x \mapsto x_0$. The free rank- ν description above shows that $\mathcal{O}_{U, x_0}$ has dimension ν over $\mathbb{F}$. A finite-dimensional $\mathbb{F}$ -algebra has at most as many homomorphisms to $\mathbb{F}$ as its dimension, so the fiber has size at most ν . $\square$

Lemma 3.17. *Let $f \in \mathcal{O}_U$ and let $u = (x_0, y_1, \dots, y_t) \in U(\mathbb{F})$ be a zero of f . Then x_0 is a zero of the norm of f , namely, $N_X(f)(x_0) = 0$. Moreover, if f has a zeros in the preimage $\pi_U^{-1}(x_0)$, then $N_X(f)$ is divisible by $(x - x_0)^a$.*

Proof. Fix $x_0 \in \mathbb{F}$ and let

$$\mathcal{O}_{U, x_0} = \mathcal{O}_U \otimes_{\mathbb{F}_2[x, 1/\sigma]} \mathbb{F}$$

be the algebra from the proof of Lemma 3.16. Multiplication by f induces the linear map

$$\begin{aligned} m_{f, x_0} : \mathcal{O}_{U, x_0} &\rightarrow \mathcal{O}_{U, x_0} \\ g &\mapsto f \cdot g \end{aligned}$$

Let $M_f(x)$ denote the matrix of multiplication by f in the basis chosen above; its specialization at x_0 represents m_{f, x_0} . Assume that f has a zeros in the preimage $\pi_U^{-1}(x_0)$. Each such point u gives a linear functional $e_u : \mathcal{O}_{U, x_0} \rightarrow \mathbb{F}$ by evaluation, and these linear functionals are linearly independent. For each u , we have

$$e_u(m_{f, x_0}(g)) = e_u(gf) = f(u)g(u)$$

so, whenever $f(u) = 0$, the composition $e_u \circ m_{f, x_0}$ is the zero map. Thus the left kernel of $M_f(x_0)$ has dimension at least a . A square matrix and its transpose have the same rank, so

$$q := \dim \ker M_f(x_0) \geq a.$$

Choose a basis of the fiber such that the last q basis vectors span $\ker M_f(x_0)$. In this basis,

$$M_f(x_0) = \begin{pmatrix} * & \cdots & * & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ * & \cdots & * & 0 & \cdots & 0 \end{pmatrix},$$

where the last q columns are zero.

Using the corresponding lifted basis near x_0 , we therefore have

$$M_f(x) = \begin{pmatrix} * & \cdots & * & (x - x_0)a_{1,1}(x) & \cdots & (x - x_0)a_{1,q}(x) \\ \vdots & & \vdots & \vdots & & \vdots \\ * & \cdots & * & (x - x_0)a_{\nu,1}(x) & \cdots & (x - x_0)a_{\nu,q}(x) \end{pmatrix}.$$

Factoring $(x - x_0)$ from each of the last q columns gives

$$\det M_f(x) = (x - x_0)^q \det \begin{pmatrix} * & \cdots & * & a_{1,1}(x) & \cdots & a_{1,q}(x) \\ \vdots & & \vdots & \vdots & & \vdots \\ * & \cdots & * & a_{\nu,1}(x) & \cdots & a_{\nu,q}(x) \end{pmatrix}.$$

Hence, since $q \geq a$,

$$(x - x_0)^a \mid \det M_f(x) = N_X(f)(x).$$

□

Corollary 3.18. *For every nonzero element $f \in \mathcal{O}_U$, we have*

$$|\{u \in U(\mathbb{F}) : f(u) = 0\}| \leq \deg_{\text{num}}(N_X(f)) \quad (3.3)$$

where $\deg_{\text{num}}$ is the degree of the numerator polynomial in reduced form.

Proof. Write $N_X(f) = a(x)/b(x)$ in reduced form. Since the norm belongs to $\mathbb{F}_2[x, 1/\sigma]$, the denominator b divides a power of σ and is therefore nonzero at the first coordinate of every point of $U(\mathbb{F})$. By Lemma 3.17, if f has a_{x_0} zeros above $x_0 \in \mathbb{F}$, then $(x - x_0)^{a_{x_0}} \mid a(x)$. Counting over all zeros gives the desired inequality. □

Thus, to get a uniform bound on the number of zeros of elements of H , we need to bound the degree of the norms of all elements in H .

Proposition 3.19. *Let $(\mathbb{F}_2(X), H, U, e)$ be an algebraic evaluation code as above, with encoding map C_* . Given any basis $\{h_i\}_{i=0}^{k_*-1}$ of H , let*

$$h_T = \sum_{i=0}^{k_*-1} T_i h_i$$

be a symbolic combination in $\mathbb{F}_2(X)[T_0, \dots, T_{k_*-1}]$. Let

$$N_T = N_X(h_T) = \det \left(\sum_{i=0}^{k_*-1} T_i m_{h_i} \right)$$

be the symbolic norm. Then we can write $N_T = P(x, T_0, \dots, T_{k_*-1})/\sigma(x)^\mu$. Denote

$$B_{\text{norm}} = \deg_x(P(x, T_0, \dots, T_{k_*-1})). \quad (3.4)$$

Then every nonzero element $f \in H$ has at most B_{norm} zeros on $U(\mathbb{F})$. The same conclusion holds for nonzero $\mathbb{F}$ -linear combinations of the h_i after extending constants to $\mathbb{F}$, provided the scalar-extended function field remains a field.

Proof. Every element $f \in H$ is a linear combination of the h_i and therefore a specialization of h_T in the coefficients $f_0, \dots, f_{k_*-1}$. Thus

$$N_X(f)(x) = \det \left(\sum_i f_i m_{h_i} \right) = N_T(x, f_0, \dots, f_{k_*-1})$$

so its numerator degree is at most B_{norm} . The conclusion follows from Corollary 3.18. Because this argument is a symbolic specialization in the coefficients f_i , it is unchanged after extending the constants to $\mathbb{F}$. □

Given a specific basis, the symbolic norm N_T in Proposition 3.19 can be computed by a small script. This gives the practical bound used to analyze our explicit code.

More generally, any bound on the reduced numerator degrees that holds uniformly over nonzero $\mathbb{F}$ -linear combinations may be used for B_{norm} , by Eq. (3.3). If the basis functions are only defined on U_η , with $\eta h_i \in \mathcal{O}_U$, apply the norm calculation to ηH on U . Since η is nonzero on U_η , the zero bound there is unchanged.

3.4 The size of the evaluation set

An algebraic evaluation code $(\mathbb{F}_2(X), H, U, e)$ with encoding map C_* gives rise to a linear code in $\mathbb{F}^n$, where $n = |U(\mathbb{F})|$. Proving large distance also requires lower bounding n , which can be done using the Lang–Weil theorem: a variant of the Weil bound with input conditions that can be stated in the elementary field-theoretic language above.

Theorem 3.20 (Explicit Lang–Weil bound for equation models, [CM06, Theorem 7.1]). *Let U be an affine equation model for $\mathbb{F}_2(X)$, with nonzero defining polynomials $P_0, \dots, P_{c_{\text{eq}}-1}$, and put*

$$\Delta := \prod_{j=0}^{c_{\text{eq}}-1} \deg P_j,$$

where the degrees are total degrees and an empty product is 1. Suppose that $\mathbb{F}_2$ is the full constant field of $\mathbb{F}_2(X)$: every element of $\mathbb{F}_2(X)$ algebraic over $\mathbb{F}_2$ already lies in $\mathbb{F}_2$.

Then, for every finite extension $\mathbb{F}/\mathbb{F}_2$ with $q := |\mathbb{F}| > 4\Delta^2$ and every nonzero $\sigma \in \mathbb{F}_2[x]$,

$$|U_\sigma(\mathbb{F})| \geq q - (\Delta - 1)(\Delta - 2)\sqrt{q} - 5\Delta^{13/3} - \Delta \deg \sigma. \quad (3.5)$$

Proof. The field-quotient condition in Definition 3.2 gives an *irreducible component* $V \subseteq U$, defined over $\mathbb{F}_2$, with function field $\mathbb{F}_2(X)$. Finiteness over $\mathbb{F}_2(x)$ gives dimension one; the full-constant-field condition over the perfect field $\mathbb{F}_2$ gives *absolute irreducibility*. Bézout bounds its *degree* δ by Δ . Apply [CM06, Theorem 7.1] with $r = 1$, since $q > 4\Delta^2 \geq 4\delta^2$. Each fiber of x on V has at most δ points by Bézout, so excluding the roots of σ removes at most $\Delta \deg \sigma$ points. The inclusion $V_\sigma(\mathbb{F}) \subseteq U_\sigma(\mathbb{F})$ gives the stated bound. $\square$

If U_σ also satisfies the fiber bound of Lemma 3.16, excluding the roots of a further nonzero polynomial $\eta \in \mathbb{F}_2[x]$ removes at most $\nu \deg \eta$ points. Thus Eq. (3.5) gives the lower bound

$$|U_{\sigma\eta}(\mathbb{F})| \geq n_0 := q - (\Delta - 1)(\Delta - 2)\sqrt{q} - 5\Delta^{13/3} - \Delta \deg \sigma - \nu \deg \eta. \quad (3.6)$$

3.5 Efficient sampling

The projection $\pi_U : U(\mathbb{F}) \rightarrow \mathbb{F}$ from Lemma 3.16 has fibers of size at most ν . This gives a generic *bounded-fiber rejection sampler* whenever the solutions above a chosen x can be enumerated and ordered:

1. Sample $x \in \mathbb{F}$ uniformly.
2. Compute and order the fiber $\pi_U^{-1}(x)$.
3. Sample $j \in \{1, \dots, \nu\}$ uniformly. Output the j th point if it exists, and otherwise restart.

