Zellic

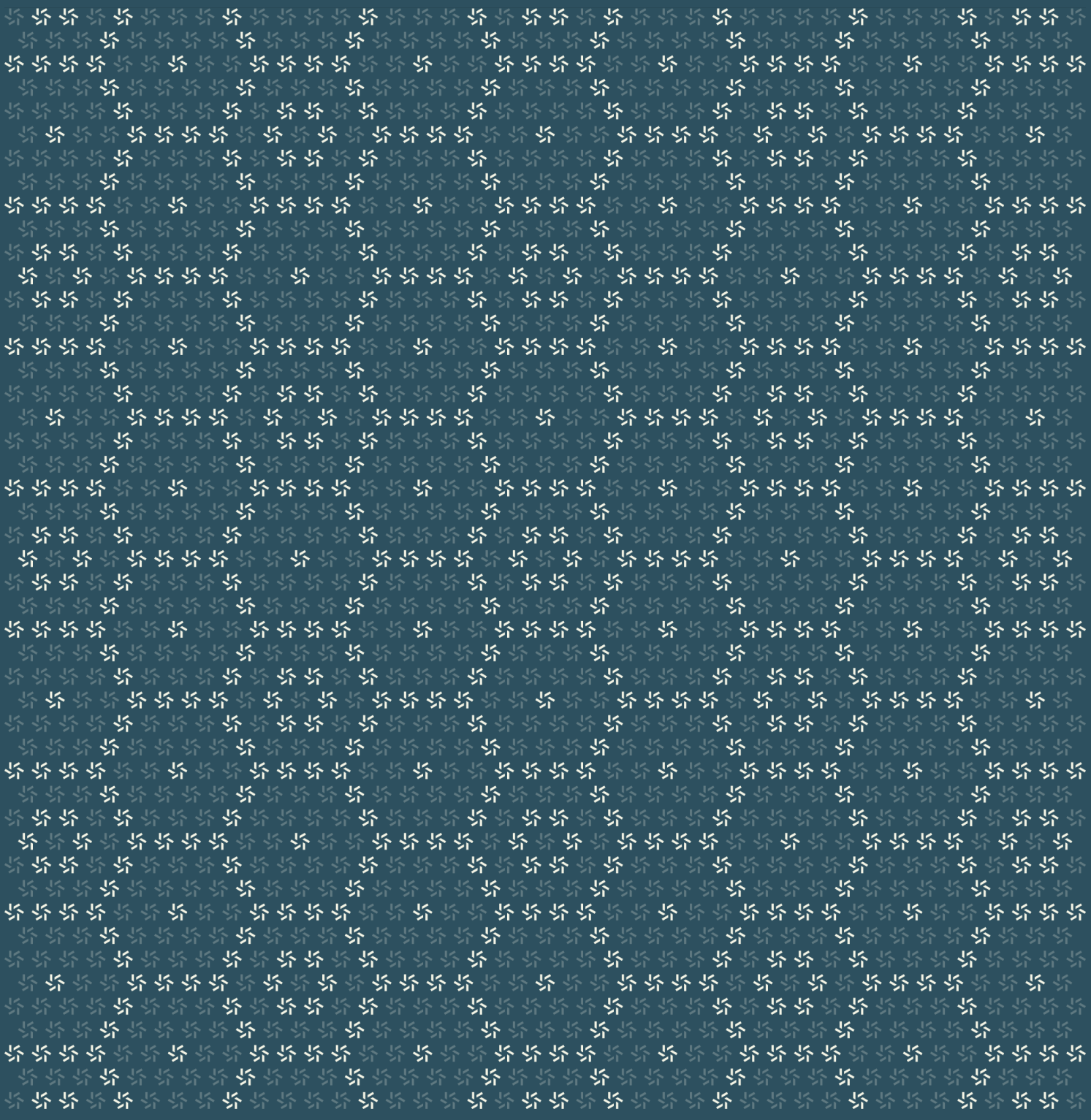
Prepared for
Uma Roy
John Guibas
Succinct Inc.

Prepared by
Malte Leip
Mark Pedron
Zellic

December 23, 2025

Hypercube Protocol

Cryptographic Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Succinct Inc. from August 18th to December 5th, 2025. During this engagement, Zellic reviewed Hypercube Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the shard verifier in `sp1_hypercube` soundly verify that the prover knows a computational trace and chips such that the chips' constraints hold and the interaction lookup tables match?
- Are the audited code parts in the individual `slop_*` crates sound and robust as separate libraries?

In addition to these general questions, Succinct Inc. tasked us to analyze key investigation points, which are stated and answered in section 5.1. ↗.

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Verifier denial of service due to large proof size
- The prover
- Precise claims about the bits of security

Throughout the engagement we assumed the following:

- Low-level memory wrappers like buffers and tensors are assumed to be validly instantiated.
- The verifier is run on a 64-bit machine.
- The prime of the base field has 31 bits and is bigger than $2^{30} + 2^{29}$.

Additionally, configurable parts of the verifier are assumed to be reasonably configured, and only configuration variants present in the audited code were considered. In particular, while the `JaggedPcsVerifier` has implementation generic over a `JaggedConfig`, we only audited

`JaggedPcsVerifier` with respect to the `JaggedBasefoldConfig` implementation of `JaggedConfig`. This is in turn generic over a `JaggedEvalConfig`, where we only considered the implementation for `JaggedEvalSumcheckConfig`.

For the shard verifier, the top-level verifier that we reviewed, we document some assumptions that we made on its intended use and limitations in section 5.3. ↗.

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Hypercube Protocol targets, we discovered 53 findings. Three critical issues were found. Seven were of high impact, six were of medium impact, 22 were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Succinct Inc. in the Discussion section (5. ↗).

The project successfully demonstrates the viability of the Hypercube protocol for the shard verifier, a significant milestone for a novel cryptographic protocol.

However, given the current level of code maturity, the complexity of the project, and the number of severe findings identified during the audit, we strongly recommend a follow-up assessment prior to production deployment. Further details are provided in section 6. ↗, which also outlines recommended patterns to improve the project's overall maturity.

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	7
Medium	6
Low	22
Informational	15

2. Introduction

2.1. About Hypercube Protocol

Succinct Inc. contributed the following description of Hypercube Protocol:

SP1 is a zero-knowledge virtual machine (zkVM) which allows developers to prove an execution of arbitrary Rust (or other LLVM-compiled language) programs. This means

- A developer can write a normal Rust code, which is then compiled to an ELF.
- Given the ELF, SP1 sets up a proving key and a verifying key.
- SP1 zkVM generates a proof on the execution of the Rust code using the proving key.
- The verifying key can be used to verify the generated proof.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Protocol design. For novel/nonstandard protocols, we investigate the core design for potential flaws. This does not, however, apply to implementations of established protocols and algorithms. For example, when auditing a library for Yao's garbled circuits, we would not spend time on the garbling scheme itself. On the other hand, the white paper for a novel protocol would be very much considered in scope and examined thoroughly.

Applied cryptography. Zellic has a dedicated team of strong theoretical and applied cryptographers. Implementing cryptographic applications securely, like Web3 wallets, is incredibly difficult, and we help clients navigate a minefield of potential pitfalls and mistakes. We have ensured secure implementation of noncustodial wallets, ERC-4337 (AA), MPC, Shamir's secret sharing (SSS), EOAs, native multi-sig support, enclave solutions, social log-in, and key recovery, among other technologies.

Cryptographic components. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities. We ensure all of the cryptographic primitives used in the project are used and configured correctly.

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. For cryptographic applications, this includes insufficient checks on the inputs, incorrect handling of edge cases, and completeness issues in low-level arithmetic operations.

> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (5. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Hypercube Protocol Targets

Type	Rust
Platform	Rust
Target	sp1-wip
Repository	https://github.com/succinctlabs/sp1-wip ↗
Version	`353904f13ff3dcbdb169d3c9e130d1c823f167c9`

Programs	`slop/crates/multilinear/src/*.rs`
	`slop/crates/sumcheck/src/verifier.rs`
	`slop/crates/sumcheck/src/proof.rs`
	`slop/crates/algebra/src/univariate.rs`
	`slop/crates/merkle-tree/src/tcs.rs`
	`slop/crates/basefold/src/config.rs`
	`slop/crates/basefold/src/verifier.rs`
	`slop/crates/basefold/src/code.rs`
	`slop/crates/stacked/src/verifier.rs`
	`slop/crates/jagged/src/verifier.rs`
	`slop/crates/jagged/src/poly.rs`
	`slop/crates/jagged/src/jagged_eval/sumcheck_eval.rs (only the` `JaggedEvalConfig` `implementation, so excluding the prover)`
	`crates/hypercube/src/verifier/shard.rs`
	`crates/hypercube/src/logup_gkr/verifier.rs`
	`crates/hypercube/src/folder.rs`
	`(GenericVerifierPublicValuesConstraintFolder only)`

Changes to the scope

During the audit, a code refactor was recommended by Zellic and carried out by Succinct Inc. Therefore, the final code commit this report refers to differs from the initially targeted commit. In the initially targeted code version, the slop and SP1 code was distributed over two separate repositories (6b785e8f ↗ and 24e3ebaa ↗). The commit this report refers to contains both slop and SP1 in a combined repository.

In addition to the results of the audit for the final scope (section 3. ↗), this report also contains findings that refer to the initial code prior to the refactor (section 4. ↗).

For details of the refactor, see section 5.2. ↗

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 13.5 person-weeks. The assessment was conducted by two consultants over the course of 16 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Malte Leip
Engineer
malte@zellic.io ↗

Mark Pedron
Engineer
mark.pedron@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below. See section 2.3. ↗ for more information about the refactor and scope change.

August 18, 2025	Kick-off call
August 18, 2025	Start of primary review period
September 18, 2025	Zellic suggests a refactor
September 30, 2025	The audit switches to a new commit
December 5, 2025	End of primary review period

3. Detailed Findings

This section discusses the issues that were found within the final scope of the audit, which followed the scope refactor; see section 2.3. ↗ for more information on the change.

3.1. Shard verifier does not observe chips: Constraints

Target	sp1_hypercube		
Category	Soundness	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

> **Note:** Other issues related to missing-chip observations are described in Findings 3.2. ↗ and 3.3. ↗.

The shard verifier is intended to check that the prover knows a set of chips and corresponding traces, such that the chip constraints hold for each trace row and the interactions match. A chip constraint is given by a polynomial in the prover provided and preprocessed columns of a chip. The constraint given by the polynomial is said to hold for a row if the polynomial vanishes on its trace values. The verification protocol of the constraints depends on the constraint polynomials to be known.

We found that the function `verify_shard` does not let the `challenger` observe the used chips in a way that fixes the constraint polynomials.

The prover sends the names of the used chips in the field `shard_chips` of the proof.

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    let ShardProof {
        shard_chips,
        // ...
    } = proof;
```

The function `verify_shard` calls the function `verify_zerocheck`. The chips' constraints are not used outside of the function `verify_zerocheck`. Inside the function `verify_zerocheck`, the constraints are used in exactly two places, which are the function calls to `compute_padded_row_adjustment` and `eval_constraints`.

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_zerocheck(
    &self,
    shard_chips: &BTreeSet<Chip<GC::F, A>>,
    opened_values: &ShardOpenedValues<GC::F, GC::EF>,
    gkr_evaluations: &LogUpEvaluations<GC::EF>,
    // ...
{
    // ...
    let alpha = challenger.sample_ext_element::<GC::EF>();
    // ...
    let gkr_batch_open_challenge = challenger.sample_ext_element::<GC::EF>();
    // ...
    let lambda = challenger.sample_ext_element::<GC::EF>();
    // ...
    for ((chip, (_, openings)), zerocheck_eq_val) in
        shard_chips.iter().zip_eq(opened_values.chips.iter())
        .zip_eq(zerocheck_eq_vals)
    {
        // ...
        let padded_row_adjustment =
            Self::compute_padded_row_adjustment(chip, alpha, public_values);

        let constraint_eval =
            Self::eval_constraints(chip, openings, alpha, public_values)
            - padded_row_adjustment * geq_val;
        // ...
        rlc_eval =
            rlc_eval * lambda
            + zerocheck_eq_val * (constraint_eval + openings_batch);
    }
```

The function `verify_zerocheck` is supposed to reduce the claim that the constraints hold on the committed trace together with the claim that `gkr_evaluations` are evaluations of the committed traces' columns (as multilinear polynomials), to the claim that `opened_values` are evaluations of the committed traces' columns (as multilinear polynomials).

This reduction is done by combining all these different claims together in several steps:

- For each chip constraint, the claim that the constraint holds for each individual row up to the committed height is reduced to a single evaluation claim per chip column at a random point `gkr_evaluations.point`.

- These individual constraint evaluations per chip are multiplied with increasing powers of the random challenge `alpha` and added together.
- The resulting constraint sums per chip are multiplied with increasing powers of the random challenge `lambda` and added together.
- The evaluation claims from `gkr_evaluations` are multiplied with powers of `gkr_batch_open_challenge` and added to the previous sum.

If all the individual claims of the random linear combination are known to the challenger before the random challenges `alpha`, `lambda`, `gkr_batch_open_challenge`, and `gkr_evaluations.point` are generated, and the random linear combination is verified to equal zero, it can be concluded by an application of the Schwartz–Zippel lemma that all the individual summands are zero except with negligible soundness error.

If the chips' constraints are not fixed, the individual claims can depend on the generated random challenges, which makes the verification reduction unsound.

Impact

The constraints can be changed in two ways: 1) when the constraints for a given chip are chosen and 2) when the prover chooses the chips used in the provided proof.

The impact of being able to choose the constraint polynomials without changing the random challenges has been described in detail in Finding 3.3 ("STARK verifiers do not hash the constraint system into challenger") in the previous report (Zellic — SP1 Cryptographic Security Assessment, May 6th, 2025). We will therefore only describe the impact of the choice of chips by the prover here. In both cases, the result is the unsound acceptance of an incorrect proof by the verifier.

The chips have three defining characteristics that will change the verifier's behavior when changing the used chip: 1) the constraint polynomials, 2) the interaction lookup-table definitions, and 3) the widths.

Because the `main_commitment` binds the prover to the used chips' widths, the widths of different chips have to agree for them to be interchangeable by the prover. The impact of this finding will depend on the availability of interchangeable chips. If the used machine has enough such chips of common widths available, the prover can use this freedom of changing the chips to let `verify_shard` unsoundly succeed.

As a proof of concept, we describe an attack under the assumption that for enough interchangeable chips, there is a common trace such that the interaction digests do not depend on the chip. This is the case if there is a common way to disable interactions, for example, if zeroing the complete chip traces disables the interactions. In the following, we assume that the machine has `N` pairs of interchangeable chips.

The shard verification proof proceeds roughly in the following steps:

1. In `verify_logup_gkr`, the interaction claims are reduced to a single trace-evaluation claim per chip column (`logup_gkr_proof.logup_evaluations`).

2. In `verify_zerocheck`, the logUp trace-evaluation claim and the chip constraints together are reduced to a single trace-evaluation claim per chip column (`proof.opened_values`).
3. The `jagged_pcs_verifier` is used to verify these evaluation claims.

The attacking prover does the following:

- The prover chooses one chip for each pair of interchangeable chips.
- The prover commits to traces that disable all interactions.
- The prover incorrectly claims, to pass `verify_logup_gkr`, that a single chip (which is not changed later) produces the total interaction digest to match the expected `cumulative_sum` and claims the correct interaction contributions for all other chips (for which the interactions are disabled). This reduces the interaction claim to incorrect evaluation claims `logup_gkr_proof.logup_evaluations`.
- The prover will choose the correct trace openings for `proof.opened_values`.
- The prover, in `verify_zerocheck`, will reduce the incorrect `logup_gkr_proof.logup_evaluations` claim via sumcheck to an incorrect `proof.zerocheck_proof.point_and_eval.1` claim.

Because neither the evaluation claim to produce the correct interaction digest is correct, nor do the chip constraints hold for the committed traces, the last verification step in the following segment of `verify_zerocheck` would fail:

```
let mut rlc_eval = GC::EF::zero();
for ((chip, (_, openings)), zerocheck_eq_val) in
    shard_chips.iter().zip_eq(opened_values.chips.iter())
    .zip_eq(zerocheck_eq_vals)
{
    // ...
    // Zellic: Constraints do not hold
    let constraint_eval =
        Self::eval_constraints(chip, openings, alpha, public_values)
        - padded_row_adjustment * geq_val;

    // Zellic: These are the correct openings
    let openings_batch = openings
        .main
        .local
        .iter()
        .chain(openings.preprocessed.local.iter())
        .copied()
        .zip(gkr_batch_open_challenge.powers().skip(1))
        .map(|(opening, power)| opening * power)
        .sum::<GC::EF>();

    // Horner's method.
```

```
    rlc_eval =
        rlc_eval * lambda
        + zerocheck_eq_val * (constraint_eval + openings_batch);
}
// Zellic: This will fail
if proof.zerocheck_proof.point_and_eval.1 != rlc_eval {
```

When the prover switches the chosen chips, only the constraint evaluations will change. The possibly failing check `proof.zerocheck_proof.point_and_eval.1 != rlc_eval` is a linear equation in the constraint evaluations. The prover can compute the individual contributions of the different chip options. For the checked equation to hold, the prover has to find a combination of choices summing up to the correct value. This is a subset sum problem (where the subset is the set of chips the prover switches, and the summed values are the difference between the contributions of the switched and the original chip).

In the field configurations under consideration (degree 4 extension of a 31-bit prime), this is an equation over a field with about 2^{124} elements. One way to solve this problem for N interchangeable chips is as follows.

With t the target sum, the prover first computes and stores for all possible combinations of the first $N/2$ chips the value $t - x$, where x is the sum of the contributions of these chips. The prover then iterates through all possible combinations for the second $N/2$ chips, computing the sum of contributions y and looking for a match in the precomputed values $t - x$.

The total amount of memory needed for such an attack is about $2^{N/2}$ extension field elements (about 16 bytes each). Each attempt (with one fixed commitment) requires the effort to compute all contributions to the targeted `proof.zerocheck_proof.point_and_eval.1 != rlc_eval` check, including everything these contributions depend on (which thus includes all data observed prior to the sampling of relevant challenges). This effort can be roughly estimated as the effort to compute one shard proof. Additionally, $2 \cdot 2^{N/2}$ linear combinations of $N/2$ extension field elements must be computed.

Each attempt will have a success probability of about 2^{N-124}. An attacker thus expects to require roughly 2^{124-N} attempts, for a total compute effort of roughly 2^{124-N} shard proofs and $2^{124-N/2+1}$ linear combinations of $N/2$ extension field elements.

For $N = 60$, the memory requirements for the data itself are 16 GiB, which, including overhead for the hash map, is still very feasible. With only 2^{64} partial shard proofs and 2^{95} linear combinations and lookups needed, the total amount of compute required is likely to be lower than anticipated for an expected 100 bits of security.

Using larger N (requiring more memory) as well as generalizing the attack to not consider only pairs of chips but sets of chips out of which one is picked (resulting in a k-list problem) may further increase the feasibility of the attack, though requiring stronger assumptions on the available chips.

In practice, more targeted attacks using specific arithmetic properties of the used chips could be possible.

Recommendations

The challenger must observe a commitment that binds the constraint polynomials before the random challenges are generated.

Remediation

This issue has been acknowledged by Succinct Inc., and a partial fix was implemented in c9bc3514 ↗. This fix lets the challenger observe the names of the used chips at the beginning of the shard verifier. This commits the prover to the used chips, but it does not mitigate the risk of the chip constraints being chosen using the knowledge of the random challenges.

3.2. Shard verifier does not observe chips: Interactions

Target	sp1_hypercube		
Category	Soundness	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

> **Note:** Other issues related to missing chip observations are described in Findings 3.1. ↗ and 3.3. ↗.

The shard verifier is intended to check that the prover knows a set of chips and corresponding traces, such that the chip constraints hold for each trace row and the interactions match. A chip interaction is given by a sequence of affine linear functions in the prover-provided and preprocessed columns of a chip. One of these functions defines the lookup multiplicity; the other ones define the virtual columns of the lookup table. The verification protocol of the interaction lookup depends on these affine functions to be known.

We found that the function `verify_shard` does not let the `challenger` observe the used chips in a way that fixes the chip interactions' affine functions.

The prover sends the names of the used chips in the field `shard_chips` of the proof.

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    let ShardProof {
        shard_chips,
        // ...
    } = proof;
```

The function `verify_shard` calls the function `LogUpGkrVerifier::verify_logup_gkr`. The chips' interactions are not used outside of the function `verify_logup_gkr`. Inside the function

`verify_logup_gkr`, the interactions are used in exactly two places, which are the function calls to `interaction.eval`.

```
// crates/hypercube/src/logup_gkr/verifier.rs
pub fn verify_logup_gkr(
    shard_chips: &BTreeSet<Chip<F, A>>,
    // ...
    cumulative_sum: EF,
    // ..
    proof: &LogupGkrProof<EF>,
    // ..
) -> Result<(), LogupGkrVerificationError<EF>> {
    // ...
    for ((chip, openings), threshold) in

    shard_chips.iter().zip_eq(chip_openings.values()).zip_eq(degrees.values())
    {
    // ...
        for (interaction, is_send) in chip
            .sends()
            .iter()
            .map(|s| (s, true))
            .chain(chip.receives().iter().map(|r| (r, false)))
        {
            let (real_numerator, real_denominator) = interaction.eval(
                preprocessed_trace_evaluations.as_ref(),
                main_trace_evaluations,
                alpha,
                betas.as_slice(),
            );
            // ...
            let (padding_numerator, padding_denominator) = interaction.eval(
                padding_preprocessed_opening.as_ref(),
                &padding_trace_opening,
                alpha,
                betas.as_slice(),
            );
```

The function `verify_logup_gkr` is supposed to reduce the claim that the chips' interactions, together with the global interactions tracked by the function's argument `cumulative_sum`, match up, to the claim that `proof.logup_evaluations.chip_openings` are evaluations of the committed traces' columns (as multilinear polynomials) at the point `proof.logup_evaluations.point`.

The reduction is implemented via the GKR protocol, which reduces the interaction claim to an evaluation claim at the random point `interaction_point` for the multilinear polynomial, which evaluates to interaction digests on hypercube, with each hypercube point evaluating to the digest for a single interaction, evaluated using the actual trace at the point

`proof.logup_evaluations.point`. This claim is then checked by evaluating the chip interactions on the `proof.logup_evaluations.chip_openings` row and evaluating the multilinear polynomial obtained by multilinear extension at `interaction_point`.

If `interaction_point` is generated after the challenger observed commitments that bind the evaluations of the chips' interactions on the `proof.logup_evaluations.chip_openings`, it can be concluded by an application of the Schwartz–Zippel lemma that the coefficients of the two polynomials evaluated at `interaction_point` agree.

If the chips' constraints are not fixed, the interaction evaluations can depend on the `interaction_point`. In that case, instead of forcing the multilinear polynomials derived from interaction evaluations to be the same as the ones for which the interaction verification succeeded, this multilinear polynomial only has to agree with it at one known point. This makes the verification reduction unsound.

Impact

The interaction definitions can be changed in two ways: 1) when the interaction definitions for a given chip are chosen and 2) when the prover chooses the chips used in the provided proof.

The impact of being able to choose the interaction definitions without changing the random challenges has been described in detail in Finding 3.3 ("STARK verifiers do not hash the constraint system into challenger") in the previous report (Zellic — SP1 Cryptographic Security Assessment, May 6th, 2025). The result is the unsound acceptance of an incorrect proof by the verifier.

The ability of the prover to choose chips after knowing the random challenges always affects both the interaction definitions and the constraints. Because both evaluations are bound by the same trace commitment apart from the chip choice, and the constraint evaluation happens in the protocol after the interaction evaluation, any exploit of this finding is likely to also use Finding 3.1. ↗, unless there is a trace that fulfills the constraints of different interchangeable chips.

Recommendations

The challenger must observe a commitment that binds the interaction evaluation before the random challenges are generated.

Remediation

This issue has been acknowledged by Succinct Inc., and a partial fix was implemented in c9bc3514 ↗. This fix lets the challenger observe the names of the used chips at the beginning of the shard verifier. This commits the prover to the used chips, but it does not mitigate the risk of the chip interaction definitions being chosen using the knowledge of the random challenges.

3.3. Shard verifier does not observe chips: Preprocessed width

Target	sp1_hypercube		
Category	Soundness	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

> **Note:** Other issues related to missing chip observations are described in Findings 3.1. ↗ and 3.2. ↗.

The shard verifier is intended to check that the prover knows a set of chips and corresponding traces, such that the chip constraints hold for each trace row and the interactions match. Part of the defining properties of a chip are the widths of its preprocessed and main trace. The interpretation of the jagged evaluation of the combined chips' traces as individual openings depends on these widths to be known.

We found that the function `verify_shard` does not let the `challenger` observe the used chips in a way that fixes the width of the preprocessed trace.

While the prover commits to an opening shape of the main trace via the `main_commitment`, this only fixes the widths of the main trace and not the preprocessed trace.

Concretely, for the main trace, the commitment for the jagged PCS also contains shape information via `row_counts_and_column_counts`, consisting of `(height, width)` tuples. The jagged PCS itself flattens these tuples (except the last two entries, which are for padding) into just a list of heights, and operates over a commitment to a jagged table that is just a single flat list of columns with the given heights. However, given that the commitment contains these `(height, width)` pairs, we can consider the jagged PCS commitment to also come with a commitment to a particular way to interpret the flat list of columns as a list of tables.

For soundness of, for example, zerocheck, which applies the Schwartz–Zippel lemma to polynomials that depend on the width of the chips, these widths should to be fixed by the challenger at that point in the protocol. On application of Schwartz–Zippel, these widths arise from `shard_chips`.

Thus the widths that `shard_chips` encodes must have been fixed by that time in the protocol. These widths are towards the end of the `verify_shard` checked against the `row_counts_and_column_counts` shape data from the jagged PCS commitment:

```
// crates/hypercube/src/verifier/shard.rs
if !proof
```

```
    .evaluation_proof
    .row_counts_and_column_counts
    .iter()
    .cloned()
    .zip(
        once(
            shard_chips
                .iter()
                .map(MachineAir::<GC::F>::preprocessed_width)
                .filter(|&width| width > 0)
                .collect::<Vec<_>>(),
        )
        .chain(once(shard_chips.iter().map(Chip::width).collect())),
    )
    // The jagged verifier has already checked that `a.len()>=2`,
    // so this indexing is safe.
    .all(|(a, b)| a[..a.len() - 2].iter()
    .map(|(_, c)| *c).collect::<Vec<_>>() == b)
{
    Err(ShardVerifierError::InvalidShape)
} else {
    Ok(())
}
```

As the jagged PCS commitment is observed by the challenger before the challenge to that Schwartz–Zippel step is sampled, this closes the link. In the end, the width going into the polynomial used must have been the unique one from the previously observed commitment, so the prover had no choice in still changing the polynomial after the challenge point was sourced.

The situation differs, however, for the preprocessed widths. There is no way around the preprocessed trace working somewhat differently, because the preprocessed commitment is fixed in the verification key. The proof, however, can vary in which chips are included. As the preprocessed commitment must work for different proofs that may have different chips included, it is not possible to enforce that the preprocessed commitment always includes the same chips as the main commitment.

The way one can think about this is that there is an ordered set `shard_chips` of the used chips and separately an ordered set of tables committed in each of the two commitments. We require an order-preserving injection from the ordered set of tables to the chips. In the main commitment case, the intention is that the two sets are of equal size, so with this enforced, there is then a unique order-preserving injection as stated. So checking that the sizes are equal and then checking that the shape of the committed tables equals the shape claimed by `shard_chips` and the opening heights via this unique order-preserving bijection pins down all this data on the `shard_chip` and opening side — and because the main commitment is observed by the challenger early, this data is then fixed early in the protocol.

However, for the preprocessed commitment, the size of these two ordered sets is not necessarily

equal, so there is more than one injection possible. However, the challenger does not observe data that fixes what injection will be used.

The code snippet quoted above enforces that the committed preprocessed table shapes match the claimed preprocessed widths of the `shard_chips`, which claim nonzero preprocessed widths.

We may assume that the verification key is well-formed and compatible with the verifier in the following way:

1. Exactly those machine chips with nonzero preprocessed width appear (with their names) as keys in `preprocessed_chip_information`. These are also precisely those chips for which a preprocessed table is committed in `preprocessed_commit`, and the corresponding dimensions in `preprocessed_chip_information` are precisely the dimensions of the data in `preprocessed_commit`.
2. The verifier is running with the machine's chips being identical to how they were on verification key generations (so that, for example, the name used back then still refers to exactly the same chip).

These assumptions are relevant in the following snippet, in which the shard verifier enforces that `shard_chips` includes all chips from `preprocessed_chip_information` (the keys of `heights` agree with the names of the chips in `shard_chips`, enforced elsewhere):

```
// crates/hypercube/src/verifier/shard.rs
for (chip, dimensions) in vk.preprocessed_chip_information.iter() {
    if let Some(height) = heights.get(chip) {
        if *height != dimensions.height {
            return Err(

    ShardVerifierError::PreprocessedChipHeightMismatch(chip.clone())
            );
        }
    } else {
        return Err(
            ShardVerifierError::PreprocessedChipHeightMismatch(chip.clone())
        );
    }
}
```

Thus, under these assumptions, we may conclude that `shard_chips` includes all machine chips with nonzero preprocessed width. As the preprocessed commitment is assumed to have exactly one table for each machine chip with nonzero preprocessed width, there is a unique ordered injection from the tables in the preprocessed commitment to `shard_chips`, if we take the ordered list of machine chips to be fixed.

However, if we do not assume that the machine chips are fixed for an attacker, then the challenger does not fix a particular injection. By distributing the preprocessed-width zeros among the chips differently, the injection can be changed.

Impact

The list of preprocessed width can be changed in two ways: 1) when the preprocessed widths for the ordered list of machine chips is chosen and 2) when the prover chooses which chips to use in the provided proof.

In contrast to Findings 3.1. ↗ and 3.2. ↗, the scenario of the prover making use of this issue with their choice among the fixed list of machine chips is not relevant for preprocessed width. As explained above, if we assume that the machine chips are fixed, the prover retains no remaining flexibility; they must include all chips with nonzero preprocessed width.

Attacks in which the attacker may change the chip definitions are in principle possible, however. The attacker could sample challenges and then choose the discussed injection and assign the preprocessed widths accordingly to the `shard_chips`, without changing the challenges. For example, if the attacker uses two chips, and the preprocessed commitment commits to a single preprocessed table, then they can still change which chip to attach this preprocessed table to.

However, usefulness for an attack appears severely constrained. A legitimate-looking chip with preprocessed columns (in the sense of `preprocessed_width()`) should have constraints or interactions using them. At the same time, panics could happen if a nonexisting column is referenced by constraints or interactions, so if `chip.preprocessed_width() = 0`, then such a legitimate-looking chip must not have any interaction or constraint referencing preprocessed columns. But that means that in practice the attacker has no choice anymore to swap anything with regards to only the preprocessed widths, as they would also have to change constraints and/or interactions, bringing them back to Findings 3.1. ↗ and 3.2. ↗.

Thus, assuming the attacker is restricted to legitimate-looking chips (such as in a scenario in which they submit backdoored precompiles for inclusion), lack of the challenger fixing the preprocessed widths appears highly unlikely to be relevant for actual exploitation.

This issue thus amounts rather to a theoretical gap in the soundness analysis of the protocol.

Recommendations

The challenger must observe a commitment that binds the preprocessed widths before the random challenges are generated. This can be achieved by observing a commitment that binds all defining characteristics of the used chips, including the preprocessed width.

Remediation

This issue has been acknowledged by Succinct Inc..

3.4. Jagged PCS does not bind the prover to separation into rounds

Target	slop_jagged		
Category	Soundness	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The function `JaggedPcsVerifier::verify_trusted_evaluations` has the following signature:

