Audit of PeerDAS KZG libraries

Date: September 9th, 2025

Introduction

On July 21st, 2025, the Ethereum Foundation engaged zkSecurity to perform a security assessment of the KZG libraries used by PeerDAS. These libraries include blst, c-kzg-4844, rust-eth-kzg, and go-eth-kzg. The goal of this assessment was to conduct an in-depth analysis of the various KZG polynomial commitment implementations (C, Rust, and Go) used for Ethereum's EIP-4844 and EIP-7594, as well as some of their dependencies in the `blst` BLS12-381 library. Over the course of the engagement, we evaluated compliance with relevant specifications, compared cross-implementation behaviors, and identified potential safety issues. Specified bindings and assembly routines were excluded from the scope as defined.

The engagement lasted four weeks and was conducted by two consultants. During this period, several observations and findings were identified and communicated to the respective library development teams. The detailed findings and their implications are discussed in the subsequent sections of this report.

Scope

- **`blst` library** at target commit v0.3.15: The scope includes implementations for `blst_p1s_mult_pippenger_scratch_sizeof` and `blst_p1s_mult_pippenger`, covering lower-level elliptic curve operations. However, the ASM implementations of finite field arithmetic are excluded.
- **`c-kzg-4844` library** at target commit 669e7484: The scope includes code associated with EIP-7594, such as `common/lincomb.c` and all of `src/eip7594`. It also includes `bindings`, except for `bindings/python`, `bindings/elixir`, and `bindings/node.js`.
- **`rust-eth-kzg` library** at target commit v0.7.1: The scope covers code related to both EIP-4844 and EIP-7594, including bindings, except for `bindings/golang`.
- **`go-eth-kzg` library** at target commit v1.3.0: The scope includes code associated with EIP-7594, excluding `internal/kzg`, `internal/multiexp`, `internal/utils`, `tests`, and all `_test.go` files.

Overview

EIP-7594

Following EIP-4844, a blob-carrying transaction in Ethereum can publish a blob containing 4096 field elements along with its KZG commitment. Validators must retrieve the entire blob and verify the commitment against it. EIP-7594 introduces 1-D peer DAS (Data Availability Sampling) for Ethereum. The blob is first extended using erasure coding (doubling its size). The extended blob is then divided into 128 cells, each containing 64 field elements. The blob is associated with a KZG commitment, and each cell has a KZG multi-proof to verify that its 64 field elements are consistent with the blob commitment. To ensure data availability, each validator only needs to receive and verify a small subset (e.g., 8) of the cells from the network, instead of downloading and verifying the entire blob as required in EIP-4844.

Cell Proof Generation

During block proposal, the proposer splits the blob into cells and generates a proof for each cell. First, the blob is extended with erasure coding to double its size, and then the extended blob is divided into cells, each containing 64 field elements. Next, the KZG multi-point proof for each cell is computed. The FK20 paper describes an efficient algorithm for computing all cell proofs of a blob. Below is the function signature for cell proof generation in `c-kzg-4844`:

```
C_KZG_RET compute_cells_and_kzg_proofs(
    Cell *cells, // The output cells
    KZGProof *proofs, // The output proofs for each cell
    const Blob *blob, // The input blob data
    const KZGSettings *s // The settings, including the trusted setup and
precomputation
);
```

Cell Proof Verification

Validators retrieve a subset of cells and proofs from the network and verify these KZG multi-point proofs against the blob commitment. Since a block can contain multiple blobs, validators use a universal verification method to efficiently verify cells from different blobs in a single batch. Below is the function signature for cell proof verification in `c-kzg-4844`:

```
C_KZG_RET verify_cell_kzg_proof_batch(
    bool *ok, // The verification result
    const Bytes48 *commitments_bytes, // The commitment of the entire blob that the
cell belongs to
    const uint64_t *cell_indices, // The indices of the cells in the blob
    const Cell *cells, // The array of cell data
    const Bytes48 *proofs_bytes, // The proofs for the cells
    uint64_t num_cells, // The number of cells in this batch, specifying the length
of the above arrays
    const KZGSettings *s // The settings, including the trusted setup and
precomputation
);
```

Blob Recovery

To recover the original blob, other nodes (e.g., index nodes) can retrieve cell data from the network. Once more than half of the cell data is retrieved, the entire blob can be reconstructed using erasure coding. Below is the function signature for blob recovery in `c-kzg-4844`:

```
C_KZG_RET recover_cells_and_kzg_proofs(
    Cell *recovered_cells, // The output recovered cells
    KZGProof *recovered_proofs, // The output recovered proofs for each cell
    const uint64_t *cell_indices, // The input indices of each cell
    const Cell *cells, // The input cell data
    uint64_t num_cells, // The number of input cells
    const KZGSettings *s  // The settings, including the trusted setup and
precomputation
);
```

Multi-Scalar Multiplication (MSM) in blst

The `blst` library implements the BLS12-381 elliptic curve and pairing operations in C and assembly. Our scope focuses on the MSM (Multi-Scalar Multiplication) implementation, which primarily uses the Pippenger algorithm. Note that `blst` is designed for cryptographic signatures and implements constant-time operations to prevent side-channel attacks. However, for data availability sampling use cases, constant-time operations are not required and may reduce efficiency.

Given a list of base points $[P_0, P_1, P_2, \ldots, P_{n-1}]$ and scalars (all of fixed bit length, e.g., 256) $[s_0, s_1, s_2, \ldots, s_{n-1}]$, the Pippenger algorithm efficiently computes $\sum_{i=0}^{n-1} s_i P_i$.

The main idea is to divide the scalars into smaller windows. Within each window, the scalars are grouped into buckets based on their values, allowing for efficient computation. For example, consider four points $[P_0, P_1, P_2, P_3]$ and four 4-bit scalars $[11, 9, 3, 7]$. If the window size is 2 bits, the scalars are split into two parts: the lower half $[3, 1, 3, 3]$ and the upper half $[2, 2, 0, 1]$. The computation is then performed separately for each window:

- $W_0 = 3P_0 + 1P_1 + 3P_2 + 3P_3$
- $W_1 = 2P_0 + 2P_1 + 0P_2 + 1P_3$

The final result is obtained by summing the contributions from each window: $W_0 + 4W_1$. Within each window, points with the same scalar value are grouped into the same bucket to reduce the number of operations. For example, W_0 can be computed as $1P_1 + 3(P_0 + P_2 + P_3)$, and W_1 as $2(P_0 + P_1) + 1P_3$.

Booth Encoding is an optimization technique that reduces the bucket size by half. Instead of using scalar values in the range $[0, 1, 2, 3]$, Booth encoding maps them to $[-2, -1, 0, 1, 2]$. This optimization leverages the fact that the negative of a point P (denoted $-P$) can be easily computed by negating its y coordinate.

For example, if a scalar is -2, the algorithm negates the point and places it in the bucket for 2, as $-2P = 2(-P)$. Given a scalar s and a window of n bits, Booth encoding splits the scalar into smaller components w_i within the range $[-2^{n-1}, 2^{n-1}]$, while ensuring that $\sum 2^{ni} w_i = s$. This approach reduces the number of buckets required and improves efficiency.