Every point of $U(\mathbb{F})$ is proposed with probability $1/(\nu|\mathbb{F}|)$ on each attempt, so the output is uniform. The expected number of attempts is $\nu|\mathbb{F}|/|U(\mathbb{F})|$. For the explicit code, the same procedure is implemented by solving the displayed equations one coordinate at a time and rejecting the values for which a denominator vanishes.

3.6 Summary: algebraic evaluation codes

The preceding bounds give the following criterion for a single evaluation code.

Theorem 3.21 (Algebraic evaluation codes). *Let $(\mathbb{F}_2(X), H, U, e)$ be an algebraic evaluation code as in Definition 3.12, with $U = V_{\sigma\eta}$ and the assumptions of Section 3.3 on V_σ and ηH . Assume the conditions of Theorem 3.20: $\mathbb{F}_2$ is the full constant field of $\mathbb{F}_2(X)$, and $q = |\mathbb{F}| > 4\Delta^2$, where $\Delta = \prod_j \deg P_j$ for the defining equations of V .*

Let P be the polynomial numerator in Proposition 3.19, applied to ηH . As in Eqs. (3.4) and (3.6), put

$$\begin{aligned} B_{\text{norm}} &:= \deg_x P, \\ n_0 &:= q - (\Delta - 1)(\Delta - 2)\sqrt{q} - 5\Delta^{13/3} - \Delta \deg \sigma - \nu \deg \eta. \end{aligned}$$

A smaller uniform norm bound may replace B_{norm} , as described immediately following Proposition 3.19.

If $n_0 > B_{\text{norm}}$, then C_* is injective, with length $n \geq n_0$, dimension k_* , and relative minimum distance

$$\delta_* \geq 1 - \frac{B_{\text{norm}}}{n} \geq 1 - \frac{B_{\text{norm}}}{n_0}.$$

Whenever the fibers can be enumerated and ordered, the sampler of Section 3.5 is uniform on $U(\mathbb{F})$ and takes at most $\nu|\mathbb{F}|/n_0$ expected attempts.

Proof. Equation (3.6) applied to V gives $n \geq n_0$. Applying Proposition 3.19 and Eq. (3.3) to ηH on V_σ bounds the number of zero coordinates for each nonzero message by B_{norm} , since η is nonzero on U . Thus $n \geq n_0 > B_{\text{norm}}$ gives injectivity and the distance bound. The sampling claim is Section 3.5 with $n \geq n_0$. $\square$

If we also have a k -dimensional multiplicative subspace $M \subseteq H$ as in Definition 3.13, then the associated code $C = C_* \circ R$ is a binary-preserving multiplication code with product code C_* , by Proposition 3.15.

4 Searching for algebraic evaluation codes

We now explain how curves and divisors produce the tuple defining an algebraic evaluation code as in Definition 3.12 and an appropriate multiplicative subspace as in Definition 3.13. For a given $k = \dim M$, we want to keep $k_* = \dim H$ and the cost of the message product small.

4.1 Curves, points at infinity, and divisors

For the abstract construction in this section, a *curve* X means a one-dimensional object over $\mathbb{F}_2$ that is also *smooth*, *projective*, and *geometrically connected*. Smoothness is the hypothesis that gives one local parameter at every point; projectivity adds the finitely many locations missing from an affine equation model; and geometric connectedness says that the curve remains in one piece after extending the field. Such a curve has a nonnegative integer invariant called its *genus*, denoted g_X . We use the standard correspondence between these curves and algebraic function fields as a black box: the explicit equations are used for validation, while the smooth projective curve is used for the divisor and Riemann–Roch arguments.

For the geometrically connected curves considered here, the scalar-extended function field remains a field, as required by Proposition 3.19.

For a field extension K of $\mathbb{F}_2$, write $X(K)$ for the K -valued points of the abstract curve. The affine equation model displays all but finitely many of them; the remaining locations are called its *points at infinity*. This phrase is relative to the chosen equations. For example, the affine line has K as its set of points, while the corresponding *projective line* has

$$\mathbb{P}^1(K) = K \cup \{\infty\}.$$

Near ∞ , the coordinate $t = 1/x$ turns a degree- d polynomial into

$$a_d x^d + \cdots + a_0 = t^{-d}(a_d + a_{d-1}t + \cdots + a_0 t^d),$$

so polynomial degree is exactly a pole budget at the point at infinity.

We next record the abstract language needed in the construction. A *closed point* Q may be pictured as a *Galois orbit* of points over an algebraic extension; its *degree* $\deg Q$ is the number of points in that orbit. In particular, an $\mathbb{F}_2$ -rational point has degree one. A *divisor* on X is a finite formal sum

$$D = \sum_Q n_Q Q, \quad n_Q \in \mathbb{Z},$$

with degree $\deg D = \sum_Q n_Q \deg Q$. A divisor is *effective* if every n_Q is nonnegative, and its *support* is the finite set of Q for which $n_Q \neq 0$. The coefficient n_Q is a local budget: a positive coefficient permits a pole at Q , while a negative coefficient requires a zero there.

For a nonzero function $f \in \mathbb{F}_2(X)$, its *order of vanishing* $\text{ord}_Q(f)$ is positive when f has a zero at Q and negative when it has a pole. Define

$$L(D) := \{0\} \cup \{0 \neq f \in \mathbb{F}_2(X) : \text{ord}_Q(f) + n_Q \geq 0 \text{ for every } Q\}, \quad \ell(D) := \dim_{\mathbb{F}_2} L(D).$$

Thus $L(D)$ is the vector space of functions obeying the pole and zero budgets in D , while $\ell(D)$ denotes only its dimension. Directly from the budgets,

$$D - E \text{ effective} \implies L(E) \subseteq L(D), \quad L(D)L(E) \subseteq L(D + E).$$

Reed–Solomon is the special case $X = \mathbb{P}^1$: for the point ∞ above,

$$L(d\infty) = \mathbb{F}_2[x]_{\leq d}.$$

This is the usual polynomial degree bound expressed as a pole budget.

4.2 Local coordinates and jet interpolation

To choose the basis map e and make the message product inexpensive, the construction below uses more than the value of a function at a point. Let $Q \in X(\mathbb{F}_2)$. A *local parameter* t_Q is a coordinate that vanishes to order one at Q . A function with no pole near Q then has a unique power-series expansion

$$f = a_0 + a_1 t_Q + a_2 t_Q^2 + \cdots.$$

For $f \in L(D)$, the same statement holds after choosing a *local scaling* that clears the pole permitted by D . The j th *jet evaluation* is the coefficient

$$J_Q^j(f) := a_j.$$

The parameter and scaling are part of the choice of jet coordinates. This formulation is more useful in characteristic two than repeated differentiation, since factorials can vanish. When the local scalings for $L(D)$, $L(E)$, and $L(D + E)$ are chosen compatibly, multiplication of power series gives

$$J_Q^j(fg) = \sum_{a+c=j} J_Q^a(f) J_Q^c(g). \quad (4.1)$$

For the equation model in Example 3.6, consider the point $(0, 0)$ and take $t = x$. Solving its equation in formal power series gives

$$y = t^3 + t^6 + t^{12} + t^{24} + \cdots.$$

The jets of an expression $a(x) + b(x)y$ can therefore be computed by substitution and truncation, and Eq. (4.1) becomes ordinary convolution of the coefficient vectors.

Now let $P \subseteq X(\mathbb{F}_2)$ have size k , and also write P for the divisor obtained by summing its points. Evaluation defines

$$\text{ev}_P : L(D) \longrightarrow \mathbb{F}_2^P, \quad f \longmapsto (f(Q))_{Q \in P},$$

using local scalings when needed. Its kernel is $L(D - P)$. Consequently,

$$\ell(D) = k, \quad \ell(D - P) = 0$$

say exactly that evaluation is an isomorphism $L(D) \simeq \mathbb{F}_2^P$.

