

## Scroll zstd Compression

Security assessment by HashEye · prepared for Scroll

HASHEYE AUDITED

PROJECT	Scroll zstd Compression
CLIENT	Scroll
CATEGORY	Scroll
PUBLISHED	June 1, 2024
REPORT ID	research-scroll-zstd-compression-2024-06-01-1migcf

This report was produced under HashEye's layered review process – **automated detection**, **pattern correlation**, and **senior manual verification** – with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at [hashey.io/audits/research-scroll-zstd-compression-2024-06-01-1migcf](https://hashey.io/audits/research-scroll-zstd-compression-2024-06-01-1migcf).

Scroll Compression Circuits Security Assessment July 9, 2024 Prepared for: Rohit Narurkar Scroll  
Prepared by: Opal Wright, Will Song, Filipe CasaL, Joe Doyle, and Marc Ilunga

About HashEye Founded in 2012 and headquartered in New York, HashEye provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high- end security research with a real -world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel. We maintain an exhaustive list of publications at <https://github.com/hasheye-io/publications> , with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon. We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom. HashEye also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash. To keep up to date with our latest news and announcements, please follow hasheye on Twitter and explore our public repositories at <https://github.com/hasheye-io> . To engage us directly, visit our "Contact" page at <https://www.hasheye.io/contact> , or email us at [info@hasheye.io](mailto:info@hasheye.io) . HashEye, Inc. 497 Carroll St., Space 71, Seventh Floor Brooklyn, NY 11215 <https://www.hasheye.io> [info@hasheye.io](mailto:info@hasheye.io) HashEye 1  
Scroll zstd Compression Security Assessment PUBLIC

Notices and Remarks Copyright and Distribution © 2024 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. This report is considered by HashEye to be public information; it is licensed to Scroll under the terms of the project statement of work and has been made public at Scroll's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of HashEye. The sole canonical source for HashEye publications is the HashEye Publications page . Reports accessed through any source other than that page may have been modified and should not be considered authentic. Test Coverage Disclaimer All activities undertaken by HashEye in association with this project were performed in accordance with a statement of work and agreed upon project plan. Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase. HashEye uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.  
HashEye 2 Scroll zstd Compression Security Assessment PUBLIC

Table of Contents About HashEye 1 Notices and Remarks 1 Table of Contents 3 Project Summary 5  
Executive Summary 7 Project Goals 9 Project Targets 10 Project Coverage 11 Automated Testing 12  
Codebase Maturity Evaluation 13 Summary of Findings 15 Detailed Findings 17 1. Multiple missing  
Boolean constraints on Boolean advice columns 17 2. Column annotations do not match lookup table  
columns 19 3. Unexpected BlockType for LiteralsHeader reaches unreachable! macro 22 4.  
RomTagTransition table does not allow ZstdBlockSequenceHeader → BlockHeader transitions 23 5. The  
back referencing phase is not properly constrained to a monotone behavior once activated 26 6. The  
blob-based public input commitment scheme is poorly documented 28 7. Left shift leads to undefi ned  
behavior 31 8. Missing constraints for Block\_Maximum\_Size 32 9. Apparent discrepancy between  
bitwise-op-table configuration and code comment 34 10. The compression mode reserved field is not  
enforced to equal zero 36 11. The tag\_config.is\_change witness is partially unconstrained 38 12.  
The is\_llt/is\_mot/is\_mlt constraints are only valid if self.table\_kind is in {1, 2, 3} 41 13.  
Values larger than 23 satisfy the "spans\_three\_bytes" constraints 43 14. Missing a large number of  
test cases that should fail 44 A. Vulnerability Categories 46 B. Code Maturity Categories 48 C.  
Code Quality Findings 50 D. Automated Analysis Tool Configuration 56 Expression is reconstrained  
rule 56 HashEye 3 Scroll zstd Compression Security Assessment PUBLIC

IsEqualChip rule 57 HashEye 4 Scroll zstd Compression Security Assessment PUBLIC

Project Summary Contact Information The following project manager was associated with this project: Anne Marie Barry , Project Manager annemarie.barry@hasheye.io The following engineering director was associated with this project: Jim Miller , Engineering Director, Cryptography james.miller@hasheye.io The following consultants were associated with this project: Will Song , Consultant Opal Wright , Consultant will.song@hasheye.io opal.wright@hasheye.io Filipe Casal , Consultant Marc Ilunga , Consultant filipe.casal@hasheye.io marc.ilunga@hasheye.io Joe Doyle , Consultant joseph.doyle@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event May 20, 2024 Pre-project kickoff call May 24, 2024 Status update meeting #1 May 31, 2024 Status update meeting #2 June 7, 2024 Status update meeting #3 June 14, 2024 Delivery of report draft June 14, 2024 Report readout meeting June 21, 2024 Delivery of updated report draft June 21, 2024 Updated report meeting HashEye 5 Scroll zstd Compression Security Assessment PUBLIC