The `ptype##s_mult_pippenger` function computes the multiplication for each window dynamically, while the `ptype##s_mult_wbits` function precomputes the multiplications for each window (e.g., $[1P, 2P, 3P, 4P]$) and caches them. This allows for faster selection of points during computation but consumes more memory.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	go-eth-kzg/internal/kzg_multi/kzg_verify.go	Fiat-Shamir Challenge in go-eth-kzg Batch Verifier Is Not Sound	High
#01	blst/src/multi_scalar.c	`ptype##s_to_affine_row_wbits` in blst Fails to Handle Infinity Point Input	Medium
#02	rust-eth-kzg/bindings/csharp/.../ethkzg.cs	C# Bindings in rust-eth-kzg May Use Invalid Pointers Due to Improper Pinning	Medium
#03	blst/src/ec_mult.h	Scalar Input Greater Than the Group Order Lead to Incorrect Result in `ptype##_mult_w##SZ`	Low
#04	blst/src/multi_scalar.c	Out-of-bounds Read in `ptype##s_mult_wbits` May Cause Segmentation Fault	Low
#05	c-kzg-4844/bindings/go/main.go	Missing Nil Check in c-kzg-4844 Go Bindings	Low
#06	c-kzg-4844/bindings/node.js/src/kzg.cxx	Incorrect nullptr Check in `RecoverCellsAndKzgProofs` Function in c-kzg-4844 Node.js Bindings	Low
#07	c-kzg-4844/bindings/node.js/src/kzg.cxx	Memory Leak in c-kzg-4844 Node.js Bindings	Low
#08	go-eth-kzg/api_eip7594.go	Missing Nil Check in go-eth-kzg API	Low

ID	COMPONENT	NAME	RISK
#09	rust-eth-kzg/crates/.../fk20/verifier.rs	Missing Length Validation in Internal FK20 Verifier Can Cause Fiat-Shamir Weakness	Low
#0a	c-kzg-4844/bindings	Minor Issues in c-kzg-4844 Bindings	Informational
#0b	c-kzg-4844/bindings	Large Stack Allocation Risk in c-kzg-4844 Rust Bindings	Informational
#0c	rust-eth-kzg and go-eth-kzg	The FK20 Prover Implementation Deviates from the Paper	Informational
#0d	go-eth-kzg/internal/poly/poly.go	Panic When Computing `PolyMul` With Empty Polynomial in go-eth-kzg	Informational
#0e	go-eth-kzg/serialization.go	Panic When Given Short `poly` for `SerializePoly`	Informational
#0f	rust-eth-kzg and go-eth-kzg	Incorrect Comments	Informational
#10	rust-eth-kzg/crates/.../fk20/verifier.rs	Footguns in Internal FK20 Verifier Constructor	Informational
#11	rust-eth-kzg/crates/.../reed_solomon.rs	Polynomial Recovery Could be ~40% More Efficient by Exploiting Block Structure	Informational
#12	rust-eth-kzg/crates/eip4844/src/verifier.rs	The EIP-4844 Verifier in rust-eth-kzg Can Be More Efficient	Informational

#00 - Fiat-Shamir Challenge in go-eth-kzg Batch Verifier Is Not Sound

Severity: High **Location:** go-eth-kzg/internal/kzg_multi/kzg_verify.go

Description. The batch cell proof verifier in go-eth-kzg does not correctly implement the Fiat-Shamir challenge as specified in the EIP-7594 KZG batch verification spec. Specifically, the implementation omits the `proof` values when hashing to generate the random combiner `r`.

The random combiner `r` is intended to securely combine multiple proofs for a single pairing check. If the proofs are not included in the hash, a malicious prover can adapt the proofs after seeing `r`, potentially crafting invalid proofs that still pass batch verification. This undermines the soundness of the verification process.

Additionally, the implementation serializes integers as 16 bytes, while the spec requires 8-byte serialization. While this does not currently pose a security risk, strict adherence to the spec is recommended for compatibility and future-proofing.

Impact. Omitting proofs from the Fiat-Shamir hash compromises the soundness of batch verification, allowing invalid cell proofs to pass.

Recommendation. It is recommended to update the Fiat-Shamir challenge computation to include the proofs in the hash input, as required by the spec. Besides, consider changing integer serialization to use 8 bytes instead of 16 bytes for full compliance.

Client Response. Fixed in https://github.com/crate-crypto/go-eth-kzg/pull/111.

#01 - `ptype##s_to_affine_row_wbits` in blst Fails to Handle Infinity Point Input

Severity: Medium **Location:** blst/src/multi_scalar.c

Description. The function `ptype##s_to_affine_row_wbits` (defined in `blst/src/multi_scalar.c`) fails to correctly handle projective points at infinity (`Z = 0`).

Specifically, in the section where the accumulator is built:

```
for (i = 0; i < npoints; i++)
    for (j = nwin; --src, --j; acc++)
        mul_##field(acc[0], acc[-1], src->Z);
```

the `Z` coordinate is multiplied directly into the accumulator without guarding against the case where `Z = 0`. This means that if *any* point in the input batch is infinity, the accumulator becomes zero. Since batch inversion relies on this accumulator, the reciprocal computation:

```
--acc; reciprocal_##field(acc[0], acc[0]);
```

will fail silently, propagating invalid zeroes into the affine coordinates of *all* points.

By contrast, the related function `ptype##s_to_affine` correctly applies:

```
vec_select(acc++, BLS12_381_Rx.p, point->Z, sizeof(vec##bits),
vec_is_zero(point->Z, sizeof(point->Z)));
```

which substitutes a neutral Montgomery radix (`1`) when `Z = 0`, ensuring that infinity points do not corrupt the batch.

Fuzzer Context. To find this and another findings we developed and ran a custom fuzzer. The fuzzer targets the `blst` BLS12-381 scalar multiplication routines, including both single-point, wbits-precompute, and Pippenger multi-point paths. Inputs are variable-length byte arrays that are split into a scalar (up to 320 bits) and a compressed curve point.

For each input, the fuzzer:

1. Parses and validates the scalar and curve point.
2. Executes all three multiplication paths. Using the zero scalar and point for the second and third paths as required.
3. Compares results to detect inconsistencies or crashes.

The fuzzer uses AFL++ in persistent mode but can also run standalone for testing. Any mismatches trigger an immediate abort to report potential bugs.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#include <unistd.h>
#include "bindings/blst.h"

#ifndef __AFL_FUZZ_TESTCASE_LEN
ssize_t fuzz_len;
#define __AFL_FUZZ_TESTCASE_LEN fuzz_len
unsigned char fuzz_buf[1024*1024];
#define __AFL_FUZZ_TESTCASE_BUF fuzz_buf
#define __AFL_FUZZ_INIT() void sync(void);
#define __AFL_LOOP(x) ((fuzz_len = read(0, fuzz_buf, sizeof(fuzz_buf))) > 0 ? 1 : 0)
#define __AFL_INIT() sync()
#endif

__AFL_FUZZ_INIT();

#define SCALAR_SIZE 32
#define POINT_SIZE 48
#define MAX_SCALAR_SIZE 40
#define MAX_INPUT_SIZE (MAX_SCALAR_SIZE + POINT_SIZE)
#define PIPPENGER_NPOINTS 32

static limb_t *g_scratch;
static size_t g_scratch_size;
static byte *g_zero_scalars;
static blst_p1_affine *g_zero_points;
static blst_p1_affine *g_points_array;
static byte *g_scalars_array;
static const blst_p1_affine **g_point_ptrs;
static const byte **g_scalar_ptrs;

static void init_fuzzer() {
    g_scratch_size = blst_p1s_mult_pippenger_scratch_sizeof(PIPPENGER_NPOINTS);
    size_t wbits_scratch = blst_p1s_mult_wbits_scratch_sizeof(PIPPENGER_NPOINTS);
    if (wbits_scratch > g_scratch_size) g_scratch_size = wbits_scratch;
    g_scratch = malloc(g_scratch_size);

    g_zero_scalars = calloc(PIPPENGER_NPOINTS, MAX_SCALAR_SIZE);
    g_zero_points = calloc(PIPPENGER_NPOINTS, sizeof(blst_p1_affine));
    g_points_array = malloc(PIPPENGER_NPOINTS * sizeof(blst_p1_affine));
    g_scalars_array = malloc(PIPPENGER_NPOINTS * MAX_SCALAR_SIZE);
    g_point_ptrs = malloc(PIPPENGER_NPOINTS * sizeof(blst_p1_affine *));
    g_scalar_ptrs = malloc(PIPPENGER_NPOINTS * sizeof(byte *));
}

static size_t calc_scalar_nbits(const byte *scalar, size_t max_bytes) {
    for (size_t i = max_bytes; i > 0; i--) {
        if (scalar[i-1]) {
            size_t bits = i*8;
            byte b = scalar[i-1];
```