More generally, write $k_* = sk + b$ with $0 \leq b < k$, and choose $P_{\text{extra}} \subseteq P$ of size b . Taking the first s jet coefficients at every point of P and one additional coefficient at the points of P_{extra} defines

$$\text{jet}_{P, P_{\text{extra}}} : L(2D) \longrightarrow (\mathbb{F}_2^P)^s \times \mathbb{F}_2^{P_{\text{extra}}}.$$

Its kernel is $L(2D - sP - P_{\text{extra}})$. Therefore

$$\ell(2D) = k_*, \quad \ell(2D - sP - P_{\text{extra}}) = 0$$

say exactly that these k_* jet coordinates give an isomorphism. These are the four interpolation conditions used in Theorem 4.1; in these coordinates, the message product is the coefficient convolution in Eq. (4.1).

4.3 Dimension, distance, and point counts

When choosing D , we need to control both $\ell(D)$ and $\ell(2D)$. The *Riemann–Roch theorem* relates these dimensions to the pole budgets:

$$\ell(D) = \deg D - g_X + 1 + h^1(D).$$

Here $h^1(D)$ is a nonnegative correction term, and it vanishes whenever $\deg D > 2g_X - 2$. Thus Riemann–Roch is the curve analogue of the fact that degree- d polynomials form a $(d+1)$ -dimensional space. This vector-space dimension is separate from the dimension-one check for the equation system above.

There is also an abstract zero bound for the divisor spaces used below. If $0 \neq f \in L(D)$, then the divisor $\text{div}(f) + D$ is effective and has degree $\deg D$. Its zeros, counted with *multiplicity* and extension degree, therefore have total degree at most $\deg D$. In particular, evaluating $L(D)$ at a set $\mathcal{Y}$ of rational points, with compatible local scalings at points in the support of D , gives a code of relative distance at least

$$1 - \frac{\deg D}{|\mathcal{Y}|}.$$

This generalizes the fact that a degree- d polynomial has at most d roots. It also implies that $\deg D < 0$ forces $L(D) = 0$.

The code length is controlled by a point count. For a smooth projective geometrically connected curve over $\mathbb{F}_q$, the *Hasse–Weil bound* says

$$||X(\mathbb{F}_q)| - (q + 1)| \leq 2g_X \sqrt{q}.$$

This is the estimate used in the abstract construction. Higher genus permits, but does not by itself guarantee, more than q rational points.

4.4 Abstract curve-and-divisor construction

From now on, fix $\mathbb{F} = \mathbb{F}_{2^{128}}$. The following theorem gives the conditions on the curve, divisor, and interpolation points that we search for. The construction is a jet-interpolation variant of a standard algebraic-geometry code; such codes were studied as “Differential Goppa codes” in [GCV26].

Theorem 4.1. *Let X be a smooth projective geometrically connected curve over $\mathbb{F}_2$ of genus $g_X \leq 1000$, and let $P \subseteq X(\mathbb{F}_2)$ have size k . Choose $k_* > k$, write $k_* = sk + b$ with $0 \leq b < k$, and choose $P_{\text{extra}} \subseteq P$ of size b . Suppose a divisor D satisfies the following conditions, where set names also represent the divisors obtained by summing their elements:*

$$\ell(D) = k, \quad \ell(2D) = k_*, \quad \ell(D - P) = 0, \quad \ell(2D - sP - P_{\text{extra}}) = 0.$$

Let $\mathcal{Y} \subseteq X(\mathbb{F})$ be any set of $n > 2 \deg D$ points. Then there is a binary-preserving multiplication code

$$C : \mathbb{F}^k \longrightarrow \mathbb{F}^n$$

with product code $C_* : \mathbb{F}^{k_*} \rightarrow \mathbb{F}^n$, systematic extension map R , and relative distance

$$\delta_* \geq 1 - \frac{2 \deg D}{n}. \tag{4.2}$$

In particular, taking $\mathcal{Y} = X(\mathbb{F})$ gives $\delta_* > 1 - 2^{-100}$. Writing product-code messages as

$$u = (u_0, \dots, u_{s-1}, u_s) \in (\mathbb{F}^k)^s \times \mathbb{F}^b,$$

define $B(u, v) = (z_0, \dots, z_s)$ by

$$z_j = \sum_{a+c=j} u_a \circ v_c, \quad (4.3)$$

with the last block restricted to the coordinates in P_{extra} . If $R: \mathbb{F}^k \rightarrow \mathbb{F}^{k_*}$ is the extension map, the message product is $v \times w = B(Rv, Rw)$, using

$$\frac{s(s+1)}{2}k + (s+1)b$$

coordinate multiplications. Together with the two applications of R , this gives product complexity $O(kk_* + s^2k)$ field operations. For $\mathcal{Y} = X(\mathbb{F})$, the code is therefore sumcheck-friendly with security parameter $\lambda = 100$ and this product-complexity bound.

Proof. The divisor need not be effective. Since $\ell(D) > 0$, choose $0 \neq a \in L(D)$ and put $D' := D + \text{div}(a)$. Then D' is effective, division by a identifies $L(D)$ with $L(D')$, and division by a^2 identifies $L(2D)$ with $L(2D')$ compatibly with multiplication. These identifications preserve all four dimension conditions. We may therefore replace D by D' and suppress the prime.

Extend scalars in $L(D)$ and $L(2D)$ from $\mathbb{F}_2$ to $\mathbb{F}$; their dimensions do not change. At each point of $\mathcal{Y}$, choose compatible local scalings for D and $2D$. Evaluation then respects multiplication. By assumption, $n > 2 \deg D$, so the zero bound makes evaluation on both spaces injective. The availability of such a large set is automatic in our parameter range: Riemann–Roch gives

$$\deg D \leq \ell(D) + g_X - 1 = k + g_X - 1,$$

and Hasse–Weil gives

$$|X(\mathbb{F})| \geq 2^{128} + 1 - 2g_X 2^{64}.$$

Because $k \leq |X(\mathbb{F}_2)| \leq 3 + 2g_X \sqrt{2}$ and $g_X \leq 1000$, the full set $X(\mathbb{F})$ has more than $2 \deg D$ points; the same estimates give $2 \deg D / |X(\mathbb{F})| < 2^{-100}$.

At the points of P , choose local parameters and compatible local scalings over $\mathbb{F}_2$. The first pair of dimension conditions then identifies $\mathbb{F}^P$ with $L(D)$ by inverse evaluation. The second pair identifies $(\mathbb{F}^P)^s \times \mathbb{F}^{P_{\text{extra}}}$ with $L(2D)$ by inverse jet interpolation. Composing these isomorphisms with evaluation on $\mathcal{Y}$ gives C and C_* ; composing the inclusion $L(D) \subseteq L(2D)$ with the two coordinate maps gives R . Its first block records the zeroth jets, so R is systematic.

The convolution formula Eq. (4.3) follows from Eq. (4.1), and hence

$$C(v) \circ C(w) = C_*(B(Rv, Rw)).$$

All points and coordinate choices used for R are defined over $\mathbb{F}_2$, so R and B preserve binary vectors. Finally, the zero bound for $L(2D)$ gives Eq. (4.2); the estimates above give the stated bound when $\mathcal{Y} = X(\mathbb{F})$. $\square$

Note. The parameters of the construction in Theorem 4.1 depend on the size of k_* , which can be minimized as follows: let X have genus g_X . By Riemann–Roch,

$$\begin{aligned} k &= \ell(D) = \deg D - g_X + 1 + h^1(D), \\ k_* &= \ell(2D) = 2 \deg D - g_X + 1 + h^1(2D), \end{aligned}$$

giving that

$$k_* = 2k + (g_X - 1) + h^1(2D) - 2h^1(D).$$

Therefore we want to choose X and D to minimize

$$(g_X - 1) + h^1(2D) - 2h^1(D).$$

For a typical choice of D , the h^1 -terms are expected to be zero, so this reduces to minimizing g_X .

4.5 Summary: the curve-and-divisor construction

In this construction, we search for a curve X , divisor D , and interpolation points satisfying the four dimension conditions in Theorem 4.1: Riemann–Roch controls $\ell(D)$ and $\ell(2D)$, local expansions provide the product rule, and Hasse–Weil provides enough evaluation points. Optimizing the parameters of the code reduces to optimizing the genus g_X of X .

After the effective-divisor replacement in the proof of Theorem 4.1, take $H = L(2D)$ and $M = L(D) \subseteq H$, and let e be inverse jet interpolation. Choosing an affine equation model U on which the basis functions are defined gives the tuple $(\mathbb{F}_2(X), H, U, e)$. Its evaluation-code properties can then be checked using Theorem 3.21.

5 AI search for explicit parameters

Finding explicit parameters to optimize the construction of Theorem 4.1 required heavy AI assistance, and we believe that the project would have been infeasible before modern LLMs. This section records how the models were used. The explicit data are described in Section A.