```
//  slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &JaggedPcsProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>>
```

The jagged PCS scheme as implemented is intended as a commitment scheme for multilinear polynomials, each on `self.max_log_row_count` variables. These polynomials can be committed to in rounds, and so the argument `commitments` consists of a slice of individual round commitments, each of which is intended to commit to some number of multilinear polynomials.

The commitment scheme then allows opening all of these polynomials simultaneously at a point `point`. The corresponding openings are given in `evaluation_claims`, which also come as a batch of batches, with the outer layer corresponding to the rounds the commitment is made in.

Note that in this finding description we will ignore that the jagged PCS also involves committing to column heights, as this is not relevant to this finding.

The verifier should ensure that `commitments` indeed committed to a batch of batches of multilinear polynomials in the same shape as `evaluation_claims` and that these committed polynomials indeed evaluate at `point` to the values given in `evaluation_claims`.

However, the jagged verifier fails to enforce that the polynomials were committed in the claimed batch. For this, we consider the case of at least two rounds.

The commitments for the jagged PCS consist of hashing together a commitment for an underlying PCS as well as shape information for that batch:

```
for (
    round_column_counts,
    round_row_counts,
    modified_commitment,
    original_commitment
) in izip!(
    column_counts.iter(),
    row_counts.iter(),
    commitments.iter(),
    original_commitments.iter()
) {
    let (hasher, compressor) = GC::default_hasher_and_compressor();

    let hash = hasher.hash_iter(
        once(GC::F::from_canonical_usize(round_column_counts.len()))
        .chain(

    round_row_counts.clone().into_iter().map(GC::F::from_canonical_usize)
            .chain(
                round_column_counts.clone().into_iter()
                    .map(GC::F::from_canonical_usize)
            ),
        ),
    );
    let expected_commitment = compressor.compress([*original_commitment,
    hash]);

    if expected_commitment != *modified_commitment {
        return Err(JaggedPcsVerifierError::IncorrectTableSizes);
    }
}
```

The `column_counts` and `row_counts` contain shape information that is also checked against `evaluation_claims`. Thus, the jagged commitment for round `i` does bind the prover to a particular claimed opening shape for round `i` already, so they cannot later use a different shape for `evaluation_claims[i]`. Furthermore, they are bound to a particular commitment for the underlying PCS, `original_commitments[i]`. However, what has not yet been checked is that `original_commitments[i]` already binds the prover to the polynomials that are opened for `evaluations_claims[i]`.

While `original_commitments` is not used elsewhere in the verifier, this variable is equal to `proof.merkle_tree_commitments`, which is used in one more location, the call to the verifier of the underlying PCS:

```
self.pcs_verifier
    .verify_trusted_evaluation(
```

```
        proof.merkle_tree_commitments.as_slice(),
        evaluation_point,
        expected_eval,
        pcs_proof,
        challenger,
    )
    .map_err(JaggedPcsVerifierError::DensePcsVerificationFailed)
```

This underlying PCS verifier allows evaluating a single high-dimensional multilinear polynomial constructed from the individual ones of the smaller dimension `self.max_log_row_count`. This underlying PCS verifier does not, however, receive any information with which it could check how committed data for the polynomial should be distributed over the different rounds of `proof.merkle_tree_commitments`. The jagged PCS verifier also does not access the proof for this underlying PCS. Hence, there is no connection established between the expected data that should be contained in `proof.merkle_tree_commitments[i]` according to the jagged verifier's promises to its callers and what the prover actually supplies.

Impact

The reason this PCS splits the commitment into multiple rounds is that many protocols that may make use of a PCS like this do use several rounds (sometimes also called phases) in which the prover commits to polynomials (or other data).

One example is the SP1 shard verifier, where the first round consists of a preprocessed commitment that is fixed by the verification key, while the second round consists of a prover-generated commitment.

In other contexts, proving systems might allow the prover to first commit to one batch of polynomials, then the verifier samples a challenge, and then the prover commits to another batch of polynomials, while knowing the challenge. This is a common method to allow using constraints that compress checks via random linear combinations within circuit. In those situations, the prover controls the data committed in both rounds.

Whenever the prover can commit to data in two or more rounds, the discussed issue allows them to supply all actual polynomials only in the last commitment. For all but the last commitment, they commit to an empty underlying commitment, together with the expected non-empty shape information. Then, in the last commitment they use the shape information that is expected for the last round, together with an underlying commitment containing all polynomials for all rounds.

This critically breaks soundness of most protocols making use of multiple rounds with commitments generated by the prover.

In the case of the SP1 shard verifier, this does not apply, as only the main round is prover-controlled, while the preprocessed round commitment is part of the verification key and considered trusted. Thus, we can assume that the preprocessed commitment does include polynomials that match the committed shape, which then implies that the only other commitment must also have consistent committed polynomials and shape.

Recommendations

The jagged verifier should pass to the underlying PCS verifier information on the area expected to be committed to in each commitment. The `MultilinearPcsVerifier` trait should be adjusted by adding one more argument for this to `verify_trusted_evaluation`. The stacked verifier should be adjusted to check this new argument for consistency with `proof.batch_evaluations`.

Alternatively, if the jagged PCS verifier is only intended to be used in situations in which constraining how the prover distributes polynomials between the commitments is not required, remove one layer of batching from the type of `evaluation_claims` to avoid suggesting that the outer batching layer corresponds to the commitments, or document clearly that this is not guaranteed.

Remediation

This issue has been acknowledged by Succinct Inc., and fixes were implemented in the following commits:

- 0a09ba99 ↗
- 7310e655 ↗
- fbfefe13 ↗
- 242c5755 ↗

3.5. Random challenge `beta_seed` not guaranteed to be long enough for `verify_public_values`

Target	sp1_hypercube		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In `ShardVerifier::verify_shard`, the random challenge `beta_seed` is generated to compute a random linear combination of the virtual columns of the lookup tables. The number of virtual columns must not exceed `2^beta_seed.len() - 1`.

The number of elements `beta_seed_dim` of `beta_seed` is chosen to accommodate the largest number of virtual columns among all used chips' interactions:

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    // ...
    let max_interaction_arity = shard_chips
        .iter()
        .flat_map(|c| c.sends().iter().chain(c.receives().iter()))
        .map(|i| i.values.len() + 1)
        .max()
        .unwrap();
    let beta_seed_dim = max_interaction_arity.next_power_of_two().ilog2();
    // ...
```

However, in addition to the chip interactions, the function `verify_public_values` uses `beta_seed` as well to contribute to the lookup tables:

```
// ...
let cumulative_sum =
```

```
    -self.verify_public_values(pv_challenge, &alpha, &beta_seed,
    public_values)?;
// ...
```

Inside `verify_public_values`, `beta_seed` is expanded to betas and passed to a `VerifierPublicValuesConstraintFolder`:

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_public_values(
    // ...
    beta_seed: &Point<GC::EF>,
    // ...
) -> Result<GC::EF, ShardVerifierConfigError<GC, C>> {
    let betas = slop_multilinear::partial_lagrange_blocking(beta_seed)
        .into_buffer().into_vec();
    let mut folder = VerifierPublicValuesConstraintFolder::<GC> {
        perm_challenges: (alpha, &betas),
        alpha: challenge,
        accumulator: GC::EF::zero(),
        local_interaction_digest: GC::EF::zero(),
        public_values,
        _marker: PhantomData,
    };
    A::Record::eval_public_values(&mut folder);
```

The call to `A::Record::eval_public_values` will then call `VerifierPublicValuesConstraintFolder::{send/receive}` to add rows to the lookup tables. In these functions, `betas` (there called `perm_challenges.1`) are assumed to have enough elements to fit the argument message:

```
// crates/hypercube/src/folder.rs
fn send(&mut self, message: AirInteraction<Expr>, _scope: InteractionScope) {
    let mut denominator: Expr = (*self.perm_challenges.0).into();
    let mut betas = self.perm_challenges.1.iter().cloned();
    denominator +=
        betas.next().unwrap() * F::from_canonical_usize(message.kind
    as usize);
    for value in message.values {
        denominator += value * betas.next().unwrap();
    }
    let digest = message.multiplicity / denominator;
    self.local_interaction_digest += digest;
}
```

If `perm_challenges.1` does not have enough elements, a call to `betas.next().unwrap()` will panic.

Impact

If the prover sends a proof containing only chips whose lookup tables contain less virtual columns than the largest lookup in `verify_public_values`, the call to `verify_shard` will panic.

Recommendations

Ensure that `beta_seed_dim` is large enough to accommodate the largest interaction generated in `verify_public_values`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit bf7d5463 ↗.

3.6. Shard verifier panics if the prover selects no chips

Target	sp1_hypercube		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In the function `verify_shard`, the variable `shard_chips` first refers to the field `proof.shard_chips`, in which the prover is supposed to send the list of used chip names.

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    let ShardProof {
        shard_chips,
        // ...
    } = proof;
    // ...
```

Then, `shard_chips` is rebound to a handle for the list of actual chips with the names in `proof.shard_chips`:

```
// ...
let shard_chips = self
    .machine
    .chips()
    .iter()
    .filter(|chip| shard_chips.contains(&chip.name()))
    .cloned()
    .collect::<BTreeSet<_>>();
// ...
```

If `proof.shard_chips` did not contain any name of a chip in `self.machine.chips()`, `shard_chips` is now empty. In this case, the following call `.max().unwrap()` will panic:

```
// ...
let max_interaction_arity = shard_chips
    .iter()
    .flat_map(|c| c.sends().iter().chain(c.receives().iter()))
    .map(|i| i.values.len() + 1)
    .max()
    .unwrap();
```

Impact

The prover can send an empty list as `proof.shard_chips`, in which case `verify_shard` will panic.

Recommendations

Use a default value 0 for `max_interaction_arity`:

```
// ...
let max_interaction_arity = shard_chips
    .iter()
    .flat_map(|c| c.sends().iter().chain(c.receives().iter()))
    .map(|i| i.values.len() + 1)
    .max()
    .unwrap();
    .unwrap_or_else(0);
```

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit bf7d5463 ↗.

3.7. Malicious proofs can trigger a panic in the basefold verifier via FRI opening shape

Target	slop_basefold		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

A basefold proof contains a field `query_phase_openings_and_proofs`:

```
// slop/crates/basefold/src/verifier.rs
pub struct BasefoldProof<GC: IopCtx> {
    /// The univariate polynomials that are used in the sumcheck part
    ///  of the BaseFold protocol.
    pub univariate_messages: Vec<[GC::EF; 2]>,
    /// The FRI parts of the proof.
    /// The commitments to the folded polynomials produced in the commit phase.
    pub fri_commitments: Vec<GC::Digest>,
    /// The query openings for the individual multilinear polynmomials.
    ///
    /// The vector is indexed by the batch number.
    pub component_polynomials_query_openings_and_proofs:
        Vec<MerkleTreeOpeningAndProof<GC>>,
    /// The query openings and the FRI query proofs for the FRI query phase.
    pub query_phase_openings_and_proofs: Vec<MerkleTreeOpeningAndProof<GC>>,
    /// The prover performs FRI until we reach a polynomial of degree 0, and
    /// return the constant value of this polynomial.
    pub final_poly: GC::EF,
    /// Proof-of-work witness.
    pub pow_witness: <GC::Challenger as GrindingChallenger>::Witness,
}
```

The verifier `BasefoldVerifier::verify_mle_evaluations` only uses this in the call to `BasefoldVerifier::verify_queries`, where it is used as the argument `query_openings`.

```
fn verify_mle_evaluations(
    // ...
    self.verify_queries(
        &proof.fri_commitments,
        &query_indices,
```

```
        proof.final_poly,
        batch_evals,
        &proof.query_phase_openings_and_proofs,
        &betas,
    )?;
```

The components of `query_openings` are used in the main loop, as `query_opening`:

```
fn verify_queries(
    // ...
    for (round_idx, ((commitment, query_opening), beta)) in (
        self.fri_config.log_blowup()
        ..log_max_height)
        .rev()

    .zip_eq(commitments.iter().zip_eq(query_openings.iter()).zip_eq(betas))
    {

        let openings = &query_opening.values;
        if indices.len() != folded_evals.len()
            || indices.len() != openings.dimensions.sizes()[0]
            || indices.len() != xis.len()
        {
            return Err(BaseFoldVerifierError::IncorrectShape);
        }

        // ...

        // Check that the opening is consistent with the commitment.
        self.tcs
            .verify_tensor_openings(
                commitment,
                &indices,
                &query_opening.values,
                &query_opening.proof,
            )
            .map_err(BaseFoldVerifierError::TcsError)?;
    }
```

Note that `query_opening.values.dimensions.sizes()` is indexed at 0 with no prior checks on the shape of `query_opening.values`. Thus, a malicious proof in which the tensor `query_opening.values` has rank zero will cause the verifier to panic due to an attempted out-of-bound access.

These openings are later in the proof verified with `MerkleTreeTcs::verify_tensor_openings`, which contains the following check:

```
// slop/crates/merkle-tree/src/tcs.rs
if proof.paths.dimensions.sizes().len() != 2 ||
    opening.dimensions.sizes().len() != 2 {
    return Err(MerkleTreeTcsError::IncorrectShape);
}
```

This check that `opening.dimensions.sizes().len()` is equal to the expected value `2` is the one that is missing before accessing `query_opening.values.dimensions.sizes()` at index 0.

Impact

An attacker can easily produce a proof `proof` for basefold that causes an unconditional verifier panic by making `proof.query_phase_openings_and_proofs[0].values` a rank-zero tensor. The panic will be triggerable via any proof for a protocol whose verifier uses the basefold verifier, including the jagged PCS or shard verifier, if they are configured to use basefold as the underlying dense PCS.

This finding can be demonstrated by modifying the test `test_jagged_basefold` and running the `test_koala_bear_jagged_basefold` function. The test creates a proof for the jagged PCS and verifies it at the end. The verifier can be made to panic by inserting the following code just before the call to the verifier at the very end:

```
// slop/crates/jagged/src/basefold.rs
// #[tokio::test]
async fn test_jagged_basefold<
// ..
{
    // ..
    let mut proof = proof;
    use slop_tensor::Tensor;
    let empty_values : Tensor<<GC as IopCtx>::F> = Tensor::with_sizes_in([],
        CpuBackend);
    proof.pcs_proof.pcs_proof.query_phase_openings_and_proofs[0].values =
        empty_values;

    jagged_verifier
        .verify_trusted_evaluations(
```

The test will then panic on verification with the message `index out of bounds: the len is 0 but the index is 0` at the `openings.dimensions.sizes()[0]` attempted access.

Recommendations

Check that `openings.dimensions.sizes().len()` is equal to 2 before the check involving `openings.dimensions.sizes()[0]`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 9faa6dc1 ↗.

3.8. The `JaggedEvalSumcheckConfig::jagged_evaluation` verifier rejects proofs where the number of columns is maximal

Target	slop_jagged		
Category	Completeness	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The function `JaggedEvalSumcheckConfig::jagged_evaluation` has the following signature:

```
// crates/jagged/src/jagged_eval/sumcheck_eval.rs
pub fn jagged_evaluation<EF: ExtensionField<F>, Challenger:
    FieldChallenger<F>>(
    params: &JaggedLittlePolynomialVerifierParams<F>,
    z_row: &Point<EF>,
    z_col: &Point<EF>,
    z_trace: &Point<EF>,
    proof: &JaggedSumcheckEvalProof<EF>,
    challenger: &mut Challenger,
) -> Result<EF, JaggedEvalSumcheckError<EF>>
```

The argument `params` is intended to encode an admissible sequence of cumulative heights t of length K (see page 6 of *Jagged Polynomial Commitments (or: How to Stack Multilinears)* ↗ by Hemo, Jue, Rabinovich, Roh, and Rothblum for this terminology), and the purpose of this function is to verify a proof of the evaluation below, with the resulting value returned.

$$\sum_{0 \le y < K} \mathsf{eq}(z_c, \texttt{int\_to\_bits\_be}(y)) \cdot \hat{g}(z_r, z_t, t_{y-1}, t_y) \tag{3.1}$$

In the expression above, we use z_r, z_c, z_t for `z_row`, `z_col`, and `z_trace` respectively, and $\hat{g}$ and eq are as defined in the jagged paper.

The relevance of this expression is that (under some additional assumptions) it is equal to $\hat{f}_t\,(z_r, z_c, z_t)$, which is a crucial function used in the jagged PCS, linking the underlying dense index z_t with the row index z_r and the column index z_c.

The number K in the above sum arises from `params.col_prefix_sums`, which is a vector that contains what we denote by $[t_{-1}, t_0, \ldots, t_{K-1}]$. Here, each t_i is interpreted as the total area of the i-th column together with all previous columns, with $t_{-1} = 0$, and $t_0, \ldots, t_{K-1}$ is assumed to be admissible in the sense of the jagged paper. Note that then K is the number of columns that this

sequence encodes, with the columns indexed by 0 through $K - 1$.

Let k be the dimension of the point z_c. As this point represents the column index, we would expect k to be large enough that each column index of the columns defined by t can be adressed with k bits. This is the case as long as $2^k \geq K$ — the maximum number with k bits is $2^k - 1$, and the columns are indexed by 0 through $K - 1$.

With this in mind, the following checks have meaning:

```
let z_col_partial_lagrange = Mle::blocking_partial_lagrange(z_col);
let z_col_partial_lagrange = z_col_partial_lagrange.guts().as_slice();

if z_col_partial_lagrange.len() < branching_program_evals.len() {
    return Err(JaggedEvalSumcheckError::IncorrectShape);
}

if branching_program_evals.len() + 1 != params.col_prefix_sums.len() {
    return Err(JaggedEvalSumcheckError::IncorrectShape);
}
```

Here, the first check ensures that $2^k \geq$ `branching_program_evals.len()` and the second that `branching_program_evals.len()` $+ 1 = K + 1$, and thus jointly they imply $2^k \geq K$.

However, later in the function, there is another check also involving k and K:

```
if params.col_prefix_sums.len() > z_col_partial_lagrange.len() {
    return Err(JaggedEvalSumcheckError::IncorrectShape);
}
```

This returns an error whenever $K + 1 > 2^k$, so whenever $K \geq 2^k$. Note that, taking into account the previous checks, this additionally rules out precisely the case $K = 2^k$.

However, $K = 2^k$ appears to be a reasonable configuration; it is the situation in which k bits are used to address columns, and all columns are indeed in use (in the sense of their height, which may be zero, being encoded in t).

Impact

There does not seem to be a reason for `JaggedEvalSumcheckConfig::jagged_evaluation` itself to rule out $K = 2^k$. Doing so could thus be considered a completeness issue, as valid proofs for this case are rejected.

From the perspective of only `JaggedEvalSumcheckConfig::jagged_evaluation`, it is unclear what precise specification is intended, so the overarching `JaggedPcsVerifier` can clarify this. Here, `JaggedEvalSumcheckConfig::jagged_evaluation` is called as follows:

```
// crates/jagged/src/verifier.rs
let jagged_eval = JaggedEvalSumcheckConfig::jagged_evaluation(
    params,
    &z_row,
    &z_col,
    &sumcheck_proof.point_and_eval.0,
    jagged_eval_proof,
    challenger,
)
.map_err(|_| JaggedPcsVerifierError::JaggedEvalProofVerificationFailed)?;
```

Recall that K is equal to `params.col_prefix_sums.len() - 1`, while k is the dimension of `z_col`.

We find that they are closely related, with k essentially computed from K:

```
let num_col_variables = (params.col_prefix_sums.len() - 1)
    .next_power_of_two().ilog2();
let z_col = (0..num_col_variables)
    .map(|_| challenger.sample_ext_element::<GC::EF>())
    .collect::<Point<_>>();
```

Here, `num_col_variables` and hence the dimension of `z_col`, so k, is set equal to the smallest integer so that $K \leq 2^k$. Should K be a power of two, this will result in the case $K = 2^k$, which will cause verification to fail due to the discussed check in `JaggedEvalSumcheckConfig::jagged_evaluation`. Note that the computation of `z_col` from `params.col_prefix_sums` happens on the side of the verifier, so the prover cannot directly influence this and decide to provide an extra (unnecessary) component for `z_col` to circumvent the overly strict check.

This completeness impact can be verified with a test. The file crates/jagged/src/basefold.rs contains the test `test_koala_bear_jagged_basefold`, which is a wrapper around `test_jagged_basefold`, which tests the jagged PCS with some specific settings for the number of columns:

```
let column_counts_rounds = vec![vec![128, 45, 32], vec![512]];
```

The actual column numbers used at the level of `JaggedEvalSumcheckConfig::jagged_evaluation` will be one higher for both rounds due to the addition of an empty (i.e., height zero) padding column. Thus, the total number of columns in this test is $K = 719$. Changing the 45 to 350 changes this to $K = 1024$. In this case, the mentioned check will be triggered and verification will fail, demonstrating the completeness impact.

Recommendations

The overly strict check ruling out $K = 2^k$ should likely be removed.

Additionally, we recommend to implement more comprehensive tests. In this case, while a test `test_jagged_basefold` exists to test the jagged PCS, it only tests a single very specific configuration in terms of numbers of rows and columns. Testing only on a single configuration is prone to missing issues that only arise in edge cases (such as K being a power of two). We thus recommend to carry out such tests across entire ranges of configurations as well as randomized configurations.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit b39c6e6a ↗.

The fix changed the relevant check to return an error on `params.col_prefix_sums.len() - 1 > z_col_partial_lagrange.len()`.

3.9. Missing check that all public values are observed

Target	sp1_hypercube		
Category	Soundness	**Severity**	Critical
Likelihood	Low	**Impact**	Medium

Description

The following code snippet checks the number of elements of `public_values` of the shard verifier:

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    let ShardProof {
        // ...
        public_values,
        // ...
    } = proof;
    // ...
    if public_values.len() < self.machine.num_pv_elts() {
        tracing::error!("invalid public values length: {}",
    public_values.len());
        return Err(ShardVerifierError::InvalidPublicValues);
    }

    // Observe the public values.
    challenger.observe_slice(&public_values[0..self.machine.num_pv_elts()]);
    // ...
    let cumulative_sum =
        -self.verify_public_values(pv_challenge, &alpha, &beta_seed,
    public_values)?;
    // ...
    self.verify_zerocheck(
        // ...
        public_values,
```

```
            // ...
        )?;
```

Only the slice of the expected length `self.machine.num_pv_elts()` is observed. But the complete `public_values` are passed to the constraint validators later.

Impact

If the chips use the excess public values, these excess values have not been observed and could be chosen by the prover after all the challenges were generated by the challenger. This would make the proof verification unsound.

Recommendations

Do not allow more public values than expected.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 368eec5e ↗.

3.10. On the small total area compared to maximum column height, the jagged verifier repeats column values rather than padding with zero

Target	slop_jagged		
Category	Soundness	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

This finding is about the jagged verifier in combination with the `JaggedEvalSumcheckConfig::jagged_evaluation` verifier for evaluation of the jagged polynomial and in combination with `BranchingProgram::eval`.

To begin, `BranchingProgram::eval` is intended to evaluate the function $\hat{g}$ (see Hemo, Jue, Rabinovich, Roh, and Rothblum's paper *Jagged Polynomial Commitments (or: How to Stack Multilinears)* ↗). It does so using `self.num_vars + 1` bits of the four arguments, so it in the end returns the following:

$$\hat{g}\Big(\mathtt{tr}_{\mathtt{self.num\_vars}+1}(\mathtt{self.z\_row}), \mathtt{tr}_{\mathtt{self.num\_vars}+1}(\mathtt{self.z\_index}), \mathtt{tr}_{\mathtt{self.num\_vars}+1}(\mathtt{prefix\_sum}), \mathtt{tr}_{\mathtt{self.num\_vars}+1}(\mathtt{next\_prefix\_sum})\Big)$$

Here we use $\mathtt{tr}_n(x)$ as notation for truncation of x to length n by zero-extending on the left if necessary and removing excessive components on the left.

This means it is important for any caller that wishes to obtain $\hat{g}(\mathtt{self.z\_row}, \mathtt{self.z\_index}, \mathtt{prefix\_sum}, \mathtt{next\_prefix\_sum})$ instead to ensure that those four arguments only have nonzero entries in the least significant `self.num_vars+1` components.

The function `JaggedEvalSumcheckConfig::jagged_evaluation` uses `BranchingProgram::eval` towards the end:

```
// slop/crates/jagged/src/jagged_eval/sumcheck_eval.rs
pub fn jagged_evaluation<EF: ExtensionField<F>, Challenger:
    FieldChallenger<F>>(
    // ...
    z_row: &Point<EF>,
    z_col: &Point<EF>,
    z_trace: &Point<EF>,
    // ...
) -> Result<EF, JaggedEvalSumcheckError<EF>> {
    // ...
```

```
    let branching_program = BranchingProgram::new(z_row.clone(),
    z_trace.clone());
    jagged_eval_sc_expected_eval *=
        branching_program.eval(&first_half_z_index, &second_half_z_index);
    // ...
}
```

This first creates a new `BranchingProgram` with `BranchingProgram::new`:

```
// slop/crates/jagged/src/poly.rs
pub fn new(z_row: Point<K>, z_index: Point<K>) -> Self {
    let log_m = z_index.dimension();

    Self {
        z_row,
        z_index,
        memory_states: all_memory_states(),
        bit_states: all_bit_states(),
        num_vars: log_m,
    }
}
```

So we will have `branching_program.num_vars = z_trace.dimension()`, and the value returned by `branching_program.eval` will be as follows:

$$\hat{g}\Big(\mathtt{tr}_{\mathtt{z\_trace.dimension()}+1}(\mathtt{z\_row}), \mathtt{tr}_{\mathtt{z\_trace.dimension()}+1}(\mathtt{z\_index}),$$
$$\mathtt{tr}_{\mathtt{z\_trace.dimension()}+1}(\mathtt{first\_half\_z\_index}), \mathtt{tr}_{\mathtt{z\_trace.dimension()}+1}(\mathtt{second\_half\_z\_index})\Big)$$

We have $\mathtt{tr}_{\mathtt{z\_trace.dimension()}+1}(\mathtt{z\_trace}) = \mathtt{z\_trace}$ (up to zero padding). The dimension of both `first_half_z_index` and `second_half_z_index` is `params.col_prefix_sums[0].len()`:

```
// slop/crates/jagged/src/jagged_eval/sumcheck_eval.rs
pub fn jagged_evaluation<EF: ExtensionField<F>, Challenger:
    FieldChallenger<F>>(
    // ...
    let result = partially_verify_sumcheck_proof(
        partial_sumcheck_proof,
        challenger,
        2 * params.col_prefix_sums[0].len(),
        2,
    );
```

```
    if let Err(result) = result {
        return Err(JaggedEvalSumcheckError::SumcheckError(result));
    }

    let (first_half_z_index, second_half_z_index) = partial_sumcheck_proof
        .point_and_eval
        .0
        .split_at(partial_sumcheck_proof.point_and_eval.0.dimension() / 2);
```

Thus, for the last two arguments to not be truncated, it is necessary that `params.col_prefix_sums[0].len() <= z_trace.dimension()+1`.

For the first argument, `z_row`, we directly conclude that we need `z_row.dimension() <= z_trace.dimension()+1`.

Thus, these two are assumptions that `JaggedEvalSumcheckConfig::jagged_evaluation` imposes on the caller in the current implementation.

Impact

Let us consider what happens when these assumptions are not satisfied, focusing on the case of `z_row`. The branching program evaluation will then truncate, so ignore the more significant (left) components of `z_row`. The goal of `JaggedEvalSumcheckConfig::jagged_evaluation` is to compute $\sum_{0<=y<\texttt{t.len()}-1} eq(\texttt{z\_col}, \texttt{int\_to\_bits\_be}(y)) \cdot \hat{g}(\texttt{z\_row}, \texttt{z\_trace}, t_{y-1}, t_y)$, where `[t_{-1}, ...] = params.col_prefix_sums`. The `z_row` argument is only used in the branching program evaluation, which computes the $\hat{g}$ factor after a sumcheck compressed the sum.

Thus, `JaggedEvalSumcheckConfig::jagged_evaluation` will act on overly long `z_row` arguments as if it had been truncated to dimension `z_trace.dimension()+1`.

This can be confirmed with the following proof of concept:

```
#[cfg(test)]
mod tests {
    use slop_algebra::extension::BinomialExtensionField;
    use slop_alloc::Buffer;
    use slop_koala_bear::KoalaBear;
    use slop_multilinear::Point;
    use crate::jagged_eval::CpuBackend;
    use crate::JaggedEvalProver;
    use crate::JaggedLittlePolynomialVerifierParams;
    use slop_challenger::IopCtx;
    use crate::JaggedEvalSumcheckConfig;
    use crate::JaggedLittlePolynomialProverParams;
    use slop_koala_bear::KoalaBearDegree4Duplex;
    use slop_algebra::AbstractField;
    use slop_alloc::GLOBAL_CPU_BACKEND;
```

```
use crate::{JaggedAssistSumAsPolyCPUImpl, JaggedEvalSumcheckProver};

#[tokio::test]
async fn zellic_test_jagged_eval_truncation() {
    type F = KoalaBear;
    type EF = BinomialExtensionField<F, 4>;
    type Challenger = <KoalaBearDegree4Duplex as IopCtx>::Challenger;
    let prover : JaggedEvalSumcheckProver<
            F,
            JaggedAssistSumAsPolyCPUImpl<
                F,
                EF,
                Challenger,
            >,
            CpuBackend,
            Challenger,
        > = JaggedEvalSumcheckProver::default();
    let params_prover : JaggedLittlePolynomialProverParams =
        JaggedLittlePolynomialProverParams::new(vec![1,2], 4);
    let params_verifier : JaggedLittlePolynomialVerifierParams<F> =
        params_prover.clone().into_verifier_params();
    // Max column height is 16, so 4 bits
    // Total area 3, so needs 2 bits
    // So we use `z_trace` of dimension 2
    // With 2 columns, need dimension 1 for `z_col`. Due to F710 need to
use one more, so use 2.
    // Will use dimension 4 for `z_row`

    fn vec_to_point(v : &Vec<usize>) -> Point<EF> {
        let mut buffer : Buffer<EF, CpuBackend> =
            Buffer::with_capacity(v.len());
        for x in v.iter() {
            buffer.push(EF::from_canonical_usize(*x));
        }
        Point::new(buffer)
    }
    fn vec_to_int_be(v : &Vec<usize>) -> usize {
        let mut result = 0;
        for x in v.iter() {
            result = (result << 1) + x;
        }
        result
    }

    fn int_to_vec_be(value: usize, l: usize) -> Vec<usize> {
```

```
            let mut result : Vec<usize> = vec![];
            for i in 0..l {
                result.push((value >> i) & 1);
            }
            result.reverse();
            result
        }

        let one : EF = EF::one();
        let zero : EF = EF::zero();

        for r in 0..16 {
            for c in 0..2 {
                for i in 0..4 {
                    let z_row_int = int_to_vec_be(r, 4);
                    let mut z_row_int_prover = z_row_int.clone();
                    z_row_int_prover[0] = 0;
                    let z_col_int = int_to_vec_be(c, 2);
                    let z_trace_int = int_to_vec_be(i, 2);

                    let z_row = vec_to_point(&z_row_int);
                    let z_row_prover = vec_to_point(&z_row_int_prover);
                    let z_col = vec_to_point(&z_col_int);
                    let z_trace = vec_to_point(&z_trace_int);

                    let mut challenger : Challenger =
                        KoalaBearDegree4Duplex::default_challenger();

                    let proof = prover.prove_jagged_evaluation(
                        &params_prover,
                        &z_row_prover,
                        &z_col,
                        &z_trace,
                        &mut challenger.clone(),
                        GLOBAL_CPU_BACKEND
                    ).await;
                    let result =
    JaggedEvalSumcheckConfig::<F>::jagged_evaluation(
                        &params_verifier,
                        &z_row,
                        &z_col,
                        &z_trace,
                        &proof,
                        &mut challenger.clone()
                    );
                    let result = result.unwrap();
                    if result == one {
```

```
                        println!(
                            "\x1b[31mResult 1 for
    z_row={}   z_col={}   z_trace={}\x1b[0m",
                            r,
                            c,
                            i
                        );
                    }
                    else if result == zero {
                        println!("Result 0 for
    z_row={}   z_col={}   z_trace={}", r, c, i);
                    }
                    else {
                        panic!("Should not happen!");
                    }
                }
            }
        }
    }
}
```

This test uses a configuration in which the maximum column height is 2^4, with two columns present, the first of height 1 and the second of height 2. This makes the total area 3, requiring two bits. We thus use `z_trace` of dimension 2. We use `z_col` of dimension 2; while one bit should suffice, the proof would be erroneously rejected due to a bug described in Finding 3.8. ↗. For `z_row`, we use dimension 4, matching the maximum column height.

The test produces and verifies proofs while going through all hypercube points for these arguments in which the unnecessary `z_col` component is zero. To create the proof, the prover is passed a modified `z_row`, in which the first component is replaced by zero.

With this, all proofs pass. The return values are zero or one, and one occurs for these inputs:

```
Result 1 for z_row=0   z_col=0   z_trace=0
Result 1 for z_row=0   z_col=1   z_trace=1
Result 1 for z_row=1   z_col=1   z_trace=2
Result 1 for z_row=8   z_col=0   z_trace=0
Result 1 for z_row=8   z_col=1   z_trace=1
Result 1 for z_row=9   z_col=1   z_trace=2
```

The first three of these are correct, but the last three are incorrect and should have resulted in zero.

The verifier is thus unsound if used on `z_row` inputs that can have arbitrary dimension, as it is possible to prove incorrect results.

By slightly modifying the above code and passing `z_row` to the prover instead of the modified `z_row_prover`, the test also demonstrates the corresponding completeness issue: it is not

possible to prove the actually correct result for those values.

While the test only prints the results for points on the hypercube, the result will be incorrect for many other points in which a truncated component of `z_row` is nonzero as well.

Note that there is a function analogous to `JaggedEvalSumcheckConfig::jagged_evaluation` that carries out the computations directly, without assistance from a prover, `JaggedLittlePolynomialVerifierParams::full_jagged_little_polynomial_evaluation`. In contrast to the case we discussed above, `full_jagged_little_polynomial_evaluation` explicitly ensures that longer `z_row`s are supported:

```
// The program below reads only the first log_m +1 bits of z_row, but z_row
    could in theory
// be longer than that if the total trace area is less than the padded height.
    This
// correction ensures that the higher bits are zero.
let log_m = z_index.dimension();
let z_row_correction: EF = z_row
    .reversed()
    .to_vec()
    .iter()
    .skip(log_m + 1)
    .cloned()
    .map(|x| EF::one() - x)
    .product();
```

Let us now turn towards the usage in the jagged verifier, which is intended to open columns at `z_row = point`. In the verifier, apart from checking that the dimension corresponds to the maximal column dimension `self.max_log_row_count`, `z_row` is only used in the call to `JaggedEvalSumcheckConfig::jagged_evaluation`. Note that while `self.max_log_row_count` is used in some other places (in particular as padding columns that are added can have at most height `1 << self.max_log_row_count`), there appears to be no check that prevents `1<<self.max_log_row_count` from being large compared to the total area.

Note that while padding is done, this is to a multiple of `1<<C::log_stacking_height(&self.pcs_verifier)`, and it is possible that `C::log_stacking_height(&self.pcs_verifier)` is smaller than `self.max_log_row_count`. In particular, the default settings have smaller log-stacking height than the maximum column dimension:

```
// crates/prover/src/core.rs
pub const CORE_LOG_STACKING_HEIGHT: u32 = 21;
pub const CORE_MAX_LOG_ROW_COUNT: usize = 22;
```

The call to `JaggedEvalSumcheckConfig::jagged_evaluation` performed by the jagged verifier then looks as follows:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    // ...
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    // ...
    let jagged_eval = JaggedEvalSumcheckConfig::jagged_evaluation(
        params,
        &z_row,
        &z_col,
        &sumcheck_proof.point_and_eval.0,
        jagged_eval_proof,
        challenger,
    )
    .map_err(|_| JaggedPcsVerifierError::JaggedEvalProofVerificationFailed)?;
```

To avoid truncation, we would need `params.col_prefix_sums[0].len() <= sumcheck_proof.point_and_eval.0.dimension()+1` and `z_row.dimension() <= sumcheck_proof.point_and_eval.0.dimension()+1`.