```c
            while ((b & 0x80) == 0 && bits > (i-1)*8) { bits--; b <<= 1; }
            return bits;
        }
    }
    return 0;
}

static int parse_input(const uint8_t *data, size_t size, byte *scalar, size_t
*scalar_nbits, blst_p1_affine *point) {
    byte tmp[MAX_INPUT_SIZE];
    if (size > MAX_INPUT_SIZE) return 0;
    memcpy(tmp, data, size); if (size < MAX_INPUT_SIZE) memset(tmp+size, 0,
MAX_INPUT_SIZE-size);
    memcpy(scalar, tmp, MAX_SCALAR_SIZE);
    *scalar_nbits = calc_scalar_nbits(scalar, MAX_SCALAR_SIZE);
    return blst_p1_uncompress(point, tmp + MAX_SCALAR_SIZE) == BLST_SUCCESS;
}

static void test_path(const byte *scalar, size_t nbits, const blst_p1_affine *point,
blst_p1 *r1, blst_p1 *r2, blst_p1 *r3) {
    const blst_p1_affine *p1[1] = {point};
    const byte *s1[1] = {scalar};
    blst_p1s_mult_pippenger(r1, p1, 1, s1, nbits, g_scratch);

    g_point_ptrs[0]=point; g_point_ptrs[1]=g_zero_points;
    g_scalar_ptrs[0]=scalar; g_scalar_ptrs[1]=g_zero_scalars;
    blst_p1s_mult_pippenger(r2, g_point_ptrs, 2, g_scalar_ptrs, nbits, g_scratch);

    memcpy(g_points_array, point, sizeof(blst_p1_affine));
    memcpy(g_scalars_array, scalar, MAX_SCALAR_SIZE);
    g_point_ptrs[0]=g_points_array; g_point_ptrs[1]=NULL;
    g_scalar_ptrs[0]=g_scalars_array; g_scalar_ptrs[1]=NULL;
    blst_p1s_mult_pippenger(r3, g_point_ptrs, PIPPENGER_NPOINTS, g_scalar_ptrs,
nbits, g_scratch);
}

static int points_equal(const blst_p1 *a, const blst_p1 *b) { return
blst_p1_is_equal(a,b); }

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    byte scalar[MAX_SCALAR_SIZE]; blst_p1_affine point; size_t scalar_nbits;
    if (!parse_input(data, size, scalar, &scalar_nbits, &point)) return 0;

    size_t max_nbits = scalar_nbits + 10; if (max_nbits > MAX_SCALAR_SIZE*8)
max_nbits = MAX_SCALAR_SIZE*8;
    for (size_t nbits = scalar_nbits; nbits <= max_nbits; nbits++) {
        blst_p1 r1,r2,r3; test_path(scalar, nbits, &point, &r1,&r2,&r3);
        if (!points_equal(&r1,&r2) || !points_equal(&r1,&r3) ||
!points_equal(&r2,&r3)) abort();
    }
    return 0;
}

int main(int argc, char **argv) {
    init_fuzzer();
```

```
    unsigned char *buf = __AFL_FUZZ_TESTCASE_BUF;
    while (__AFL_LOOP(10000)) { int len=__AFL_FUZZ_TESTCASE_LEN; if(len>0 &&
len<=MAX_INPUT_SIZE){memset(g_scratch,0,g_scratch_size);
LLVMFuzzerTestOneInput(buf,len);}}
    if(argc==2){ FILE *fp=fopen(argv[1],"rb"); uint8_t data[MAX_INPUT_SIZE]; size_t
read=fread(data,1,MAX_INPUT_SIZE,fp); fclose(fp);
LLVMFuzzerTestOneInput(data,read);}
    free(g_scratch); free(g_zero_scalars); free(g_zero_points);
free(g_points_array); free(g_scalars_array); free(g_point_ptrs);
free(g_scalar_ptrs);
    return 0;
}
```

Impact.

`ptype##s_to_affine_row_wbits()` is used in `ptype##s_precompute_wbits()` to batch-convert the computed point table from Jabobi to affine representation. In turn, `ptype##s_precompute_wbits()` is used as precomputation step of the main MSM routine `prefix##s_mult_pippenger()`, in the case that the number of points is between 2 and 31:

```
void prefix##s_mult_pippenger(ptype *ret, \
                              const ptype##_affine *const points[], \
                              size_t npoints, \
                              const byte *const scalars[], size_t nbits, \
                              ptype##xyzz scratch[]) \
{ \
    // ...
    if ((npoints * sizeof(ptype##_affine) * 8 * 3) <= SCRATCH_LIMIT && \
        npoints < 32) { \
        ptype##_affine *table = alloca(npoints * sizeof(ptype##_affine) * 8); \
        ptype##s_precompute_wbits(table, 4, points, npoints); \
        ptype##s_mult_wbits(ret, table, 4, npoints, scalars, nbits, NULL); \
        return; \
    } \
    ptype##s_mult_pippenger(ret, points, npoints, scalars, nbits, scratch, 0); \
}
```

According to our analysis, the issue is unlikely to affect the KZG libraries that depend on `prefix##s_mult_pippenger()`. For example, `g1_lincomb_fast()` in `c-kzg-4844` depends on the problematic code path when the number of points is between 8 and 31 input points. However, it avoids zero input points by filtering them out before calling into `blst`. The same is true for the Rust library.

Nevertheless, the issue is subtle and could affect the soundness of similar protocols or of the EIP-7594 verifier if point filtering was removed. Inherently, `g1_lincomb_fast()` is called on prover-supplied input points (the KZG cell proofs) and is therefore vulnerable to any mishandling of invalid points.

Recommendation. Adopt the same protective logic used in `ptype##s_to_affine`:

- Use `vec_select` to substitute `Z = 1` when `Z = 0` before contributing to the batch product.
- Apply `vec_czero` on the affine outputs to zero them out if the original point was infinity.

This ensures that infinity points are handled locally and do not corrupt other valid points in the batch.

Client Response. Fixed in https://github.com/supranational/blst/commit/f48500c1fdbefa7c0bf9800bccd65d28236799c1.

#02 - C# Bindings in rust-eth-kzg May Use Invalid Pointers Due to Improper Pinning

Severity: Medium **Location:** rust-eth-kzg/bindings/csharp/.../ethkzg.cs

Description. In the C# bindings for rust-eth-kzg, several methods (such as `ComputeCellsAndKZGProofs` and `RecoverCellsAndKZGProofs`) pin individual byte arrays inside a loop using the `fixed` statement to obtain pointers for native interop. However, the scope of each `fixed` statement only lasts for the duration of the fixed scope. After the scope ends, the arrays are no longer pinned, and the garbage collector (GC) may move them before the native function is called. This means the pointers stored in the pointer arrays may become invalid, leading to undefined behavior or memory corruption.