First, the project was the result of a long series of brainstorming sessions with GPT 5.3-5.5 Pro and Claude Opus 4.6-4.8 to try to find novel codes useful in cryptographic proof systems. The LLMs were presented with various human-generated ideas and asked to sanity-check their plausibility, fill in details, and refine constructions. We were not able to get the majority of these ideas to work, and none of them would have been worth examining in detail before LLMs: due to mathematical complexity, each would have been far too involved for how likely it was to actually produce something useful. Furthermore, the range of mathematical expertise needed to analyze all of them was unreasonable. The authors, in particular, did not have enough even to attempt to do so; only enough intuition about what tools from various fields could achieve to suggest ideas.

For example, the study of curves with many points over small fields is very well developed, with significant literature. Any one of the ~ 20 experts in the field worldwide would likely also have been able to find our genus-95 example. However, finding such an expert, cold-emailing them, and requesting that they spend at the very least a few days of their time on a speculative project for an application not even necessarily in their area of interest was completely impractical.

Beyond the difficulty of coming up with an abstract construction, implementing the code explicitly would have been impractically time-consuming before modern LLMs, requiring thousands of lines of Sage scripts for searches and computations that depended on specialized knowledge of algebraic geometry. Through better mathematical intuition, a human expert would have been able to make more efficient strategic choices in algorithms and search directions. However, the LLMs’ overwhelming advantage in quick script writing and testing let them produce a final answer much faster.

We believe that this project is a great example of how LLMs could be useful specifically in applied mathematics research. The usefulness of the deepest parts of modern mathematics in applications previously seemed to be capped by exactly these difficulties: the breadth of expertise needed to find a practical connection and the complexity of the specialized computations needed to actually make it explicit. Modern LLMs are particularly suited to resolving these issues, making feasible the application of vast and powerful new portions of modern mathematical literature.

5.1 Prompting

The majority of the AI work involved finding examples of algebraic objects (the X and D) satisfying certain properties. Until an example was found, the models were instructed to:

- guess at a plausible family of candidates to search through based on their understanding of the research literature;
- see if they could find an example in the family, running Sage scripts if they felt that these would be helpful; and

- record what they tried in a running Markdown file and repeat with the next most plausible family.

We often augmented these instructions with human-supplied hints when we felt that the AI was on the wrong track. The most common hint was “Do not write Sage scripts just yet. First do a careful abstract analysis to check if the search you are planning is actually likely to find a candidate”, since the models were more biased toward running computations than toward checking whether the computations were actually helpful to run.

Before starting the curve search, the AI model was also asked to prove an abstract lower bound on the genus. This gave the genus-89 Serre–Oesterlé bound. It was then asked to run this loop to find an example in genus exactly 89. After this did not work, it was asked to find an example with genus close to 89, which produced the above genus-95 example. Finally, it was asked to try to beat genus 95, which it was not able to do even after a day or two.

Once the examples were found, computing the details of the code in coordinates to generate Eq. (A.2) and the $\mathbf{R}, \mathbf{E}, \eta$ of Sections A.2 and A.3 was extremely fast. GPT 5.5 Pro generated them more or less in one shot by writing and running Sage scripts for about an hour or two.

5.2 Validation

Any AI-generated math needs to be carefully validated. In this case, we luckily do not care about the provenance of the code, only that it satisfies the combinatorial and algebraic properties in Definition 1.2. These can be checked elementarily without needing the heavy algebraic-geometry theory that went into constructing the code—see Section B.

We therefore do not rely on the AI’s geometric derivation of the curve and divisor. Instead, Section B checks the final equation model, product identities, matrix properties, norm bounds, and point-count hypotheses directly. Nevertheless, given the difficulty of finding Eq. (A.2) and the data $\mathbf{E}, \eta$ together with a matrix $\mathbf{R}$ satisfying Eq. (B.1), it is very unlikely that a correct code could have been generated from incorrect algebraic geometry.

Acknowledgments

We thank Lev Soukhanov for helpful conversations. We also thank Aleksander Makelov for providing a Chat GPT Pro subscription.

References

- [ARR25] Noor Athamnah, Noga Ron-Zewi, and Ron D. Rothblum. Linear prover IOPs in log star rounds. In *Theory of Cryptography - TCC 2025*, pages 335–368. Springer, 2025.
- [BCF⁺25] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D. Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. In *Advances in Cryptology - EUROCRYPT 2025*, pages 123–152. Springer, 2025.
- [BCG20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Theory of Cryptography - TCC 2020*, pages 19–46. Springer, 2020.
- [BFRW25] Benedikt Bünz, Giacomo Fenzi, Ron D. Rothblum, and William Wang. TensorSwitch: Nearly optimal polynomial commitments from tensor codes. *IACR Cryptol. ePrint Arch.*, 2025:2065, 2025. To appear in CRYPTO 2026.
- [BHI⁺24] Nir Bitansky, Prahladh Harsha, Yuval Ishai, Ron D. Rothblum, and David J. Wu. Dot-product proofs and their applications. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 806–825. IEEE, 2024.

- [BKK⁺16] Eli Ben-Sasson, Yohay Kaplan, Swastik Kopparty, Or Meir, and Henning Stichtenoth. Constant rate pcps for circuit-SAT with sublinear query complexity. *J. ACM*, 63(4):32:1–32:57, 2016.
- [BLNR22] Sarah Bordage, Mathieu Lhotel, Jade Nardi, and Hugues Randriam. Interactive oracle proofs of proximity to algebraic geometry codes. In *37th Computational Complexity Conference (CCC 2022)*, LIPIcs, pages 30:1–30:45, 2022.
- [BMMS25a] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, and Matan Shtepel. FICS and FACS: Fast IOPPs and accumulation via code-switching. *IACR Cryptol. ePrint Arch.*, 2025:737, 2025. To appear in CRYPTO 2026.
- [BMMS25b] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, and Matan Shtepel. Query-optimal IOPPs for linear-time encodable codes. *IACR Cryptol. ePrint Arch.*, 2025:1588, 2025. To appear in EUROCRYPT 2026.
- [BRW26] Benedikt Bünz, Ron D. Rothblum, and William Wang. Flock: Fast proving for batch Boolean computations. *CoRR*, abs/2607.27491, 2026.
- [BSCKL21] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Elliptic curve fast Fourier transform (ECFFT) part I: Fast polynomial algorithms over all finite fields. arXiv:2107.08473, 2021.
- [BSCKL22] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Scalable and transparent proofs over all large fields, via elliptic curves (ECFFT part II). In *Theory of Cryptography Conference (TCC)*, 2022.
- [CCH⁺19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat–Shamir: From practice to theory. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 1082–1090. ACM, 2019.
- [CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In *Advances in Cryptology – CRYPTO 2018*, 2018.
- [CM06] Antonio Cafure and Guillermo Matera. Improved explicit estimates on the number of solutions of equations over a finite field. *Finite Fields and Their Applications*, 12(2):155–185, 2006.
- [CR20] Alain Couvreur and Hugues Randriambololona. Algebraic geometry codes and some applications. arXiv:2009.01281, 2020.
- [DM13] Iwan Duursma and Kit-Ho Mak. On lower bounds for the Ihara constants $A(2)$ and $A(3)$. *Compositio Mathematica*, 149(7):1108–1128, 2013.
- [DP25] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In *Advances in Cryptology - EUROCRYPT 2025*, pages 93–122. Springer, 2025.
- [DV83] V. G. Drinfeld and S. G. Vlăduț. The number of points of an algebraic curve. *Functional Analysis and Its Applications*, 17:53–54, 1983.
- [GCV26] David González González, Ángel Luis Muñoz Castañeda, and Luis Manuel Navas Vicente. Differential goppa codes. *arXiv preprint arXiv:2603.04049*, 2026.
- [GLS⁺23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *Advances in Cryptology - CRYPTO 2023*, pages 193–226. Springer, 2023.