The dimension of `sumcheck_proof.point_and_eval.0` is `params.col_prefix_sums[0].len() - 1`:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    // ...
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    // ...
    partially_verify_sumcheck_proof(
        sumcheck_proof,
        challenger,
        params.col_prefix_sums[0].len() - 1,
        2,
    )
    .map_err(JaggedPcsVerifierError::SumcheckError)?;
```

Thus, this boils down to `params.col_prefix_sums[0].len() <= params.col_prefix_sums[0].len()` and `z_row.dimension() <= params.col_prefix_sums[0].len()`. The first holds, so we are only left with the potential truncation of `z_row`, which can actually happen if the dimension of the prefix sum points is smaller than the maximum column dimension.

More concretely, the dimension of `params.col_prefix_sums[0]` is `log_m`, which is the log of the total area:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    // ...
```

```
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    // ...
    let log_m = log2_ceil_usize(usize_prefix_sums.last().copied().unwrap());
    // ...
    let point_prefix_sums: Vec<Point<GC::F>> =
        usize_prefix_sums.iter().map(|&x| Point::from_usize(x, log_m +
    1)).collect();

    if point_prefix_sums != params.col_prefix_sums {
        return Err(JaggedPcsVerifierError::IncorrectShape);
    }
```

The upshot is the following. Suppose we are in a situation where the total column area is small compared to the maximum column height. On evaluation, columns should be treated as padded with zeros. However, what actually happens is that they are padded with zeros up to a dimension one higher than the total area, and then this medium-size padded column is padded to the maximum height by duplicating it as often as needed, rather than padding with zero.

Concretely, let us go back to the example from the proof-of-concept code. Let us assume for simplification that the log-stacking height is small enough that no padding occurs. In this example, this would mean a log-stacking height of zero — in a real example, both maximum column dimension as well as the log-stacking height would be bigger, and the problem can occur as long as the log-stacking height is smaller than the maximum column dimension minus one.

In the example, the total area is 3, with columns of height 1 and 2, with maximum column height $16 = 2^4$. Then `log_m` will be 2, meaning that `z_row`, which is of dimension 4, will be truncated to length `log_m + 1 = 3`, so its most significant (leftmost) component is treated as if it were zero.

Suppose the values in the dense commitment are `[A, B, C]`. Then, the two columns will be opened as if they had contents `[A, 0, 0, 0, 0, 0, 0, 0, A, 0, 0, 0, 0, 0, 0, 0]` and `[B, C, 0, 0, 0, 0, 0, 0, B, C, 0, 0, 0, 0, 0, 0]`. In particular, it is possible to open the columns in a way where they exceed the height they are supposed to have.

Similarly to the `JaggedEvalSumcheckConfig::jagged_evaluation`, this has a soundness aspect (can open things incorrectly) and a completeness aspect (cannot open them correctly).

While it is unclear how usefully this can be exploited at the level of the shard verifier in practice, it also impacts the shard verifier.

There, the traces take part both in constraint checks as well as in computing contributions for interaction as part of the logUp argument. If `trace` is the trace for a chip and `C` a constraint, then what is roughly checked with regards to this constraint is that `C(trace[i]) - geq(i, height)*C(0) = 0` for every $0 \leq i < 2^m$, where 2^m is the total height of the (virtual) trace, and `height` is the actual committed height. For `i < height`, this should amount to `C(trace[i]) - 0*C(0) = 0`, so `C(trace[i]) = 0`, while for `i >= height`, we obtain `C(0) - C(0) = 0`, which is true if `height` is accurate and the padding is with zeros. Thus, `C(trace[i]) - geq(i, height)*C(0) = 0` for all $0 \leq i < 2^m$ captures the intent that the constraint should evaluate to zero on all real rows (so those below the height).

With regards to the logUp cumulative sum, what is computed is roughly the sum over all $0 \leq i < 2^m$ over all interactions of a term of the form $\frac{p(\texttt{trace}[i]) - \texttt{geq}(i, \texttt{height}) \cdot p(0)}{q(\texttt{trace}[i]) + \texttt{geq}(i, \texttt{height}) \cdot (1 - q(0))}$, with p and q computing the numerator and denominator of the logUp contribution for this interaction at row `i`. The intention is that for `i > height`, this will evaluate to $\frac{0}{1} = 0$, while it will evaluate to the intended contribution $\frac{p(\texttt{trace}[i])}{q(\texttt{trace}[i])}$ otherwise.

Now given the described issue with the jagged PCS verifier means that it may happen that for $i \geq$ `height`, it does not hold that `trace[i] = 0` but instead evaluates to some row `row` that already occurred as `trace[j]` for $j <$ `height`. In that case, for constraint `C`, we must have `C(row) = 0` as well as `C(row) - C(0) = 0`. It is thus only possible to prove such openings in the context of the shard verifier, if `C(0) = 0` for every constraint. The row will then contribute both with the intended $\frac{p(\texttt{row})}{q(\texttt{row})}$ as well as with $\frac{p(\texttt{row}) - p(0)}{q(\texttt{row}) + 1 - q(0)}$ to the logUp cumulative sum.

Considering for example a chip with an interaction for which `q(0) = 1`, this amounts to the multiplicity of the interaction from `row` to be `p(row) + p(row) - p(0)` instead of `p(row)`. If for example `p(row) = 1` and `p(0) = 2`, this could mean that this row does not actually net contribute with interactions, even though it should, potentially bypassing crucial checks. However, for this to be useful to an attacker, this row must take part in other interactions, and in general it may be rare to find a chip in which constraints and interactions happen to have the precise form to make successful use of this issue.

Recommendations

Ensure that `num_vars` of the branching program is big enough so that no truncation happens in any argument. This could happen either at the level of `JaggedEvalSumcheckConfig::jagged_evaluation` or at the level of `JaggedPcsVerifier::verify_trusted_evaluations`.

Remediation

This issue has been acknowledged by Succinct Inc..

3.11. Jagged verifier can panic on some points if columns are short

Target	slop_jagged		
Category	Completeness	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The jagged verifier towards the end reduces the claim to be checked to

```
// slop/crates/jagged/src/verifier.rs
sumcheck_proof.point_and_eval.1 = Q(evaluation_point) * jagged_eval
```

where `Q` is the underlying dense polynomial.

The way this is checked is by computing `expected_eval = sumcheck_proof.point_and_eval.1 / jagged_eval` and then verifying with the PCS verifier for the underlying dense PCS that `Q` evaluates to this value at `evaluation_point`:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &JaggedPcsProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    // ...

    let expected_eval = sumcheck_proof.point_and_eval.1 / jagged_eval;

    // Verify the evaluation proof using the (dense) stacked PCS verifier.
    let evaluation_point = sumcheck_proof.point_and_eval.0.clone();
    self.pcs_verifier
        .verify_trusted_evaluation(
            proof.merkle_tree_commitments.as_slice(),
            evaluation_point,
            expected_eval,
            pcs_proof,
            challenger,
```

```
        )
        .map_err(JaggedPcsVerifierError::DensePcsVerificationFailed)
    // ...
}
```

However, this is only correct if `jagged_eval` is nonzero. For cases in which it is zero, the verifier will panic.

Impact

The jagged PCS verifier can panic if `jagged_eval` in the snippet above is zero. This amounts to a completeness issue for those cases as well as a denial-of-service vector for malicious provers.

That `jagged_eval` is zero can indeed occur. The value of `jagged_eval` is computed as follows:

```
let jagged_eval = JaggedEvalSumcheckConfig::jagged_evaluation(
    params,
    &z_row,
    &z_col,
    &sumcheck_proof.point_and_eval.0,
    jagged_eval_proof,
    challenger,
)
.map_err(|_| JaggedPcsVerifierError::JaggedEvalProofVerificationFailed)?;
```

It is evaluation of the jagged polynomial $\hat{f}$, concretely:

$$\texttt{jagged\_eval} = \hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \texttt{z\_col}, \texttt{sumcheck\_proof.point\_and\_eval.0})$$

Here, the prefix sums `params.col_prefix_sums` are bound to the commitment and reflect its shape; we assume these to be choosable by the prover (subject to the relevant restrictions). The point `z_row` is the point of dimension `self.max_log_row_count` that is supplied to the verifier and is the point at which the committed polynomials are to be evaluated. Considering the jagged PCS verifier as a standalone verifier, we assume that this could be any point, possibly choosable by a malicious prover.

In contrast, `z_col` and `sumcheck_proof.point_and_eval.0` are sampled from the challenger, and so, as we assume that the commitment and `z_row` have been observed by the challenger before `JaggedPcsVerifier::verify_trusted_evaluations` was called, these are random points that the prover cannot predict before choosing `params.col_prefix_sums` and `z_row`.

Consider the multilinear polynomial $\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \_, \_)$ (i.e., where we fix the first argument to be `z_row` and consider this a polynomial in the second and third slot only). Then, if this is not the zero polynomial, the probability that this will evaluate to zero at a random point is very low. This is the usual Schwartz–Zippel lemma.

Considering the case where $\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \_, \_)$ is not zero but

$\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \texttt{z\_col}, \texttt{sumcheck\_proof.point\_and\_eval.0})$ is, this is thus an extremely unlikely case, in which the verifier would panic. As the code could be restructured to avoid a panic in this case, this may still be considered suboptimal. However, for practical purposes, this situation will not occur.

So we then ask whether $\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \_, \_)$ could be the zero polynomial. Indeed, this can be the case. Concretely, consider for example a configuration in which all columns have height at most h, and `z_row = int_to_bits_be(i)` for `i > h`. Then, it is easy to check that on the hypercube, $\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \_, \_)$ will be identically zero, because there is no column for which this row index matches any dense index. Then $\hat{f}_{\texttt{params.col\_prefix\_sums}}(\texttt{z\_row}, \_, \_)$ itself will be identically zero.

This can be tested with the test `test_koala_bear_jagged_basefold` from slop/crates/jagged/src/basefold.rs by changing `test_jagged_basefold` as follows. Remove the previous random sampling of `eval_point`, and replace the initial settings' lines with the following:

```
let row_counts_rounds = vec![vec![1]];
let column_counts_rounds = vec![vec![2]];
let num_rounds = row_counts_rounds.len();

let log_blowup = 1;
let log_stacking_height = 1;
let max_log_row_count = 4;


use slop_alloc::Buffer;
use slop_algebra::AbstractField;

let v : Vec<usize> = vec![0,0,1,1];

let mut buffer : Buffer<GC::EF, CpuBackend> = Buffer::with_capacity(v.len());
for x in v.iter() {
    buffer.push(GC::EF::from_canonical_usize(*x));
}
let eval_point : Point<GC::EF> = Point::new(buffer);
```

The test will panic in the verifier on the discussed line, with a `Tried to invert zero` error.

Recommendations

The panic can be avoided by restructuring the code so that the prover supplies the claimed value of `expected_eval` and then by verifying both that `expected_eval` is correct via the underlying PCS verifier and that `sumcheck_proof.point_and_eval.1 = expected_eval * jagged_eval`. This will avoid the division.

Remediation

This issue has been acknowledged by Succinct Inc., and fixes were implemented in the following commits:

- fee987dd ↗
- 052c8f6a ↗

The switch from `verify_trusted_evaluations` to `verify_untrusted_evalutions` addresses the following issue:

If `jagged_eval` is nonzero, the value of `expected_eval` is fixed by the challenger state via the new check `sumcheck_proof.point_and_eval.1 = expected_eval * jagged_eval` together with both `sumcheck_proof.point_and_eval.1` as well as `jagged_eval` being fixed by the challenger before the call to the underlying PCS verifier:

```
// slop/crates/jagged/src/verifier.rs
self.pcs_verifier
    .verify_trusted_evaluation(
        proof.merkle_tree_commitments.as_slice(),
        evaluation_point,
        expected_eval,
        pcs_proof,
        challenger,
    )
    .map_err(JaggedPcsVerifierError::DensePcsVerificationFailed)
```

If `jagged_eval` is zero, the new check will return an error should `sumcheck_proof.point_and_eval.1` be nonzero and otherwise continue without error for any prover-chosen value of `expected_eval`.

As `expected_eval` is not observed by the challenger, this means that in those specific cases, the call to `self.pcs_verifier.verify_trusted_evaluation` will not ensure that the challenger has fixed `expected_eval`. In this case, `self.pcs_verifier` is generic, satisfying trait `MultilinearPcsVerifier`, and the distinction between `MultilinearPcsVerifier::verify_trusted_evaluation` and `MultilinearPcsVerifier::verify_untrusted_evaluation` is precisely that the former assumes that the evaluation claim is fixed by the challenger, for example as described in the key investigation points for this audit:

> For context, in `slop`, there's `verify_trusted_evaluations` and `verify_untrusted_evaluations`, which is basically whether or not if you need to observe all the evaluation claims before proceeding further.

Thus, in principle, the soundness assumptions of the underlying PCS might be broken for the case in which `sumcheck_proof.point_and_eval.1` and `jagged_eval` are zero (and such cases can

occur, as described above). In practice, we considered the configuration of `JaggedPcsVerifier` with respect to the `JaggedBasefoldConfig` implementation of `JaggedConfig`. In this case, the underlying PCS verifier being called is the `StackedPcsVerifier`, which does not require the assumption that the expected evaluation has been fixed by the challenger. For other PCS verifiers, it is also possible that not observing `expected_eval` is not a problem, even if they in general require it for soundness. If the only guarantee that a malicious prover can break due to `expected_eval` not having been fixed is `expected_eval` itself, then there is no impact in the particular situation that arises in the jagged PCS verifier, as any opening value would have been valid (it gets multiplied by zero in the crucial equation anyway). So the only potential case of concern would be an underlying PCS verifier where soundness of other guarantees, such as that the commitment was valid and of a certain shape, relies on `expected_eval` to have been fixed by the challenger.

To support generic `MultilinearPcsVerifier`, the challenger now observes `expected_eval` before the call to `self.pcs_verifier.verify_trusted_evaluation`. Should only specific implementations of `MultilinearPcsVerifier` be intended to be supported, this may be skipped should each of them be safe in the context described above.

3.12. Some shape information only fixed modulo prime in jagged PCS commitment

Target	slop_jagged		
Category	Soundness	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Note: This finding discusses an issue impacting how binding commitments are for the jagged PCS verifier that stems from usage of `GC::F::from_canonical_usize`. For a related finding allowing a malicious prover to cause panics in the verifier if compiled in debug mode, see Finding 3.17. ↗. For related issues in the Merkle-tree verifier, see Findings 3.18. ↗ and 3.24. ↗.

The jagged PCS verifier contains the following snippet:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &JaggedPcsProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    let JaggedPcsProof {
        pcs_proof,
        sumcheck_proof,
        jagged_eval_proof,
        params,
        row_counts_and_column_counts,
        merkle_tree_commitments: original_commitments,
    } = proof;

    let (row_counts, column_counts): (Vec<Vec<usize>>, Vec<Vec<usize>>) =
        row_counts_and_column_counts.iter().map(|r_c|
    r_c.clone().into_iter().unzip()).unzip();

    // ...

    for (round_column_counts, round_row_counts, modified_commitment,
```

```
    original_commitment) in izip!(
        column_counts.iter(),
        row_counts.iter(),
        commitments.iter(),
        original_commitments.iter()
    ) {
        let (hasher, compressor) = GC::default_hasher_and_compressor();

        let hash = hasher.hash_iter(

    once(GC::F::from_canonical_usize(round_column_counts.len())).chain(

    round_row_counts.clone().into_iter().map(GC::F::from_canonical_usize).chain(

    round_column_counts.clone().into_iter().map(GC::F::from_canonical_usize),
                ),
            ),
        );
        let expected_commitment = compressor.compress([*original_commitment,
    hash]);

        if expected_commitment != *modified_commitment {
            return Err(JaggedPcsVerifierError::IncorrectTableSizes);
        }
    }


    // ...
}
```

For relevant fields, `GC::F::from_canonical_usize` tends to be implemented to panic in debug mode if the argument is not a standard representative (standard representatives are nonnegative integers that are smaller than the modulus). In release mode, this check is omitted, and the value is reduced modulo the fields characteristic p.

The above snippet is intended to make the prover bound to the lengths of `column_counts[i]` and `column_counts[i]` as well as their values. The wrap modulo p implies that for any of those values, the check only binds the prover modulo p, so that one opening might use k and another $k + c \cdot p$ for some integer c.

Impact

The column and row counts are subject to several other checks, so we first discuss what flexibility remains to the malicious prover.

We begin by noting that `row_counts` and `column_counts` have the same shapes by construction,

and thus `round_column_counts` and `round_row_counts` have the same lengths. The preimage of `hash` then has length `1 + 2*round_column_counts.len()`. If the hash function used is a cryptographically secure hash function on inputs of variable length, then this means that the hash value commits to the length `1 + 2*round_column_counts.len()` and hence to `round_column_counts.len()`. Incorporating `round_column_counts.len()` into the preimage directly is thus not directly needed, and hence the wraparound in the first field element of the preimage does not introduce an issue.

Below we consider multiple different cases for column or row counts being larger than or equal to p. One distinction we must make is whether we are considering one of the last two row/column counts for each round or not. This is because the last two row/column counts are for the padding columns and are thus handled and interpreted differently. The following table gives an overview, where we consider $(h, c) =$ `row_counts_and_column_counts[i][j]` for some i and j. The perspective here is that the correct opening of the commitment would involve only nonnegative integers smaller than p, so we consider cases where at least one of h and c is bigger than or equal to p and investigate whether this is caught by other checks the verifier performs.

Case	Padding	Condition on h and c	Outcome
A	No	$max(c, h) \geq p$ and $min(c, h) > 0$	Rejected by verifier
B	No	$h \geq p$ and $c = 0$	Rejected by verifier
C	No	$h = 0$ and $c \geq p$	Can be opened in more than one way but proof is large
D	Yes	$h \geq p$ or $c \geq p$	Rejected by verifier

Case A: No padding, both h and c are nonzero, and at least one is bigger than or equal to p

In this case, `round_areas` will have an entry that is bigger than or equal to $h \cdot c \geq p$:

```
let round_areas: Vec<usize> = row_counts
    .iter()
    .zip(column_counts.iter())
    .map(|(rc, cc)| {
        // The counts have been checked above to be at least length 2, so it's
    safe to subtract 2.
        let rc_len = rc.len() - 2;
        let cc_len = cc.len() - 2;
        rc.iter()
            .take(rc_len)
```

```
            .zip_eq(cc.iter().take(cc_len))
            .map(|(r, c)| r.saturating_mul(*c))
            .fold(0usize, |a, b| a.saturating_add(b))
    })
    .collect();
```

We are assuming that the underlying prime field has characteristic $p > 2^{30}$. Thus, the verifier will return an error on this check that limits the area for each round:

```
if round_areas.iter().any(|&area| area == 0 || area >= (1 << 30)) {
    return Err(JaggedPcsVerifierError::AreaOutOfBounds);
}
```

Case B: No padding, $h \geq p$ and $c = 0$

The round areas that were just discussed will not distinguish between h and $h \,\%\, p$, due to $c = 0$.

However, the following checks will result in a rejection of the proof:

```
if proof_added_columns
    != expected_added_vals_and_cols.iter().map(|(_,
    cols)| *cols).collect::<Vec<_>>()
    || last_cols.iter().any(|&x| x != 1)
    || added_rows_uniform.iter().any(|&x| x != 1 << self.max_log_row_count)
    ||
    added_rows_final.iter().zip_eq(expected_added_vals_and_cols.iter()).any(
        |(&x, &expected)| {
            x != expected.0 - (expected.1 - 1) * (1 << self.max_log_row_count)
                || row_counts
                    .iter()
                    .any(|rc| rc.iter().any(|&r| r > 1 <<
    self.max_log_row_count))
        },
    )
{
    return Err(JaggedPcsVerifierError::IncorrectShape);
}
```

Here, we assume that $2^{\texttt{self.max\_log\_row\_count}} < p$, which causes the check `r > 1 << self.max_log_row_count` to return an error for $h \geq p$.

Case C: No padding, $h = 0$ and $c \geq p$

Again, the area checks will not see this large count.

However, this time, there is no check on a bound for the column counts. Converting the row and column counts to the list of column heights does then insert a large number of additional columns of height zero:

```
let usize_column_heights: Vec<usize> = row_counts
.iter()
.zip_eq(column_counts.iter())
.flat_map(|(rc, cc)| {
    rc.iter().zip_eq(cc.iter()).flat_map(|(r, c)| std::iter::repeat_n(*r, *c))
})
.collect();
```

If we compare to the proof that uses $c' = c \,\%\, p$ (with $c = c' + k \cdot p$) instead of c, then it will be required to adjust the evaluation claims by inserting $k \cdot p$–many zeros and to also adjust the prefix sums by duplicating a value an additional $k \cdot p$ times. There does not appear to be a check that rules out large lengths for the evaluation claims and prefix sums.

Note that while the attacker cannot reuse the proof for the sumcheck and underlying PCS, the malicious prover can compute a fresh one for this second opening, as the underlying dense representation does not change when inserting columns of height zero into the sparse one.

As a limitation, the proof will have substantial size. The prefix heights `proof.params.col_prefix_sums` will need to have length bigger than p (which we assume satisfies $p > 2^{30}$). Each entry can consist of up to 30 field elements, depending on the total area committed to. If each entry consists of m field elements, the proof will need to be at least $4 \cdot m \cdot 2^{30}$ bytes, so $2 \cdot m$ GiB, large.

Case D: With padding

Now we consider the case of padding columns.

These are actually checked to occur in number and heights in a manner deterministically computed from the unpadded area for each round. This means that even if an overflow should happen, the commitment is still binding to the sole intended value.

However, this opens another question of whether there is a completeness issue in debug mode, should those deterministic values ever be larger than or equal to p.

For the column heights, this is not the case, as they are at most `1 << self.max_log_row_count`, which we assume is smaller than p. The maximum count occurring will be `(1 << C::log_stacking_height(&self.pcs_verifier)) / (1 << self.max_log_row_count)`. We similarly assume that `1 << C::log_stacking_height(&self.pcs_verifier)` is smaller than p, in which case the counts will also be in range.

Recommendations

Before calling `GC::F::from_canonical_usize`, the verifier should check that the values are smaller than p and return an error if they are not.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 7389c219 ↗.

3.13. Challenger `DuplexChallenger` and `MultiField32Challenger` by Plonky3 are not usable for Fiat–Shamir with variable-length prover messages

Target	sp1_hypercube, slop_jagged, slop_basefold		
Category	Soundness	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

Many of the verifiers in scope for this audit are implemented in a manner that can be conceptualized by considering a public-coin interactive protocol to which the Fiat–Shamir transform has been applied, with prover messages being observed by a challenger and verifier messages being replaced by sampling from the challenger.

While the implementation and configuration of the challenger was not in scope for this audit, files such as slop/crates/baby-bear/src/baby_bear_poseidon2.rs suggest that the Plonky3 `DuplexChallenger` is used as a challenger for the shard verifier. This raises the question of whether the `DuplexChallenger` is used correctly by the verifiers in our scope.

This challenger appears to be using the construction described in the paper *A Fiat–Shamir Transformation From Duplex Sponges* ↗ by Alessandro Chiesa and Michele Orrù. However, this paper appears to only analyze security in a setting in which prover messages have a fixed length.[1] The shard verifier (and various other in-scope verifiers that are components of it) are, however, not obtained from an interactive protocol where prover messages have fixed length. We are not aware of any analysis of the security of the `DuplexChallenger` for such a use case.

Indeed, challenges of the `DuplexChallenger` do not bind the prover to the earlier transcript if variable-length messages are allowed (i.e., there exist easily obtainable pairs of different transcripts that end in the same challenge). This is described below. We also explain below how direct usage of this challenger in a Fiat–Shamir transformation of an interactive protocol with variable-length messages can indeed in principle result in unsound noninteractive protocols.

Fiat–Shamir for variable-length messages

Fiat–Shamir is used to transform interactive protocols to noninteractive protocols by replacing messages from the verifier with some value that is deterministically computed from the prover messages so far. This replacement must have the property that these values change randomly if any part of what "happened so far" in the interactive version is changed. What "happened so far" is

[1] This is in the sense that the protocol prescribes the length; the messages in different rounds can still have different lengths.

the transcript of messages sent by the prover and verifier. Note that this crucially includes what was sent in which round. It is thus not enough for the challenges to fix the concatenation of prover messages. For example, it must make a difference whether the prover first sent values `[1,2,3]`, then the verifier sent a challenge, and then the prover sent `[4,5]` or whether the prover just sent `[1,2]` first and then `[3,4,5]` after the challenge by the verifier. These two are different transcripts, and thus, for usual soundness analysis of Fiat–Shamir to be applicable, a challenge generated afterwards must reflect this difference.

Note that this is related but not the same for the situation in which the prover in a single message sends a tuple of variable-length components. In this case, a potential vulnerability arising from nondeterministic parsing would arise already in the interactive version, so this does not relate to Fiat–Shamir. For example, if the prover may send both `([1,2,3], [4,5])` or `([1,2], [3,4,5])`, then an interactive protocol in which this is operationalized by the prover just sending `[1,2,3,4,5]` — resolving how to parse this as a tuple of two lists (by sending the lengths) only in a later message — may be vulnerable. In an actually implemented interactive version, it would be unusual to just send a flat list first and only later get the lengths needed to parse it. So in practice, such a vulnerability may occur more likely in implemented noninteractive versions in which the proof consists of structured data, like a pair of vectors, but then the verifier only observes the flattened concatenation of field elements. This is a potential issue to watch out for, where it in many cases might be required for soundness to also observe the shapes with the data.

In contrast, in the interactive version of protocols, if the prover sends two messages but there is a message by the verifier in between, then there is no flexibility regarding which part of the combined prover messages will later be considered as being part of the first message and how long that message was. Applying Fiat–Shamir to public-coin interactive protocols by observing prover messages with a challenger and replacing verifier messages by sampling with the challenger requires the challenger implementation to preserve this property.

To make this more concrete, we consider the protocol to alternate between the prover sending messages `m[i]` of length `l[i]` and the verifier sampling a single challenge (sampling multiple challenges corresponds to dummy rounds where the prover needs to send messages of length `l[i] = 0`). Consider the following pseudocode utilizing the type of challenger used in applying the Fiat–Shamir transform.

```
challenger_1 = a fresh challenger
for i in 0..n_1:
    challenger_1.observe(m_1[i])
    a_1[i] = challenger_1.sample()

challenger_2 = a fresh challenger
for i in 0..n_2:
    challenger_2.observe(m_2[i])
    a_2[i] = challenger_2.sample()
```

What we must require is that it is infeasible for an attacker to produce such `m_1[i]` and `m_2[i]` so that there is `0 <= i < min(n_1,n_2)` with `m_1[i] != m_2[i]`, yet `a_1[n_1-1] = a_2[n_2-1]`. Additionally, it must be infeasible to produce such `m_1[i]` and `m_2[i]` so that `n_1 != n_2`, yet

`a_1[n_1-1] = a_2[n_2-1]`.

However, the Plonky3 `DuplexChallenger` challenger as well as the `MultiField32Challenger` challenger do not have this property and are thus not usable in general for Fiat–Shamir for protocols with variable-length prover messages. They do have such a property if we restrict to cases where `len(m_1[i]) = len(m_2[i])` for all `i`. So these challengers are usable for protocols that have fixed message lengths.

For the remainder of this discussion, we will only focus on the case of `DuplexChallenger`.

The `DuplexChallenger`

The `DuplexChallenger` follows the overwriting-duplex method for sponge-construction hashes, as seen in the following snippets from the implementation:

```
fn observe(&mut self, value: F) {
    // Any buffered output is now invalid.
    self.output_buffer.clear();

    self.input_buffer.push(value);

    if self.input_buffer.len() == RATE {
        self.duplexing();
    }
}

fn sample(&mut self) -> EF {
    EF::from_base_fn(|_| {
        // If we have buffered inputs, we must perform a duplexing so that the
    challenge will
        // reflect them. Or if we've run out of outputs, we must perform a
    duplexing to get more.
        if !self.input_buffer.is_empty() || self.output_buffer.is_empty() {
            self.duplexing();
        }

        self.output_buffer
            .pop()
            .expect("Output buffer should be non-empty")
    })
}

fn duplexing(&mut self) {
    assert!(self.input_buffer.len() <= RATE);

    // Overwrite the first r elements with the inputs.
```

```
    for (i, val) in self.input_buffer.drain(..).enumerate() {
        self.sponge_state[i] = val;
    }

    // Apply the permutation.
    self.permutation.permute_mut(&mut self.sponge_state);

    self.output_buffer.clear();
    self.output_buffer.extend(&self.sponge_state[0..RATE]);
}
```

We can describe this as follows, considering only the case in which every message is nonempty and the verifier only sends a single field element on their turn.