```
public unsafe (byte[][], byte[][]) ComputeCellsAndKZGProofs(byte[] blob)
{
    ...
    fixed (byte* blobPtr = blob)
    fixed (byte** outCellsPtrPtr = outCellsPtrs)
    fixed (byte** outProofsPtrPtr = outProofsPtrs)
    {

        // Get the pointer for each cell
        for (int i = 0; i < numCells; i++)
        {
            fixed (byte* cellPtr = outCells[i])
            {
                outCellsPtrPtr[i] = cellPtr; // The ptr is only pinned in this scope
            }
        }

        // Get the pointer for each proof
        for (int i = 0; i < numCells; i++)
        {
            fixed (byte* proofPtr = outProofs[i])
            {
                outProofsPtrPtr[i] = proofPtr; // The ptr is only pinned in this
scope
            }
        }

        CResult result = eth_kzg_compute_cells_and_kzg_proofs(_context, blobPtr,
outCellsPtrPtr, outProofsPtrPtr);
        ThrowOnError(result);
    }
    return (outCells, outProofs);
}
```

Similar patterns exist in other methods, such as `RecoverCellsAndKZGProofs` and `VerifyCellKZGProofBatch`.

Impact. If the GC moves any of the arrays after the loop, the native function may dereference invalid pointers, causing crashes, data corruption, or unpredictable behavior.

Recommendation. It is recommended to allocate and fix one large continuous array for the 2-D array so that the pointers are fixed during the Rust call.

Client Response. Fixed in https://github.com/crate-crypto/rust-eth-kzg/pull/413 and https://github.com/crate-crypto/rust-eth-kzg/pull/425.

#03 - Scalar Input Greater Than the Group Order Lead to Incorrect Result in `ptype##_mult_w##SZ`

Severity: Low **Location:** blst/src/ec_mult.h

Description. This issue was identified using the same custom fuzzer referenced in the blst infinity point finding.

The function `ptype##_mult_w##SZ` in `blst/src/ec_mult.h` assumes that `ret` and `row` can only be equal in the final loop iteration, where `ptype##_dadd` (double or add) is explicitly called.

The relevant code section is:

```
if (bits > 0) ptype##_add(sum, ret, row);
else          ptype##_dadd(sum, ret, row, NULL);
```

Here, `ptype##_add` is called under the assumption that `ret != row`. However, this is not always true. For example, if the scalar input exceeds the group order, Booth-encoded windowing can select the same point for both `ret` and `row`. In such cases, `ptype##_add` is invoked with identical inputs, which is undefined behavior, as elliptic curve addition formulas do not apply to doubling. This can result in incorrect group operations.

Impact. The `ptype##_mult_w##SZ` function may produce incorrect results when the input scalar is greater than the group order. This affects scalar multiplication functions such as `blst_p1_unchecked_mult`, `blst_p2_unchecked_mult`, `blst_p1s_mult_pippenger`, and `blst_p2s_tile_pippenger`.

Recommendation. Ensure the code correctly handles the case where `ret == row` in all iterations, not just the last one.

Client Response. Fixed in https://github.com/supranational/blst/commit/7c535f1afcea92a4ff3103e73d937604122cce5e.

#04 - Out-of-bounds Read in `ptype##s_mult_wbits` May Cause Segmentation Fault

Severity: Low **Location:** blst/src/multi_scalar.c

Description. In `blst`, the `ptype##s_mult_wbits` macro (expanding to `blst_p1s_mult_wbits` and `blst_p2s_mult_wbits`) implements multi-scalar multiplication (MSM) with a fixed window size. When extracting the most-significant (top) window, the code incorrectly uses `wbits` instead of the computed `window = nbits % wbits` in the call to `get_wval_limb`. This can cause the function to read past the end of the scalar byte array when the top window is shorter than `wbits` (including when `window == 0`). Although the out-of-bounds value is masked and discarded, the memory access still occurs, which may cause a segmentation fault if the scalar buffer is followed by a protected or unmapped page, or trigger memory sanitizers.

```
/* top excess bits modulo target window size */
window = nbits % wbits; /* yes, it may be zero */
wmask = ((limb_t)1 << (window + 1)) - 1;

nbits -= window;
z = is_zero(nbits);

/* BUG: uses wbits instead of window for the top slice length */
wval = (get_wval_limb(scalar, nbits - (z^1), wbits + (z^1)) << z) & wmask;
wval = booth_encode(wval, wbits);
ptype##_gather_booth_wbits(&scratch[0], row, wbits, wval);
```

If `nbits` is an exact multiple of `wbits`, then `window == 0` and `nbits -= window` leaves `nbits` unchanged. The call `get_wval_limb(scalar, nbits - (z^1), wbits + (z^1))` then requests `wbits+1` bits at the top boundary, which reads one byte past the end of the scalar. Even when `window > 0` but `< wbits`, using `wbits` instead of `window` can extend the read beyond the buffer.

For example, with `wbits = 4` and `nbits = 320` (40 bytes scalar):

- `window = nbits % wbits = 0`
- The buggy line requests `wbits + (z^1)` bits, causing a 1-byte over-read.
- The correct line should request `window + (z^1)` bits, i.e., 1 bit in this case.

The bug was validated with Valgrind by allocating the scalar on the heap so that the next byte is outside the allocated block:

```
valgrind --tool=memcheck --track-origins=yes ./test_pippenger
==33966== Invalid read of size 1
==33966==    at 0x10A321: get_wval_limb (in /home/marco/repo/blst/test_pippenger)
==33966==    by 0x118990: POINTonE1s_mult_wbits (in /home/marco/repo/blst/test_pippenger)
==33966==    by 0x121367: blst_p1s_mult_pippenger (in /home/marco/repo/blst/test_pippenger)
```

```
==33966==    by 0x109D79: main (test_pippenger.c:130)
==33966==  Address 0x4a88482 is 0 bytes after a block of size 2 alloc'd
==33966==                           at       0x4846828:       malloc       (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==33966==    by 0x1098BD: main (test_pippenger.c:58)
```

The bug can be reliably triggered by placing a protected page immediately after the scalar buffer. Any read of `ptr[size]` will fault, exposing the over-read:

```c
// Allocate a vector of bytes and right after a protected page
// so that any call right after size crashes; i.e., ptr[size] faults.
uint8_t* alloc_vec_with_protected_page_end(size_t size) {
    size_t pagesize = sysconf(_SC_PAGESIZE);
    size_t alloc_size = ((size + pagesize - 1) / pagesize) * pagesize;
    size_t total_size = alloc_size + pagesize;

    void *base = mmap(NULL, total_size, PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (base == MAP_FAILED) return NULL;

    // Protect the last page (immediately after data region)
    if (mprotect((uint8_t*)base + alloc_size, pagesize, PROT_NONE) != 0) {
        munmap(base, total_size);
        return NULL;
    }

    // Return pointer so that ptr[size] crashes
    return (uint8_t*)base + alloc_size - size;
}

void free_vec_with_protected_page_end(uint8_t *ptr, size_t size) {
    size_t pagesize = sysconf(_SC_PAGESIZE);
    size_t alloc_size = ((size + pagesize - 1) / pagesize) * pagesize;
    size_t total_size = alloc_size + pagesize;
    void *base = (void*)(ptr - (alloc_size - size));
    munmap(base, total_size);
}
```

Note: If scalars are stored contiguously (e.g., `scalar_0` immediately followed by `scalar_1`), the over-read will typically fetch the first byte of `scalar_1`, and the result gets masked out and discarded. This hides the issue but is still incorrect behavior.

Impact. `blst_p1s_mult_wbits` and `blst_p2s_mult_wbits` can read out of bounds of the scalar array and may cause segmentation faults in some cases. As KZG libraries typically store scalars contiguously, this is unlikely to be triggered in those contexts, but the bug remains.

Recommendation. Patch `ptype##s_mult_wbits` to use `window` (not `wbits`) for the top-window call to `get_wval_limb` to avoid reading past the end of the scalar buffer:

```diff
- wval = (get_wval_limb(scalar, nbits - (z^1), wbits + (z^1)) << z) & wmask;
+ wval = (get_wval_limb(scalar, nbits - (z^1), window + (z^1)) << z) & wmask;
```

Client Response. Fixed in https://github.com/supranational/blst/commit/01d167c8bfb0a76a6f44dd479902e2662983a1e9.

#05 - Missing Nil Check in c-kzg-4844 Go Bindings

Severity: Low **Location:** c-kzg-4844/bindings/go/main.go

Description. The `ComputeCells` and `ComputeCellsAndKZGProofs` functions in the `c-kzg-4844` Go bindings do not check if the `blob` argument is `nil` before passing it to the underlying C function. If a `nil` value is provided, this can result in a crash or undefined behavior.

As an example, the following test will crash:

```go
func TestCrash(t *testing.T) {
    _, _, err := ComputeCellsAndKZGProofs(nil)
    if err != nil {
        fmt.Println("Error:", err)
    }
}
```

Recommendation. Add a `nil` check for the `blob` argument at the beginning of both functions. If `blob` is `nil`, return an appropriate error (e.g., `ErrBadArgs`), similar to the pattern used in other functions like `VerifyBlobKZGProof`:

```go
if blob == nil {
    return [CellsPerExtBlob]Cell{}, ErrBadArgs
}
```

Apply this check to both `ComputeCells` and `ComputeCellsAndKZGProofs` to prevent crashes and improve robustness.

Client Response. Fixed in https://github.com/ethereum/c-kzg-4844/pull/590.

#06 - Incorrect nullptr Check in `RecoverCellsAndKzgProofs` Function in c-kzg-4844 Node.js Bindings

Severity: Low **Location:** c-kzg-4844/bindings/node.js/src/kzg.cxx

Description. In the `RecoverCellsAndKzgProofs` function, memory is allocated for `recovered_proofs`, and the code checks if the allocation is successful. However, there is a logic error in the check: the code checks `recovered_cells == nullptr` instead of `recovered_proofs == nullptr`. This may cause the function to proceed with a null pointer for `recovered_proofs`, leading to undefined behavior or crashes.

```
/**
 * Given at least 50% of cells, reconstruct the missing cells/proofs.
 *
 * @param[in] {number[]}  cellIndices - The identifiers for the cells you have
 * @param[in] {Cell[]}    cells - The cells you have
 *
 * @return {[Cell[], KZGProof[]]} - A tuple of cells and proofs
 *
 * @throws {Error} - Invalid input, failure to allocate or error recovering
 * cells and proofs
 */
Napi::Value RecoverCellsAndKzgProofs(const Napi::CallbackInfo &info) {
    ...
    recovered_proofs = (KZGProof *)calloc(CELLS_PER_EXT_BLOB, BYTES_PER_PROOF);
    if (recovered_cells == nullptr) { // Incorrect check here
        Napi::Error::New(
            env, "Error while allocating memory for recovered proofs"
        )
            .ThrowAsJavaScriptException();
        goto out;
    }
    ...
```

Impact. An incorrect null check for `recovered_proofs` may result in dereferencing a null pointer, causing application crashes or unpredictable behavior.

Recommendation. It is recommended to correct the allocation check for `recovered_proofs` to `if (recovered_proofs == nullptr)`.

Client Response. Fixed in https://github.com/ethereum/c-kzg-4844/pull/595.

#07 - Memory Leak in c-kzg-4844 Node.js Bindings

Severity: Low **Location:** c-kzg-4844/bindings/node.js/src/kzg.cxx

Description. A memory leak exists in the `RecoverCellsAndKzgProofs` function of the `c-kzg-4844` Node.js binding. The `cell_indices` array is allocated with `calloc` but is never freed, resulting in leaked memory on each invocation.

```
/**
 * Given at least 50% of cells, reconstruct the missing cells/proofs.
 *
 * @param[in] {number[]}  cellIndices - The identifiers for the cells you have
 * @param[in] {Cell[]}    cells - The cells you have
 *
 * @return {[Cell[], KZGProof[]]} - A tuple of cells and proofs
 *
 * @throws {Error} - Invalid input, failure to allocate or error recovering
 * cells and proofs
 */
Napi::Value RecoverCellsAndKzgProofs(const Napi::CallbackInfo &info) {
    ...
    num_cells = cells_param.Length();
    cell_indices = (uint64_t *)calloc(num_cells, sizeof(uint64_t));
    if (cell_indices == nullptr) {
        Napi::Error::New(env, "Error while allocating memory for cell_indices")
            .ThrowAsJavaScriptException();
        goto out;
    }
    ...
out:
    free(cells);
    free(recovered_cells);
    free(recovered_proofs);
    return result;
}
```

Impact. Unfreed `cell_indices` allocations can accumulate, especially in long-running processes, leading to increased memory usage and potential exhaustion.

Recommendation. It is recommended to ensure `cell_indices` is freed before returning from the function, including all error paths.

Client Response. Fixed in https://github.com/ethereum/c-kzg-4844/pull/595.

#08 - Missing Nil Check in go-eth-kzg API

Severity: Low **Location:** go-eth-kzg/api_eip7594.go

Description. The function `DeserializeBlob` in go-eth-kzg lacks a check for `nil` values on its `blob` input parameter. Passing a `nil` blob causes a runtime panic, crashing the Go process.

```
// DeserializeBlob implements [blob_to_polynomial].
func DeserializeBlob(blob *Blob) (kzg.Polynomial, error) {
    poly := make(kzg.Polynomial, ScalarsPerBlob)
    for i := 0; i < ScalarsPerBlob; i++ {
        chunk := blob[i*SerializedScalarSize : (i+1)*SerializedScalarSize]
        if err := poly[i].SetBytesCanonical(chunk); err != nil {
            return nil, ErrNonCanonicalScalar
        }
    }
    return poly, nil
}
```

Several functions are affected and may crash if called with a `nil` blob:

1. `ComputeCells` (`go-eth-kzg/api_eip7594.go`)
2. `ComputeCellsAndKZGProofs` (`go-eth-kzg/api_eip7594.go`)
3. `BlobToKZGCommitment` (`go-eth-kzg/proof.go`)
4. `ComputeBlobKZGProof` (`go-eth-kzg/proof.go`)
5. `ComputeKZGProof` (`go-eth-kzg/proof.go`)
6. `VerifyBlobKZGProof` (`go-eth-kzg/verify.go`)

For example, the following test will crash:

```
func TestCrash(t *testing.T) {
    _, _, err := ComputeCellsAndKZGProofs(nil)
    if err != nil {
        fmt.Println("Error:", err)
    }
}
```

Similarly, the function `deserializeCell` is missing a `nil` check, impacting:

1. `RecoverCellsAndComputeKZGProofs` (`api_eip7594.go`)
2. `VerifyCellKZGProofBatch` (`api_eip7594.go`)

Recommendation. It is recommended to add a `nil` check in the affected functions, following the pattern used in `VerifyBlobKZGProof` :

```
if blob == nil {
    return false, ErrBadArgs
}
```

Client Response. Fixed in https://github.com/crate-crypto/go-eth-kzg/pull/114.

#09 - Missing Length Validation in Internal FK20 Verifier Can Cause Fiat-Shamir Weakness

Severity: Low **Location:** rust-eth-kzg/crates/.../fk20/verifier.rs

Description. The core KZG verifier method allows each entry of `bit_reversed_coset_evals` to be a vector of dynamic size, and does not check that their sizes are all equal:

```
pub fn verify_multi_opening(
    &self,

    deduplicated_commitments: &[G1Point],
    commitment_indices: &[CommitmentIndex],

    bit_reversed_coset_indices: &[CosetIndex],
    // [ZKSECURITY] the sizes of these vectors could be different
    bit_reversed_coset_evals: &[Vec<Scalar>],
    bit_reversed_proofs: &[G1Point],
) -> Result<(), VerifierError> {
```

However, the Fiat-Shamir logic implicitly assumes the sizes to be the same, and does not commit the sizes of these vectors individually.