- [GNR26] Kobi Gurkan, Andrija Novakovic, and Ron D. Rothblum. Bolt: Faster SNARKs from sketched codes. *IACR Cryptol. ePrint Arch.*, 2026:310, 2026. To appear in CRYPTO 2026.
- [GRS25] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. Essential coding theory. <https://cse.buffalo.edu/faculty/atri/courses/coding-theory/book/web-coding-book.pdf>, 2025. Book draft, August 26, 2025.
- [Gru24] Angus Gruen. Some improvements for the PIOP for zerocheck. *IACR Cryptol. ePrint Arch.*, 2024:108, 2024.
- [HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Paper 2024/278, 2024.
- [HR22] Justin Holmgren and Ron Rothblum. Faster sounder succinct arguments and IOPs. In *Advances in Cryptology - CRYPTO 2022*, pages 474–503. Springer, 2022.
- [LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.
- [Mei13] Or Meir. IP = PSPACE using error-correcting codes. *SIAM J. Comput.*, 42(1):380–403, 2013.
- [NA25] Andrija Novakovic and Guillermo Angeris. Ligerito: A small and concretely fast polynomial commitment scheme. *IACR Cryptol. ePrint Arch.*, 2025:1187, 2025.
- [NST24] Vineet Nair, Ashish Sharma, and Bhargav Thankey. Brakingbase - a linear prover, poly-logarithmic verifier, field agnostic polynomial commitment scheme. *IACR Cryptol. ePrint Arch.*, 2024:1825, 2024.
- [QGZ26] Wenjie Qu, Yanpei Guo, and Jiaheng Zhang. Lightning, field-agnostic super-efficient polynomial commitment scheme. Cryptology ePrint Archive, Paper 2026/258, 2026.
- [RR24] Noga Ron-Zewi and Ron Rothblum. Local proofs approaching the witness length. *J. ACM*, 71(3):18, 2024. Preliminary version in FOCS 2020.
- [RR25] Noga Ron-Zewi and Ron Rothblum. Proving as fast as computing: Succinct arguments with constant prover overhead. *J. ACM*, 72(2):15:1–15:54, 2025. Preliminary version in STOC 2022.
- [Sch92] René Schoof. Algebraic curves over $\mathbf{F}_2$ with many rational points. *Journal of Number Theory*, 41(1):6–14, 1992.
- [Ser20] Jean-Pierre Serre. *Rational Points on Curves over Finite Fields*. Documents Mathématiques. Société Mathématique de France, 2020.
- [Sou] Lev Soukhanov. Hashcaster: Frobenius sumcheck for vectorized Boolean ops. <https://github.com/morgana-proofs/hashcaster>. Retrieved August 2026.
- [Sou26a] Lev Soukhanov. Hashcaster2. <https://github.com/levs57/hashcaster2>, 2026. Retrieved September 2026.
- [Sou26b] Lev Soukhanov. RMFE-skip. <https://hackmd.io/@levs57/HJkT2mEXMx>, 2026. Retrieved September 2026.
- [vdGHLR09] Gerard van der Geer, Everett W. Howe, Kristin E. Lauter, and Christophe Ritzenthaler. Tables of curves with many points. <http://www.manypoints.org>, 2009. Retrieved June 2026.
- [XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Advances in Cryptology - CRYPTO 2022*, pages 299–328. Springer, 2022.

A The explicit instance

This appendix records the instance-specific inputs produced by the search. The generic parameter accounting appears in Section 4.3, the abstract construction appears in Theorem 4.1, and the direct checks appear only in Section B.

A.1 Curve and divisor data

The search found a smooth projective geometrically connected curve $X/\mathbb{F}_2$ of genus 95 with $|X(\mathbb{F}_2)| = 64$. It is a $(\mathbb{Z}/2)^3$ -Artin–Schreier cover of a genus-4 curve X_0 from the manYPoints database [vdGHLR09]. An affine equation model for X_0 is

$$y^4 + (x+1)y^2 + (x^3+x)y + x^7 + x^3 = 0.$$

The cover adjoins z_i satisfying $z_i^2 + z_i = g_i$, where

$$\begin{aligned} g_1 &= \frac{(x^9 + x^6 + x^5 + x^4 + x^3 + x^2) + (x^5 + x^4 + x^3 + x^2 + 1)y + (x^4 + x^3 + x^2)y^2 + (x^3 + 1)y^3}{x^{10} + x^4 + 1}, \\ g_2 &= \frac{(x^8 + x^5 + x^4 + x^3 + x^2 + x) + (x^6 + x^5 + x^2 + x)y + (x^3 + x)y^2}{x^{10} + x^4 + 1} + \frac{(x^4 + x^2 + x)y^3}{(x+1)(x^{10} + x^4 + 1)}, \\ g_3 &= \frac{(x^6 + x^4 + x^3 + x^2) + (x^5 + 1)y + (x^3 + x^2 + x + 1)y^2 + (x^3 + x^2)y^3}{x^{10} + x^4 + 1}. \end{aligned} \quad (\text{A.1})$$

This adapts a construction from [Sch92]. The Serre–Oesterlé bound permits genus as low as 89, but we do not know an example improving genus 95.

Put

$$d(x) := (x+1)(x^{10} + x^4 + 1), \quad \gamma_i(x, y) := d(x)g_i(x, y) \in \mathbb{F}_2[x, y].$$

The affine equation model U is the polynomial system

$$\begin{cases} y^4 + (x+1)y^2 + (x^3+x)y + x^7 + x^3 = 0, \\ z_i^2 + z_i = \tau\gamma_i(x, y), \\ \tau d(x) = 1. \end{cases} \quad i = 1, 2, 3, \quad (\text{A.2})$$

The associated quotient algebra over $\mathbb{F}_2(x)$ is

$$K_U := \frac{\mathbb{F}_2(x)[y, z_1, z_2, z_3]}{(y^4 + (x+1)y^2 + (x^3+x)y + x^7 + x^3, z_i^2 + z_i - g_i \ (1 \leq i \leq 3))}.$$

The validation below certifies directly that this algebra is a field of degree 32 over $\mathbb{F}_2(x)$. The coordinate τ is uniquely determined by x , so for points of U we suppress it and write (x, y, z_1, z_2, z_3) . This notation does not distinguish the branches of the base equation at its singular points or at infinity. To list the rational points of X exactly, we therefore use the two-generator *place labels* printed by Sage. The eight rational points of X_0 ,

in the order used by the code, are

$$\begin{aligned}
q_0 &= \text{Place}(x^{-1}, x^{-2}y), \\
q_1 &= \text{Place}\left(x, \frac{y^3}{x^3+x} + \frac{y}{x^2+x}\right), \\
q_2 &= \text{Place}\left(x, \frac{y^3}{x^3+x} + \frac{(x^2+x+1)y}{x^2+x} + x+1\right), \\
q_3 &= \text{Place}\left(x, \frac{y^3}{x^3+x} + \frac{y^2}{x^2+x} + \frac{y}{x+1} + 1\right), \\
q_4 &= \text{Place}\left(x, \frac{y^3}{x^3+x} + \frac{y^2}{x^2+x} + \frac{xy}{x+1} + 1\right), \\
q_5 &= \text{Place}\left(x+1, \frac{y^3}{x^3+x} + \frac{y}{x^2+x}\right), \\
q_6 &= \text{Place}\left(x+1, \frac{y^3}{x^3+x} + \frac{y^2}{x^2+x} + \frac{y}{x+1} + 1\right), \\
q_7 &= \text{Place}\left(x+1, \frac{y^2}{x^2+x} + \frac{(x+1)y}{x} + 1\right).
\end{aligned}$$

Each q_r splits into eight points of X . Write $(q_r; \epsilon_1 \epsilon_2 \epsilon_3)$ for the point at which $z_i = \epsilon_i$, and put

$$\begin{aligned}
\mathcal{T} &= (000, 001, 010, 011, 100, 101, 110, 111), \\
\mathcal{T}' &= (000, 001, 010, 011, 101, 110, 100, 111).
\end{aligned}$$

For an ordered list $\mathcal{S}$ of triples, $(q; \mathcal{S})$ denotes the corresponding ordered block of eight points. The exact point order is

$$(P_1, \dots, P_{64}) = ((q_0; \mathcal{T}), (q_1; \mathcal{T}), (q_2; \mathcal{T}), (q_3; \mathcal{T}'), (q_4; \mathcal{T}), (q_5; \mathcal{T}), (q_6; \mathcal{T}), (q_7; \mathcal{T})). \quad (\text{A.3})$$

Thus the selected thirty points are, explicitly,

$$\begin{aligned}
P_{\text{extra}} &= \{(q_r; \epsilon) : 0 \leq r \leq 2, \epsilon \in \mathbb{F}_2^3\} \\
&\cup \{(q_3; 000), (q_3; 001), (q_3; 010), (q_3; 011), (q_3; 101), (q_3; 110)\} = \{P_1, \dots, P_{30}\}.
\end{aligned} \quad (\text{A.4})$$

Set $P := \{P_1, \dots, P_{64}\} = X(\mathbb{F}_2)$. Not every thirty-point subset of P satisfied the fourth interpolation condition, so selecting P_{extra} required a second search after the curve and divisor had been fixed.