First, `m[0]` is split into one or more blocks, with all but potentially the last of size `RATE`, with the last block potentially smaller (but not size zero). Let us call the blocks `b[0], ..., b[k]`. Then for `j` in `0,...,k-1`, the first `len(b[j])` elements of the state are overwritten by `b[j]`, followed by applying the permutation. The element `a[0]` is then obtained as the first element of the current state. The analogous procedure then continues with `m[1], m[2], ...` to sample `a[1], a[2], ...`.

Below we describe three ways in which this fails the required property described above.

Defect 1: Overwriting may not change the state

Consider `m[0] = b[0] || ... || b[k]` and `m'[0] = b[0] || ... || b[k-1] || b'[k]`. So we consider two messages that only (potentially) differ in the last block. The state is thus the same before overwriting with `b[k]` or `b'[k]`. Let us assume that `b'[k] = b[k] || c`, and the state before overwriting with `b[k]` or `b'[k]` is `s`. Then the new states can only differ potentially at indexes `j` in `len(b[k]), ..., len(b[k])+len(c)-1`, where overwriting with `b[k]` will yield `s[j]` and overwriting with `b'[k]` will yield `c[j-len(b[k])]`. The new states will thus be identical if `c` is chosen such that `c[j] = s[j+len(b[k])]`. Note that a malicious prover does have access to the challenger state when they decide on what to send for `c`, so it is possible in practice to construct messages `m[0]` and `m'[0]` so that the challenges `a[0]` and `a'[0]` agree.

This means an attacker can already know the later challenge `a[0]` by the time they must decide whether they include `m[0]` or `m'[0]` as the first message into the proof.

Defect 2: Future challenges may not depend on separation of prover data into messages

Consider the following sequence of operations:

1. Observe `n*RATE` field elements as `m[0]` (where the prover can control `n`).
2. Sample one challenge `a[0]`.
3. Observe `m*RATE` field elements as `m[1]` (where the prover can control `m`).
4. Sample one challenge `a[1]`.

Then `a[1]` will only depend on `m[0] || m[1]`, not on where `a[0]` was sampled, as long as `n, m > 0`.

This means an attacker can first choose this concatenation, already obtain `a[1]`, and then they still have multiple options available for how to split this as a concatenation, determining `a[0]`.

Defect 3: Transcripts with different numbers of prover messages may collide

Suppose we have two separate transcripts. It possible to use variable-length prover messages to match different challenges across the two transcripts when they have different number of rounds. Consider for example the first transcript having `m[0]` of length `1`, sampling `a[0]`, then having another prover message `m[1]` of length `1` before sampling `a[1]`. Now, on the second transcript, the prover may use `m'[0]` that consists of `m[0]`, concatenated with the not-overwritten `RATE` part of the state after the first observation, concatenated with `m[1]`. Then `a'[0]` sampled afterwards will be equal to `a[1]`.

This is another violation of the principle that the challenge should fix the preceding transcript; this is a computationally feasible way to obtain two transcripts that in this case have both different messages and a different number of messages but with the final challenges equal.

An attacker can thus compute `a'[0] = a[1]` and still decide whether they would like to send a proof where this value occurs as the first or second challenge.

Impact

All three variants described above show that the required property of sampled challenges fixing (for feasible amount of compute by the attacker) the previous transcript does not hold for `DuplexChallenger`, breaking usual soundness analysis of the Fiat–Shamir transform. The case of `MultiField32Challenger` is similar.

This means that `DuplexChallenger` and `MultiField32Challenger` are not actually usable for applying Fiat–Shamir to protocols in which the prover can send variable-length data.

To demonstrate that there are sound interactive protocols where directly applying these challengers to obtain a noninteractive version introduces a soundness problem, consider the following example. We consider a toy interactive subprotocol that replaces just sampling a challenge for the higher-level protocol by the following:

1. The prover sends `n*RATE` field elements as `m[0]` (where the prover controls `n`).
2. The verifier samples one challenge `alpha`.
3. The prover sends `m*RATE` field elements as `m[1]` (where the prover controls `m`).
4. The verifier samples one challenge `beta`.
5. The prover sends `k*RATE` field elements as `m[2]` (where the prover controls `k`).
6. The verifier uses `alpha + beta` in the higher-level protocol.

In the interactive setting, this is safe, as `alpha+beta` is still uniformly random if both `alpha` and `beta` were. And while having the prover send messages that are not used is wasteful, it does not change the fact that `alpha` and `beta` are uniformly random and independent.

Using a suitable challenger for the Fiat–Shamir transform, the resulting noninteractive protocol would also be safe. However, using `DuplexChallenger` or `MultiField32Challenger`, the resulting noninteractive protocol would be unsafe. The attacker can now choose `alpha` and `beta` out of a set of possibilities by changing how to write `n+m+k` as a sum of three summands, while already knowing all future challenges, as they only depend on the concatenation `m[0] || m[1] || m[2]`, which they keep fixed. Getting something particular for `alpha+beta` is then a k-list problem (see e.g., *A Generalized Birthday Problem* and *A Concrete Analysis of Wagner's k-List Algorithm over $\mathbb{Z}_p$*). This can be solved faster than normal brute forcing (getting easier if we allow even more summands than just two), breaking soundness.

In the shard verifier and other verifiers in scope, there are multiple spots where variable-length prover messages are observed. We are not aware of any concrete impacts, however. While incorrect usage of the challenger in the shard verifier may be a theoretical issue, it appears unlikely that this can be exploited usefully in practice. In many cases in the shard verifier, the lengths of variable-length prover messages are checked against other data later during verification, introducing various constraints and hurdles for a hypothetical attacker; and for an attack to be viable, there also must be a sufficiently approachable relationship between challenges where being able to change messages or earlier challenges in the described way without changing later challenges can be used in a manner to save the attacker compute.

Overall, we rate the likelihood that this issue can be used to help for exploits in practice as Low. However, as there is no established security proof showing that this usage of the challengers is sound, this amounts to a gap in the theoretical justification for this usage.

Recommendations

We recommend to use a challenger that is suitable for variable-length–message protocols or alternatively to adjust the protocols to be fixed-length. The latter could be done by replacing variable-length messages by fixed-length commitments, followed by opening them.

Remediation

This issue has been acknowledged by Succinct Inc.

Succinct Inc. suggested that current usage may also be sound with `DuplexChallenger`, based on the following two specific properties of the shard verifier:

1. Initial fixed-length commitments that are observed prior to any variable-length data commit the prover to a chip-cluster shape, and this commitment is opened without requiring further use of the challenger.
2. All variable-length observations have their lengths checked against the chip-cluster shape.

We can confirm that property 1 holds. While we have not checked property 2, we are not aware of any counterexample, and property 2 appears plausible to us. We also assess that it is plausible that

usage of `DuplexChallenger` in the Fiat–Shamir transform of an interactive protocol with prover messages of variable length, where the length is however constrained in the above manner, is sound, and we provided Succinct Inc. with more detailed thoughts about this point. However, a comprehensive assessment of the theoretical basis for usage of challengers like `DuplexChallenger` under these kinds of assumptions was beyond the scope of this audit.

3.14. Jagged verifier panics if the number of rounds is zero

Target	slop_jagged		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The jagged verifier `JaggedPcsVerifier::verify_trusted_evaluations` contains the following snippet:

```
// slop/crates/jagged/src/verifier.rs
let mut usize_prefix_sums: Vec<usize> = usize_column_heights
    .iter()
    .scan(0usize, |state, &x| {
        let result = *state;
        *state += x;
        Some(result)
    })
    .collect();

usize_prefix_sums
    .push(*usize_prefix_sums.last().unwrap() +
    *usize_column_heights.last().unwrap());
```

If the number of rounds configured for the underlying PCS is zero, then a prover can (and must) submit a proof with zero rounds, which will result in `usize_column_heights` being empty. Consequently, `usize_prefix_sums` will be empty after the assignment in the snippet, causing `usize_prefix_sums.last().unwrap()` to panic.

Impact

Should the jagged PCS verifier be used in a context where the prover can influence the configured number of rounds, or the number of rounds happens to be zero for other reasons, a malicious prover can construct a proof that will cause the verifier to panic.

On the other hand, an honest prover will not be able to produce a valid proof, even though opening an empty list of polynomials at a point, producing an empty list of evaluations, is an edge case of a PCS like the jagged PCS that in itself has meaning, albeit one that is not useful and unlikely to occur in most protocols.

Thus, the particular way in which `usize_prefix_sums` is computed amounts to a completeness issue for legitimately proving openings with zero rounds and a denial-of-service vector with respect to malicious provers.

Note that the shard verifier uses two rounds, so this situation cannot occur through the shard verifier.

While hypothetical protocols are thinkable in which a PCS, such as the jagged PCS, might also be used with zero rounds, this is an unlikely edge case.

To demonstrate the panic, one may use the `test_koala_bear_jagged_basefold` test while modifying `test_jagged_basefold` as follows.

```
// slop/crates/jagged/src/basefold.rs
// #[tokio::test]
async fn test_jagged_basefold<
// ..
{
    // ..

    // Check a normal proof, this should work
    jagged_verifier
        .verify_trusted_evaluations(
            &commitments,
            eval_point,
            eval_point.clone(),
            &evaluation_claims,
            &proof,
            &mut challenger,
        )
        .unwrap();

    // New verifier with zero rounds:
    let jagged_verifier = JaggedPcsVerifier::<GC, JaggedBasefoldConfig<GC>>::
        new(
        log_blowup,
        log_stacking_height,
        max_log_row_count as usize,
        0,
    );
    let mut challenger = jagged_verifier.challenger();
    let mut proof = proof;
    proof.row_counts_and_column_counts = Rounds::new();
    proof.merkle_tree_commitments = Rounds::new();
```

```
    let result = jagged_verifier
        .verify_trusted_evaluations(
            &[],
            eval_point,
            &[],
            &proof,
            &mut challenger,
        );
    println!("{:?}", result);
    ()
```

This will reuse the legitimate proof with multiple rounds as a shortcut to creating a type-correct proof (the sumcheck proof and underlying PCS proof for example will never be used, but one must construct a proof of appropriate type), then modify it to pass all verification up to the panic.

Recommendations

We recommend to revise the `usize_prefix_sums` logic to also support the case of zero rounds. This can be done by changing the current way the last entry is appended to conditional on the vector being empty appending zero:

```
if usize_column_heights.len() == 0 {
    usize_prefix_sums.push(0);
}
else {
   usize_prefix_sums
       .push(*usize_prefix_sums.last().unwrap() +
    *usize_column_heights.last().unwrap());
}
```

Alternatively, add a check to reject proofs/configurations with zero rounds.

If supporting the case of zero rounds is intended, then note that currently (as of the time of writing), the last step in the verification, the call to the verifier of the underlying dense PCS, will error on zero rounds, at least in the case of the stacked PCS, due to the following check:

```
if point.dimension() < self.log_stacking_height as usize {
    return Err(StackedVerifierError::IncorrectShape);
}
```

So to fully support zero rounds, a workaround would be needed for the invocation of the stacked verifier, such as special-casing the case of zero rounds in the stacked verifier.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 3746fed7 ↗. The commit removes support for zero rounds and returns an error in the jagged PCS verifier before `usize_prefix_sums.last().unwrap()` is invoked should `usize_prefix_sums` be empty.

3.15. Tensor shape: Undocumented and unchecked assumptions

Target	slop_multilinear		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The crate `slop_multilinear` uses tensors to represent different objects related to multlinear algebra, for example points and multilinear polynomials. While the `Tensor` struct can be used to represent tensors of different ranks, in each use case, there is usually exactly a specific rank that is suitable to represent the intended object.

The code in several places does not document and check the necessary rank assumptions. Note that there is a separate finding (Finding 3.16. ↗) related to additional size assumptions on these tensors. Additionally, while the code introduces the types `Mle` and `MleEval` to hide the chosen memory representation, we found multiple instances where this abstraction is broken and the code expects a specific memory representation.

Due to the lack of documentation, there is ambiguity about the intent of different interfaces. In the following, the affected points will be listed, and for one resolution of the ambiguity, the subsequent problems will be described. If the resolution of the ambiguity does not align with the intended use, the results of the subsequent analysis might not apply as written here.

Tensor representation of multilinear polynomials `Mle`

The struct `Mle<T, A: Backend>` is used to represent a batch of multlinear polynomials in a common number of variables. It uses a field `guts: Tensor<T,A>` to store field values that encode the tensor. There are several points in the code that communicate that the memory representation depends on the backend `A`, for example this comment:

```
// slop/crates/multlinear/src/mle.rs
/// The tensor must be in the correct shape for the given backend.
#[inline]
pub const fn new(guts: Tensor<F, A>) -> Self {
    Self { guts }
}
```

In addition, there is an interface `MleBaseBackend` that has to be implemented to retrieve the number of polynomials and number of variables from the `guts`:

```
// slop/crates/multlinear/src/base.rs
pub trait MleBaseBackend<F: AbstractField>: Backend {
    /// Returns the number of polynomials in the batch.
    fn num_polynomials(guts: &Tensor<F, Self>) -> usize;

    /// Returns the number of variables in the polynomials.
    fn num_variables(guts: &Tensor<F, Self>) -> u32;
    // ...
}
```

Despite this backend dependency, there are backend-independent functions that assume a specific memory layout, for example `transpose`:

```
// slop/crates/multlinear/src/mle.rs
pub fn transpose(&self) -> Mle<F, A>
where
    F: AbstractField,
    A: TransposeBackend<F>,
{
    Mle::new(self.guts.clone().transpose())
}
```

This function will panic if `guts` is not a tensor of rank 2, and its effect on the represented multilinear polynomial will depend on the memory representation. Thus, switching the backend of the `Mle` might not only change the internal representation but also the behavior of the code using its backend-independent interface, which is unexpected.

Another example is `from_buffer`:

```
// slop/crates/multlinear/src/mle.rs
pub fn from_buffer(buffer: Buffer<F, A>) -> Self
where
    F: AbstractField,
    A: MleBaseBackend<F>,
{
    // First, we need to convert the buffer into an arbitrary 2 dimensional
    tensor.
    let size = buffer.len();
    let mut tensor = Tensor::from(buffer).reshape([size, 1]);

    // Then, we need to convert the tensor into the correct shape, which is
    determined by the
    // backend.
    let dim_0 = A::num_polynomials(&tensor);
    let dim_1 = A::num_non_zero_entries(&tensor);
```

```
    tensor.reshape_in_place([dim_1, dim_0]);
    Self::new(tensor)
}
```

The reshaping at the end indicates that `[size, 1]` is not the correct shape for the tensor. But `num_polynomials` and `num_non_zero_entries` are called on the tensor in this shape; thus, these functions are expected to return sensible numbers even if their argument does not have a valid shape. This expectation is unexpected and is not apparent from their interface. Also, the implementation of `from_buffer` assumes that the valid layout for any backend is `[num_non_zero_entries, num_polynomials]`, breaking the abstraction.

Another example is the following segment in the logUp verifier:

```
// crates/hypercube/src/logup_gkr/verifier.rs
pub fn verify_logup_gkr(
    // ...
) -> Result<(), LogupGkrVerificationError<EF>> {
    // ...
    let expected_size = 1 << (number_of_interaction_variables + 1);
    if numerator.guts().dimensions.sizes() != [expected_size, 1]
        // ...
    {
        return Err(LogupGkrVerificationError::InvalidShape);
    }
    // ...
    let initial_number_of_variables = numerator.num_variables();
    if initial_number_of_variables != number_of_interaction_variables + 1 {
        return Err(LogupGkrVerificationError::InvalidFirstLayerDimension(
            initial_number_of_variables,
            number_of_interaction_variables + 1,
        ));
    }
```

Here, the shape of `numerator` is verified by direcly comparing it with `[expected_size, 1]` and by using `num_variables`. For `CpuBackend`, the second check is redundant. Because the first check already fixed the dimension vector, any backend deviating from the `CpuBackend` layout will make the second check fail, resulting in a completeness problem.

Tensor representation of evaluations of multilinear polynomials `MleEval`

The struct `MleEval` is used to represent the evaluation of a batch of multilinear polynomials `Mle` at a common point. Thus, if `Mle` represented `n` multilinear polynomials in a common number of variables, the result is an instance of `MleEval` with `n` values.

Similar to `Mle`, it is unclear in which way the backend is allowed to influence the memory

representation via the field `evaluations`. The following function is backend-independent:

```
// slop/crates/multilinear/src/mle.rs
//. It is expected that `self.evaluations.sizes()` is one of the three options:
/// `[1, num_polynomials]`, `[num_polynomials,1]`, or `[num_polynomials]`.
#[inline]
pub fn num_polynomials(&self) -> usize {
    self.evaluations.total_len()
}
```

It is unclear in which way the three shapes mentioned in the comment relate to the backend. Also, not all backend-independent functions allow all three mentions shapes, as documented in Finding 4.11. ↗.

Backend functions expect their arguments to have valid shape

The crate defines backend interfaces that can be implemented to define operations on `Mle` and `MleEval` instances. While their arguments are never typed as `Mle` or `MleEval` and use `Tensor` instead, they expect these arguments to be `guts` and `evaluations` of valid `Mle` and `MleEval` instances.

We found that the functions do not document or check the tensor shape assumptions on their arguments.

- The function interface `MleEvaluationBackend::eval_mle_at_point` expects its argument `mle: Tensor` to be the guts of an `Mle`.
- The function interface `MleEvaluationBackend::eval_mle_at_eq` expects its arguments `mle: Tensor` and `eq: Tensor` to be the guts of an `Mle`.
- The function interface `eval_mle_at_point_blocking` expects its argument `mle: Tensor` to be the guts of an `Mle`.
- The function interface `MleFoldBackend::fold_mle` expects its argument `guts: Tensor` to be the guts of an `Mle`.
- The function interface `MleFixLastVariableBackend::mle_fix_last_variable_constant_padding` expects its argument `mle: Tensor` to be the guts of an `Mle`.
- The function interface `MleFixLastVariableBackend::mle_fix_last_variable` expects its argument `mle: Tensor` to be the guts of an `Mle`.
- The function interface `MleFixLastVariableInPlaceBackend::mle_fix_last_variable_in_place` expects its argument `mle: Tensor` to be the guts of an `Mle`.
- The function interface `MleFixAtZeroBackend::fixed_at_zero` expects its argument `mle: Tensor` to be the guts of an `Mle`.

In all the cases above, the corresponding implementations for `CpuBackend` fail to check that the tensors have the required rank. Their behavior on tensors of unintended shape varies between

panics and unintended reshaping.

Functions return tensors that could be `Mle` or `MleEval`, but this is not documented:

- The function interface `MleEvaluationBackend::eval_mle_at_point` returns a tensor that could be an `MleEval`.
- The function interface `MleEvaluationBackend::eval_mle_at_eq` returns a tensor that could be an `MleEval`.
- The function interface `ZeroEvalBackend::zero_evaluations` returns a tensor that could be an `MleEval`.
- The function interface `MleFoldBackend::fold_mle` returns a tensor that could be an `MleEval`.
- The function interface `PartialLagrangeBackend::partial_lagrange` returns a tensor that could be an `Mle`.

Impact

Lack of clarity in documentation leads to mismatching expectations between the callers and implementation of a function.

The backend separation architecture invites users of the `slop_multilinear` crate as a library to implement their own backends with differing memory representations, without communicating the restrictions they must obey. Given the restrictions in the present code as described above, it is very likely that any memory representation differing from the representation of `CpuBackend` will break supposedly backend-independent code.

Recommendations

We recommend the following.

- Decide which details of the implementation of `Mle` and `MleEval` can depend on the backend and which are independent, and document the result.
- Do not allow creation of instances of `Mle` and `MleEval` with invalid `guts` and `evaluations`.
- Add checks that assert the expected assumptions on the tensor arguments.
- Use the types `Mle` and `MleEval` instead of `Tensor` when applicable.

Remediation

This issue has been acknowledged by Succinct Inc.

3.16. Tensor dimension sizes: Undocumented and unchecked assumptions

Target	slop_multilinear		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

Several functions in the crate `slop_multilinear` have a clear intention when operating on arguments that satisfy size-matching conditions, for example a function in a certain number of variables evaluated on a point of the same dimension. If these conditions are not met, the application of these functions may either not be intended at all or one of the arguments will be padded or truncated.

In several cases, the functions either do not document and check the natural assumptions, or they lack documentation about the padding. In the following, we list the assumptions that are missing checks, documentation, or both. If the actual function's behavior outside of these assumptions is intended, the specific padding or truncating behavior should be documented.

- The function interface `MleEvaluationBackend::eval_mle_at_point(mle: &Tensor<F, Self>, point: &Point<EF, Self>)` assumes that `num_variables(mle) == point.dimension()` without documenting it.
- The function interface `MleEvaluationBackend::eval_mle_at_eq(mle: &Tensor<F, Self>, eq: &Tensor<EF, Self>)` assumes that `num_variables(mle) == num_variables(eq)` and `num_polynomials(eq) == 1` without documenting it.
- The function implementation `CpuBackend::eval_mle_at_point(mle: &Tensor<F, Self>, point: &Point<EF, Self>)` assumes that `num_variables(mle) == point.dimension()` without checking it.
- The function implementation `CpuBackend::eval_mle_at_eq(mle: &Tensor<F, Self>, eq: &Tensor<EF, Self>)` assumes that `num_variables(mle) == num_variables(eq)` and `num_polynomials(eq) == 1` without checking it.
- The function implementation `eval_mle_at_point_blocking_with_basis(mle: &Tensor<F, CpuBackend>, point: &Point<EF, CpuBackend>, basis: Basis)` assumes that `num_variables(mle) == point.dimension()` without documenting and checking it.
- The function interface `MleFoldBackend::fold_mle(guts: &Tensor<F, Self>, beta: F)` assumes that `num_polynomials(guts) == 1` without documenting it.
- The function implementation `Mle::eval_at(&self, point: &Point<EF, A>)` assumes that `num_variables(self) == point.dimension()` without documenting and checking it.
- The function implementation `Mle::eval_at_eq(&self, eq: &Mle<EF, A>)` assumes that `num_variables(self) == num_variables(eq)` and `num_polynomials(eq) == 1`

without documenting and checking it.

- The function implementation `Mle::fold(&self, beta: F)` assumes that `num_polynomials(self) == 1` without documenting it.
- The function implementation `Mle::fixed_at_zero(&self, point: &Point<EF>)` assumes that `num_variables(self) == point.dimension() + 1` without documenting and checking it.
- The function implementation `Mle::blocking_eval_at(&self, point: &Point<E>)` assumes that `num_variables(self) == point.dimension()` without documenting and checking it.
- The function implementation `MleEval::add_evals(&self, other: Self)` assumes that `self.len() == other.len()` without documenting and checking it.
- The function interface `MleFixLastVariablesBackend::mle_fix_last_variable(mle: &Tensor<F, Self>, alpha: EF, padding_values: F)` assumes that `num_polynomials(mle) == padding_values.len()` without documenting it.
- The function implementation `CpuBackend::mle_fix_last_variable(mle: &Tensor<F, Self>, alpha: EF, padding_values: F)` assumes that `num_polynomials(mle) == padding_values.len()` without checking it.
- The function interface `MleFixedAtZeroBackend::fixed_at_zero(mle: &Tensor<F, Self>, point: &Point<EF>)` assumes that `num_variables(mle) == point.dimension() + 1` without documenting it.
- The function implementation `CpuBackend::fixed_at_zero(mle: &Tensor<F, Self>, point: &Point<EF>)` assumes that `num_variables(mle) == point.dimension() + 1` without checking it.

Impact

Lack of clarity in documentation leads to mismatching expectations between the callers and implementation of a function.

Several of the mentioned functions exhibit (possibly unintended) padding behavior; for example, `mle_fix_last_variable` will panic in a `rayon` thread if the argument `padding` is not long enough. If the calling function is not aware of this restriction, necessary checks might be left out, which could lead to verifier panics that can be triggered by incorrect proofs.

Silent truncation of arguments can lead to relevant prover input being ignored and unchecked. Depending on the usage in the verifier, this can lead to soundness errors that are preventable by adding the corresponding assertion checking the length matching of the arguments.

Recommendations

We recommend the following.

- Add documentation for the intended restrictions on the arguments.
- Add assertions checking the intended restrictions.
- Alter the calls that rely on implicit truncation or padding.

- Do not silently truncate any argument.
- Do not silently pad any argument (use `PaddedMle` if appropriate instead).

Remediation

This issue has been acknowledged by Succinct Inc.

3.17. Malicious proofs can trigger a panic in the jagged verifier compiled for debug mode

Target	slop_jagged		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

Note: This finding discusses a verifier panic in the jagged PCS verifier that stems from usage of `GC::F::from_canonical_usize`. For a related issue impacting how binding the commitment is, see Finding 3.12. ↗. For related issues in the Merkle-tree verifier, see Findings 3.18. ↗ and 3.24. ↗.

The jagged PCS verifier contains the following snippet:

```
// slop/crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &JaggedPcsProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    let JaggedPcsProof {
        pcs_proof,
        sumcheck_proof,
        jagged_eval_proof,
        params,
        row_counts_and_column_counts,
        merkle_tree_commitments: original_commitments,
    } = proof;

    let (row_counts, column_counts): (Vec<Vec<usize>>, Vec<Vec<usize>>) =
        row_counts_and_column_counts.iter().map(|r_c|
    r_c.clone().into_iter().unzip()).unzip();

    // ...
```

```
    for (round_column_counts, round_row_counts, modified_commitment,
    original_commitment) in izip!(
        column_counts.iter(),
        row_counts.iter(),
        commitments.iter(),
        original_commitments.iter()
    ) {
        let (hasher, compressor) = GC::default_hasher_and_compressor();

        let hash = hasher.hash_iter(

    once(GC::F::from_canonical_usize(round_column_counts.len())).chain(

    round_row_counts.clone().into_iter().map(GC::F::from_canonical_usize).chain(

    round_column_counts.clone().into_iter().map(GC::F::from_canonical_usize),
                ),
            ),
        );
        let expected_commitment = compressor.compress([*original_commitment,
    hash]);

        if expected_commitment != *modified_commitment {
            return Err(JaggedPcsVerifierError::IncorrectTableSizes);
        }
    }


    // ...
}
```

For relevant fields, `GC::F::from_canonical_usize` is implemented as follows:

```
#[inline]
fn from_canonical_usize(n: usize) -> Self {
    debug_assert!(n < P as usize);
    Self::from_canonical_u32(n as u32)
}
```

This will thus panic on the assert when compiled with debug mode and if the value is bigger than or equal to the prime modulus of the field.

In the context of the jagged verifier, the three calls to `GC::F::from_canonical_usize` all depend on data provided by the prover, without the verifier having checked that these values are less than the prime modulus of the field. It is thus possible for a malicious prover to send a proof that will cause a verifier panic.

Note that while, for `round_column_counts.len()`, this would require the prover to send a long vector (making the proof have substantial size), for the other two calls, this is very cheap for the prover, as only a single `usize` being transmitted needs to be changed from a legitimate to a too large value.

Impact

The prover can cause a panic in the jagged PCS verifier. This can be tested as follows.

Modify the `test_jagged_basefold` function:

```
// slop/crates/jagged/src/basefold.rs
// #[tokio::test]
async fn test_jagged_basefold<
// ..
{
    // ..
    use slop_algebra::Field;
    let mut proof = proof;
    let order_as_usize: usize = GC::F::order().try_into().unwrap();
    proof.row_counts_and_column_counts.rounds[0][0].0 += order_as_usize;

    jagged_verifier.verify_trusted_evaluations(
        &commitments,
        eval_point,
        &evaluation_claims,
        &proof,
        &mut challenger,
    );
```

This will then cause tests such as `test_koala_bear_jagged_basefold` to succeed in release mode but fail with a panic in debug mode, with the panic arising from the debug assert described above.

In the context of SP1, the shard verifier uses the jagged PCS verifier. This panic can also be reached by the malicious prover through the shard verifier. While the shard verifier does carry out some additional checks on `row_counts_and_column_counts` of the jagged PCS proof, those checks appear after the call to the jagged PCS verifier.

Recommendations

The lengths of `row_counts_and_column_counts[i]` and all contained `usize` values should be checked to be smaller than the characteristic of `GC::F`, with an error returned if they are not, before the above snippet with the calls to `GC::F::from_canonical_usize`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 7389c219 ↗.

3.18. Malicious proofs can trigger a panic in the Merkle-tree verifier compiled for debug mode

Target	slop_merkle_tree		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

> **Note:** This finding discusses a verifier panic in the Merkle-tree verifier that stems from usage of `GC::F::from_canonical_usize`. For a related issue impacting how binding the commitment is, see Finding 3.24. ↗. For related issues in the jagged PCS verifier, see Findings 3.17. ↗ and 3.12. ↗.

The Merkle-tree verifier contains the following snippet:

```
// slop/crates/merkle-tree/src/tcs.rs
pub fn verify_tensor_openings(
    &self,
    commit: &GC::Digest,
    indices: &[usize],
    opening: &Tensor<GC::F>,
    proof: &MerkleTreeTcsProof<GC::Digest>,
) -> Result<(), MerkleTreeTcsError> {
    // ...
    let hash = self.hasher.hash_slice(&[
        GC::F::from_canonical_usize(proof.log_tensor_height),
        GC::F::from_canonical_usize(proof.width),
    ]);
```

For relevant fields, `GC::F::from_canonical_usize` is implemented as follows:

```
#[inline]
fn from_canonical_usize(n: usize) -> Self {
    debug_assert!(n < P as usize);
    Self::from_canonical_u32(n as u32)
}
```

This will thus panic on the assert when compiled with debug mode and if the value is bigger than or

equal to the prime modulus of the field.

In the context of the Merkle-tree verifier, the calls to `GC::F::from_canonical_usize` all depend on data provided by the prover, without the verifier having checked that these values are less than the prime modulus of the field. It is thus possible for a malicious prover to send a proof that will cause a verifier panic.

Note that if `indices` is not empty, having one of the two values exceed the field size would require `opening` or `proof.paths` to have a huge size. As of the time of writing, a check that `indices` is not empty prevents the panic from occurring. However, this check that prevents the fields `proof.log_tensor_height` and `proof.width` from being inspected for empty `indices` is incorrect (see Finding 3.23. ↗).

Impact

If the incorrect early exit (Finding 3.23. ↗) is removed, a prover could send a proof with `proof.log_tensor_height` or `proof.width` being equal to or exceeding the fields characteristic, which would cause the verifier to panic in debug mode.

Recommendations

Check that `proof.log_tensor_height` and `proof.width` are smaller than the characteristic of `GC::F`. Return an error if they are not.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 7389c219 ↗.

3.19. Jagged verifier rejects rounds with zero area

Target	slop_jagged		
Category	Completeness	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The jagged verifier does not support round commitments with zero area.

Due to the following check, it rejects a proof in which any round has area zero:

```
// slop/crates/jagged/src/verifier.rs
if round_areas.iter().any(|&area| area == 0 || area >= (1 << 30)) {
    return Err(JaggedPcsVerifierError::AreaOutOfBounds);
}
```

While the code segment above makes it clear that this restriction is recognized in the implementation, the interface lacks documentation on that restriction, which makes it unexpected to someone using the jagged verifier.

In contrast to this, the upper bound on the area was explicitly asked for in the key investigation points (see section 5.1. ↗); thus, we assume this to be a desired restriction, which the soundness of the interaction verification in hypercube depends on.

Impact

The shard verifier uses the jagged verifier. In this case, it is used in two rounds, corresponding to the preprocessed and the main trace.

If all used chips in a shard proof do not have a preprocessed trace or all do not have a main trace, the failure of the jagged verifier to support zero round area means that no acceptable proof to the shard verifier can be produced.

Because all chips with a preprocessed trace have to be used by a shard proof, this can only happen if either there are no chips with preprocessed trace or the chips with preprocessed trace have no main trace. While both conditions seem unlikely, they are not explicitly ruled out in the shard verifier documentation. In some designs, it might be possible that a shard does not use any chip with preprocessed trace.

Recommendations

Document the assumption that the round areas cannot be zero for the jagged and shard verifiers. Ensure that the code using the shard verifier (not in scope) only uses it in cases where neither the preprocessed trace nor the main trace are empty.

Alternatively, allow the jagged verifier to support zero round area.

If all rounds have zero area (which is then possible), the evaluation of the jagged polynomial would always be zero. Therefore, it is necessary to remediate Finding 3.11. ↗ to avoid the zero division in the following segment:

```
let expected_eval = sumcheck_proof.point_and_eval.1 / jagged_eval;
```

Remediation

This issue has been acknowledged by Succinct Inc..