```rust
fn compute_fiat_shamir_challenge(
    // [ZKSECURITY] ...
) -> Scalar {
    const DOMAIN_SEP: &str = "RCKZGCBATCH__V1_";
    let hash_input_size = DOMAIN_SEP.len()
            // [ZKSECURITY] ...
            + coset_evals.len() * verification_key.coset_size * size_of::<Scalar>()
            + proofs.len() * G1Point::compressed_size();

    let mut hash_input: Vec<u8> = Vec::with_capacity(hash_input_size);

    hash_input.extend(DOMAIN_SEP.as_bytes());
    hash_input.extend((verification_key.num_coefficients_in_polynomial as
u64).to_be_bytes());
    // [ZKSECURITY] we commit to the _intended_ size of each cell of coset evals
    hash_input.extend((verification_key.coset_size as u64).to_be_bytes());

    // [ZKSECURITY] ...

    for k in 0..num_cosets {
        hash_input.extend(row_indices[k as usize].to_be_bytes());
        hash_input.extend(coset_indices[k as usize].to_be_bytes());

        // [ZKSECURITY] we hash each cell of coset evals without hashing their sizes
individually
        for eval in &coset_evals[k as usize] {
            hash_input.extend(eval.to_bytes_be());
        }
        hash_input.extend(proofs[k as usize].to_compressed());
    }

    // [ZKSECURITY] this assertion forces the _total_ size of coset evals to be as
expected
    assert_eq!(hash_input.len(), hash_input_size);
```

In another place in the verifier, coset eval sizes are forced to be powers of two, but other than that, the verifier places no restrictions on these sizes; the polynomial interpolation logic resizes them to the expected length before performing an IFFT, potentially truncating or padding with zeros.

For example, empirically, the verifier runs through to the end when resizing the first three coset evals from 64, 64, 64 to 128, 32, 32 (keeping the total size the same, as required by the final assertion above).

This means that we could shift values from the coset evals to other places like the `proofs` and `row_indices` while keeping the same Fiat-Shamir challenge. This is undesirable as it represents a potential weakness in the soundness of Fiat-Shamir.

Impact. In the main EIP-7594 verifier, coset evals are static-size vectors, so this issue does not affect production code.

However, the core FK20 library `ekzg-multi-open` is published as a standalone crate where `verify_multi_opening()` is part of the public API. In general, the core library aims to enforce soundness on its own and frequently adds assertions to prevent invalid inputs. That’s why we report this issue, even though it seems unlikely that `verify_multi_opening()` will be used in a deployment that allows dynamic-sized evaluation inputs.

Recommendation. At the beginning of `verify_multi_opening()`, add an assertion that every entry of `bit_reversed_coset_evals` is of size `verification_key.coset_size`.

Client Response. Fixed in https://github.com/crate-crypto/rust-eth-kzg/pull/415.

#0a - Minor Issues in c-kzg-4844 Bindings

Severity: Informational **Location:** c-kzg-4844/bindings

Description. The following minor issues were identified in the c-kzg-4844 bindings. These do not affect security but may impact correctness or code quality.

1. **Incorrect Constant Used in Error Message**

In the Rust binding, `BYTES_PER_PROOF` is used in the error message instead of `BYTES_PER_COMMITMENT`. Source

```
// ...existing code...
if bytes.len() != BYTES_PER_COMMITMENT {
    return Err(Error::InvalidKzgCommitment(format!(
        "Invalid byte length. Expected {} got {}",
        BYTES_PER_PROOF, // Should be BYTES_PER_COMMITMENT
        bytes.len(),
    )));
}
// ...existing code...
```

2. **Incorrect Return Type in Java Binding**

The JNI function should return a `jobject` type instead of `jbyteArray`. Source

```
JNIEXPORT jbyteArray JNICALL
Java_ethereum_ckzg4844_CKZG4844JNI_recoverCellsAndKzgProofs(
    JNIEnv *env, jclass thisCls, jlongArray cell_indices, jbyteArray cells) {
    // ...existing code...
}
```

3. **Incorrect Null Return in Java Binding**

The function should return `NULL` instead of `0` when an error occurs.
Source

```
if (cells_size != count * BYTES_PER_CELL) {
    throw_invalid_size_exception(env, "Invalid cells size.", cells_size,
                                 count * BYTES_PER_CELL);
    return 0; // Should be return NULL;
}
```

4. **Incorrect Parameter Name in C# Binding**

The error message should use `nameof(commitment)` instead of `nameof(proof)`. Source

```
// ...existing code...
ThrowOnInvalidLength(commitment, nameof(proof), BytesPerCommitment); // Should be nameof(commitment)
// ...existing code...
```

Recommendation. Correct the above issues to improve code clarity and correctness.

Client Response. Fixed in https://github.com/ethereum/c-kzg-4844/pull/595.

#0b - Large Stack Allocation Risk in c-kzg-4844 Rust Bindings

Severity: Informational **Location:** c-kzg-4844/bindings

Description. In the `c-kzg-4844` Rust bindings, the following code allocates a large array (about 262KB) on the stack, which may risk stack overflow on platforms with limited stack size (e.g., 2MB).

```
pub fn compute_cells(&self, blob: &Blob) -> Result<Box<CellsPerExtBlob>, Error> {
    let mut cells = [Cell::default(); CELLS_PER_EXT_BLOB];
    unsafe {
        let res = compute_cells_and_kzg_proofs(cells.as_mut_ptr(), ptr::null_mut(),
blob, self);
        if let C_KZG_RET::C_KZG_OK = res {
            Ok(Box::new(cells))
        } else {
            Err(Error::CError(res))
        }
    }
}

pub fn compute_cells_and_kzg_proofs(
    &self,
    blob: &Blob,
) -> Result<(Box<CellsPerExtBlob>, Box<ProofsPerExtBlob>), Error> {
    let mut cells = [Cell::default(); CELLS_PER_EXT_BLOB];
    let mut proofs = [KZGProof::default(); CELLS_PER_EXT_BLOB];
    unsafe {
        let res =
            compute_cells_and_kzg_proofs(cells.as_mut_ptr(), proofs.as_mut_ptr(),
blob, self);
        if let C_KZG_RET::C_KZG_OK = res {
            Ok((Box::new(cells), Box::new(proofs)))
        } else {
            Err(Error::CError(res))
        }
    }
}
```

There is no direct evidence of failure, but such large stack allocations could be problematic.

Impact. Allocating large arrays on the stack may cause stack overflow, especially in environments with small stack limits.

Recommendation. Consider allocating large arrays on the heap instead of the stack to avoid potential stack overflow issues.

Client Response. Fixed in https://github.com/ethereum/c-kzg-4844/pull/595.

#0c - The FK20 Prover Implementation Deviates from the Paper

Severity: Informational **Location:** rust-eth-kzg and go-eth-kzg

Description. The FK20 paper describes an efficient algorithm for computing all cell proofs of a blob. The `rust-eth-kzg` and `go-eth-kzg` libraries implement this algorithm with minor deviations.

In the paper, when composing the CirculantMatrix from the ToeplitzMatrix, the element at index `r` is set to `0`. In the code, this element is set to `f_{d-i}`, which equals to the first element in the array. As a result, `f_{d-i}` appears twice in the CirculantMatrix column.

```
// Embed toeplitz matrix within a circulant matrix
func (tm *toeplitzMatrix) embedCirculant() circulantMatrix {
	n := len(tm.row)
	row := make([]fr.Element, len(tm.col)+n)

	// Copy tm.Col
	copy(row, tm.col)

	// Append rotated and reversed tm.Row
	for i := 0; i < n; i++ {
		row[len(tm.col)+i] = tm.row[(n-i)%n]
	}
	return circulantMatrix{row: row}
}
```

This deviation does not affect the correctness of the final result, as it is cancelled out when multiplying with the zeros in the extended vectors, and only the first half of the final vector is used.

Additionally, the paper specifies the ToeplitzMatrix size as `(r-1) * (r-1)`, while the code uses `r * r`. The upper-left corner matches the paper, but the upper-right (extended) element is non-zero. The ToeplitzMatrix thus differs from the paper. Since the vector `s` is padded to size `r` with the last element set to `0`, the matrix-vector multiplication still yields the same result as in the paper.

Recommendation. The implementation deviation is subtle and the behavior is not obvious from looking at the code. It is recommended to document the deviation or update the code to strictly follow the paper.

Client Response. Partially fixed in https://github.com/crate-crypto/go-eth-kzg/pull/113.

#0d - Panic When Computing `PolyMul` With Empty Polynomial in go-eth-kzg

Severity: Informational **Location:** go-eth-kzg/internal/poly/poly.go

Description. The `PolyMul` function does not correctly compute `productDegree` when there are zero coefficients in both `a` and `b`. As a result, it will try to allocate a slice of size $2^{64} - 1$, which will lead to a panic.