For completeness, we also give residue-field coordinates for the three non-rational points in the divisor. Put

$$\begin{aligned}
h_8(T) &= T^8 + T^4 + T^3 + T^2 + 1, & \kappa_8 &= \mathbb{F}_2[T]/(h_8), \\
h_{10}(T) &= T^{10} + T^6 + T^5 + T^3 + T^2 + T + 1, & \kappa_{10} &= \mathbb{F}_2[T]/(h_{10}),
\end{aligned}$$

and write a_8 and a_{10} for the images of T . The point Q_{10} is the unique ramified point of X above the point of X_0 with

$$\begin{aligned}
x &= a_{10}^9 + a_{10}^8 + a_{10}^3 + a_{10}^2 + a_{10}, \\
y &= a_{10}^9 + a_{10}^8 + a_{10}^5 + a_{10}^4 + a_{10}^3 + 1.
\end{aligned}$$

Equivalently, its base point has the Sage label

$$\text{Place}(x^5 + x^2 + 1, y^2 + (x^4 + x^2 + 1)y + x^2 + x).$$

A closed point of degree d is specified by one $\mathbb{F}_{2^d}$ -valued solution below, with its *Frobenius conjugates* understood. The degree-eight point A_8 is

$$\begin{aligned}
x &= a_8^5 + a_8^4 + a_8^3 + a_8^2, & y &= a_8^3 + 1, \\
z_1 &= a_8^6 + a_8^4, & z_2 &= a_8^7 + a_8^6 + a_8^5 + a_8^4 + a_8^3, \\
z_3 &= a_8^5 + a_8^4 + a_8^3 + a_8.
\end{aligned}$$

The degree-ten point B_{10} is

$$\begin{aligned} x &= a_{10}^9 + a_{10}^7 + a_{10}^6 + a_{10}^5 + a_{10}^4, & y &= a_{10}^9 + a_{10}^5 + a_{10}^4 + a_{10}^2, \\ z_1 &= a_{10}^9 + a_{10}^7 + a_{10}^6 + 1, & z_2 &= a_{10}^7 + a_{10}^6 + a_{10}^5 + a_{10}^4 + a_{10}^3 + 1, \\ z_3 &= a_{10}^9 + a_{10}^7 + a_{10}^6 + a_{10}^3 + a_{10}. \end{aligned}$$

The divisor found by the search is therefore

$$D = 16Q_{10} + A_8 - B_{10}, \quad \deg D = 158,$$

where the subscripts record the point degrees. The exact computations made during the search give

$$\ell(D) = 64, \quad \ell(2D) = 222, \quad \ell(D - P) = 0, \quad \ell(2D - 3P - P_{\text{extra}}) = 0. \quad (\text{A.5})$$

Thus $k = 64$, $k_* = 222 = 3 \cdot 64 + 30$, $s = 3$, and $b = 30$. The divisor D is not assumed effective; the proof of Theorem 4.1 explains why this is harmless. The first equality includes the computed condition $h^1(D) = 0$. The second also follows from Riemann–Roch because $\deg(2D) = 316 > 2g_X - 2 = 188$, so $h^1(2D) = 0$. The degree checks behind the two vanishing statements are

$$\deg(D - P) = 94, \quad \deg(2D - 3P - P_{\text{extra}}) = 94.$$

Their vanishing in Eq. (A.5) is an additional output of the search, not a consequence of negative degree.

A.2 Code data and maps

The final code is specified by a binary matrix and a vector of rational functions:

$$\mathbf{R} = \begin{pmatrix} \mathbf{I}_{64} \\ \mathbf{R}_0 \end{pmatrix} \in \mathbb{F}_2^{222 \times 64}, \quad \tilde{E} = (\tilde{E}_0, \dots, \tilde{E}_{221}) \in K_U^{222}.$$

All entries of $\tilde{E}$ have the common denominator

$$\eta(x) = x^{49} + x^{47} + x^{45} + x^{37} + x^{35} + x^{31} + x^{23} + x^{21} + x^9 + x^7 + x^5 + x. \quad (\text{A.6})$$

The evaluation set and the three maps are

$$\begin{aligned} \mathcal{Y} &:= \{Q \in U(\mathbb{F}) : \eta(x(Q)) \neq 0\}, \\ R(v) &:= \mathbf{R}v, \\ C_*(w)(Q) &:= w \cdot \tilde{E}(Q), \\ C(v) &:= C_*(Rv). \end{aligned}$$

This is exactly the concrete framework of Section 3, with $k = 64$, $k_* = 222$, and $\nu = 32$.

On the affine equation model U from Eq. (A.2), elements of $L(D)$ and $L(2D)$ are represented by rational expressions in x, y, z_1, z_2, z_3 . We use the local scalings and local parameters from Theorem 4.1 to interpret their values and jet coordinates at P .

Inverse evaluation at P gives a basis $(f_Q)_{Q \in P}$ of $L(D)$ characterized by

$$f_Q(Q) = 1, \quad f_Q(Q') = 0 \quad (Q' \in P \setminus \{Q\}).$$

The AI computed rational expressions for these 64 functions. It then computed their jet coefficients to obtain the matrix $\mathbf{R}$ of the extension map. The same functions give the optional direct evaluation rule for C .

Likewise, inverse jet interpolation gives a basis of $L(2D)$ indexed by the 222 jet coordinates, with each basis function having one jet coordinate equal to 1 and all others equal to 0. The vector $\tilde{E}$ consists of these basis functions and therefore defines the evaluation rule for C_* .

These geometric facts explain how the data were found; the validation below checks the resulting identities directly in K_U .

A.3 Machine-readable representation

With zero-based indices $0 \leq i < 222$ and $0 \leq j < 64$, encode row i of $\mathbf{R}$ as a u64 whose bit j is $\mathbf{R}_{ij}$; thus column j , which corresponds to P_{j+1} , is bit j , and column 0 is the least-significant bit. The exact values are listed in increasing row order, four row-value pairs per printed line:

row u64	row u64	row u64	row u64
<i>Rows 0–63: systematic block I_{64}</i>			
0 0x0000000000000001	1 0x0000000000000002	2 0x0000000000000004	3 0x0000000000000008
4 0x0000000000000010	5 0x0000000000000020	6 0x0000000000000040	7 0x0000000000000080
8 0x0000000000000100	9 0x0000000000000200	10 0x0000000000000400	11 0x0000000000000800
12 0x0000000000001000	13 0x0000000000002000	14 0x0000000000004000	15 0x0000000000008000
16 0x0000000000010000	17 0x0000000000020000	18 0x0000000000040000	19 0x0000000000080000
20 0x0000000000100000	21 0x0000000000200000	22 0x0000000000400000	23 0x0000000000800000
24 0x0000000001000000	25 0x0000000002000000	26 0x0000000004000000	27 0x0000000008000000
28 0x0000000010000000	29 0x0000000020000000	30 0x0000000040000000	31 0x0000000080000000
32 0x0000000100000000	33 0x0000000200000000	34 0x0000000400000000	35 0x0000000800000000
36 0x0000001000000000	37 0x0000002000000000	38 0x0000004000000000	39 0x0000008000000000
40 0x0000010000000000	41 0x0000020000000000	42 0x0000040000000000	43 0x0000080000000000
44 0x0000100000000000	45 0x0000200000000000	46 0x0000400000000000	47 0x0000800000000000
48 0x0001000000000000	49 0x0002000000000000	50 0x0004000000000000	51 0x0008000000000000
52 0x0010000000000000	53 0x0020000000000000	54 0x0040000000000000	55 0x0080000000000000
56 0x0100000000000000	57 0x0200000000000000	58 0x0400000000000000	59 0x0800000000000000
60 0x1000000000000000	61 0x2000000000000000	62 0x4000000000000000	63 0x8000000000000000
<i>Rows 64–221: remaining block</i>			
64 0x12524642a1f3a14e	65 0x76d50d084b8d65ee	66 0xf73d75bcacf36d76f	67 0x9e68e02699537209
68 0x5b083dc890e78433	69 0x845267390dbb374e	70 0xa0e0dc4e833cdf6a	71 0xa8519d5bc0472d03
72 0xa0acf147ed3c1bfe	73 0xdf039dead59a9e8a	74 0x987786f8322bd66e	75 0xc9ac91b7e43a7d61
76 0x9e819bcd70937a1a	77 0xded2c75fe805f35e	78 0xcdceef1cb20a34f7f	79 0xe0698a169dd8aef4
80 0x4ce510edee796eb6	81 0x0834fea9c96429a8	82 0x615bfe59c00b20a8	83 0xd3ea80ae42779f7d9
84 0x2af4458b9968197a	85 0xa11a58ff114a6e97	86 0xb1ef586886b5eb80	87 0xdcd4bc14a098707c
88 0xedb536fd0b3cd73d	89 0x4d83336da2f9e2fb	90 0x6b133635c6959d70	91 0x133e284ebe8716ad
92 0xbe8003e902d833ad	93 0xc1759c715c484043	94 0x814fff25a2273c92	95 0x57d0f51b36d2dae9
96 0xa00d02a37322f1544	97 0x09579bfe54ba8321	98 0xeb8f2adfd0ce6866	99 0x4dd54c89b9ab01f3
100 0x7da702e38c58bf99	101 0xe61f7a5ac52577c6	102 0x5c2629f1071031ed	103 0x323e0b2d1cdca61d
104 0x99f2bb0469231fcf	105 0xf18a308183a1d7a0	106 0x6815a088aa7be39	107 0xc06127c29cd685aa
108 0x2707e4c29c8c042b	109 0x4eae8da759196db2	110 0xbb6182abcfbf321d	111 0x868f6047e3545027
112 0x6ae400a5608a8541	113 0xc430e42549972282	114 0x667ec05ca01f4021	115 0xc948c5fff05d9491b
116 0x3ef9d2545d6235a6	117 0xc205e268019e3a95	118 0x3d931d0808f85563	119 0x27952b33fa0449b1
120 0x65b8e998c9701e6e	121 0x6d0a9782d60bdc38	122 0x2d0a0bbc8bf1a32c	123 0x80edd00cfe20cb2f
124 0xb3397a82c7f42362	125 0xe88d31f77d791732	126 0x09f3492c5934554c	127 0xbaa01582319d2304
128 0x03db7670a07f1021	129 0x89d745db5b46f639	130 0x5eac57a07c612ebd	131 0xadac949c20aedd8d
132 0x9be41f7e62f1d7f5	133 0xed1b201a9af701e1	134 0xe300aefcd92c3611	135 0xadb7d13d6399bc787
136 0x2df0d02c8fa1f3a5	137 0x20d102c0fc499d63	138 0xd027f4a9334f6d6c	139 0x72fcdce5baadf65f
140 0x21b70e20c2b35595	141 0xcb4208173fb386b4	142 0x5ca91fcb8c914ed4	143 0x27b801b0fbccc50a
144 0xd81bfbb8ce5d105f	145 0xdfa5896d8126e1f9	146 0x327438bcc5cd6321	147 0x953040397fb36772
148 0xdf000d3705349a14	149 0xff5a57f937a4169a	150 0x08b1b9234b16def5	151 0x742e29bedc4368b1
152 0xc0cb49fcd8e51702	153 0x7f671651bf70618a	154 0x1f0789384f86678a	155 0xdd1597db9c6efa4c
156 0x3d25320de597b515	157 0x4eef040fe23e0032	158 0x62ecf4924f6d4fdf	159 0x7def0bfccee02333e
160 0xf5af4d40073e9fbf	161 0x1424caae5457134	162 0x92213aafbdd3022a	163 0x0ee403ffe425f8ef
164 0xb5b81d8830b58f8f	165 0x35e47dda582f938	166 0x08a5af5377cecd9d	167 0xfdc5006a179da1fd
168 0xf25bcb8f51f204f7	169 0x11fea142cfad88b5	170 0xb5b68c5419d313e0	171 0x52ae584f64076e04
172 0xf498086eb364e58f	173 0x3d9bc752bc97eab3	174 0xc83bfeae2a37bc83	175 0xb1c732c58e31a55b
176 0x4cd4edf2ebdd6492	177 0x5d9ca5dc5aaabad6	178 0xaa7144ff2f877d26	179 0xd3428b6424426402
180 0x424e4795fcd524a7	181 0xf77287015accd4f	182 0xe421d89c5d05e198	183 0x871bd1cd0a700e792
184 0x339427131bbcdede	185 0x4fc7448e83da674a	186 0x71d6c062cc9d3c26	187 0x693e1bb641767d0d
188 0x5875f6cc2ab99d91	189 0x6664282f56920f57	190 0x9a548e2cf05450f1	191 0x5dda30c643e0a2a8
192 0x430dfb71df88cb91	193 0x3391a8ef4bdd82d5	194 0xc0a1f1ec1274247c	195 0x874cd067c999a0a2
196 0x81b573527709cf3d	197 0x70a85e8e5be1a207	198 0xdfa2c2216182316b	199 0xeb6b1962446c5a58
200 0xc10e9d506eb47847	201 0x3ef1916f3177b848	202 0x93d94c7028ff4c7b	203 0x6efb6e6fab271242
204 0x24cb1f1f4421103c	205 0xe708e3d01421dff0	206 0x001094023f6c6b18	207 0x64ab1c48d92d6047
208 0x24b70eb317efa9cb	209 0x422268e52d75c092	210 0x924df2bbb61fb08f	211 0x22de9e252179c63b
212 0x023938a6cdcedf51	213 0x96b270e3788c28fd	214 0x0bf6a7aa6d0e4f34	215 0x4d39a4fe99c4a623
216 0x70f67229ae1f5434	217 0x8744ccf04268a59e	218 0xd141395fb76b5f62	219 0x6a53242b6fb6e88c
220 0xe155882d033de134	221 0x2aedc3a41a2c1a07		

To store $\tilde{E}$, set

$$(m_0, \dots, m_7) = (1, z_3, z_2, z_2 z_3, z_1, z_1 z_3, z_1 z_2, z_1 z_2 z_3)$$

and record a polynomial table

$$\mathbf{E} \in \mathbb{F}_2[x]^{222 \times 4 \times 8}.$$

The rational functions are recovered by

$$\tilde{E}_i(x, y, z_1, z_2, z_3) = \eta(x)^{-1} \sum_{j=0}^3 \sum_{a=0}^7 \mathbf{E}_{ija}(x) y^j m_a. \quad (\text{A.7})$$

The accompanying repository, <https://github.com/succinctlabs/AG-skip-Verification-Certificate>, contains a machine-readable copy of the array above and the coefficients of $\mathbf{E}$. With SageMath installed, run `python3 verify.py` from the repository root to check the equation-level certificate in Section B. The repository documentation maps each check to the corresponding claim below.

A.4 Sampler for the explicit model

The generic bounded-fiber sampler of Section 3.5 specializes as follows. Sample $x \in \mathbb{F}$ and reject if $d(x)\eta(x) = 0$. Order the at most four solutions y of the first equation in Eq. (A.2), choose a uniform index in $\{1, 2, 3, 4\}$, and reject if that solution does not exist. If the three Artin–Schreier equations have solutions, they give exactly eight triples (z_1, z_2, z_3) ; choose one uniformly. Otherwise, reject. This is uniform on $\mathcal{Y}$.

The quartic equation for y can be rewritten as two successive equations of the form $a^2 + a = \alpha$, and the three equations for the z_i already have that form. Over $\mathbb{F}$, such an equation is solvable exactly when $\text{tr}_{\mathbb{F}/\mathbb{F}_2}(\alpha) = 0$, and its two solutions then differ by 1. Since $a \mapsto a^2 + a$ is $\mathbb{F}_2$ -linear, a right inverse on its image can be precomputed once.

B Independent validation of the explicit code

This section applies Theorem 3.21 to the evaluation code C_* in Section A.2 and checks the multiplication identity separately. It does not use the divisor D , the dimension calculations in Eq. (A.5), or the asserted genus of X .

Let

$$B : \mathbb{F}^{222} \times \mathbb{F}^{222} \longrightarrow \mathbb{F}^{222}$$

be the block convolution in Eq. (4.3) for $222 = 3 \cdot 64 + 30$. The exact product identity to be checked is

$$C_*(B(\mathbf{R}u, \mathbf{R}v)) = C_*(\mathbf{R}u) \circ C_*(\mathbf{R}v) \quad (u, v \in \mathbb{F}^{64}). \quad (\text{B.1})$$

The validation script reduces both sides in the coordinate algebra of Eq. (A.2) for all $64^2 = 4096$ pairs of standard basis vectors (u, v) . Bilinearity proves Eq. (B.1) for every pair. The displayed form of $\mathbf{R}$ proves rank 64 and makes the extension map systematic, while the block formula proves product-systematicity. Because $\mathbf{R}$ and B are defined over $\mathbb{F}_2$, they also prove binary preservation. Random evaluation tests used during development are useful smoke tests, but are not part of this certificate.

It remains to certify the size of $\mathcal{Y}$ and the distance of C_* .

B.1 Checking the evaluation-code hypotheses

We first check Definition 3.12 for the data in Section A.2, then the additional norm and point-count assumptions of Theorem 3.21. For the point count, take the system V obtained by clearing the denominators in the four equations for (x, y, z_1, z_2, z_3) . Then $V_d = U$ is the model of Eq. (A.2), and $V_{d\eta}(\mathbb{F}) = \mathcal{Y}$.

The direct algebra certificate verifies that the quotient of the cleared equations over $\mathbb{F}_2(x)$ is a field

$$K_V := K_U \quad \text{with} \quad [K_V : \mathbb{F}_2(x)] = \nu = 32. \quad (\text{B.2})$$

Concretely, the script gives a 32-element basis and a degree-32 *minimal polynomial* for the *primitive element*

$$\alpha = y + z_1 + xz_2 + x^2z_3.$$

It verifies that this polynomial remains irreducible over $\overline{\mathbb{F}_2}(x)$. Thus the scalar-extended field remains a field, or equivalently, $\mathbb{F}_2$ is the full constant field of K_V .

The same 32 monomials form a free basis of the equation quotient over $\mathbb{F}_2[x, 1/d]$. Its map to K_V is therefore injective, verifying Eq. (3.2), and each fiber away from $d = 0$ has at most $\nu = 32$ points. Adjoining $1/\eta$ gives the same identification for the evaluation model U_η over $\mathbb{F}_2[x, 1/(d\eta)]$. Notice that the field in Eq. (B.2) is finite-dimensional over $\mathbb{F}_2(x)$, not over $\mathbb{F}_2$. This is the required one-free-parameter certificate.