A comment on the area restriction was added in the implementation of the jagged verifier in 09cc6065 ↗ and 4ecf6f6d ↗.

3.20. Hardcoded chip-constraint degree in zerocheck

Target	sp1_hypercube		
Category	Completeness	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The zerocheck verifier performs the sumcheck as follows:

```
// crates/hypercube/src/verifier/shard.rs
partially_verify_sumcheck_proof(&proof.zerocheck_proof, challenger,
    max_log_row_count, 4)
    .map_err(|e| {
    ShardVerifierError::<
        _,
        <C::PcsVerifier as MultilinearPcsVerifier<GC>>::VerifierError,
    >::ConstraintsCheckFailed(e)
})?;
```

This hardcodes 4 as the maximum degree of the polynomial. The polynomial is a random linear combination of expressions of the form `zerocheck_eq_val * (constraint_eval + openings_batch)`. The term `zerocheck_eq_val` has degree one. The term `openings_batch` is a random linear combination of multilinear polynomial evaluations and thus has degree one.

The term `constraint_eval` is computed as follows:

```
let constraint_eval = Self::eval_constraints(chip, openings, alpha,
    public_values)
    - padded_row_adjustment * geq_val
```

The call to `eval_constraints` is a random linear combination of the evaluations of the constraint polynomials of the chip, Thus, its degree is the maximum degree of constraint polynomials of that chip. The `padded_row_adjustment` is a constant that does not depend on the trace, and `geq_val` has degree one.

Therefore, the degree of the polynomial in the sumcheck proof is `max(2, 1 + max_constraint_degree)`. Given that 4 is hardcoded, it needs to be ensured that the maximum constraint degree is at most 3.

The chip constructor `Chip::new` contains the following code segment:

```
// crates/hypercube/src/chip.rs
let mut max_constraint_degree =
    get_max_constraint_degree(&air, air.preprocessed_width(),
    PROOF_MAX_NUM_PVS);

if !sends.is_empty() || !receives.is_empty() {
    max_constraint_degree = std::cmp::max(max_constraint_degree,
    MAX_CONSTRAINT_DEGREE);
}
assert!(max_constraint_degree > 0);
let max_constraint_degree = 3;
let log_quotient_degree = log2_ceil_usize(max_constraint_degree - 1);
```

This does not ensure that the constraint polynomial degree is bounded by 3. Instead, what happens is that the maximum constraint degree is computed. Then the max with `MAX_CONSTRAINT_DEGREE = 3` is taken if there are interactions.[2]

But then, instead of checking that `max_constraint_degree` is at most 3, it is just checked that this is positive and then reset to the value 3 with no justification.

Impact

The verifier only supports chips whose constraints are polynomials of degree at most 3, but the chips' constructor does not enforce this restriction. The verifier will reject proofs for higher constraint degrees.

Recommendations

Assert that chips actually have constraints whose degree is bounded by 3.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 53da5ad3 ↗.

[2] The reason for this is not apparent from the code segment and may be deprecated code left over from the previous logUp argument architecture.

3.21. Function `VirtualGeq::eval_at_usize` is incorrect for `index == (1 << self.num_vars)`

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `virtual_geq::VirtualGeq<F>::eval_at_usize` is expected to return the linear combination of the boolean functions `index == self.threshold` and `index >= self.threshold` (interpreted as functions with values $\{0, 1\}$) with coefficients `self.eq_coefficient` and `self.geq_coefficient`.

```
// slop/crates/multilinear/src/virtual_geq.rs
// ..
impl<F: Field> VirtualGeq<F> {
    // ..
    pub fn eval_at_usize(&self, index: usize) -> F {
        assert!(index <= (1 << self.num_vars));
        if index < self.threshold as usize {
            F::zero()
        } else if index == self.threshold as usize {
            self.eq_coefficient + self.geq_coefficient
        } else if index < (1 << self.num_vars) {
            self.geq_coefficient
        } else {
            F::zero()
        }
    }
```

The documentation of the type `VirtualGeq` explains this behavior for the values `index < (1 << self.num_vars)`. The type is documented to represent a linear combination of the `eq` and `geq` polynomials, which are the bitwise multilinear polynomial extensions of the respective boolean functions.

```
// crates/multilinear/src/virtual_geq.rs
// ..
/// A struct capturing a dense representation of a linear combination of a geq
    and eq polynomial,
/// both with the same threshold and number of variables.
```

```
///
/// In terms of "guts", a `VirtualGeq` is a
/// vector of length `2^num_vars` where the first `threshold` entries are zero,
    the next entry is
/// `eq_coefficient + geq_coefficient`, and the rest are `geq_coefficient`.
    (In the edge case
/// threshold == 2^num_vars, this means the vector consists of all zeroes.)
#[derive(Debug, Copy, Clone)]
pub struct VirtualGeq<F> {
    pub threshold: u32,
    pub geq_coefficient: F,
    pub eq_coefficient: F,
    pub num_vars: u32,
}
```

While the above documentation comments only explicitly reference the range of integers `0 <= index < (1 << self.num_vars)`, the `eval_at_usize` function only asserts that `index` satisfies `index <= (1 << self.num_vars)`, thus suggesting that `index = (1 << self.num_vars)` is also a supported argument.

The return value is incoherent for `index == (1 << self.num_vars)`, depending on whether `self.threshold == (1 << self.num_vars)`:

- If `self.threshold == (1 << self.num_vars)`, the function returns `self.eq_coefficient + self.geq_coefficient`, which is the linear combination of the boolean functions `index == self.threshold` and `index >= self.threshold` with the given coefficients.
- If `self.threshold < (1 << self.num_vars)`, the function returns `F::zero()`, which is *not* the linear combination of the boolean functions `index == self.threshold` and `index >= self.threshold` with the given coefficients.

Impact

The function is not used in the audited code.

The incoherent return value is not communicated in the documentation to users of the slop library and thus could break code relying on one of the alternatives being consistently returned.

Recommendations

Document the desired behavior of the function in the case `index == (1 << self.num_vars)`, and change the implementation accordingly. If the function should be the linear combination of the boolean functions == and >= in all cases, change the implementation as follows:

```
// crates/multilinear/src/virtual_geq.rs
// ..
impl<F: Field> VirtualGeq<F> {
    // ..
    pub fn eval_at_usize(&self, index: usize) -> F {
        // ..
        } else if index < (1 << self.num_vars) {
        } else {
            // ..
        } else {
            F::zero()
        }
    }
```

Alternatively, change the assert such that `index == (1 << self.num_vars)` is not allowed.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 16280648 ↗.

3.22. Constructors for `Padded` do not check type assumptions

Target	slop_multilinear		
Category	Code Maturity	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `PaddedMle` struct is intended to represent a batch of multilinear polynomials whose guts are given elementwise by `inner` padded with additional values according to one of three padding patterns `Constant`, `Generic`, or `Zero`.

```
// crates/multilinear/src/padded.rs
pub enum Padding<F, A: Backend> {
    Constant((F, usize, A)),
    Generic(Arc<MleEval<F, A>>),
    Zero((usize, A)),
}
// ...
impl<F: Clone, A: Backend> Padding<F, A> {
    pub fn num_polynomials(&self) -> usize {
        match self {
            Padding::Constant((_, num_polynomials, _)) => *num_polynomials,
            Padding::Generic(ref eval) => eval.num_polynomials(),
            Padding::Zero((num_polynomials, _)) => *num_polynomials,
        }
    }
}
// ...
pub struct PaddedMle<T, A: Backend = CpuBackend> {
    inner: Option<Arc<Mle<T, A>>>,
    padding_values: Padding<T, A>,
    num_variables: u32,
}
```

The `Constant(value,num_polynomials,backend)` and `Zero(num_polynomials,backend)` paddings are intended to pad an `inner` having `num_polynomials` as a number of polynomials. The `Generic(values)` padding is intended to pad an `inner` such that the number of polynomials in `inner` and `values` agree. Note that in all cases, there is a condition requiring the number of polynomials of the padding and `inner` to match for `PaddedMle` to be valid.

There are two constructors for `PaddedMle`:

```
impl<T: AbstractField, A: MleBaseBackend<T>> PaddedMle<T, A> {
    #[inline]
    pub const fn new(
        inner: Option<Arc<Mle<T, A>>>,
        num_variables: u32,
        padding_values: Padding<T, A>,
    ) -> Self {
        Self { inner, num_variables, padding_values }
    }

    pub fn padded(inner: Arc<Mle<T, A>>, num_variables: u32, padding_values:
    Padding<T, A>) -> Self
    where
        A: MleBaseBackend<T>,
    {
        assert!(inner.num_non_zero_entries() <= 1 << num_variables);
        match padding_values {
            Padding::Generic(ref p) => {
                assert!(p.num_polynomials() == inner.num_polynomials());
            }
            Padding::Constant(_) => {}
            Padding::Zero(_) => {}
        }
        Self { inner: Some(inner), num_variables, padding_values }
    }
```

The `new` constructor does not check the number of polynomials to agree. The `padded` constructor only checks the condition for one of the three padding patterns.

Impact

The constructors can be used to instantiate invalid instances of `PaddedMle`. The return values of the methods of the instantiated `PaddedMle` are then inconsistent; for example, the method `PaddedMle::num_polynomials` returns the number of polynomials of the padding, while the method `PaddedMle::eval_at_eq` returns as many values as the number of polynomials of `inner`.

Recommendations

Add checks for the number of polynomials in all cases.

Remediation

This issue has been acknowledged by Succinct Inc., and a partial fix was implemented in commit 5172693b ↗. This commit adds the missing checks to the function `padded`.

3.23. Function `MerkleTreeTcs::verify_tensor_openings` does not verify `proof.width` and `proof.log_tensor_height` if `indices` is empty

Target	slop_merkle_tree		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

From the interface of the function `MerkleTreeTcs::verify_tensor_openings`, it is not clear whether the contents of proof are part of the statement that is verified or if it is opaque data that is just meant to support the verification of the actual statement.

The way it is used in the basefold verifier makes clear that the `MerkleTreeTcs` verification is expected to guarantee the correctness of the fields of proof, as they are used and not treated as opaque:

```
// slop/crates/basefold/src/verifier.rs
if opening_and_proof.proof.log_tensor_height != log_len +
    self.fri_config.log_blowup() {
    return Err(BaseFoldVerifierError::IncorrectShape);
}
```

Thus, these fields should always be checked by `MerkleTreeTcs::verify_tensor_openings` to be correct. However, if `indices` is empty, this check is skipped, so `proof.width` and `proof.log_tensor_height` can differ from the committed data:

```
// slop/crates/merkle-tree/src/tcs.rs
if indices.is_empty() {
    return Ok(());
}
```

Note that this would be okay if the proof would not be treated as part of the statement, because then the statement to check would be empty if `indices` were empty.

Impact

The function is never called with empty indexes in basefold, so this behavior has no impact there.

For users of slop as a library, the behavior is inconsistent. Because the shape is always verified if

`indices` is not empty, users might rely on this behavior always. To check the shape without opening the commitment, calling the function with empty `indices` appears to be appropriate, but in this case the check would not be performed and users would rely on the unchecked shape to be correct.

Recommendations

Remove the early exit.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 60700be6 ↗.

3.24. Some shape information only fixed modulo prime in Merkle-tree commitment

Target	slop_jagged		
Category	Soundness	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Note: This finding discusses an issue impacting how binding commitments are for the Merkle-tree verifier that stems from usage of `GC::F::from_canonical_usize`. For a related issue allowing a malicious prover to cause panics in the verifier if compiled in debug mode, see Finding 3.18. For related issues in the jagged PCS verifier, see 3.17. and 3.12.

The Merkle-tree verifier contains the following snippet:

```
// slop/crates/merkle-tree/src/tcs.rs
pub fn verify_tensor_openings(
    &self,
    commit: &GC::Digest,
    indices: &[usize],
    opening: &Tensor<GC::F>,
    proof: &MerkleTreeTcsProof<GC::Digest>,
) -> Result<(), MerkleTreeTcsError> {
    // ...
    let hash = self.hasher.hash_slice(&[
        GC::F::from_canonical_usize(proof.log_tensor_height),
        GC::F::from_canonical_usize(proof.width),
    ]);
    let expected_commit = self.compressor.compress([proof.merkle_root, hash]);

    if expected_commit != *commit {
        return Err(MerkleTreeTcsError::InconsistentCommitmentShape);
    }
```

In release mode, the function `GC::F::from_canonical_usize` wraps at the fields characteristic. Thus, in this case, the values are only bound by the commitment up to a multiple of the fields characteristic.

Impact

The prover could open the same commitment with different values for `proof.log_tensor_height` and `log.width`.

When opened with empty `indices`, exceeding the characteristic would not require the prover to send actual data of that length. In this case, the calling function to the verifier could expect that `proof.log_tensor_height` and `log.width` are verified to match the values bound by the commitment, which is not the case.

Recommendations

Check that `proof.log_tensor_height` and `proof.width` are smaller than the characteristic of `GC::F`. Return an error if they are not.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 7389c219 ↗.

3.25. Undocumented panics in `ShardVerifier::shape_from_proof`

Target	sp1_hypercube		
Category	Code Maturity	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

The function `ShardVerifier::shape_from_proof` is part of the public interface of `ShardVerifier`. Its only argument is the shard proof `proof`. Depending on the proof and the type `C` the `ShardVerifier` is instantiated with, the function panics under the following circumstances:

- If `C` is `JaggedBasefoldConfig`, it panics if `proof.evaluation_proof.pcs_proof.batch_evaluations` has less than two elements.
- If `C` is `WhirBasefoldConfig`, it panics if `proof.evaluation_proof.pcs_proof.merkle_proof` is empty.
- If `C` is `WhirBasefoldConfig`, it panics if `proof.evaluation_proof.pcs_proof.merkle_proof[0]` has less than two elements.
- It panics if `proof.evaluation_proofs.row_counts_and_column_counts` has less than two elements.
- It panics if any element of `proof.evaluation_proofs.row_counts_and_column_counts` has less than two elements.
- In debug mode, it panics if `proof.shard_chips` contains a string that is not a name of a chip of the machine.
- In debug mode, it panics due to arithmetic overflow if for any `i` and `cc = proof.evaluation_proof.row_counts_and_column_counts[i]`, the value of `cc[cc.len() - 2].1` is the maximum `usize::MAX`.

These panics occur because of asserted conditions not being met, due to access of array elements without the necessary length checks or due to arithmetic overflows.

Impact

The function itself is not used in the audited code.

Recommendations

If the function is intended to be used in a verifier, either document the necessary checks the caller has to ensure before calling this function, or check the panic conditions in this function and return an error if they are met.

If this function is not intended to be used in a verifier, document the expected preconditions.

Remediation

This issue has been acknowledged by Succinct Inc.

Succinct Inc. added the following comment:

> The function is used only for proving.

3.26. Field `proof.shard_chips` in argument `proof` of `verify_shard` is redundant and can contain strings that are not chip names

Target	sp1_hypercube		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The proof field `proof.shard_chips` should contain the names of the chips used in the proof. These names must agree with the keys of `opened_values.chips`; thus, this field is redundant.

In the function `verify_shard`, the value of `proof.shard_chips` is used in a single place:

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
where
    A: for<'a> Air<VerifierConstraintFolder<'a, GC>>,
{
    // ...
    let shard_chips = self
        .machine
        .chips()
        .iter()
        .filter(|chip| shard_chips.contains(&chip.name()))
        .cloned()
        .collect::<BTreeSet<_>>();
```

In this place, only elements of `proof.shard_chips` that occur as a name of a chip are relevant; other entries are ignored. This means the verification can be successful even though `proof.shard_chips` contains strings that are not chip names, which may be unexpected.

Impact

Depending on the proof encoding, the redundant field can unnecessarily increase the proof size.

It is a common occurrence in the audited code that proof contents are not treated as opaque data for the verifier, but calling functions sometimes expect explicit proof fields to be verified. A caller might expect `verify_shard` to verify that the contents of `proof.shard_chips` are the names of the used chips, which it does not verify.

Recommendations

Remove the field and use the keys of `opened_values.chips` instead. Return an error if the keys contain strings that are not names of chips on the machine.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit bf7d5463 ↗.

3.27. Field `proof.opened_values[].local_cumulative_sum` is unused

Target	sp1_hypercube		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The argument `proof` of `verify_shard` contains a vector `proof.opened_values` of elements of type `ChipOpenedValues`. Each element has a field `local_cumulative_sum`, which is not used in `verify_shard`.

Impact

Depending on the proof encoding, the unused field can unnecessarily increase the proof size.

It is a common occurrence in the audited code that proof contents are not treated as opaque data for the verifier, but calling functions sometimes expect explicit proof fields to be verified. A caller might expect `verify_shard` to verify something about `proof.opened_values[].local_cumulative_sum`.

Recommendations

Remove the field from the proof.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit bf7d5463 ↗.

3.28. Unnecessary vector `zerocheck_eq_vals` in `verify_zerocheck`

Target	sp1_hypercube		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `verify_zerocheck`, the vector `zerocheck_eq_vals` is defined as follows:

```
// crates/hypercube/src/verifier/shard.rs
let zerocheck_eq_vals = vec![zerocheck_eq_val; shard_chips.len()];
```

The only place this is used is in the main loop directly below, where the components are accessed but not modified:

```
for ((chip, (_, openings)), zerocheck_eq_val) in

    shard_chips.iter().zip_eq(opened_values.chips.iter()).zip_eq(zerocheck_eq_vals)
{
    // ...
    rlc_eval = rlc_eval * lambda + zerocheck_eq_val * (constraint_eval +
    openings_batch);
}
```

The vector could thus be removed.

Impact

The code clarity suffers from the unnecessary `zerocheck_eq_vals`. Note that the local variable `zerocheck_eq_val` additionally shadows the outer local variable `zerocheck_eq_val` of the same name.

Recommendations

Remove `zerocheck_eq_vals`, and replace `zerocheck_eq_val` in the loop with the outer `zerocheck_eq_val` (which has the same value).

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 67dddbb6 ↗.

3.29. Unnecessarily and confusingly nested check in jagged verifier

Target	slop_jagged		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The jagged verifier has the following checks:

```
// slop/crates/jagged/src/verifier.rs
if proof_added_columns
    != expected_added_vals_and_cols.iter().map(|(_,
    cols)| *cols).collect::<Vec<_>>()
    || last_cols.iter().any(|&x| x != 1)
    || added_rows_uniform.iter().any(|&x| x != 1 << self.max_log_row_count)
    ||
    added_rows_final.iter().zip_eq(expected_added_vals_and_cols.iter()).any(
        |(&x, &expected)| {
            x != expected.0 - (expected.1 - 1) * (1 << self.max_log_row_count)
                || row_counts
                    .iter()
                    .any(|rc| rc.iter().any(|&r| r > 1 <<
    self.max_log_row_count))
        },
    )
{
    return Err(JaggedPcsVerifierError::IncorrectShape);
}
```

This finding is about the last condition of the outer disjunction:

```
added_rows_final.iter().zip_eq(expected_added_vals_and_cols.iter()).any(
    |(&x, &expected)| {
        x != expected.0 - (expected.1 - 1) * (1 << self.max_log_row_count)
            || row_counts
                .iter()
                .any(|rc| rc.iter().any(|&r| r > 1 << self.max_log_row_count))
    },
)
```

The inner condition here is a disjunction again, but the second condition (that all row counts are smaller than or equal to the maximum height) does not actually depend on what is being iterated over. Instead, this check should actually be performed independently of the outer loop.

Impact

Structuring the check like this is confusing. It raises the question of whether a bypass of this necessary check could happen if the outer iterator is empty. It turns out that this is not the case, because the outer iterator will only be empty if there are no rounds, in which case `row_counts` is also empty.

Also, this is inefficient if there is more than one round, as the same check will be performed multiple times.

Recommendations

Move this check one level up to the main outer disjunction.

Remediation

This issue has been acknowledged by Succinct Inc., and fixes were implemented in the following commits:

- f38b8c1f ↗

The part of the commit that fixes this finding is the following change:

```
|| added_rows_final.iter().zip_eq(expected_added_vals_and_cols.iter()).any(
    |(&x, &expected)| {
        x != expected.0 - (expected.1 - 1) * (1 << self.max_log_row_count)
            || row_counts
                .iter()

    .any(|rc| rc.iter().any(|&r| r > 1 << self.max_log_row_count))
    },
)
|| row_counts.iter().any(|rc| rc.iter().any(|&r| r > 1 << self.max_log_row_
    count))
```

3.30. Function `JaggedEvalSumcheckConfig::jagged_evaluation` uses `len()` for `Point` rather than `dimension()`

Target	slop_jagged		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Point` type has a `dimension()` member function. While it does not have an explicit implementation of `len`, calling `.len()` on a `Point` still compiles via dereferencing to a `Slice`. The `JaggedEvalSumcheckConfig::jagged_evaluation` function uses `.len()` on `Point` multiple times. In this setting, using `.dimension()` would be more idiomatic. Mixing `dimension()` and `len()` can be slightly confusing because the reader might doubt whether `.len()` returns the same value as `.dimension()`.

Impact

Using `len` to access the dimension breaks the abstraction layer of `Point::dimension`. Future implementations may not guarantee that `len` returns the same value as `dimension`.

Recommendations

Always use `dimension` to retrieve the dimension of a `Point`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit b4de67db ↗.

3.31. Field `max_log_row_count` of `JaggedLittlePolynomialVerifierParams` is unused

Target	slop_jagged		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The field `max_log_row_count` of the `JaggedLittlePolynomialVerifierParams` struct does not appear to be used anywhere, except in `JaggedPcsVerifier::verify_trusted_evaluations`, where it is checked to be `JaggedPcsVerifier::max_log_row_counts`. But as this is its only appearance, it does not appear to have use; the prover is forced to set it to the value it is checked against, which they can, and then nothing further depends on it.

Impact

The redundant field increases the proof size.

Recommendations

Remove the redundant field.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 41ff5b09 ↗.

3.32. Polynomial interpolation might panic if an interpolation point occurs more than once

Target	slop_algebra		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `interpolate_univariate_polynomial` function is intended to compute the interpolated polynomial that evaluates to `ys[i]` at `xs[i]`:

```
// slop/crates/algebra/src/univariate.rs
pub fn interpolate_univariate_polynomial<K: Field>(xs: &[K], ys: &[K]) ->
    UnivariatePolynomial<K> {
    let mut result = UnivariatePolynomial::new(vec![K::zero()]);
    for (i, (x, y)) in xs.iter().zip(ys).enumerate() {
        let (denominator, numerator) = xs.iter().enumerate().filter(|(j,
    _)| *j != i).fold(
            (K::one(), UnivariatePolynomial::new(vec![*y])),
            |(denominator, numerator), (_, xj)| {
                (denominator * (*x - *xj), numerator.mul_by_x() + numerator *
    (-*xj))
            },
        );
        result = result + numerator * denominator.inverse();
    }
    result
}
```

The function assumes that `xs` and `ys` have the same length and the values in `xs` are unique. Note that should `xs` include the same value twice, then on `x` taking that value in the outer loop, one will still get this same value at least once as `xj` in the inner loop. This will make `denominator` zero, which will then cause the call `denominator.inverse()` to panic.

Impact

Calling this function on data that could contain a duplicate value in `xs` could result in a panic.

Calling this function on `xs` and `ys` of different lengths will truncate the longer one, which likely is unintended.

Recommendations

Document the assumption on `xs` and `ys`. Wrap the return value in an `Option` or `Result` type to allow the caller to handle the error.

Remediation

This issue has been acknowledged by Succinct Inc. The behavior of the function was documented in commit 4125c566 ↗.

Succinct Inc. stated the following:

> We only use this function in prove methods, so [...] the panic is ok, though as you suggest it would be good to document. Ultimately, though, we're ok with assigning responsibility to the caller to ensure the x points are not duplicates and that the `xs` and `ys` lengths are the same.

3.33. Proof component `branching_program_evals` is not required for `JaggedEvalSumcheckConfig::jagged_evaluation` and not checked

Target	slop_jagged		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `JaggedEvalSumcheckConfig::jagged_evaluation` has the following signature:

```
// crates/jagged/src/jagged_eval/sumcheck_eval.rs
pub fn jagged_evaluation<EF: ExtensionField<F>, Challenger:
    FieldChallenger<F>>(
    params: &JaggedLittlePolynomialVerifierParams<F>,
    z_row: &Point<EF>,
    z_col: &Point<EF>,
    z_trace: &Point<EF>,
    proof: &JaggedSumcheckEvalProof<EF>,
    challenger: &mut Challenger,
) -> Result<EF, JaggedEvalSumcheckError<EF>>
```

The argument `params` is intended to encode an admissible sequence of cumulative heights t of length K (see page 6 of *Jagged Polynomial Commitments (or: How to Stack Multilinears)* by Hemo, Jue, Rabinovich, Roh, and Rothblum for this terminology), and the purpose of this function is to verify a proof of the evaluation below, with the resulting value returned.

$$\sum_{0 \le y < K} \mathsf{eq}(z_c, \texttt{int\_to\_bits\_be}(y)) \cdot \hat{g}(z_r, z_t, t_{y-1}, t_y)$$

In the expression above, we use z_r, z_c, z_t for `z_row`, `z_col`, and `z_trace` respectively, and $\hat{g}$ and eq are as defined in the jagged paper.

The `proof` contains a field `branching_program_evals`, from which the value `jagged_eval` is computed:

```
let jagged_eval = z_col_partial_lagrange
    .iter()
    .zip(branching_program_evals.iter())
    .map(|(partial_lagrange, branching_program_eval)| {
        *partial_lagrange * *branching_program_eval
```

```
    })
    .sum::<EF>();
```

Here, `z_col_partial_lagrange[y]` is equal to eq$(z_c, \texttt{int_to_bits_be}(y))$, and the intention is that the value of `branching_program_eval[y]` that the prover provides should be equal to $\hat{g}(z_r, z_t, t_{y-1}, t_y)$.

The value `jagged_eval` is the returned result, so one potential implementation of `JaggedEvalSumcheckConfig::jagged_evaluation` would be to verify that each `branching_program_eval[y]` is correct.

However, the actual implementation instead uses a different method to validate `jagged_eval`, and `branching_program_eval` plays no further role. Verification thus succeeds as long as the prover provides values as `branching_program_eval` with the property that the linear combination used to compute `jagged_eval` evaluates to the correct result.

Impact

There is no impact on soundness or completeness of the verifier.

Transmitting the vector `branching_program_eval` as part of the proof is unnecessary, so the proof could be made smaller by instead having the proof contain `jagged_eval` directly.

A minor risk in the current implementation is that developers of higher-level protocols making use of `JaggedEvalSumcheckConfig::jagged_evaluation` expect that the values of `branching_program_eval` are being checked to be correct (in the sense of containing the values $\hat{g}(z_r, z_t, t_{y-1}, t_y)$) by `JaggedEvalSumcheckConfig::jagged_evaluation`, which they, however, are not. Within the scope of this audit, we did not detect any such usage.

Recommendations

Consider replacing `branching_program_eval` in the proof by `jagged_eval`.

Remediation

This issue has been acknowledged by Succinct Inc., and fixes were implemented in the following commits:

- 1d575292 ↗
- 56bb8ef7 ↗

3.34. Redundant prover messages in sumcheck and basefold

Target	slop_basefold, slop_sumcheck		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the protocol's sumcheck and basefold, there is a phase where the prover sends a sequence of univariate polynomials. In each round of that phase, the prover sends a univariate polynomial (observed by the challenger); this is checked to evaluate correctly at some point. Then, the challenger generates a random value, which updates one coordinate of the point.

The evaluation at the point determines one of the coefficients of the univariate polynomial. So instead of sending that coefficient and then having the verifier check the evaluation, the prover could skip sending one coefficient, and the verifier deduces that coefficient instead of verifying the evaluation.

This shortens the transaction transcript, because less coefficients have to be sent/observed.

In sumcheck, there is a comment at the initial check that might refer to this optimization.

```
// There is a way to structure a sumcheck proof so that this check is not
    needed, but it doesn't
// actually save the verifier work.
let first_poly = &proof.univariate_polys[0];
if first_poly.eval_one_plus_eval_zero() != proof.claimed_sum {
    return Err(SumcheckError::InconsistencyWithClaimedSum);
}
```

Succinct Inc. stated the following:

> The comment did indeed reflect that we are aware of this optimization, but it is from a time when we were mostly concerned about verifier arithmetic and less worried about proof size. But now we are more concerned about proof size and so it's worth revisiting.

Impact

The redundant observations increase the proof size.

Recommendations

Remove the constant coefficient from each univariate polynomial. Replace the evaluation check by the computation of the constant coefficient.

Remediation

This issue has been acknowledged by Succinct Inc.

3.35. Redundant `Padding::Zero`

Target	slop_multilinear		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The enum value `Padding::Zero` is redundant. It functions the same way as `Padding::Constant` with value 0. The following code already uses `Padding::Constant` where `Padding::Zero` could be used:

```
// slop/crates/multilinear/src/padded.rs
pub fn padded_with_zeros(inner: Arc<Mle<T, A>>, num_variables: u32) -> Self
where
    A: MleBaseBackend<T>,
{
    let num_polys = inner.num_polynomials();
    let backend = inner.backend().clone();
    Self::padded(inner, num_variables, Padding::Constant((T::zero(),
    num_polys, backend)))
}
```

Impact

The case `Padding::Zero` always has to be considered in implementations in addition to `Padding::Constant`, leading to redundant code, which bears the risk of introducing errors.

Recommendations

Remove `Padding::Zero`. If this special case of `Padding::Constant` should be specifically optimized, check whether the constant padding value is zero at the relevant points.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 5eb22cd0 ↗.

3.36. Variables swapped in `all_bit_states`

Target	slop_jagged		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the jagged crate, the following function creates a list of all states of the branching program used in the evaluation of the jagged function:

```
// slop/crates/jagged/src/poly.rs
pub fn all_bit_states() -> Vec<BitState<bool>> {
    (0..2)
        .flat_map(|row_bit| {
            (0..2).flat_map(move |index_bit| {
                (0..2).flat_map(move |last_col_bit| {
                    (0..2).map(move |curr_col_bit| BitState {
                        row_bit: row_bit != 0,
                        index_bit: index_bit != 0,
                        curr_col_prefix_sum_bit: last_col_bit != 0,
                        next_col_prefix_sum_bit: curr_col_bit != 0,
                    })
                })
            })
        })
        .collect()
}
```

Note that `last_col_bit` and `curr_col_bit` are used in a manner that can be confusing; `curr_col_bit` sounds like a variable one might use for `curr_col_prefix_sum_bit`, not `next_col_prefix_sum_bit`. The function still does what is intended; however, the order of the bit states will be different than one might think on skimming the code and mistakenly assuming `curr_col_bit` corresponds to `curr_col_prefix_sum_bit`.

Impact

It is important in other parts of the implementation, in particular `BranchingProgram::eval`, that `BitStates` are ordered in a consistent manner. While the audited code did indeed sort `BitStates` consistently, the above implementation of `all_bit_states` could lead to confusion about the

correct ordering on future changes.

Recommendations

Rename the local variable `last_col_bit` to something like `curr_col_bit` and `curr_col_bit` to something like `next_col_bit` (in each case in both occurrences, the function should still behave exactly as it does currently, as of the time of writing).

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 1932e4ee ↗.

3.37. Jagged proof can be made substantially smaller by removing `params`

Target	slop_jagged		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The jagged PCS proof contains a field `params` of type `JaggedLittlePolynomialVerifierParams`:

```
// slop/crates/jagged/src/verifier.rs
pub struct JaggedPcsProof<GC: IopCtx, C: JaggedConfig<GC>> {
    pub pcs_proof: <C::PcsVerifier as MultilinearPcsVerifier<GC>>::Proof,
    pub sumcheck_proof: PartialSumcheckProof<GC::EF>,
    pub jagged_eval_proof: JaggedSumcheckEvalProof<GC::EF>,
    pub params: JaggedLittlePolynomialVerifierParams<GC::F>,
    pub row_counts_and_column_counts: Rounds<Vec<(usize, usize)>>,
    pub merkle_tree_commitments: Rounds<GC::Digest>,
}
```

The data contained in `params` is as follows:

```
pub struct JaggedLittlePolynomialVerifierParams<K> {
    pub col_prefix_sums: Vec<Point<K>>,
    pub max_log_row_count: usize,
}
```

The `usize` `max_log_row_count` holds the logarithm to basis 2 of the maximum height of a column. The first field `col_prefix_sums` is intended to contain the prefix sums (including $t_{-1} = 0$ in the terminology of Hemo, Jue, Rabinovich, Roh, and Rothblum from the *Jagged Polynomial Commitments (or: How to Stack Multilinears)* ↗ paper), stored in the form of their binary big-endian representation, with each bit saved as a field element. This makes `params` of substantial size, as each field element usually takes four bytes to store and the number of components (bits) may be up to 30. Thus, `params.col_prefix_sums` might take up to $120 \cdot n$ bytes, where n is the total number of columns.

However, `params` is deprecated and not required anymore. After the refactor (see the discussion in section 5.2. ↗), `params` is not used by the shard verifier.

The jagged PCS verifier itself uses `params` as follows:

```rust
// crates/jagged/src/verifier.rs
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &JaggedPcsProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), JaggedPcsVerifierError<GC::EF, JaggedError<GC, C>>> {
    let JaggedPcsProof {
        pcs_proof,
        sumcheck_proof,
        jagged_eval_proof,
        params,
        row_counts_and_column_counts,
        merkle_tree_commitments: original_commitments,
    } = proof;

    let (row_counts, column_counts): (Vec<Vec<usize>>, Vec<Vec<usize>>) =
            row_counts_and_column_counts.iter().map(|r_c|
    r_c.clone().into_iter().unzip()).unzip();

    if params.col_prefix_sums.is_empty() || params.max_log_row_count
    != self.max_log_row_count {
        return Err(JaggedPcsVerifierError::IncorrectShape);
    }

    let num_col_variables = (params.col_prefix_sums.len() -
    1).next_power_of_two().ilog2();

    // ...

    if point_prefix_sums != params.col_prefix_sums {
        return Err(JaggedPcsVerifierError::IncorrectShape);
    }

    // ...