July 9, 2024 Delivery of comprehensive report HashEye 6 Scroll zstd Compression Security Assessment PUBLIC

Executive Summary Engagement Overview Scroll engaged HashEye to review the security of their halo2 zstd decoder. The associated circuits are to be integrated into Scroll's zkEVM and reduce the size of Blob-carrying transactions through compression using the zstd compression algorithm. A team of five consultants conducted the review from May 20 to June 21, 2024, for a total of 12 engineer-weeks of effort. Our testing efforts focused on completeness and soundness of circuits that implement decompression according to the zstd standard. With full access to source code and documentation, we performed static and dynamic testing of the zstd decoder, using automated and manual processes. Observations and Impact Overall, the code shows signs of having small gaps in oversight, where typos are inevitably introduced in variable names or certain features are re-enabled. Constraints sometimes appear to be non-local to their actual use/initialization, which causes some values to become underconstrained in certain scenarios. Although practicing better code writing practices may alleviate some of these issues, the code's complexity ultimately requires multiple sets of eyes—whether internal or external—to review the code for small errors. We found the provided documentation to be particularly helpful in providing examples when it came to our internal review process, which greatly alleviated many pain points often seen in ZK audits. Recommendations Based on the codebase maturity evaluation and findings identified during the security review, HashEye recommends that Scroll take the following steps: • Remediate the findings disclosed in this report. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations. • Strengthen inline documentation. With a ZK circuit this large, it is often useful to document the exact constraints expected of variables and perhaps where to find these constraints. There are multiple instances of columns—for example, TagConfig.is\_reverse and FseDecoder.table\_kind—whose values are indirectly constrained via lookup\_any , and these constraints can be particularly laborious to track down. • Use wrapper types on highly constrained values. The codebase often uses advice columns that are highly specialized in nature, and need to lie in a small set of HashEye 7 Scroll zstd Compression Security Assessment PUBLIC

possible values. In particular, the circuit uses many Boolean advice columns as well as a table\_kind column, all of which must fall in the expected range for their derived expressions to be correct. Such columns deserve their own wrapper type to ensure proper constraints over their assigned witness values. Finding Severities and Categories The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 2 Medium 0 Low 4 Informational 6 Undetermined 2 CATEGORY BREAKDOWN Category Count Cryptography 7 Data Validation 5 Denial of Service 1 Undefined Behavior 1 HashEye 8 Scroll zstd Compression Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the Scroll halo2 zstd decoder. Specifically, we sought to answer the following non-exhaustive list of questions: • Does the zstd decoder correctly decode the encoded data output by their custom encoder? • Is the zstd decoder underconstrained? If so, a proof consisting of invalid assignments for a specific decoder instance will pass validation. • Do the zstd decoder tests accurately target the specific edge cases Scroll is concerned about? • Does the witness generator correctly implement the zstd decoding algorithm on the given input blob and assign the correct intermediate values to the DecoderConfig ? HashEye 9 Scroll zstd Compression Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the following targets. zkEVM Compression Circuits Repository <https://github.com/scroll-tech/zkevm-circuits/> Version 475bc1f7c0531d8c5fd90de19b4e997f5367843b 4658b6e995400d789698ed0bc2a021d6b40788f5 Type Rust Library HashEye 10 Scroll zstd Compression Security Assessment PUBLIC

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- FixedTable . We conducted a manual review of the fixed lookup table values to assess correctness.
- FseTable / FseSortedStatesStable . We conducted a manual review of the two finite state entropy tables to assess the correct computation of the finite state entropy logic of the zstd decoder.
- BitstringTable . We conducted a manual review of the bitstring table to test the correct translation of certain bitstrings of the input blob to their corresponding integer values. We also checked lookups into the bitstring table for correctness.
- LiteralHeadersTable . We performed a manual review to assess the completeness and soundness of the circuit parsing the literals header, and whether it follows the specification.
- SeqInstTable/SeqExecConfig . We performed a manual review of the sequence execution table and the sequence instruction table to assess the correctness of instructions decoding, paying special attention to the offset update rules.
- DecoderConfig . We conducted a manual review to ensure the correct linkage of components, as well as the constraints for controlling the specific instances of computation managed by the components. We took special care to check the various conditions activating constraints, the variables used, and their respective constraints.