For comparison,

$$(x, y, z_1, z_2, z_3) = (0, 0, 0, 0, 0)$$

is an $\mathbb{F}_2$ -point on the locus $d(x) \neq 0$ (with $\tau = 1$ in Eq. (A.2)), but the existence of this point alone would not prove that $\mathbb{F}_2$ is the full constant field. The constant-field irreducibility check above is the indispensable Lang–Weil input.

To finish checking Definition 3.12, the entries $\eta\tilde{E}_i$ are polynomial expressions in x, y, z_1, z_2, z_3 , so they belong to $\mathcal{O}_U$, and the $\tilde{E}_i$ are defined on U_η . Their linear independence is the nonvanishing assertion in Eq. (B.4), checked in the next subsection. Thus

$$H := \text{span}_{\mathbb{F}_2}\{\tilde{E}_0, \dots, \tilde{E}_{221}\}, \quad e(w) := w \cdot \tilde{E}, \quad \dim_{\mathbb{F}_2} H = 222.$$

This gives the algebraic evaluation code (K_U, H, U_η, e) with $k_* = 222$. The dimension check uses only the displayed functions, not the Riemann–Roch calculation. The norm-degree assertion in Eq. (B.4) supplies $B_{\text{norm}} = 2248$ by the denominator-clearing calculation in Section 3.3.

It remains to check the numerical conditions in Theorem 3.21. The cleared equations have degrees 7, 12, 12, 13, so

$$\Delta = 7 \cdot 12 \cdot 12 \cdot 13 = 13104, \quad 2^{128} > 4\Delta^2.$$

Since $\deg d = 11$ and $\deg \eta = 49$, Eq. (3.6) with $\sigma = d$ gives

$$\begin{aligned} n = |\mathcal{Y}| \geq n_0 &= 2^{128} - (\Delta - 1)(\Delta - 2)2^{64} - 5\Delta^{13/3} - 11\Delta - 49\nu \\ &\geq 2^{128} - 2^{92} > 2248. \end{aligned} \tag{B.3}$$

The sampler in Section A.4 enumerates the fibers, so the sampling conclusion of Theorem 3.21 also applies, with at most $32 \cdot 2^{128} / (2^{128} - 2^{92})$ expected attempts.

B.2 Norm and distance

For $w \in \mathbb{F}^{222}$, put

$$p_w := \eta(x)(w \cdot \tilde{E}).$$

Since η is the common denominator of the entries of $\tilde{E}$, we have

$$p_w \in \mathcal{O}_U \otimes_{\mathbb{F}_2} \mathbb{F}.$$

Put $K_{V, \mathbb{F}} := K_V \otimes_{\mathbb{F}_2} \mathbb{F}$. The full-constant-field certificate in the preceding subsection shows that this is a field of degree 32 over $\mathbb{F}(x)$. Multiplication by p_w on this 32-dimensional vector space therefore has field norm

$$N_{K_{V, \mathbb{F}}/\mathbb{F}(x)}(p_w) \in \mathbb{F}(x).$$

There are three useful levels at which to bound the number of zeros. The first uses only degree accounting in the explicit equations and coefficient table. Give x, y, z_1, z_2, z_3 weights 1, 7/4, 0, 0, 0, respectively. The reduction rules do not increase weight, and inspection of **E** gives weighted degree at most 49 for p_w . Multiplying its multiplication matrix by $(x^{10} + x^4 + 1)^3(x + 1)$, of degree 31, clears all denominators. If the i -th basis

monomial has weight ω_i , the cleared entry in row i , column j has degree at most $80 + \omega_j - \omega_i$. The weights cancel in each determinant term, so the norm has numerator degree at most

$$32 \cdot 80 = 2560.$$

The implication that a zero of p_w lies above a zero of this determinant, together with the fiber bound 32 from Lemma 3.16 and a conservative allowance for the $49 + 11 = 60$ excluded values of x , gives

$$Z_{\text{raw}} \leq 32 \cdot 2560 + 32 \cdot 60 = 83840.$$

Together with Eq. (B.3), this proves the equation-level bound

$$\delta_* \geq 1 - \frac{83840}{2^{128} - 2^{92}} > 1 - 2^{-111}.$$

The second level uses the multiplicity statement in Lemma 3.17. If p_w vanishes at a points in the fiber over x_0 , then its norm is divisible by $(x - x_0)^a$; thus the factor of 32 in the raw fiber count is unnecessary. Moreover, an exact symbolic calculation from the table **E** gives the sharper degree bound

$$0 \neq p_w \quad \text{and} \quad \deg_x \text{num } N_{K_{V,\mathbb{F}}/\mathbb{F}(x)}(p_w) \leq 2248 \quad \text{for every } 0 \neq w \in \mathbb{F}^{222}. \quad (\text{B.4})$$

For this calculation, the equations assign fractional x -degrees to y, z_1, z_2, z_3 , which bound the pole at infinity, while a polynomial of degree 16 clears all finite poles contributed by the displayed denominators and powers of the z_i . Multiplication of the resulting pole bound by the extension degree 32, together with inspection of the exact coefficients of **E**, gives 2248. A direct Bézout calculation gives another equation-level bound, but a weaker one. Because η is nonzero on $\mathcal{Y}$, the functions p_w and $w \cdot \tilde{E}$ have the same zeros there. The norm criterion Eq. (3.3) therefore gives at most 2248 zero coordinates. Combining this with Eq. (B.3) proves that C_* is injective and

$$\delta_* \geq 1 - \frac{2248}{2^{128} - 2^{92}} > 1 - 2^{-116}.$$

This completes the hypotheses of Theorem 3.21 for the single evaluation code C_* . Together with the separate matrix and product checks above, it proves all the properties required in Definition 1.2 directly from the explicit data.

The third level uses the geometric construction. If one additionally trusts that the claimed smooth projective curve and divisor construction are correct, the zero bound for $L(2D)$ is

$$Z_{\text{geom}} \leq 2 \deg D = 316.$$

By Eq. (4.2), this gives more than 119 bits of security. The three bounds therefore give, respectively, 111, 116, and 119 bits of security; the first two use only the equation-level validation, while the last uses the full curve-and-divisor interpretation.

C Comparison with the asymptotic behavior

In this appendix, we explain why the parameters of our code are better than what one would expect from the asymptotic theory of algebraic-geometry codes.

Consider codes of the type constructed in this work: the k bits of the message are placed on k rational points of a curve X over $\mathbb{F}_2$ of genus g , and the divisor D is non-special of degree $k + g - 1$, which is the best case. By Riemann–Roch, the product message has length

$$k_* = 2k + g - 1,$$

so its suffix has $k + g - 1$ bits, that is, $1 + (g - 1)/k$ bits per data bit. The quality of such a code is therefore governed by the ratio g/N for curves over $\mathbb{F}_2$ with $N \geq k$ rational points.

The asymptotic picture. Let $N_2(g)$ denote the maximum number of rational points of a curve of genus g over $\mathbb{F}_2$, and let $A(2) := \limsup_{g \rightarrow \infty} N_2(g)/g$ be the corresponding Ihara constant. The Drinfeld–Vlăduț bound [DV83] states that $A(q) \leq \sqrt{q} - 1$, so $A(2) \leq \sqrt{2} - 1 \approx 0.414$. Consequently, as k grows, any curve with at least k rational points has genus $g \geq (1 + \sqrt{2} - o(1))k \approx 2.41k$. Every construction of the above type therefore has

$$k_* \gtrsim 4.41k \quad \text{and a suffix of at least} \quad \approx 3.41k$$

bits. With the best known families of curves over $\mathbb{F}_2$ (for which N/g is roughly 0.317; see [DM13, Proposition 5.5]), the suffix is roughly $4.15k$ bits.

Our code. For $k = 64$, our curve has genus 95, giving $k_* = 222 \approx 3.47k$ and a suffix of $158 \approx 2.47k$ bits. This is well below the asymptotic *lower bound* of about $3.41k$, which applies to every family. The reason is that the asymptotic bounds are far from tight for curves of small genus: such curves can have many more rational points relative to their genus than is possible asymptotically. For example, for $N = 64$ rational points, the Serre–Oesterlé bound (see, e.g., [Ser20]) only forces $g \geq 89$, that is, $g/N \approx 1.39$ rather than about 2.41.

Consequences. The same reasoning shows that, for constructions that place all message bits on rational points, the cost per data bit necessarily grows toward the asymptotic regime as the message length k increases. Thus, parameters as favorable as ours here are a feature of the small message lengths that are relevant in practice. Nevertheless, the suffix of the Reed–Solomon product message has about $\log_2(2k)$ bits per data bit, which grows without bound. Hence, even asymptotically, AG-based codes eventually outperform Reed–Solomon, though by a smaller margin than at $k = 64$.