    // Further uses of `params`

    // ...
}
```

The above snippets include all uses up to the `point_prefix_sums != params.col_prefix_sums` check. All checks involving `params` below it could thus replace `params.col_prefix_sums` by `point_prefix_sums` and `params.max_log_row_count` by `self.max_log_row_count`. The only two

uses that occur potentially before `point_prefix_sums` was computed only depend on `params.col_prefix_sums.len()`.

Looking into the definition of `point_prefix_sums`, we find the following:

```
// crates/jagged/src/verifier.rs
let (row_counts, column_counts): (Vec<Vec<usize>>, Vec<Vec<usize>>) =
    row_counts_and_column_counts.iter().map(|r_c|
    r_c.clone().into_iter().unzip()).unzip();
// ...
let usize_column_heights: Vec<usize> = row_counts
    .iter()
    .zip_eq(column_counts.iter())
    .flat_map(|(rc, cc)| {
        rc.iter().zip_eq(cc.iter()).flat_map(|(r, c)| std::iter::repeat_n(*r,
    *c))
    })
    .collect();
let mut usize_prefix_sums: Vec<usize> = usize_column_heights
    .iter()
    .scan(0usize, |state, &x| {
        let result = *state;
        *state += x;
        Some(result)
    })
    .collect();
usize_prefix_sums
    .push(*usize_prefix_sums.last().unwrap() +
    *usize_column_heights.last().unwrap());
// ...
let point_prefix_sums: Vec<Point<GC::F>> =
    usize_prefix_sums.iter().map(|&x| Point::from_usize(x, log_m +
    1)).collect();
```

So we can conclude that `params.col_prefix_sums.len()` can be replaced by `point_prefix_sums.len()`, which is equal to `usize_prefix_sums.len()`, which is equal to `usize_column_heights.len() + 1`, which could be computed immediately at the start of the function, without requiring `params` previously.

Some of the later checks involving `params` are in principle redundant and, for example, follow by construction of `point_prefix_sums`. However, it may make sense to keep them as sanity checks.

Impact

The jagged PCS proof is substantially larger than would be possible. Furthermore, keeping duplicate copies of the same data around makes the verifier code larger and thus unnecessarily

harder to read and reason about.

Recommendations

We recommend the following.

1. Remove the `params` field of `JaggedPcsProof`.
2. Remove the checks that forced `params` to be equal to a different variable.
3. Then replace the remaining code using `params` by using the value it was forced to be equal to.

Remediation

This issue has been acknowledged by Succinct Inc., and fixes were implemented in the following commits:

- f38b8c1f ↗

The commit fixes the issue, but introduces a new verifier panic if `proof.row_counts_and_column_counts` is empty.

3.38. Different batching layers in `MultilinearPcsBatchProver` and `MutlilinearPcsBatchVerifier`

Target	slop_multilinear		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The interfaces of `MultilinearPcsBatchProver::prove_*trusted_evaluations` and `MultilinearPcsBatchVerifier::verify_*trusted_evaluations` do not match. While the evaluation batching of the verifier consists of two layers, the batching in the prover has three layers.

The stuct `MleEval` used in the verifier is a rank-1 container; thus the argument `evaluations_claims`, which is a slice of `MleEval`, has rank 2. In contrast, because the struct `Rounds` has rank 1 and the struct `Evaluations` wrapping a vector of `MleEval` has rank 2, the argument `evaluation_claims` of the prover has rank 3 in total.

```
// slop/crates/multilinear/src/pcs.rs
/// * `evaluation_claims` - The evaluation claims for the multilinear
    polynomials. the slice
///   contains one [MleEval] for each round of the protocol.
/// ...
fn verify_trusted_evaluations(
    &self,
    commitments: &[GC::Digest],
    point: Point<GC::EF>,
    evaluation_claims: &[MleEval<GC::EF>],
    proof: &Self::Proof,
    challenger: &mut GC::Challenger,
) -> Result<(), Self::VerifierError>;

fn prove_trusted_evaluations(
    &self,
    eval_point: Point<GC::EF>,
    mle_rounds: Rounds<Message<Mle<GC::F, Self::A>>>,
    evaluation_claims: Rounds<Evaluations<GC::EF, Self::A>>,
    prover_data: Rounds<Self::ProverData>,
    challenger: &mut GC::Challenger,
) -> impl Future<
    Output = Result<
        <Self::Verifier as MultilinearPcsBatchVerifier<GC>>::Proof,
```

```
        Self::ProverError,
    >,
> + Send;
```

Impact

A user that generates a proof with the prover must supply rank-3 data but then flatten it to rank-2 data for the verifier. However, it is not documented which flattening should be used. Using the incorrect flattening will likely make the proof not verify.

Recommendations

Use the same type for batching in the prover and verifier interfaces. If the difference in batching is required and cannot be removed, document the correspondence between the two interfaces.

Remediation

Succinct Inc. explained the additional batching layer in the prover:

> The "reason", such as it is, is that the prover does have API to have this middle layer of batching in case it wants to breakup the MLE into contiguous and non-contiguous portions for memory usage reasons.

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 0490b491 ↗. This commit adds documentation on the difference.

3.39. Incorrect Rust toolchain version

Target	slop		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The file slop/rust-toolchain pins the Rust version to 1.82, but the code requires at least version 1.87.

Impact

The code cannot be compiled with the pinned version.

Recommendations

Change the pinned version.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 590e844c ↗.

4. Detailed Findings (Pre-refactor Scope)

This section discusses issues that were found in the state of the code as of the initial scope, which was prior to the refactor; see section 2.3. ↗ for more information on the scope change.

4.1. Malicious proofs can trigger a panic in the sumcheck verifier

Target	slop_sumcheck		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `partially_verify_sumcheck_proof` function in crates/sumcheck/src/verifier.rs (partially) checks sumcheck proofs, provided in the argument `proof`.

The implementation unconditionally accesses a vector that is part of the proof and hence attacker-controlled at index zero:

```
let first_poly = &proof.univariate_polys[0];
```

This panics should `proof.univariate_polys` be empty.

There are some checks before this access in the function that could cause the control flow to avoid the panic by returning early with an error:

```
let num_variables = proof.univariate_polys.len();
let mut alpha_point = Point::default();

// Checks for the correct proof shape.
if num_variables != proof.point_and_eval.0.dimension() {
    return Err(SumcheckError::InvalidProofShape);
}

if num_variables != expected_num_variable {
    return Err(SumcheckError::InvalidProofShape);
}
```

If `proof.univariate_polys` is empty, then `num_variables = 0`, and so the panic can only be reached if `proof.point_and_eval.0.dimension() = 0` and `expected_num_variable = 0`. The former can be arranged by the attacker, as this is also part of the proof. The latter is a separate argument to the verifier functions, so this depends on the use by the caller.

Impact

Calls to `partially_verify_sumcheck_proof` in which attacker-controlled `proof` and `expected_num_variable` are passed are vulnerable to denial-of-service attacks, with the attacker setting values in such a way that the verifier panics.

One place where `partially_verify_sumcheck_proof` is called is in `JaggedPcsVerifier::verify_trusted_evaluations` in crates/jagged/src/verifier.rs:

```
fn verify_trusted_evaluations(
    &self,
    commitments: &[C::Commitment],
    point: Point<C::EF>,
    evaluation_claims: &[Evaluations<C::EF>],
    proof: &JaggedPcsProof<C>,
    insertion_points: &[usize],
    challenger: &mut C::Challenger,
) -> Result<
    (),
    JaggedPcsVerifierError<
        C::EF,
        <C::BatchPcsVerifier as MultilinearPcsVerifier>::VerifierError,
    >,
> {
    let JaggedPcsProof {
        stacked_pcs_proof,
        sumcheck_proof,
        jagged_eval_proof,
        params,
        added_columns,
    } = proof;

    // ...

    partially_verify_sumcheck_proof(
        sumcheck_proof,
        challenger,
        params.col_prefix_sums[0].len() - 1,
        2,
    )
    .map_err(JaggedPcsVerifierError::SumcheckError)?;

    // ...
}
```

Note that here, the value passed as `expected_num_variable` is `params.col_prefix_sums[0].len() - 1`. As not just `sumcheck_proof` but also `params` comes

from the jagged PCS proof, the attacker providing this proof can set the value of `params.col_prefix_sums[0]` so that it has length one, thereby ensuring that `expected_num_variable = 0` in the sumcheck verifier.

The code in `verify_trusted_evaluations` before the call to `partially_verify_sumcheck_proof` carries out some consistency checks that also involve `params.col_prefix_sums`, but they do not stop an attacker modifying a proof in the described manner.

The following test demonstrates this. This test is essentially a copy of `test_jagged_basefold`. The only change is that we added some modifications to the proof just before running the verifier. The test will panic at the indexing at 0 as expected.

```
async fn zellic_test_jagged_basefold_panic_via_sumcheck<
    BC: BasefoldConfig<Commitment: Debug> + DefaultBasefoldConfig,
    Prover: JaggedProverComponents<
            Config = JaggedBasefoldConfig<BC, E<BC::F>>,
            F = BC::F,
            EF = BC::EF,
            Challenger = <JC<BC> as JaggedConfig>::Challenger,
            A = CpuBackend,
            Commitment = <JC<BC> as JaggedConfig>::Commitment,
        > + DefaultJaggedProver,
>()
where
    rand::distributions::Standard:
    rand::distributions::Distribution<<BC as BasefoldConfig>::F>,
    rand::distributions::Standard:
        rand::distributions::Distribution<<BC as BasefoldConfig>::EF>,
    <JC<BC> as JaggedConfig>::Commitment: Copy,
{
    let row_counts_rounds = vec![vec![1 << 10, 0, 1 << 10], vec![1 << 8]];
    let column_counts_rounds = vec![vec![128, 45, 32], vec![512]];

    let log_blowup = 1;
    let log_stacking_height = 11;
    let max_log_row_count = 10;

    let row_counts =
    row_counts_rounds.into_iter().collect::<Rounds<Vec<usize>>>();
    let column_counts =
    column_counts_rounds.into_iter().collect::<Rounds<Vec<usize>>>();

    assert!(row_counts.len() == column_counts.len());

    let mut rng = thread_rng();

    let round_mles = row_counts
```

```
        .iter()
        .zip(column_counts.iter())
        .map(|(row_counts, col_counts)| {
            row_counts
                .iter()
                .zip(col_counts.iter())
                .map(|(num_rows, num_cols)| {
                    if *num_rows == 0 {
                        PaddedMle::zeros(*num_cols, max_log_row_count)
                    } else {
                        let mle = Mle::<BC::F>::rand(&mut rng, *num_cols,
    num_rows.ilog(2));
                        PaddedMle::padded_with_zeros(Arc::new(mle),
    max_log_row_count)
                    }
                })
                .collect::<Vec<_>>()
        })
        .collect::<Rounds<_>>();

    let jagged_verifier = JaggedPcsVerifier::<JaggedBasefoldConfig<BC,
    E<BC::F>>>::new(
        log_blowup,
        log_stacking_height,
        max_log_row_count as usize,
    );

    let jagged_prover =
    JaggedProver::<Prover>::from_verifier(&jagged_verifier);

    let machine_verifier = MachineJaggedPcsVerifier::new(
        &jagged_verifier,
        vec![column_counts[0].clone(), column_counts[1].clone()],
    );

    let eval_point = (0..max_log_row_count).map(|_|
    rng.gen::<BC::EF>()).collect::<Point<_>>();

    // Begin the commit rounds
    let mut challenger = jagged_verifier.challenger();

    let mut prover_data = Rounds::new();
    let mut commitments = Rounds::new();
    for round in round_mles.iter() {
        let (commit, data) =

    jagged_prover.commit_multilinears(round.clone()).await.ok().unwrap();
```

```
        challenger.observe(commit);
        prover_data.push(data);
        commitments.push(commit);
    }

    let mut evaluation_claims = Rounds::new();
    for round in round_mles.iter() {
        let mut evals = Evaluations::default();
        for mle in round.iter() {
            let eval = mle.eval_at(&eval_point).await;
            evals.push(eval);
        }
        evaluation_claims.push(evals);
    }

    let mut proof = jagged_prover
        .prove_trusted_evaluations(
            eval_point.clone(),
            evaluation_claims.clone(),
            prover_data,
            &mut challenger,
        )
        .await
        .ok()
        .unwrap();

    let mut challenger = jagged_verifier.challenger();
    for commitment in commitments.iter() {
        challenger.observe(*commitment);
    }

    use slop_algebra::AbstractField;
    proof.params.col_prefix_sums[0] = Point::default();
    proof.params.col_prefix_sums[0]
        .add_dimension(<BC as slop_basefold::BasefoldConfig>::F::zero());
    proof.sumcheck_proof.point_and_eval =
        (Point::default(), <BC as slop_basefold::BasefoldConfig>::EF::zero());
    proof.sumcheck_proof.univariate_polys = vec![];

    let ret = machine_verifier.verify_trusted_evaluations(
        &commitments,
        eval_point,
        &evaluation_claims,
        &proof,
        &mut challenger,
    );
    println!("Verifier returned: {:?}", ret);
```

```
}

#[tokio::test]
async fn zellic_test_koala_bear_jagged_basefold_panic_via_sumcheck() {
    zellic_test_jagged_basefold_panic_via_sumcheck::<
        Poseidon2KoalaBear16BasefoldConfig,
        Poseidon2KoalaBearJaggedCpuProverComponents,
    >()
    .await;
}
```

Recommendations

To support the case of multivariate polynomials in zero variables, these checks involving the first polynomial need to be skipped in the case `expected_num_variable == 0`:

```
let first_poly = &proof.univariate_polys[0];
if first_poly.eval_one_plus_eval_zero() != proof.claimed_sum {
    return Err(SumcheckError::InconsistencyWithClaimedSum);
}

if first_poly.coefficients.len() != expected_degree + 1 {
    return Err(SumcheckError::InvalidProofShape);
}

challenger.observe_slice(
    &first_poly
        .coefficients
        .iter()
        .flat_map(|x| x.as_base_slice())
        .copied()
        .collect::<Vec<_>>(),
);
let mut previous_poly = first_poly;
```

Additionally, extending `alpha` after the loop should be skipped:

```
let alpha = challenger.sample_ext_element();
alpha_point.add_dimension(alpha);
```

Finally, the following check

```
if previous_poly.eval_at_point(alpha) != proof.point_and_eval.1 {
    return Err(SumcheckError::InconsistencyWithEval);
}
```

should, in the case `expected_num_variable == 0`, compare `proof.point_and_eval.1` to `proof.claimed_sum` instead of `previous_poly.eval_at_point(alpha)`. Together with the check `alpha_point == proof.point_and_eval.0`, which needs to be kept, this will ensure that the evaluation claim encoded by `proof.point_and_eval` enforces that evaluation of the (necessarily constant) polynomial in zero variables (note that `alpha` will be of length zero) results in the claimed value.

If the sumcheck verifier is never intended to be run legitimately on polynomials in zero variables, then it is also possible to add a check for `expected_num_variable == 0` and return an error. In this case, it should be documented that the zero-variable case is not supported, as it may naturally occur as an edge case.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit ba039b77 ↗. The fix returns an error when `expected_num_variable == 0`.

This fix was included in the state of the code we reviewed after the refactor.

4.2. Padding manipulation breaks the binding property of jagged PCS

Target	slop_jagged		
Category	Soundness	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The `MachineJaggedPcsVerifier` in crates/jagged/src/verifier.rs is the top-level verifier for the jagged PCS. The commitment for the jagged PCS consists of individual round commitments that can be sent by the prover in different steps of a higher-level protocol. (In the case of the SP1 shard verifier, there are two rounds: the preprocessed tables that are committed to in the verification key and then the main round with the witnesses committed by the prover.) Each round commitment commits to a batch of batches of multilinear polynomials defined on `max_log_row_count` variables. Each of these multilinear polynomials f has a height $h < 2^{\texttt{max\_log\_row\_count}}$, and $f(\texttt{int\_to\_bits}_{\texttt{max\_log\_row\_count}}(i)) = 0$ for $h \leq i < 2^{\texttt{max\_log\_row\_count}}$.

The verifier for the jagged PCS thus altogether handles a (round) batch of (rectangle) batches of (column) batches of multilinear polynomials. It reduces to an underlying PCS for a single large dimensional multilinear polynomial, which is committed to in rounds (which each commit to some of the evaluations). The commitment for the jagged PCS is the same as the commitment to the underlying PCS. We will refer to the individual polynomials interchangeably as columns. Thinking of them as columns, the height corresponds to the usual intuitive meaning.

See *Jagged Polynomial Commitments (or: How to Stack Multilinears)* ↗ for more information on the purpose and design of the jagged PCS. The remainder of the finding assumes the reader is familiar with the design of the jagged PCS.

As explained in section 5.2. ↗, the middle batching layer is ignored throughout, with the two bottom batching layers always flattened to a single layer. We will thus describe the rest of this finding as if each round consisted of a flat list of polynomials.

The `MachineJaggedPcsVerifier` is the outer, user-facing layer of the jagged PCS that wraps the `JaggedPcsVerifier`. As parts of its configuration, the `MachineJaggedPcsVerifier` contains the expected column counts:

```
pub struct MachineJaggedPcsVerifier<'a, C: JaggedConfig> {
    pub jagged_pcs_verifier: &'a JaggedPcsVerifier<C>,
    pub column_counts_by_round: Vec<Vec<usize>>,
}
```

The jagged PCS verifier assumes that these column counts were fixed by the time the commitment was observed (e.g., by the challenger observing them then as well or by fixing them even prior,

perhaps baking them into the protocol). In the case of the shard verifier, the heights for the chips is observed by the challenger just after the main round commitment, and `column_counts_by_round` is later computed from these heights.

The verifier checks that the evaluation claims match the shape prescribed by `column_counts_by_round` (after flattening the middle and bottom batching layer):

```
pub fn verify_trusted_evaluations(
    &self,
    commitments: &[C::Commitment],
    point: Point<C::EF>,
    evaluation_claims: &[Evaluations<C::EF>],
    proof: &JaggedPcsProof<C>,
    challenger: &mut C::Challenger,
) -> Result<
    (),
    JaggedPcsVerifierError<
        C::EF,
        <C::BatchPcsVerifier as MultilinearPcsVerifier>::VerifierError,
    >,
> {
    if evaluation_claims.len() != self.column_counts_by_round.len() {
        return Err(JaggedPcsVerifierError::IncorrectShape);
    }
    for (claims, expected_counts) in
        evaluation_claims.iter().zip_eq(&self.column_counts_by_round)
    {
        let claim_count: usize = claims
            .round_evaluations
            .iter()
            .map(slop_multilinear::MleEval::num_polynomials)
            .sum();

        let expected_count: usize = expected_counts.iter().sum();

        if claim_count != expected_count {
            return Err(JaggedPcsVerifierError::IncorrectShape);
        }
    }
    // ...
```

So far, the column counts are the user-facing column counts. Internally, the conversion to the underlying PCS requires the area of each round (summing up the height of each columns in the round) to be a multiple of $2^{\texttt{max\_log\_row\_count}}$. The jagged PCS handles this by padding the rounds with some additional columns, together with evaluation claims with value zero. Continuing from the `MachineJaggedPcsVerifier::verify_trusted_evaluations` function, after checking that the

user-facing evaluation claims match the expected shape, it then proceeds to compute `insertion_points`, which contains the indexes in the fully flattened list of evaluation claims at which the rounds end (i.e., the index of the next column after the round or equivalently the number of all columns so far, including the round being considered):

```
let insertion_points = self
    .column_counts_by_round
    .iter()
    .scan(0, |state, y| {
        *state += y.iter().sum::<usize>();
        Some(*state)
    })
    .collect::<Vec<_>>();
```

Those `insertion_points` are passed to the `JaggedPcsVerifier::verify_trusted_evaluations` function.

A proof for the jagged PCS has the following form:

```
pub struct JaggedPcsProof<C: JaggedConfig> {
    pub stacked_pcs_proof: StackedPcsProof<C::BatchPcsProof, C::EF>,
    pub sumcheck_proof: PartialSumcheckProof<C::EF>,
    pub jagged_eval_proof:
        <C::JaggedEvaluator as JaggedEvalConfig<C::F, C::EF,
    C::Challenger>>::JaggedEvalProof,
    pub params: JaggedLittlePolynomialVerifierParams<C::F>,
    pub added_columns: Vec<usize>,
}
```

The field `params` contains the prefix sums, so the cumulative sum of the heights of each column, taking into account the columns added for padding. Additionally, the proof contains `added_columns`, the number of columns added for padding at the end of each round.

Given `params`, the prover is not fully free to choose `added_columns` as they like. The prefix sums need to match, and the underlying PCS (in practice, stacked together with its underlying multilinear PCS) also checks various shape properties that constrain how freely the prover can choose `added_columns`.

However, the number of padding columns is not checked against something deterministically derived from `column_counts_by_round` or checked against the commitment. This allows a malicious prover some flexibility with which to influence the openings of the commitment.

This leads to a failure of the jagged PCS as implemented to satisfy the property of being binding. To be concrete, it is possible for a malicious prover to use `added_columns` to open the same commitment at the same point, with the same `params`, and the same verifier configuration, in more than one way.

To understand the attack idea, recall that the jagged PCS verifier reduces to an underlying PCS for a single polynomial. Thus, to translate the evaluation claims, they are fully flattened:

```
// Collect the claims for the different polynomials.
let mut column_claims =
    evaluation_claims.iter().flatten().flatten().copied().collect::<Vec<_>>();

// ...
for (insertion_point, num_added_columns) in
    insertion_points.iter().rev().zip_eq(added_columns.iter().rev())
{
    for _ in 0..*num_added_columns {
        column_claims.insert(*insertion_point, C::EF::zero());
    }
}
```

By manipulating the number of padding columns inserted between rounds, the malicious prover can thus shift evaluation claims (i.e., have the column claims belonging to a particular batch appear at a different index in two different proofs for the same commitment). What restricts this attack is that a check is performed on the total area (so sum of heights of columns) in each batch. However, this still allows moving columns with height zero between two batches, as they do not contribute to the area. Note that in particular the padding columns themselves can realistically have height zero, and even at the shard proof level, having a chip of height zero appears in principle supported.

There are two directions one can move height-zero padding columns, starting from one particular opening that we consider the base truth.

One option is that the last n padding columns belonging to one round had height zero. In that case, one can reduce `added_columns` for that round by n and increase the number of `added_columns` for the next round by n. The claimed evaluations for the first of the two rounds will not change. For the round after it, however, the first n columns that will be de-facto opened were actually the padding rounds from the previous round. The following columns will all shift by n, and the last n legitimate columns from the round will be reinterpreted as padding. The attack thus requires those columns to evaluate to zero already. So the upshot is that if one round has evaluations that end with a certain number of columns evaluating to zero, and the preceeding round has enough trailing padding columns that have height zero, then the malicious prover can cyclicly permute the evaluation claims for this round by up to that number.

The analogous second variant is a round that starts with columns that are of height zero. In that case, `added_columns` for the preceeding round can be increased, and this will amount to a cyclic permutation of the evaluation claims in the other direction.

We provided Succinct Inc. with a proof of concept for both variants, based on the `test_jagged_basefold` test in crates/jagged/src/basefold.rs. In both cases, we create a proof for the jagged PCS normally and verify it. Then we change only the evaluation claims, as well as `added_columns` of the proof, and verify again.

For the first variant, the beginning of the function is changed to the following:

```
use slop_algebra::AbstractField;
use slop_tensor::Tensor;

let row_counts_rounds = vec![vec![1 << 4, 1 << 4], vec![1 << 4, 1 << 4,
    1 << 4]];
let column_counts_rounds = vec![vec![1, 1], vec![1, 1, 1]];

let log_blowup = 1;
let log_stacking_height = 5;
let max_log_row_count = 4;

let row_counts =
    row_counts_rounds.into_iter().collect::<Rounds<Vec<usize>>>();
let column_counts =
    column_counts_rounds.into_iter().collect::<Rounds<Vec<usize>>>();

assert!(row_counts.len() == column_counts.len());

let mut rng = thread_rng();

let mut index = 0;
let const_values: [BC::F; 5] = [
    BC::F::from_canonical_u32(42),
    BC::F::from_canonical_u32(41),
    BC::F::from_canonical_u32(2),
    BC::F::from_canonical_u32(1),
    BC::F::zero(),
];

let round_mles = row_counts
    .iter()
    .zip(column_counts.iter())
    .map(|(row_counts, col_counts)| {
        row_counts
            .iter()
            .zip(col_counts.iter())
            .map(|(num_rows, num_cols)| {
                if *num_rows == 0 {
                    PaddedMle::zeros(*num_cols, max_log_row_count)
                } else {
                    //let mle = Mle::<BC::F>::rand(&mut rng, *num_cols,
    num_rows.ilog(2));
                    let tensor = Tensor::zellic_constant_values(
                        const_values[index],
                        [1 << num_rows.ilog(2), *num_cols],
                    );
```

```
                    index += 1;
                    let mle = Mle::<BC::F>::new(tensor);
                    PaddedMle::padded_with_zeros(Arc::new(mle),
    max_log_row_count)
                }
            })
            .collect::<Vec<_>>()
    })
    .collect::<Rounds<_>>();

assert_eq!(index, const_values.len());
```

And the end of the function (just the `machine_verifier` invocation) is replaced by the following:

```
use slop_multilinear::MleEval;

let evaluation_claims_two: [Evaluations<BC::EF>; 2] = [
    Evaluations {
        round_evaluations: vec![
            MleEval::new(Tensor::zellic_constant_values(
                BC::EF::from_canonical_u32(42),
                [1],
            )),
            MleEval::new(Tensor::zellic_constant_values(
                BC::EF::from_canonical_u32(41),
                [1],
            )),
        ],
    },
    Evaluations {
        round_evaluations: vec![
            MleEval::new(Tensor::zellic_constant_values(
                BC::EF::from_canonical_u32(0),
                [1],
            )),
            MleEval::new(Tensor::zellic_constant_values(
                BC::EF::from_canonical_u32(2),
                [1],
            )),
            MleEval::new(Tensor::zellic_constant_values(
                BC::EF::from_canonical_u32(1),
                [1],
            )),
        ],
    },
];
```

```
let mut proof_two = proof.clone();
proof_two.added_columns = vec![0, 2];
let mut challenger_two = challenger.clone();

println!("\nVerifying a proof with evaluation claims as follows");
for (i, eval_round) in evaluation_claims.iter().enumerate() {
    print!("Round {}: [", i);
    for x in eval_round.iter().flatten() {
        print!("{}, ", x);
    }
    println!("]");
}
let result = machine_verifier.verify_trusted_evaluations(
    &commitments,
    eval_point.clone(),
    &evaluation_claims,
    &proof,
    &mut challenger,
);
println!("Verifier result: {:?}\n", result);

println!("\nVerifying a second proof with evaluation claims as follows, with
    only the evaluation claim and `proof.added_columns` changed");
for (i, eval_round) in evaluation_claims_two.iter().enumerate() {
    print!("Round {}: [", i);
    for x in eval_round.iter().flatten() {
        print!("{}, ", x);
    }
    println!("]");
}
let result = machine_verifier.verify_trusted_evaluations(
    &commitments,
    eval_point,
    &evaluation_claims_two,
    &proof_two,
    &mut challenger_two,
);
println!("Verifier result: {:?}\n", result);
result.unwrap();
```

In this first proof of concept, the important configuration is as follows:

```
let row_counts_rounds = vec![vec![1 << 4, 1 << 4], vec![1 << 4, 1 << 4,
    1 << 4]];
let column_counts_rounds = vec![vec![1, 1], vec![1, 1, 1]];
let log_stacking_height = 5;
```

```
let max_log_row_count = 4;
```

We then arrange it so that these five columns have openings `[[42, 41], [2, 1, 0]]`. Proving this normally will cause both rounds to have one padding column. In the case of the first round, it will be of height zero. We then modify `added_columns` to be `[0,2]` instead of the previous `[1,1]`, and we now can open the same commitment with the same configuration and `params` at `[[42, 41], [0, 2, 1]]`. The `0` from the padding of the first round is now misinterpreted as the first column of the second round, and then the last column of the second round, which happened to evaluate to zero, is reinterpreted as the extra padding column that needs to evaluate to zero.

The second variant proof of concept is similar, using

```
let row_counts_rounds = vec![vec![1 << 4, 1 << 4, 1 << 4], vec![0, 1 << 4]];
let column_counts_rounds = vec![vec![1, 1, 1], vec![1, 1]];
let log_stacking_height = 6;
let max_log_row_count = 4;
```

Here we move the column in the other direction. So normally there will be one full-length padding column for the first round and three full-length ones for the second round. The second round starts with a column of height zero, so that can be moved over as a second padding column to the first round. Instead of the evaluation of the second round being verified to be `[0, 17]`, we can then instead also open it as `[17, 0]`.

Impact

This issue means that the commitment does not bind the prover to an opening with the implemented jagged PCS verifier.

While we have not tested this with a proof of concept, we believe that this issue is likely also reachable from the shard verifier, at least in principle.

The shard verifier makes things more difficult for the attacker, as there are additional checks that enforce that all padding columns (except possibly the last one in each top-level batch) need to be full height, and there must be at least one padding column for each top-level batch. Both are indeed satisfied for both openings in the second variant proof of concept above.

On top of these additional checks, in the context of the shard verifier, there are only two rounds, and the first round is the preprocessed one, with commitment already fixed by the verification key. Hence, the attacker cannot change it anymore to arrange the padding columns of the preprocessed round to all be of maximal height (as required given the previously mentioned checks in the example of the second variant proof of concept). They can thus only perform this attack (at least in the particular variant described above) against verification keys that happen to contain a preprocessed commitment in which the padding columns are all of maximal height.

If this is the case, the attacker can then use the described method to open the commitment to the main trace in more than one different way. However, further checks are performed on the

openings, as the constraints of the chips are checked.[3] As long as these checks are sound, the attacker will need to use chips so that it is possible to make the first column of the main trace height zero and so that permuting the column opening in the described way still satisfies the constraints, on top of columns interpreted as belonging to the same chip all having the same height. Such settings are likely rare, so it appears rare that an attacker might be able to find a pair of openings like this in practice.

Assuming for the moment that an attacker could find two such openings that both satisfy all constraints, this ultimately might not even be considered an issue at the level of the shard verifier, as the prover in either case would know a witness for the execution of a RISC-V program that they are claiming.

Thus, the more relevant impact is any influence that the possibility of the commitment to the jagged PCS not being binding could have on the soundness of the verification of the constraints.

Such impact can arise due to the protocol used to check the constraints on the opening having a small soundness error. The ability to switch between multiple openings can then in principle cause loss of some bits of security, as the attacker can choose from a number of openings by changing `added_columns` at a later time than they should.

In interactive protocols, one can discuss the soundness error of the protocol: the probability that the verifier will accept even though the statement was incorrect. This could, for example, be 2^{-100}, meaning that only every one in 2^{100} attempts will the prover maliciously succeed with the verifier accepting even though the statement was incorrect.

Making such a protocol noninteractive by applying the Fiat–Shamir conversion, we then obtain a situation in which the malicious prover can locally try to brute force an incorrect proof that will be accepted, with the expected number of proofs they need to generate being 2^{100}. Note that attackers are not bound to use any particular algorithm to attempt their brute force. However, the structure of the verification may enforce a certain amount of work.

As a toy example, consider a protocol in which the challenger first observes a commitment, then performs 2^{10} hashing operations to sample a challenge, and then checks an opening of the commitment and carries out a check using a random linear combination depending on the challenge. The soundness error in this protocol might then come from the reduction of certain equality checks to a single equality check via the random linear combination. In the security analysis of the entire protocol, we may then conclude that an attacker will need to make 2^{100} attempts in expectation, and on each attempt, they will need to perform at least 2^{10} hashing operations so that they can compute the challenge with which to check whether they got lucky and can pass the equality check. In total, we would then expect an attacker to need (in expectation) compute at least 2^{110} hashing operations.

Now suppose that the commitment does not bind, and the attacker, even after the commitment has been observed by the challenger, can still change some part of the proof that will cause the

[3] In this finding, we will use the term "constraints" to refer to the full collection of properties that the openings are supposed to satisfy, so this includes both the chip-level constraints as well as properties relating to interactions. At the abstract level, we discuss such properties here; the distinction does not play a role.

commitment to be openable differently. In that case, the attacker can run a brute force that is more economical; they still need to check 2^{100} openings in total, but they now split this into 2^{99} commitments, checked with respect to two openings each. This would then only require $2^{10} \cdot 2^{99} = 2^{109}$ hashing operations to be computed, only half of what would be required otherwise. Note that the attacker still needs to perform the computation of 2^{100} random linear combinations, but this is likely much cheaper than the hashing operations in our toy example.

Thus, the ability to open commitments of the underlying PCS in different ways implies that the security analysis of the higher-level protocol, in principle, loses some bits of security. If for some k an attacker can find a class of commitments that have on average 2^k openings, then in theory we might lose up to k bits of security.

In the concrete case of the non-bindingness of the jagged PCS due to this finding, plausible numbers for k are unlikely to be very large. Furthermore, while the loss of bits of security is possible in principle, the actual shard verifier does not follow the simple toy model described above, with a full security analysis regarding the bits of security significantly more complicated. Depending on the distribution of where the malicious prover is required to perform work, the malicious prover may not save as much work by utilizing the ability to open the commitment differently.

We have not performed a full analysis of the precise impact of this finding on the shard verifier's security. While we expect that the probability of an exploit of the shard verifier that makes use of this issue is low, it complicates the security analysis of the shard verifier, and we thus recommend fixing this issue also for the sake of the shard verifier.

Recommendations

The jagged PCS verifier commitments need to bindingly commit the prover to the padding for each round.

One option for this could be to commit to the padding shape together with the existing commitment.

Another option would be to require the padding to have a shape that is deterministically computed from data that is committed.

Remediation

This issue has been acknowledged by Succinct Inc. and was fixed in the codebase version audited post-refactor (see section 2.3. ↗ for more information). In this version of the codebase, the jagged PCS verifier utilizes commitments that bind to the padding shape directly. (We reported a more minor issue with this binding in Finding 3.12. ↗). Additionally, the padding shape is checked to match a shape deterministically derived from the round areas.

4.3. Duplicate drop of element returned by `Buffer::pop`

Target	slop_alloc		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The function `Buffer::pop` returns a bitwise copy of the last element of the buffer:

```
// crates/alloc/src/buffer.rs
impl<T> Buffer<T, CpuBackend> {
    // ...