```
// PolyMul multiplies two polynomials in coefficient form and returns the result.
// The degree of the resulting polynomial is the sum of the degrees of the input
polynomials.
func PolyMul(a, b PolynomialCoeff) PolynomialCoeff {
    // The degree of result will be degree(a) + degree(b) = numCoeffs(a) +
numCoeffs(b) - 1
    productDegree := numCoeffs(a) + numCoeffs(b)
    result := make([]fr.Element, productDegree-1)

    for i := uint64(0); i < numCoeffs(a); i++ {
        for j := uint64(0); j < numCoeffs(b); j++ {
            mulRes := fr.Element{}
            mulRes.Mul(&a[i], &b[j])
            result[i+j].Add(&result[i+j], &mulRes)
        }
    }

    return result
}
```

Impact. The only instance of `PolyMul` in the current repository sets `b = -x + 1`, which does not cause an issue here.

Recommendation. It is recommended to return an empty polynomial when one of the input polynomials is empty.

Client Response. Fixed in https://github.com/crate-crypto/go-eth-kzg/pull/115.

#0e - Panic When Given Short `poly` for `SerializePoly`

Severity: Informational **Location:** go-eth-kzg/serialization.go

Description. The `SerializePoly` function requires that `poly` to consist of at least 4096 terms. However, a `kzg.Polynomial` is defined to be `[]fr.Element`, which the length could be arbitrary. If there are less than 4096 entries, the function will eventually access `poly[4095]` and this will panic since it attempts to read out-of-bounds.

```
// SerializePoly converts a [kzg.Polynomial] to [Blob].
//
// Note: This method is never used in the API because we always expect a byte array
and will never receive deserialized
// field elements. We include it so that upstream fuzzers do not need to reimplement
it.
func SerializePoly(poly kzg.Polynomial) *Blob {
    var blob Blob
    for i := 0; i < ScalarsPerBlob; i++ {
        chunk := blob[i*SerializedScalarSize : (i+1)*SerializedScalarSize]
        serScalar := SerializeScalar(poly[i])
        copy(chunk, serScalar[:])
    }
    return &blob
}
```

Impact. This will not cause an issue in the repository since this function is unused.

Recommendation. It is recommended to check the length of `poly` is correct.

Client Response. Fixed in https://github.com/crate-crypto/go-eth-kzg/pull/116.

#0f - Incorrect Comments

Severity: Informational **Location:** rust-eth-kzg and go-eth-kzg

In go-eth-kzg, the comment of `nextPowerOfTwo` function is misleading. It is actually returning the next power of two exactly greater than `n`. For example, if the input `n` is `2`, it will return `4`.

```
// nextPowerOfTwo returns the next power of two greater than or equal to n
func nextPowerOfTwo(n int) int {
    if n == 0 {
        return 1
    }
    k := 1
    for k <= n {
        k <<= 1
    }
    return k
}
```

In rust-eth-kzg, `fixed_base_msm_window.rs`, the following comment is misleading. In fact, each point has 2*48 = 96 bytes, not 64.

```
// The total amount of memory is roughly (numPoints * 2^{wbits} - 1)
// where each point is 64 bytes.
```

In `lincom.rs`, this comment above `g1_lincomb()` is misleading:

```
/// A multi-scalar multiplication algorithm over G1 elements
///
/// Returns None if the points and the scalars are not the
/// same length.
```

In fact, this method will silently do the scalar multiplication on fewer points or scalars if one of them is of smaller length. The same is true for `g2_lincomb()`. Tests in the same file document the actual behavior while still having a self-contradictory comment about returning `None`:

```
fn g1_lincomb_length_mismatch_not_empty() {
    // MSM returns None when point and scalar lengths differ
    let points = vec![G1Point::generator(); 4];
    let scalars = vec![Scalar::from(1), Scalar::from(2), Scalar::from(3)];
    assert_eq!(
        g1_lincomb(&points, &scalars),
        Some(
            G1Point::generator() * Scalar::from(1)
                + G1Point::generator() * Scalar::from(2)
                + G1Point::generator() * Scalar::from(3)
        )
    );
}
```

This incorrect documentation is repeated in `fixed_base_msm.rs` above `msm()`:

```
///   Panics if the number of scalars doesn't match the number of generators.
```

In `reed-solomon.rs`, the following comment inside of `construct_vanishing_poly_from_block_erasures()` is misleading:

```
// Expand the vanishing polynomial, so that it vanishes on all blocks in the
codeword
// at the same indices.
//
// Example; consider the following polynomial f(x) = x - r
// It vanishes/has roots at `r`.
//
// Now if we expand it by a factor of three which is the process of shifting all
coefficients
// up three spaces, we get the polynomial g(x) = x^3 - r.
// g(x) has all of the roots of f(x) and a few extra roots.
//
// The roots of g(x) can be characterized as {r, \omega * r, \omega^2 * r}
// where \omega is a third root of unity.
```

In the example given, the polynomial `x^3 - r` does not actually vanish at `{r, \omega * r, \omega^2 * r}`. More generally, the operation of “expanding” alone is not what leads to getting the desired roots. Instead, this is achieved by a combination of:

- Using roots of a smaller domain, which are roots of the large domain, raised to a power.
- “Expanding” the coefficients, which correspond to evaluating the original polynomial at a power.

Client Response. Partially fixed in https://github.com/crate-crypto/go-eth-kzg/pull/118, https://github.com/crate-crypto/rust-eth-kzg/pull/416, https://github.com/crate-crypto/rust-eth-kzg/pull/417, and https://github.com/crate-crypto/rust-eth-kzg/pull/418.

#10 - Footguns in Internal FK20 Verifier Constructor

Severity: Informational **Location:** rust-eth-kzg/crates/.../fk20/verifier.rs

Description. The FK20 verifier is instantiated with three parameters:

- `num_points_to_open`, the full domain size we operate on
- `num_cosets`
- `verification_key.coset_size`

The parameters are supposed to be related and satisfy:

```
num_cosets * verification_key.coset_size = num_points_to_open
```

Furthermore, all three of these parameters are supposed to be powers of two.

The `FK20Verifier` constructor, however, takes in these parameters independently and does not validate their relation.

```rust
impl FK20Verifier {
    pub fn new(
        verification_key: VerificationKey,
        num_points_to_open: usize,
        num_cosets: usize,
    ) -> Self {
        const BIT_REVERSED: bool = true;
        let coset_gens = coset_gens(num_points_to_open, num_cosets, BIT_REVERSED);

        // [ZKSECURITY] `coset_size` is recalculated here but not linked to
`verification_key.coset_size`
        let coset_size = num_points_to_open / num_cosets;
        assert!(
            verification_key.g2s.len() >= coset_size,
            "need as many g2 points as coset size"
        );
        // [ZKSECURITY] this line will internally use the next power of two from
        // `verification_key.coset_size` to create the domain
        let coset_domain = Domain::new(verification_key.coset_size);