    #[inline]
    pub fn pop(&mut self) -> Option<T> {
        if self.is_empty() {
            return None;
        }

        // This is safe because we have just checked that the buffer is not
    empty.
        unsafe {
            let len = self.len();
            let ptr = &mut self[len - 1] as *mut _ as *mut T;
            let value = ptr.read();
            self.set_len(len - 1);
            std::ptr::drop_in_place(ptr);
            Some(value)
        }
    }
```

The ownership of everything previously held by the last buffer element is transferred to `value`. Despite this transfer, the call `std::pts::drop_in_place(ptr)` drops the last buffer element. This includes dropping everything previously owned by this element. Thus, all the owning handles included in `value` become invalid.

Impact

This finding only impacts types `T` for which `drop` has some effect. In contrast, types `T` implementing the trait `Copy` are not impacted.

If `T` owns dynamically allocated memory, dropping the value deallocates the memory. The returned value would then contain invalid handles, potentially leading to unsafe memory access. Additionally, when the returned value is dropped, a second deallocation (double free) occurs:

```
#[test]
fn zellic_buffer_double_free() {
    let mut buffer = Buffer::<Box<u32>>::with_capacity(1);
    buffer.push(Box::new(10));
    // Double free occurs here, because returned value is discarded
    // > free(): double free detected in tcache 2
    buffer.pop();
}
```

In general, the impact depends on the drop function for the type `T`. With more complicated implementations, for example decreasing reference counts, the error could be harder to detect while still allowing unsafe memory access.

Recommendations

Remove the call `std::ptr::drop_in_place(ptr);`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 3a2b8341 ↗.

4.4. The Merkle commitment does not bind the number of rows

Target	slop_merkle_tree		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The tensor commitment scheme is intended to allow commitment to a rank-2 tensor and allow querying a subset of rows. The method `verify_tensor_openings` has a parameter `expected_path_len`. The argument `commit` should bind to a tensor with `2^expected_path_len` rows.

The same commitment can be used to successfully call `verify_tensor_openings` with different `expected_path_len`. This can be done by committing the root of a Merkle-like tree with leaves of different depth. For example, the left subtree of the root node can be of constant depth `a` and the right subtree of depth `b`. Therefore, the commitment does not bind to specific `expected_path_len`, and hence does not bind to a specific number of rows.

Impact

The same commitment can be opened for different numbers of rows.

In a setting where the malicious prover has to provide the claimed depth but cannot control the row indexes being queried, they still retain a significant probability of being able to construct a valid opening proof. For example, with the left subtree of depth `a` and the right subtree of depth `b`, claiming a depth of `a`, for a single random query they will have a 1/2 probability of being able to provide a proof.

Recommendations

Include the number of committed rows in the commitment. The current interface suggests that only commitments to row counts that are a power of two should be supported, but that is not necessary, even for the Merkle-tree scheme.

In the implementation of the Merkle-tree scheme, verify that the length of the paths is the ceiling of the dual logarithm of the committed row count. In all implementations, verify that the queried indexes are below the committed row count. If one of the conditions fails, return a verifier error.

For more flexible checks, remove the argument `expected_path_len` from `verify_tensor_openings`. Instead, if required, add a second method `verify_tensor_row_count(commit,expected_row_count)` that validates that commit is a

commitment to `expected_row_count` rows.

For the tensor commitment scheme, it would be possible to retrieve the metadata `row_count` from the commitment. By not exposing this method in the interface of the tensor commitment scheme and only allowing checks, one could have a more consistent interface with other commitment schemes where larger metadata is only fixed via its hash.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 353904f1↗.

4.5. The Merkle commitment does not bind the length of rows

Target	slop_merkle_tree		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The tensor commitment scheme is intended to allow commitment to a rank-2 tensor and allow querying a subset of rows.

The same commitment can be used in different calls to `verify_tensor_openings` for openings containing values with vectors of different length. While the method checks that all opened values in a single call are vectors of equal length, a second call for different indexes could return a tensor with rows of another common length.

Impact

The same commitment can be opened for different row lengths.

Recommendations

Include the length of committed rows in the commitment. In combination with Finding 4.4. ↗, this could be done by adding the dimension of the committed tensor.

In all implementations, verify that the opened rows have the committed length, and otherwise return a verifier error.

Add the second method `verify_tensor_row_length(commit,expected_row_length)` that validates that `commit` is a commitment to rows of length `expected_row_length`. In combination with Finding 4.4. ↗, this method and `verify_tensor_row_count` could be replaced by a method `verify_tensor_size(commit, index, expected_size)`, which validates that `commit` commits to a tensor with `dimensions.sizes[index] = expected_size`.

For the tensor commitment scheme, it would be possible to retrieve the metadata `row_length` from the commitment. By not exposing this method in the interface of the tensor commitment scheme and only allowing checks, one could have a more consistent interface with other commitment schemes where larger metadata is only fixed via its hash.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 353904f1↗.

4.6. Basefold FRI commitment size must be bounded by the field

Target	slop_basefold		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The function `BasefoldVerifier::verify_mle_evaluations` invokes `BasefoldVerifier::verify_queries`, which invokes `B::F::two_adic_generator` with argument `bits = points.len + self.fri_config.log_blowup`. This function is documented to have undefined behavior unless `bits < TWO_ADICITY`.

Impact

Calling the verifier with `point`, whose dimension is too large, results in undefined behavior. The implementations of the used fields (in the current version of the used library) assert that `bits <= TWO_ADICITY`. Thus, if `bits` is larger than `TWO_ADICITY`, the function panics.

In the audited code, the `BasefoldVerifier` is used in `StackedPcsVerifier`. In this context, `points.len` equals the configuration parameter `log_stacking_height`. Both verifiers are configured together in `JaggedPcsVerifier<GC, JaggedBasefoldConfig<GC>>::new`, which is called in `ShardVerifier::from_basefold_parameter`. Unless the prover can control these configuration parameters, the error would be the result of a misconfiguration, which would likely be detected during testing.

Recommendations

Add a check that the required inequality holds before calling `two_adic_generator`.

Also, the queries to the FRI tensor require that `points.len + self.fri_config.log_blowup()` is less than or equal to the number of bits of `usize`. For the targeted primes and architectures where `usize` has at least 32 bits, `TWO_ADICITY`is always smaller than the number of bits of `usize`, but the safest option would be to add both checks to prevent verifier panics / undefined behavior.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 353904f1 ↗.

The following check was added:

```
if log_len + self.fri_config.log_blowup() > GC::F::TWO_ADICITY {
    return Err(BaseFoldVerifierError::TwoAdicityOverflow);
}
```

Succinct Inc. added the following comment on the case `== TWO_ADICITY`:

> All known implementations of this trait support `bits == TWO_ADICITY`.

4.7. The `MleFoldBackend` implementation of `CpuBackend` assumes an even input element count

Target	slop_multilinear		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The function `MleFoldBackend::fold_mle` reduces the size of a multilinear polynomial by a factor of 2. The implementation for `CpuBackend` groups consecutive elements in pairs. It then sums the first pair member and a scalar multiple of the second pair member. This corresponds to partially evaluating the multilinear polynomial at the last variable.

```
// creates/multilinear/src/fold.rs
impl<F: Field> MleFoldBackend<F> for CpuBackend {
    async fn fold_mle(guts: &Tensor<F, Self>, beta: F) -> Tensor<F, Self> {
        // ...
        slop_futures::rayon::spawn(move || {
            // ...
            let fold_guts = guts
                .as_buffer()
                .par_iter()
                .step_by(2)
                .copied()
                .zip(guts.as_buffer().par_iter().skip(1).step_by(2).copied())
                .map(|(a, b)| a + beta * b)
                .collect::<Vec<_>>();
            //...
```

We found that the implementation assumes the arguments `guts` to have an even element count. If the element count is odd, it silently ignores the last element.

Impact

If the function is called with an unpadded tensor with odd element count, ignoring the last element returns an incorrect result.

Recommendations

Either assert that the number of elements of `guts` is even, or use padding to incorporate all elements of `guts` in the evaluation. Add a test case calling the function with `guts` with an odd element count.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit bac958e7 ↗. The commit adds an assert that checks that the number of elements of `guts` is even.

4.8. The opened index for the Merkle commitment is not checked to be in range

Target	slop_merkle_tree		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `MerkleTreeTcs` is a tensor commitment scheme. It allows to commit to a rank-2 tensor of field elements. The number of rows is a power of two. The commitment can be used to query rows of the committed tensor.

The interface of the function verifying the opening of the commitment is as follows:

```
// crates/merkle-tree/src/tcs.rs
impl<GC: IopCtx, M: MerkleTreeConfig<GC>> MerkleTreeTcs<GC, M> {
    pub fn verify_tensor_openings(
        &self,
        commit: &GC::Digest,
        indices: &[usize],
        opening: &MerkleTreeOpening<GC>,
        expected_path_len: usize,
    ) -> Result<(), MerkleTreeTcsError> {
```

The arguments are

- `commit`, the commitment;
- `indices`, a vector containing the indexes of the opened rows;
- `opening`, containing the claimed row values and the proof; and
- `expected_path_len`, the log_2 of the number of committed rows.

The function does not check that the elements of `indices` are smaller than the number of rows, which is `2^expected_path_len`. Instead, the function verifies the opened values at the remainder modulo `2^expected_path_len` of the given `indices`.

Impact

The commitment could be opened outside of the committed range.

In the audited code, we did not find an instance where the function was called with indexes that could be outside of the expected range.

Recommendations

Check that `indices` is smaller than `2^expected_path_len`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 5b99a974 ↗.

4.9. Assumption that the zero field element is represented by zeroed-out memory

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Some functions are intended to fill some memory with the arithmetic value zero for some field `F`. For example, consider the following function:

```
// crates/multilinear/src/eval.rs
impl<F: AbstractField> ZeroEvalBackend<F> for CpuBackend {
    fn zero_evaluations(&self, num_polynomials: usize) -> Tensor<F, Self> {
        Tensor::zeros_in([num_polynomials], *self)
    }
}
```

The implementation of `Tensor::zeros_in` is as follows:

```
pub fn zeros_in(sizes: impl AsRef<[usize]>, allocator: A) -> Self {
    let mut tensor = Self::with_sizes_in(sizes, allocator);
    tensor.storage.write_bytes(0, tensor.total_len() *
    std::mem::size_of::<T>()).unwrap();
    tensor
}
```

Note that the actual implementation does not depend on `F::zero()` but instead fills the underlying memory with zero bytes. This means that `zero_evaluations` will only behave as intended should `F::zero()` be represented by zeroed-out memory.

Similarly, the function `Mle::zeros` also zeros out the underlying memory where arithmetic zeros are expected:

```
// crates/multilinear/src/mle.rs
pub fn zeros(num_polynomials: usize, num_non_zero_entries: usize, scope: &A)
    -> Self
where
    F: AbstractField,
    A: MleBaseBackend<F>,
```

```
{
    let mut mle = Self::uninit(num_polynomials, num_non_zero_entries, scope);
    let guts = mle.guts_mut();
    let total_len = guts.total_len();
    guts.storage.write_bytes(0, total_len *
    std::mem::size_of::<F>()).unwrap();
    mle
}
```

Impact

For field types `F` for which `F::zero()` is not represented by zeroed-out memory, the behavior of those functions will be incorrect.

For fields such as `p3_koala_bear::koala_bear::KoalaBear`, zero is indeed represented by zeroed-out memory, so these functions behave as intended.

Recommendations

Consider using the value of `F::zero()` to fill the memory, rather than zero bytes. Alternatively, document the assumption that `F::zero()` must be represented by zeroed memory.

Remediation

This issue has been acknowledged by Succinct Inc. Commit 14180de3 ↗ adds documentation about the assumption that the arithmetic zero of the field is represented by zeroed memory to the `CpuBackend` implementation of `ZeroEvalBackend::zero_evaluations`, while commit 5b99a974 ↗ removes the `Mle::zeros` function.

4.10. Hypercube iteration functions for `Mle` panic on nonmaximal columns

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The following function `Mle::hypercube_iter` (and the very similar function `Mle::hypercube_par_iter`) implemented for the `CpuBackend` backend in crates/multilinear/src/mle.rs is intended to return an iterator over the evaluations of the encoded function at the hypercube $\{0,1\}^d$, with d equal to `self.num_variables()`:

```
/// Returns an iterator over the evaluations of the MLE on the Boolean
    hypercube.
///
/// The iterator yields a slice for each index of the Boolean hypercube.
pub fn hypercube_iter(&self) -> impl Iterator<Item = &[T]>
where
    T: AbstractField,
{
    let width = self.num_polynomials();
    let height = self.num_variables();
    (0..(1 << height)).map(move |i| &self.guts.as_slice()[i * width..(i + 1) *
    width])
}
```

Note that in general, `Mle` does not necessarily store all evaluations at the hypercube but instead only the first `self.num_non_zero_entries()` many, with the remaining to be interpreted as filled with padding (by default with zero).

The functions `Mle::hypercube_iter` and `Mle::hypercube_par_iter` should thus support `self.guts` in the described form. However, only the special case `self.num_non_zero_entries() = 1 << self.num_variables()` is supported; when `self.num_non_zero_entries() < 1 << self.num_variables()`, the slice `self.guts.as_slice()` will be indexed beyond its length, resulting in a panic.

Impact

The functions `Mle::hypercube_iter` and `Mle::hypercube_par_iter` will panic unless `self.num_non_zero_entries() = 1 << self.num_variables()`. The following test

demonstrates this:

```
#[test]
fn zellic_test_mle_hypercube_iter() {
    let mut rng = rand::thread_rng();

    type F = BabyBear;

    let guts: Tensor<F, CpuBackend> = Tensor::rand(&mut rng, [3, 5]);

    let mle = Mle::<F>::new(guts);

    let data: Vec<&[BabyBear]> = mle.hypercube_iter().collect();

    println!("{:?}", data);
}
```

As of the time of writing, these functions appear to only be used in the basefold prover, which is out of scope for this audit. We did not investigate whether a call from the prover could occur with `self.num_non_zero_entries() < 1 << self.num_variables()`. As verifiers that are in scope do not call these functions, there is no impact on verifiers in the current codebase.

Recommendations

Given the current documentation comments, `Mle::hypercube_iter` and `Mle::hypercube_par_iter` should likely pad the evaluations stored in `self.guts` with zero to the required length. However, we recommend to also check in calling code what convention makes most sense for cases in which the `self.num_non_zero_entries() = 1 << self.num_variables()` is not satisfied, such as whether padding values should be passed to `Mle::hypercube_iter` and `Mle::hypercube_par_iter` instead of padding with zero.

Alternatively, if these functions are only needed in the case where `self.num_non_zero_entries() = 1 << self.num_variables()`, document this assumption and that the function will panic otherwise.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 534ac7ae ↗. The commit adds an assert to ensure `self.num_non_zero_entries() == 1 << self.num_variables()` and documents that the function panics if this is not satisfied.

4.11. The function `Mle::iter` behaves incorrectly on some input shapes

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `MleEval` struct is intended to store a batch of evaluations of multilinear polynomials, a one-dimensional array of values:

```
/// A bacth of multi-linear polynomial evaluations.
#[derive(Debug, Clone)]
#[derive_where(PartialEq, Eq, Serialize, Deserialize; Tensor<T, A>)]
pub struct MleEval<T, A: Backend = CpuBackend> {
    pub(crate) evaluations: Tensor<T, A>,
}
```

Documentation comments for this function implemented for `MleEval` suggest that the shapes allowed for the underlying `Tensor` are

- rank one
- rank two, with the first axis of dimension one
- rank two, with the second axis of dimension one

```
/// It is expected that `self.evaluations.sizes()` is one of the three options:
/// `[1, num_polynomials]`, `[num_polynomials,1]`, or `[num_polynomials]`.
#[inline]
pub fn num_polynomials(&self) -> usize {
    self.evaluations.total_len()
}
```

However, the following function behaves inconsistently across these different encodings of evaluation values:

```
pub fn iter(&self) -> impl Iterator<Item = &[T]> + '_ {
    self.evaluations.split().map(|t| t.as_slice())
}
```

Note that `split()` splits according to the first axis. Hence, if the tensor has dimensions `[d]` or

`[d,1]`, this will result in an iterator of `d` values, each of which are a slice of length `1`. If instead the tensor has dimension `[1,d]`, then an iterator over a single element, a slice of length `d`, will be returned.

Impact

The `MleEval` struct appears intended to support stored data in different shapes, but `MleEval::iter` has behavior that depends on the shape. Note that `iter` is implemented specifically for `CpuBackend`, so it could be that `CpuBackend` does not support this shape. This is not documented as of the time of writing, however, and is not otherwise obvious.

The following test can be used to observe the different behavior:

```
#[test]
fn zellic_test_mle_eval_iter() {
    let mut rng = rand::thread_rng();

    type F = BabyBear;

    let guts: Tensor<F, CpuBackend> = Tensor::rand(&mut rng, [1, 5]);

    let e = MleEval::<F>::new(guts);

    let data: Vec<&[BabyBear]> = e.iter().collect();

    println!("{:?}", data);
}
```

Should `MleEval` with an underlying tensor of shape `[1,d]` for `d > 1` be created, using `iter` without flattening the result can lead to incorrect behavior. We have not identified any concrete such instances within scope.

Recommendations

We recommend to review how `iter` is used and adjust its implementation to behave correctly for all supported input shapes.

Alternatively, document that the shape `[1,d]` with `d>1` is not supported, and verify that no `MleEval` with such a shape is created.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit add52cb9 ↗, in which the explicit `iter` function was removed, thereby causing `iter` to return a flat

`Iterator` over the underlying values via dereferencing to `[T]`.

4.12. Function `VirtualGeq::sum` incorrect for `self.threshold = 1<<self.num_vars`

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `VirtualGeq` struct encapsulates information about a particular type of linear combination of `eq` and `geq` polynomials, allowing to compute various associated data (such as evaluations) without computing and storing the evaluation on the entire hypercube first, as would be the case when using generic functions:

```
/// A struct capturing a dense representation of a linear combination of a geq
    and eq polynomial,
/// both with the same threshold and number of variables.
///
/// In terms of "guts", a `VirtualGeq` is a
/// vector of length `2^num_vars` where the first `threshold` entries are zero,
    the next entry is
/// `eq_coefficient + geq_coefficient`, and the rest are `geq_coefficient`.
    (In the edge case
/// threshold == 2^num_vars, this means the vector consists of all zeroes.)
#[derive(Debug, Copy, Clone)]
pub struct VirtualGeq<F> {
    pub threshold: u32,
    pub geq_coefficient: F,
    pub eq_coefficient: F,
    pub num_vars: u32,
}
```

Note that `threshold = 1 << num_vars` is explicitly documented as allowed, with various functions handling this case explicitly and the constructor that checks consistency of `threshold` with `num_vars` including it as well:

```
pub fn new(threshold: u32, geq_coefficient: F, eq_coefficient: F, num_vars:
    u32) -> Self {
    assert!(threshold <= (1 << num_vars));
    Self { threshold, eq_coefficient, geq_coefficient, num_vars }
}
```

However, the `sum` function behaves incorrectly in the case `threshold = 1 << num_vars`:

```
/// Sum all entries in the virtual polynomial.
pub fn sum(&self) -> F {
    F::from_canonical_usize((1 << self.num_vars) - self.threshold as usize)
        * self.geq_coefficient
        + self.eq_coefficient
}
```

In this case, all evaluations on the hypercube are zero, so the sum should be zero, while `self.eq_coefficient` is returned.

Impact

The function `VirtualGeq::sum` will compute an incorrect result should it be called on a `VirtualGeq` with `threshold = 1 << num_vars`.

We did not find any use for this function in the current codebase (as of the time of writing).

Recommendations

The summand `self.eq_coefficient` should be added only when `threshold < 1 << num_vars`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit e6613003 ↗. The fix removes the `VirtualGeq::sum` function.

4.13. Function `VirtualGeq::eval_at_usize` incorrect for `self.threshold < index < (1 << self.num_vars)`

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `virtual_geq::VirtualGeq<F>::eval_at_usize` is expected to return the linear combination of the boolean functions `index == self.threshold` and `index >= self.threshold` (interpreted as functions with values $\{0, 1\}$) with coefficients `self.eq_coefficient` and `self.geq_coefficient`.

```
// crates/multilinear/src/virtual_geq.rs
impl<F: Field> VirtualGeq<F> {
    // ..
    pub fn eval_at_usize(&self, index: usize) -> F {
        assert!(index <= (1 << self.num_vars));
        if index < self.threshold as usize {
            F::zero()
        } else if index == self.threshold as usize {
            self.eq_coefficient + self.geq_coefficient
        } else if index < (1 << self.num_vars) {
            F::one()
        } else {
            F::zero()
        }
    }
```

The function does not return the correct value if `self.threshold < index < (1 << self.num_vars)`.

Impact

The function is not used in the audited code.

Recommendations

Change the return value in the case `self.threshold < index < (1 << self.num_vars)` to `self.geq_coefficient`:

```
// crates/multilinear/src/virtual_geq.rs
impl<F: Field> VirtualGeq<F> {
    // ..
    pub fn eval_at_usize(&self, index: usize) -> F {
        // ..
        } else if index < (1 << self.num_vars) {
            F::one()
            self.geq_coefficient
        // ..
    }
```

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 8450db90 ↗.

4.14. Function `VirtualGeq::fix_last_variable` should require self to have at least one variable

Target	slop_multilinear		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The struct `VirtualGeq` represents a specific type of multilinear polynomial in `num_vars` variables. If a `VirtualGeq` instance represents the multilinear polynomial $f(X_1, \ldots, X_{\texttt{num\_vars}})$, its method `fix_last_variable` with argument `alpha` is intended to return a new `VirtualGeq` representing the multilinear polynomial $f(X_1, \ldots, X_{\texttt{num\_vars}-1}, \texttt{alpha})$ in `num_vars - 1` variables.

```
// crates/multilinear/src/virtual_geq.rs
pub struct VirtualGeq<F> {
    // ..
    pub num_vars: u32,
    // ..
}
impl<F: Field> VirtualGeq<F> {
    // ..
    pub fn fix_last_variable<EF: ExtensionField<F>>(&self, alpha: EF) ->
    VirtualGeq<EF> {
```

This requires the polynomial `f` to have at least one variable.

We found that `VirtualGeq::fix_last_variable` does not check that `num_vars` is at least 1.

Impact

If `num_vars` is 0, the method again returns a `VirtualGeq` with zero variables.

There is no clear expected or documented behavior of `VirtualGeq::fix_last_variable` when `num_vars` is 0. Calling it this way is likely the result of a coding mistake in the caller. This can lead to undetected errors in the calling code.

If the call is intended, this makes the caller dependent on unexpected and undocumented behavior of the implementation of `VirtualGeq::fix_last_variable`.

In the audited code, we did not find an instance where the method was called when `num_vars` is 0.

Recommendations

Assert that `num_vars` is at least `1`.

Remediation

This issue has been acknowledged by Succinct Inc., and a fix was implemented in commit 8450db90 ↗.

5. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

5.1. Key investigation points

This section collects the results for the key investigation points specified by Succinct Inc.

Jagged shape

> The computational trace, which has a sparse "jagged" shape, is committed in a dense form and supplemented by heights that allow the shape to be recovered. Is the distinction of "commits" (which only commits to buffer) and "commits + heights" (which is the full information of the situation at hand) done correctly?

In response to Finding 4.2. ↗ and more generally the considerations described in section 5.2. ↗, Succinct Inc. refactored the code and combined the buffer and shape commits in the jagged commitment. Thus, in the final revision of the code this audit refers to, the separation was abandoned.

In the audited version, the jagged commitment not only binds the set of used cells in the jagged shape but additionally contains the rectangle decomposition of that shape corresponding to areas used by individual chips. Due to this inclusion, the additional commitment to the heights is redundant, because the heights are committed to as the heights of the individual rectangles.

> In general, whether the shape handling is done correctly, including:
>
> - Is there a way to sneak in a column with height larger than `1 << max_log_row_count`?
> - Is there a way to make the total area not a multiple of `1 << log_stacking_height`?
> - Is there a way to make use of padding columns adversarially?

We reported an issue with padding columns in Finding 4.2. ↗. In Finding 3.10. ↗, we observed that for small total areas, opening rows exceeding the committed height can lead to row repetitions instead of padding.

The implemented padding enforces that the total area in each jagged round is a multiple of `1 << log_stacking_height`.

The heights of the chip openings are verified to match the rectangle heights of the jagged shape, and these heights are verified to be bounded by `1 << max_log_row_count`, but see Finding 3.29. ↗

for a recommendation increasing the clarity of that check.

> Make sure the prefix sums are checked for correctness sufficiently, and that these checks are done correctly, and that they ensure the maximum shard size of at most 2^{30} elements.

The area of each jagged round is enforced to be smaller than 2^{30} in the following code segment:

```
// slop/crates/jagged/src/verifier.rs
let round_areas: Vec<usize> = row_counts
    .iter()
    .zip(column_counts.iter())
    .map(|(rc, cc)| {
        // The counts have been checked above to be at least length 2, so it's
    safe to subtract 2.
        let rc_len = rc.len() - 2;
        let cc_len = cc.len() - 2;
        rc.iter()
            .take(rc_len)
            .zip_eq(cc.iter().take(cc_len))
            .map(|(r, c)| r.saturating_mul(*c))
            .fold(0usize, |a, b| a.saturating_add(b))
    })
    .collect();

if round_areas.iter().any(|&area| area == 0 || area >= (1 << 30)) {
    return Err(JaggedPcsVerifierError::AreaOutOfBounds);
}
```

This is done with saturating arithmetic to avoid overflows in the verifier computation. Then the total area of all rounds is checked as well:

```
// slop/crates/jagged/src/verifier.rs
let usize_column_heights: Vec<usize> = row_counts
    .iter()
    .zip_eq(column_counts.iter())
    .flat_map(|(rc, cc)| {
        rc.iter().zip_eq(cc.iter()).flat_map(|(r, c)| std::iter::repeat_n(*r,
    *c))
    })
    .collect();

let mut usize_prefix_sums: Vec<usize> = usize_column_heights
    .iter()
    .scan(0usize, |state, &x| {
        let result = *state;
```