        // [ZKSECURITY] however, `verification_key.coset_size` itself enters the
cryptographic protocol
        // as an essential parameter without being rounded to a power of two
        let n = verification_key.coset_size;
        // [tau^n]_2
        let tau_pow_n = G2Prepared::from(verification_key.g2s[n]
```

Empirically, the verifier can be instantiated and run successfully with `num_cosets` not equal to `num_points_to_open / verification_key.coset_size`. In this case, the relation being checked refers to different coset generators than intended, which feels like a footgun.

Even more surprising, `VerificationKey` and `FK20Verifier` can be instantiated with `verification_key.coset_size` not being a power of two. For creating roots of unity internally, `Domain::new(verification_key.coset_size)` will replace it with the next power of two. Meanwhile, for preparing the universal verification equation, we use `n = verification_key.coset_size` directly which is allowed to be a non-power of two.

(This is not the case for `num_points_to_open` and `num_cosets`: these are asserted to be powers of two.)

To see why n being a non-power of two is bad, let $k = \lceil \log(n) \rceil$. Note that the verification equation guarantees that, for a collection of coset generators h, committed polynomials $C(X)$, input values y_i we have

$$C(h\rho^i) = y_i + ((h\rho^i)^n - h^n)Q(h\rho^i)$$

for all $i < 2^k$, where ρ is a 2^k-th root of unity and $Q(X)$ is some quotient polynomial (committed to by the KZG proof).

In the intended protocol, $n = 2^k$ and the second term on the RHS vanishes, giving $C(h\rho^i) = y_i$, i.e. we prove that y_i are evaluations of $C(X)$ as desired.

However, if we allow $n < 2^k$, the verification equation becomes meaningless as the prover can arbitrarily change the purported evaluations y_i by tweaking the quotient $Q(X)$.

Incidentally, while the constructor allows $n < 2^k$, `verify_multi_opening()` will throw due to checks on the length of evaluations. Nevertheless, it seems like an unnecessary footgun. And as documented in another finding, the lengths of evaluations are only weakly restricted as well.

Impact. None of the above affects the end-to-end EIP-7594 verifier, where parameters are hardcoded to correct values.

Recommendation. Enforce both the relation `num_cosets * verification_key.coset_size = num_points_to_open` and that `verification_key.coset_size` is a power of two, in the `FK20Verifier` constructor.

Client Response. Fixed in https://github.com/crate-crypto/rust-eth-kzg/pull/420.

#11 - Polynomial Recovery Could be ~40% More Efficient by Exploiting Block Structure

Severity: Informational **Location:** rust-eth-kzg/crates/.../reed_solomon.rs

Description. For the `recover_cells_and_kzg_proofs()` method of EIP-7594, the spec uses a subroutine called recover_polynomialcoeff. The method assumes a number of given cells, interprets them as polynomial evaluations and aims to recover the full polynomial in coefficient form. We found that the algorithm could be a bit more efficient than currently specified and implemented.

Let n be the polynomial degree, and let the evaluation domain be the $2n$-th roots of unity $D = \{\omega^0, \ldots, \omega^{2n-1}\}$. We assume the domain is evenly divided into m blocks of size k, where $mk = 2n$.

A *cell* with cell index $i < k$ is the structured subset of length m of the form

$$\omega^{i+jk}, \quad j < m.$$

We are given evaluations $f(\omega^{i+jk})$ for all $j < m$ and $i \notin M$, where $M \subset [k]$ are the missing cells. Recovery works by constructing the following *vanishing polynomial* that vanishes exactly on the missing cells:

$$Z(X) = \prod\nolimits_{i \in M} (X^m - \omega^{im})$$

Indeed, for all elements $x = \omega^{i+jk}$ of a cell $i \in M$, we have

$$x^m = \omega^{im} \omega^{jkm} = \omega^{im} \omega^{j2n} = \omega^{im}$$

and therefore $Z(x) = 0$. Furthermore, these are the only roots of $Z(X)$.

Finally, observe that Z is a function of X^m and we could write $Z(X) = z(X^m)$ where $z(X) = \prod_{i \in M} (X - \omega^{im})$. As it happens, $z(X)$ is a vanishing polynomial on a subset of the small domain of k-th roots $\{\omega^{im} : i < k\}$.

Recovery, as implemented, works as follows:

1. Compute the $Z(X)$ in coefficient form. This can be done by first computing the small-domain $z(X)$ in $O(k^2)$, using naive polynomial multiplications. And then, expanding those coefficients to a sparse polynomial of m times the degree, by shifting a coefficient at position i into position im. This corresponds to replacing X by X^m.
2. Perform a size-$2n$ FFT to obtain all values of $Z(x)$, $x \in D$ on the full evaluation domain.
3. Perform a size-$2n$ coset-FFT to obtain evaluations $Z(hx)$, $x \in D$, for a coset generator $h \notin D$.
4. Compute products $f(x)Z(x)$, $x \in D$. Observe that the vanishing polynomial kills the contributions of the missing f evaluations.
5. Perform a size-$2n$ IFFT to obtain $(f \cdot Z)(X)$ in coefficient form.
6. Perform a size-$2n$ coset-FFT to obtain evaluations $f(hx)Z(hx)$, $x \in D$.
7. Compute $f(hx) = (f(hx)Z(hx))/Z(hx)$ on the coset. Thanks to evaluating on a coset, denominators are non-zero. Note: this step can be done in $O(n)$ field multiplications using batch

inversion.

8. Compute a size-$2n$ coset-IFFT to recover $f(X)$ in coefficient form.

This algorithm is dominated by 5 size-$2n$ FFTs. It will recover the original degree-n polynomial, if at most half the cell evaluations were missing.

Our observation is that steps (2) and (3) are mostly unnecessary, because the vanishing polynomial evaluations are repeating in each block:

$$Z(\omega^{i+jk}) = Z(\omega^i) = z(\omega^{im})$$

Similarly, on the coset we have

$$Z(h\omega^{i+jk}) = Z(h\omega^i) = z(h^m\omega^{im})$$

It is therefore enough to compute z's evaluations on the small size-k domain and its h^m-coset, which can be done by 2 size-k FFTs. Whenever one of the $2n$ evaluations of $Z(X)$ or $Z(hX)$ are needed, we can look them up in an array of length k.

In practice, $k = 128$ is much smaller than $2n = 8192$, and small FFTs have negligible effort compared to full-domain operations. So we expect this optimization to save close to 40% of the effort (2 out of 5 full-domain FFTs).

#12 - The EIP-4844 Verifier in rust-eth-kzg Can Be More Efficient

Severity: Informational **Location:** rust-eth-kzg/crates/eip4844/src/verifier.rs

Description. In EIP-4844, validators verify blob data and its KZG commitment. The current implementation in `rust-eth-kzg` verifies the commitment by evaluating the blob polynomial at a random point and checking the opening of the commitment at that point. To compute the evaluation, it first transforms the blob from Lagrange form to coefficient form using an inverse FFT (IFFT), which has complexity `O(n*log(n))`. However, it is possible to evaluate the polynomial directly in Lagrange form in `O(n)` time, making the current approach unnecessarily inefficient.

```
pub fn verify_blob_kzg_proof(
    &self,
    blob: BlobRef,
    commitment: Bytes48Ref,
    proof: Bytes48Ref,
) -> Result<(), Error> {
    ...
    // Compute Fiat-Shamir challenge
    let z = compute_fiat_shamir_challenge(blob, *commitment);

    // Compute evaluation at z.
    let y = blob_scalar_to_polynomial(&self.verifier.domain, &blob_scalar).eval(&z);

    // Verify KZG proof.
    self.verifier.verify_kzg_proof(commitment_g1, z, y, proof)?;

    Ok(())
}

pub(crate) fn blob_scalar_to_polynomial(domain: &Domain, blob_scalar: &[Scalar]) ->
PolyCoeff {
    let mut polynomial = blob_scalar.to_vec();
    bitreverse_slice(&mut polynomial);
    domain.ifft_scalars(polynomial)
}
```

Impact. The inefficiency increases verification time and resource usage for each blob, especially as the number of blobs per block grows.

Recommendation. Refactor the verifier to evaluate the polynomial directly in Lagrange form, avoiding the costly IFFT transformation. This can be achieved using barycentric interpolation or other standard techniques for evaluating polynomials in Lagrange form. This change will reduce the complexity from `O(n*log(n))` to `O(n)` and improve overall efficiency.