```
            *state += x;
            Some(result)
        })
        .collect();

usize_prefix_sums
        .push(*usize_prefix_sums.last().unwrap() +
        *usize_column_heights.last().unwrap());

let log_m = log2_ceil_usize(usize_prefix_sums.last().copied().unwrap());

if log_m >= 30 {
        return Err(JaggedPcsVerifierError::AreaOutOfBounds);
}
```

This time, no saturating arithmetic is used, but the already checked limit on the individual rounds' areas means that an overflow in the verifier arithmetic at this point (on a 64-bit machine) would require 2^{34} rounds. In the shard verifier, only two rounds are used; thus, no overflow can occur. This segment then checks that the total verifier area is at most 2^{29}, (which is smaller than the 2^{30} that was asked for).

Fiat–Shamir

> Whether the Fiat-Shamir transformation of the interactive verification protocol to the non-interactive version was done correctly.

In Finding 3.13. ↗, we explain a failure in the Fiat–Shamir transformation due to variable-length observations.

Findings 3.1. ↗, 3.2. ↗, and 3.9. ↗ describe missing observations at the beginning of the verification protocol. A relevant point for the Fiat–Shamir transformation is that the challenger that replaces the random oracle in the idealized protocol needs to be domain-separated. This can be achieved by seeding the challenger with observations of the parameters binding the domain, which in this case includes the chip definitions.

Lookup overflow

> Whether the logic to prevent lookup multiplicity overflow is correct.

Succinct Inc. supplied the following additional information:

> Currently, our AIRs & interactions system satisfy the following — for each interaction kind, either
>
> - all multiplicity of interactions are guaranteed to be boolean
> - there's a preprocessed table that can receive any amount, and main table that can send boolean multiplicity (like byte/program lookup)
>
> … This fact by itself is guaranteed from the AIRs, so it's not a part of the audit scope — but you can assume that this is true for the sake of the audit. … The main width (number of main column) of each chip is no less than the total number of interactions of a given kind. Therefore, for each kind, the total number of interactions happening must be at most the shard size, at most 2^{30}. Notably, there are some interactions in [the] `eval_public_values` function, but those are only $\mathcal{O}(1)$ interactions anyway, so [they] should not affect the overflow (im)possibility.

The audited code does not contain a verification that the number of send interactions in `eval_public_values` does not exceed `p-2^30`; thus we assume that this is guaranteed by the machine that defines these interactions. Under these assumptions on the AIRs, the design verifies that each row that is sent to a lookup table is also received. It is possible that the receiving amount overflows, which would entail that more rows are received than sent. This includes the case in which rows are received that are never sent. However, this is intended in the design. Any deviations from this design in the implementation are noted in the findings. For example, Finding 3.2. ↗ impacts the soundness of the verification.

Verifier panics

> Can the verifier panic when given an incorrectly formatted proof?

We found multiple instances of panics that could be triggered by the prover; see Findings 3.5. ↗, 3.6. ↗, 3.17. ↗, 3.14. ↗, 3.11. ↗, 3.7. ↗, and 3.18. ↗.

Variable-length structures

> Are any checks on the length and structure of variable length structures missing?

This question was originally asked in the context of verifier panics arising due to calls to `zip_eq` on iterators of different length. We did not find any instances of this specific type of panic. However, some of the panics listed in the subsection above arise in cases where a variable-length structure has length `0`, and Finding 3.5. ↗ is also about a length-related verifier panic.

In the initial code version before the refactor (see the discussion in section 5.2. ↗), we found that commitments to variable-length structures did not bind the length; see Findings 4.4. ↗ and 4.5. ↗.

The audited code contained a weaker version of this type of finding (Finding 3.23. ↗).

A general finding regarding the observations of variable-length structures is described in section 3.13. ↗.

The crate `slop_multilinear` is not clear about its length assumptions on its tensor arguments. This is described in detail in Finding 3.16. ↗. While the (possibly) missing checks do not impact the shard verifier, the crate as a library could help to prevent unintended usage on structures of different length by documenting its assumptions and asserting the required equalities.

5.2. Recommendation of a refactor with regards to separation of different components

During the audit, we recommended Succinct Inc. to refactor parts of the code under review. Such a refactor was then performed by Succinct Inc., and we continued the review on the new, refactored commit. In this section, we document some of the recommendations we made in connection to the refactor. The recommendation includes design that helps to improve the code's security posture. In general, checking adherence to these specific suggestions goes beyond the audit's goals. While we mention violations of the principles in findings if they contributed to a finding's cause, we did not independently audit whether the goals of the refactor were reached.

Encapsulation

The implemented protocol as a whole consists of various different building blocks. The dependencies between those can be several levels deep. For example, the shard verifier uses a jagged multilinear PCS verifier, which uses a verifier for a single large-dimension multilinear PCS (the stacked one), which in turn uses a verifier for batches of fixed-size multilinear polynomials (basefold), which in turn uses the Merkle-tree tensor commitment scheme. Separating such components cleanly into building blocks has the advantage that it becomes easier to reason about the system and mistakes are less likely. For example, on the theoretical level, it is easier to reason about the soundness of some protocol that uses some other protocols as components if one thinks of those components as black boxes satisfying a nicely packaged soundness property themselves, rather than unpacking everything to the lowest-level description of what concrete instances do. Similarly, for implementing or auditing, it is easier to keep the necessary context in mind if subcomponents are neatly encapsulated.

An ideal verifier function in this sense might thus have the shape of taking as arguments a single `proof` of appropriate type, a `challenger`, and then some other data. This other data should encode some statement that the verifier is to check, and the proof should then be the fully untrusted data sent by the prover that the verifier uses to check the statement (up to a soundness error). This proof should then generally not be something that higher-level protocols that use this verifier as a component make use of; instead, it should usually be considered as an opaque blob that only has meaning for the verifier it is intended for. An interface of this form helps cleanly separate things; anything a caller of the verifier function would need to know is contained in the other arguments

that specify the statement,[4] and the proof contains exactly the fully untrusted information by the prover that it needs to verify whatever guarantees it makes regarding that statement.[5]

Before the refactor, such encapsulation was present in the code on the level of separation into crates, files, and `impl`s. However, the actual implementation broke this separation in many places. For example, consider this snippet from the shard verifier prior to the refactor:

```
let preprocessed_round_num_poly =
    (proof.evaluation_proof.stacked_pcs_proof.batch_evaluations.rounds[0]
        .iter()
        .map(slop_multilinear::MleEval::num_polynomials)
        .sum::<usize>()) as u32;
```

Here, the shard verifier not only breaks this separation by reaching into the jagged PCS proof but breaks separation even one level further by reaching into the jagged PCS proof's encapsulated stacked PCS proof. This could suggest that a check that should be done at a lower level is missing at that level — or that the statement of the lower-level verifiers is not conceptualized in a suitable way to separate statement and proof.

The way concretely the stacked, jagged, and shard verifiers' pre-refactor interacted with each other's proofs made it harder than necessary to reason about them. For example, Finding 4.2. ↗ might have been less likely to occur with cleaner separation; the importance of checks on `added_columns` was realized in the codebase, as can be seen from the fact that the shard verifier has checks associated to it. One direct issue with this was that these checks were then missing in the jagged verifier itself, where they would have been needed also on independent usage not involving the shard verifier. Additionally, spreading such checks over two fairly complicated verifiers, which are, on their own, already difficult to reason about, increases the difficulty in making sure those checks cover everything.

Unnecessary batching layers

The multilinear PCS verifiers (like basefold) use three layers of batching polynomials in the pre-refactor version:

The top level is the slices for the `commitments` and `evaluation_claims`:

```
fn verify_trusted_evaluations(
    &self,
    commitments: &[Self::Commitment],
    point: Point<Self::EF>,
    evaluation_claims: &[Evaluations<Self::EF>],
    proof: &Self::Proof,
    challenger: &mut Self::Challenger,
```

[4] The verifier may make assumptions about this data that the caller needs to check or arrange.
[5] This is assuming some preconditions on the other arguments are satisfied.

```
) -> Result<(), Self::VerifierError>;
```

This batching layer is needed if this is to be used in a larger protocol where the prover may need to commit to polynomials at different times, as is the case for the SP1 shard verifier, where some commitments are fixed already in the verification key (preprocessed tables) and others only later provided by the prover for an individual proof.

There is also at least one more layer needed if we would like to commit to more than one polynomial at once.

If we go down the `Evaluations`, we have another middle-level batching layer, the `Vec` here:

```
pub struct Evaluations<F, A: Backend = CpuBackend> {
    pub round_evaluations: Vec<MleEval<F, A>>,
}
```

And then finally, the bottom-level batching layer is that each `MleEval` can contain a flat list of possibly more than one evaluation.

The bottom two batching layers were flattened, with the shape not actually being used. The unnecessary middle batching layer risked potential confusion where one may think that the shape associated to the two bottom batching layers might be partly checked against the commitment, while this was not actually the case.

Recommendations

For the reasons described above, we recommended a refactor to more cleanly separate various components, in particular the stacked, jagged, and shard verifiers. We also recommended to remove the middle batching layer where it does not play a role. A refactor with goals along the lines of some of these recommendations was then carried out by Succinct Inc.

5.3. Assumptions made for the shard verifier

The top-level verifier reviewed as part of this audit was `ShardVerifier::verify_shard` in crates/hypercube/src/verifier/shard.rs. Its signature is as follows:

```
// crates/hypercube/src/verifier/shard.rs
pub fn verify_shard(
    &self,
    vk: &MachineVerifyingKey<GC, C>,
    proof: &ShardProof<GC, C>,
    challenger: &mut GC::Challenger,
) -> Result<(), ShardVerifierConfigError<GC, C>>
```

In our review, we in particular assumed the following:

- That `challenger` has observed `vk.preprocessed_commit` before
- That `self.jagged_pcs_verifier.max_log_row_count < 29`
- That all chips in `self.machine.chips()` have only interactions and constraints that reference in-range (with respect to the to the chips' `width()` and `preprocessed_width()`) columns
- That the machine only has constraints and interactions referencing the first `self.machine.num_pv_elts()` public values and do not assume any specific length of the public-values vector passed (as in, they will work fine if it is longer — it will index from the start everywhere)
- That it holds that `2^30 < p`
- That `vk.preprocessed_chip_information` should include all chips that occur in `vk.preprocessed_commit`, with the correct shapes

In addition, the verifier requires that both preprocessed area and main area are nonempty, with the total area supported being allowed to be at most 2^{29}.

5.4. Basefold

The implemented basefold algorithm differs from the variants that can be found in the literature. The original implementation from Zeilberger, Chen, and Fisch, *BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes From Foldable Codes* ↗, uses FRI to commit a univariate polynomials with coefficients corresponding to the coefficients of a multilinear polynomial (see page 19, paragraph "Notation"). In *Basefold in the List Decoding Regime* ↗ by Haböck, on page 17 (equation 1), the FRI commitment instead uses a univariate polynomial whose coefficients correspond to the evaluations of a multilinear polynomial, but this source fails to alter either the protocol or FRI folding (pages 8 and 9) accordingly, so the protocol described there does not work as written.

The implemented algorithm in the audited code uses the same FRI commitment as the second source but alters the protocol to convert evaluations to coefficients by a projective coordinate change. The general protocol structure is the same, and we found no impact on the security properties of the protocol. In this section, we describe the details of the implemented algorithm.

Notation

Fix a finite field K. The space of linear polynomials in one variable $K[X]_{\leq 1}$ is linearly isomorphic to the space of pairs K^2. The implemented basefold algorithm makes use of the following two isomorphisms $M_0, M_1 : K^2 \to K[X]_{\leq 1}$:

$$(M_0(f_0, f_1))(X) = f_0 \cdot (1 - X) + f_1 \cdot X$$

$$(M_1(f_0, f_1))(X) = f_0 + f_1 \cdot X$$

These maps are inverse to evaluating at 0 and 1 and taking the coefficients of a linear polynomial, respectively. Now let $I_n = \{0,1\}^n$ be the n-dimensional boolean hypercube. We denote its elements as vectors $i = (i_0, \ldots, i_{n-1})$ starting with index 0. We can generalize the isomorphisms above to a sequence $M_0, \ldots, M_n : K^{I_n} \to K[X_0, \ldots, X_{n-1}]_{\leq 1}$ of isomorphisms:

$$(M_k f)(X_0, \ldots, X_{n-1}) = \sum_{i \in I_n} f_i \cdot \left(\prod_{j=0}^{k-1} X_j^{i_j} \right) \cdot \left(\prod_{j=k}^{n-1} X_j^{i_j} (1 - X_j)^{1-i_j} \right)$$

These isomorphisms interpolate between M_0, which is the inverse to evaluating on the boolean hypercube, and M_n, which is the inverse to taking coefficients. For $0 \leq k < n$, the two neighboring isomorphisms M_k and M_{k+1} are related by a projective coordinate change in the k-th coordinate:

$$(M_{k+1} f)(X_0, \ldots, X_{n-1}) = (1 + X_k) \cdot (M_k f)(X_0, \ldots, X_{k-1}, \frac{X_k}{1 + X_k}, X_{k+1}, \ldots, X_{n-1})$$

The protocol

We describe the implemented scheme for a commitment to a single multilinear polynomial, without the implemented reversal of coordinate indexes.

Let L be a positive integer (the log blowup), and $z \in K$ a 2^{n+L}-th primitive root of unity in K.

The prover commits to the multilinear polynomials $M_0 f$ by sending the FRI vector $(U_0(z^i))_{i=0}^{2^{n+L}-1}$, where U_0 is defined as

$$U_0(X) = (M_n f)(X, X^2, \ldots, X^{2^{n-1}})$$

For a claimed evaluation v at a query point $\omega \in K^n$, the prover and verifier interact in n rounds, where in round $k \in \{0, \ldots, n-1\}$ the following occurs:

- The prover sends the *univariate message* $(a_k, b_k) \in K^2$, where
$$a_k = (M_k f)(\beta_0, \ldots, \beta_{k-1}, 0, \omega_{k+1}, \ldots, \omega_{n-1})$$
$$b_k = (M_k f)(\beta_0, \ldots, \beta_{k-1}, 1, \omega_{k+1}, \ldots, \omega_{n-1})$$
- The verifier sends a random $\beta_k \in K$.
- The prover sends the folded FRI vector $(U_{k+1}(z^{2^k \cdot i}))_{i=0}^{2^{n+L-k-1}-1}$, where U_{k+1} is defined as follows.

$$U_{k+1}(X) = (M_n f)(\beta_0, \ldots, \beta_k, X, X^2, \ldots, X^{2^{n-2-k}})$$

Note that (a_k, b_k) are both the evaluations of $(M_k f)(\beta_0, \ldots, \beta_{k-1}, X, \omega_{k+1}, \ldots, \omega_{n-1})$, by definition, and the coefficients of $(M_{k+1} f)(\beta_0, \ldots, \beta_{k-1}, X, \omega_{k+1}, \ldots, \omega_{n-1})$, by the relation between M_k and M_{k+1}.

$$a_k \cdot (1 - X) + b_k \cdot X = (M_k f)(\beta_0, \ldots, \beta_{k-1}, X, \omega_{k+1}, \ldots, \omega_{n-1})$$

$$a_k + b_k \cdot X = (1 + X) \cdot \left(a_k \cdot (1 - \frac{X}{1+X}) + b_k \cdot \frac{X}{1+X} \right)$$

$$= (1 + X) \cdot (M_k f)(\beta_0, \ldots, \beta_{k-1}, \frac{X}{1+X}, \omega_{k+1}, \ldots, \omega_{n-1})$$

$$= (M_{k+1} f)(\beta_0, \ldots, \beta_{k-1}, X, \omega_{k+1}, \ldots, \omega_{n-1})$$

Now the verifier checks the following implied consistency relations:

$$v = a_0 \cdot (1 - \omega_0) + b_0 \cdot \omega_0 \, (= (M_0 f)(\omega_0, \ldots, \omega_{n-1}))$$

$$\forall 0 \leq k < n - 1 : a_k + b_k \cdot \beta_k = a_{k+1} \cdot (1 - \omega_{k+1}) + b_{k+1} \cdot \omega_{k+1}$$

$$(= (M_{k+1} f)(\beta_0, \ldots, \beta_k, \omega_{k+1}, \ldots, \omega_{n-1}))$$

Via random FRI queries, the following relations are checked:

$$a_{n-1} + b_{n-1} \cdot \beta_{n-1} = U_n \, (= (M_n f)(\beta_0, \ldots, \beta_{n-1}))$$

$$\forall 0 \leq k < n : U_{k+1}(X^2) = U_{k,even}(X^2) + \beta_k \cdot U_{k,odd}(X^2)$$

$$\Big(= (M_n f)(\beta_0, \ldots, \beta_{k-1}, 0, X^2, \ldots, X^{2^{n-1-k}})$$

$$+ \beta_k \cdot ((M_n f)(\beta_0, \ldots, \beta_{k-1}, 1, X^2, \ldots, X^{2^{n-1-k}}) - (M_n f)(\beta_0, \ldots, \beta_{k-1}, 0, X^2, \ldots, X^{2^{n-1-k}})) \Big)$$

Soundness heuristic

The soundness of the procotol relies on the following arguments:

- By FRI, the prover committed to a polynomial U_0 of degree at most $2^n - 1$. If we let f denote the coefficients of that polynomial, FRI additionally verifies $U_n = (M_n f)(\beta_0, \ldots, \beta_{n-1})$.
- Because a_k, b_k were chosen before β_k, the identity $a_k + b_k \cdot \beta_k = a_{k+1} \cdot (1 - \omega_{k+1}) + b_{k+1} \cdot \omega_{k+1}$ inductively (backwards) verifies that

$$a_k = (M_k f)(\beta_0, \ldots, \beta_{k-1}, 0, \omega_{k+1}, \ldots, \omega_{n-1})$$

$$b_k = (M_k f)(\beta_0, \ldots, \beta_{k-1}, 1, \omega_{k+1}, \ldots, \omega_{n-1})$$

- Finally, this yields the following.

$$v = a_0 \cdot (1 - \omega_0) + b_0 \cdot \omega_0 = (M_0 f)(\omega_0, \ldots, \omega_{n-1})$$

5.5. LogUp GKR

The audited code implements a variant of the logUp GKR algorithm described in *Improving Logarithmic Derivative Lookups Using GKR* ↗ by Papini and Haböck. In this section, we describe the relation of the implemented version to the algorithm described in the source as well as a possible optimization arising from the difference.

Description of the algorithm

The paper considers the lookup verification of a single lookup table with a single column. The rows are indexed by the elements of a boolean hypercube.

The implemented version is an extension in several ways. The lookup tables are distributed over several hypercubes, one for each chip. Additionally, the height of each chip's trace limits the used rows, and padding is required. Also, multiple lookup tables with multiple columns each are used. For notational convenience, we ignore the additional trace independent entries depending on public values only in this section.

To formalize this, we fix some notation. Let $I(a)$ denote the number of columns of the virtual interaction table a. Denote the number of chips by N. For each chip $0 \leq i < N$, we have the following data.

- Its height H_i and total width W_i (the sum of the main width and preprocessed width)
- For $0 \leq j < H_i, 0 \leq k < W_i$, the trace $t(i, j, k)$ in row j and column k — the trace is the concatenation of preprocessed and main trace, and we write $t(i, j)$ for the row vector
- The counts S_i and R_i of the send and receive interactions, respectively
- For each $0 \leq l < S_i$, a send interaction that is given by the following data:
    - The argument index $a^s_{i,l}$ indicating the interaction tables this is sent to
    - The coefficients $m^{s,c}_{i,l}, m^s_{i,l,k} (0 \leq k < W_i)$ defining the affine function, which, applied to the columns, yields the multiplicity
    - For $0 \leq k' < I(a^s_{i,l})$, the value coefficients $v^{s,c}_{i,l,k'}, v^s_{i,l,k,k'} (0 \leq k < W_i)$ defining the affine function for the k'-th virtual column of the interaction
- For the receive interactions, values $a^r_{i,l}, m^{r,c}_{i,l}, m^r_{i,l,k}, v^{r,c}_{i,l,k'}, v^r_{i,l,k,k'}$ analogous to the send interactions

We write $m^s(\cdot), m^r(\cdot), v^s(\cdot), v^r(\cdot)$ for the affine functions given by the above coefficients:

$$m^s_{i,l}(X) = m^{s,c}_{i,l} + \sum_{0 \leq k < W_i} m^{s,c}_{i,l,k} X_k$$

$$m^r_{i,l}(X) = m^{r,c}_{i,l} + \sum_{0 \leq k < W_i} m^{r,c}_{i,l,k} X_k$$

$$v^s_{i,l}(X)_{k'} = v^{s,c}_{i,l,k'} + \sum_{0 \leq k < W_i} v^{s,c}_{i,l,k,k'} X_k$$

$$v^r_{i,l}(X)_{k'} = v^{r,c}_{i,l,k'} + \sum_{0 \le k < W_i} v^{r,c}_{i,l,k,k'} X_k$$

For each interaction a and values $v = (v_0, \ldots, v_{I(a)-1})$, the goal is to verify that

$$0 = \sum_{\substack{0 \le i < N \\ 0 \le j < H_i}} \left(\sum_{\substack{0 \le l < S_i: \\ a^s_{i,l} = a, \text{ and} \\ v^s_{i,l}(t(i,j)) = v}} m^s_{i,l}(t(i,j)) - \sum_{\substack{0 \le l < R_i: \\ a^r_{i,l} = a, \text{ and} \\ v^r_{i,l}(t(i,j)) = v}} m^r_{i,l}(t(i,j)) \right)$$

This is an equation over a finite field. If the multiplicities do not overflow, this guarantees that for each entry in the lookup table, the sum of all send and receive multiplicities agree.

Via random challenges α, β (called `beta_seed` in the implementation), this is reduced to the verification of a random linear combination of the multiplicities,

$$0 = \sum_{\substack{0 \le i < N \\ 0 \le j < H_i}} \left(\sum_{0 \le l < S_i} \frac{m^s_{i,l}(t(i,j))}{d^s_{i,l}(t(i,j))} - \sum_{0 \le l < R_i} \frac{m^r_{i,l}(t(i,j))}{d^r_{i,l}(t(i,j))} \right)$$

where $d^*_{i,l}(t(i,j)) = \alpha + (M_0(a^*_{i,l}, v^*_{i,l}(t(i,j))))(\beta)$ is the interaction digest, and $(M_0 f)(p)$ evaluates the multilinear polynomial with hypercube evaluations f (zero-padded) at the point p. In this case, $1 + I(a^*_{i,l})$ hypercube evaluations are given, with the remainder of the hypercube padded with zero. The point β must have high enough dimension $\dim(\beta)$ so that $2^{\dim(\beta)} \ge 1 + I(a^*_{i,l})$.

The implemented algorithm now first lets the prover claim an evaluation for each chip interaction:

$$\frac{p^s_{i,l}}{q^s_{i,l}} = \sum_{0 \le j < H_i} \frac{m^s_{i,l}(t(i,j))}{d^s_{i,l}(\alpha, \beta)}$$

$$\frac{p^r_{i,l}}{q^r_{i,l}} = -\sum_{0 \le j < H_i} \frac{m^r_{i,l}(t(i,j))}{d^r_{i,l}(\alpha, \beta)}$$

It is then verified that these claims add up as expected. The individual claims above are then combined via a random challenge `Y` (called `interaction_point` in the code) and reduced to evaluation claims of the following two multilinear polynomials at a point `X` (called `trace_point` in the code),

$$P(X, Y) = M_0[M(X)](Y)$$

$$Q(X, Y) = M_0[D(X)](Y)$$

where $T(i, X)_k = (M_0[t(i,j,k) | 0 \le j < H_i])(X)$ is the trace row at a generic row index X,

$$\tilde{m}^*_{i,l}(X) = (M_0[m^*_{i,l}(t(i,j,k))|0 \le j < H_i])(X) = m^*_{i,l}(T(i,X)) - \texttt{geq}(X, H_i)m^*_{i,l}(0)$$

is the 0-padded multiplicity at the generic row index X,

$$\tilde{d}^*_{i,l}(X) = (M_0[d^*_{i,l}(t(i,j,k))|0 \le j < H_i])(X) = d^*_{i,l}(T(i,X)) - \texttt{geq}(X, H_i)(d^*_{i,l}(0) - 1)$$

is the 1-padded interaction digest at the generic row index X, and

$$M(X) = [\tilde{m}^s_{0,0}(X), \ldots, \tilde{m}^s_{0,S_0-1}(X), -\tilde{m}^r_{0,0}(X), \ldots, -\tilde{m}^r_{0,R_0-1}(X), \ldots$$
$$\tilde{m}^s_{N-1,0}(X), \ldots, \tilde{m}^s_{N-1,S_{N-1}-1}(X), -\tilde{m}^r_{N-1,0}(X), \ldots, -\tilde{m}^r_{N-1,R_{N-1}-1}(X)]$$
$$D(X) = [\tilde{d}^s_{0,0}(X), \ldots, \tilde{d}^s_{0,S_0-1}(X), \tilde{d}^r_{0,0}(X), \ldots, \tilde{d}^r_{0,R_0-1}(X), \ldots$$
$$\tilde{d}^s_{N-1,0}(X), \ldots, \tilde{d}^s_{N-1,S_{N-1}-1}(X), \tilde{d}^r_{N-1,0}(X), \ldots, \tilde{d}^r_{N-1,R_{N-1}-1}(X)]$$

Possible optimization

Sending and observing all individual claims for $p^*_{i,l}$ and $q^*_{i,l}$ can be avoided. Instead, the GKR protocol could be started with a single claim for the complete sum, resulting again in two evaluation claims for the polynomials $P(X,Y), Q(X,Y)$. The number of additional rounds is logarithmic in the number of chip interactions, while the size of the individual claims is linear.

5.6. Unsafe Rust code

The code in the `slop_multilinear` crate contains unsafe Rust code. The unsafe Rust code appears due to the self-implemented managements of buffer memory lifetimes. Buffers are used to hold the entries of tensors, which are themselves used in `slop_multilinear` to represent different structs related to the representation and evaluation of multilinear polynomials.

Unsafe Rust code bears the inherent risk of memory-management errors, as it disables the various guarantees Rust makes about its memory access and lifetimes. This can lead to bugs, as illustrated by Finding 4.3. ↗.

In this section, we analyze the different types of occuring unsafe code and the implicated restrictions when the unsafe functions are used as well as possible changes to avoid them.

Multiple owning handles

The function `MleFoldBackend::fold_mle` uses multiple mutable handles to the same memory to allow parallel computation. A similar pattern is used in `MleFixLastVariableBackend::mle_fix_last_variable`.

```
// slop/crates/mutlinear/src/fold.rs
impl<F: Field> MleFoldBackend<F> for CpuBackend {
    async fn fold_mle(guts: &Tensor<F, Self>, beta: F) -> Tensor<F, Self> {
        let guts = unsafe { guts.owned_unchecked() };
        // ..
        let (tx, rx) = oneshot::channel();
        slop_futures::rayon::spawn(move || {
            // Compute the random linear combination of the even and odd
    coefficients of `vals`. This is
            // used to reduce the two evaluation claims for new_point into a
    single evaluation claim.
            let fold_guts = guts
                .as_buffer()
                .par_iter()
                .step_by(2)
                .copied()
                .zip(guts.as_buffer().par_iter().skip(1).step_by(2).copied())
                .map(|(a, b)| a + beta * b)
                .collect::<Vec<_>>();
            let dim = fold_guts.len();
            let result = Tensor::from(fold_guts).reshape([dim, 1]);
            tx.send(result).unwrap();
        });
        rx.await.unwrap()
    }
}
```

This function's premise is that while multiple mutable handles to the memory owned by `guts` are created, the writes occur to different memory areas, and all handles are discarded before the next read to prevent races. Creating multiple mutable handles requires `owned_unchecked` to be unsafe.

This is required by parallel processes created by `rayon::spawn`, which requires its function to have static lifetime because it itself does not require the process to terminate. The code spawning the process then ensures that the lifetime of the value used in the process exceeds the lifetime of the process by waiting for the process to send a signal on completion.

A possible way to use safe Rust code in this case is to use a `rayon::scope`.

Unnecessary `unsafe`

The following code uses the `unsafe` keyword even though it is not unsafe. The `unsafe` could be removed. It is possible that the keyword is placed here to mark memory unsafety inside of the tensor's buffer. In this case, it should be documented and made `unsafe` there.

```
// slop/crates/multilinear/src/mle.rs
```

```
impl<T, A: Backend> MleEval<T, A> {
    // ...
    #[inline]
    pub unsafe fn evaluations_mut(&mut self) -> &mut Tensor<T, A> {
        &mut self.evaluations
    }

// slop/crates/multilinear/src/padded.rs
pub async fn eval_at_eq<ET: AbstractExtensionField<T> + Send + Sync + Eq +
    'static>(
    // ..
    unsafe { A::add_assign(evals.evaluations_mut(), geq_adjustment).await };
```

Increasing vector length

In the function `MleFixLastVariableBackend::mle_fix_last_variable`, the length of a vector is increased beyond its initialized memory. The memory is written to immediately afterwards. An identical case occurs in `JaggedLittlePolynomialVerifierParams::full_jagged_little_polynomial_evaluation` and `JaggedLittlePolynomialVerifierParams::partial_jagged_little_polynomial_evaluation`.

```
async fn mle_fix_last_variable(
    // ...
    slop_futures::rayon::spawn(move || {
        // ..
        let mut result: Vec<EF> = Vec::with_capacity(result_size);

        // ...
        #[allow(clippy::uninit_vec)]
        unsafe {
            result.set_len(result_size);
        }
```

This is not allowed according to the Rust documentation ↗: "The elements at `old_len..new_len` must be initialized".

Exposure of unsafe behavior

Several structs in the crate `slop_multilinear` expose unsafe public functions without a clear documentation of intended use cases. For example, the function `owned_unchecked_in` allows to create a duplicated owning handle with a storage allocator different from the original handle's storage allocator.

```
// slop/crates/multilinear/src/mle.r

/// # Safety
///
/// This function is unsafe because it enables bypassing the lifetime of the
    mle.
#[inline]
pub unsafe fn owned_unchecked_in(&self, storage_allocator: A) ->
    ManuallyDrop<Self> {
    let guts = self.guts.owned_unchecked_in(storage_allocator);
    let guts = ManuallyDrop::into_inner(guts);
    ManuallyDrop::new(Self { guts })
}
```

It is unclear what the intended use case is and which conditions the new storage allocator has to meet. If changing the allocator is not intended, the option to do so should be removed from the interface. Otherwise, its intended usage and necessary restrictions on the allocator should be documented.

5.7. Test suite

The code is accompanied by a narrow test suite that covers a few happy cases of several essential functions.

The coverage in general could be improved. In particular, the crate basefold did not contain any tests at all. To achieve the goal to make slop a robust and independently usable library, coverage of all features provided by the interface is desirable. In addition to ensuring correct implementation in the tested cases, this would help to clarify the interface, promises, and intended usage of the libraries. This includes testing that functions properly signal errors or panic if called on unintended data. If the crates are configurable, the tests should cover a wide range of intended configurations and input values. This is, for example, the case in the jagged crate, where the present tests only cover specific configurations and shapes.

The present tests mostly ensure the completeness of the proof system in the tested cases. Findings 3.8. ↗ and 3.19. ↗, which are concerned with completeness, could have been discovered by tests for the intended boundary cases. This is also true for some findings regarding verifier panics, which can simultaneously be completeness problems, for example Finding 3.14. ↗. Including degenerate but intended boundary cases in the tests would help to prevent these problems.

Checking soundness with tests is harder because discovering techniques to break the verifier is hard without data targeting a specific weakness. However, the code already contains several checks that produce errors when conditions required for soundness are not met. The correctness of their implementations could be checked by tests that verify that the intended error indeed is returned under the correct conditions. For example, Finding 3.23. ↗ could have been discovered by

a minimal test verifying that the shape checks of the Merkle verifier produce the correct errors.

In cases where the soundness of a verifier depends on the correctness of a value computed by a called function, unit tests for this function can directly help the soundness. This is, for example, the case for the jagged polynomial evaluation in slop/crates/jagged/src/jagged_eval/sumcheck_eval.rs. Finding 3.10. ↗ could have been found by checking small dimensions and going through all hypercube points.

6. Assessment Results

During our assessment on the scoped Hypercube Protocol targets, we discovered 53 findings. Three critical issues were found. Seven were of high impact, six were of medium impact, 22 were of low impact, and the remaining findings were informational in nature.

The audited code implements a novel verifying algorithm that combines established and new primitives in an original way, successfully demonstrating that these components integrate into a sound overall strategy. This is a significant milestone in the development of a novel cryptographic protocol.

We consider the code to be in a prototype state, characteristic of the iterative development process in fast-moving projects.

This is particularly apparent in the interfaces of the individual modules, which have been enlarged and modified to allow the connections required for the integration in the shard verifier. These modifications have been done in an ad-hoc manner to satisfy the needs of calling functions, which left the interfaces in an unclear and incoherent state.

The state of the audited code makes it hard to reason about code locally, as the required functionality is not clear from the verifier interfaces. Instead, developers and auditors must trace verification logic and protocol flow across multiple verifiers and modules. This increases the context required to verify specific correctness properties, as these guarantees often rely on incidental behavior of other modules rather than explicit local checks or documented assumptions. This makes development error-prone, review time consuming, and increases the likelihood of undiscovered bugs.

In addition to these security risks, unclear interfaces and the intertwining of verifiers with specific higher-level callers impacts the usability of the individual verifiers as separate modules. The inspection of proof fields in functions calling the corresponding verifier show that the exposed interface of the verifier does not offer the required data to use the verifier as an encapsulated building block.

To improve auditability and strengthen the security posture of the codebase, we recommend implementing the following measures.

- Proofs and commitments should be treated as mostly opaque objects except for the verifier they are intended for. While it is valid to compose a proof object to pass it to a verifier, the verifier should not be assumed to make a guarantee about the proof.
- A common pattern for interactions with commitments in this project is verifying that the committed data has a particular shape and opening the commitment. Be consistent about these interactions in the interfaces.
- Verifiers should precisely document their prerequisites and the statement they verify. The statement should be formulated in the verifier's function arguments and should not involve the proof.
- Data and verification should be grouped in a way that makes the individual verified statements as simple as possible. For example, separate the padding shape in the jagged verifier, which is an implementation detail, from the chip shapes, which are part of the interface.
- Avoid (supposedly) redundant data as much as possible. For example, in the shard verifier, there are multiple objects holding the chips' heights in different forms.

- Avoid implicit padding and truncation. Relying on undocumented implicit padding and truncation increases the mental load required for reasoning about the behavior of the code, and it risks bugs due to unintentional padding or truncation.

We recommend a reassessment of the code after revising it according to the points above.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.