Coverage Limitations Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not validate that every edge case described by the zstd documentation will not cause a proof validation failure or crash the witness generator.

HashEye 11 Scroll zstd Compression Security Assessment PUBLIC

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration We used the following tools in the automated testing phase of this project:

Tool	Description
Policy Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
Appendix D Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code
Appendix D Areas of Focus	Our automated testing and verification work focused on the following system properties:

- Identification of general code quality issues and unidiomatic code patterns
- Identification of dangerous halo2-specific and Scroll's API patterns

Test Results The results of this focused testing are detailed below.

Clippy Running Clippy in pedantic mode identifies several unidiomatic patterns and potential issues related to casting that should be investigated. In appendix D we describe how to run Clippy in pedantic mode, and how to efficiently triage these results using the SARIF file format.

Semgrep We present some of the rules that we wrote to find halo2-specific and Scroll's API patterns in appendix D .

HashEye 12 Scroll zstd Compression Security Assessment PUBLIC

Codebase Maturity Evaluation HashEye uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category Summary Result

Category	Summary	Result
Arithmetic	We identified one low-severity issue that can lead to undefined behavior in witness generation. No other arithmetic issues were identified.	Satisfactory
Complexity Management	There are multiple instances of excess business logic being placed at the top level of some functions. The DecoderConfig table is too large to effectively analyze, troubleshoot, or maintain. While the comments are helpful, the haphazard organization of the constraints makes it difficult to analyze individual constraints for correctness. As noted in the Code Quality Findings section, functions can run to over 1,000 lines; these functions need to be broken into smaller components.	Weak
Data Handling	The witness generator appears to correctly assign all intermediate values to the column variables, resulting in a correct decompressing of the zstd compressed blob.	Strong
Documentation	The provided Notion documentation was incredibly useful for getting up to speed with the code. However, we found inline documentation to be slightly lacking, especially when it came to assumptions regarding variables and their expected constraints. Furthermore, we found a few typos in a code snippet included with the documentation. Sometimes, the typos resulted in functionalities diverging from the Halo2 implementation, potentially introducing soundness issues if circuits have been implemented according to the documentation.	Moderate
Maintenance	Some submodules can only be described as monolithic, which is not recommended by modern coding standards. Other parts of the code are hard-coded according to the Weak	HashEye 13

Scroll zstd Compression Security Assessment PUBLIC

current requirements and require additional work should the requirements be expanded in the future. The most prominent example is the BitstringTable , which is hard-coded for up to three bytes and certain conditional expressions relying on bit\_index\_end column.

Memory Safety and Error Handling Overall, errors are mostly handled by the underlying halo2 library and custom error cases are not

needed. In the case of calling expect, which might panic under generic circumstances, valid reasons are given. Satisfactory Testing and Verification While several unit tests were present in the codebase, they tend to focus more on high-level zstd edge cases rather than low-level witness edge cases. While the former ensures good coverage and overall zstd decoding correctness, also testing the expected constraints on a system by including tests that should fail is critical to providing a higher guarantee of soundness in the long run. Weak HashEye 14 Scroll zstd Compression Security Assessment PUBLIC

Summary of Findings The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity 1 Multiple missing Boolean constraints on Boolean advice columns Data Validation High 2 Column annotations do not match lookup table columns Data Validation Informational 3 Unexpected BlockType for LiteralsHeader reaches unreachable! macro Denial of Service Informational 4 RomTagTransition table does not allow ZstdBlockSequenceHeader → BlockHeader transitions Cryptography Undetermined 5 The back referencing phase is not properly constrained to a monotone behavior once activated. Cryptography Undetermined 6 The blob-based public input commitment scheme is poorly documented Cryptography Informational 7 Left shift leads to undefi ned behavior Undefi ned Behavior Low 8 Missing constraints for Block\_Maximum\_Size Data Validation Low 9 Apparent discrepancy between bitwise-op-table configuration and code comment Cryptography Informational 10 The compression mode reserved field is not enforced to equal zero Data Validation Low 11 The tag\_config.is\_change witness is partially unconstrained Cryptography Informational HashEye 15 Scroll zstd Compression Security Assessment PUBLIC

12 The is\_llt/is\_mot/is\_mlt constraints are only valid if self.table\_kind is in {1, 2, 3} Data Validation High 13 Values larger than 23 satisfy the "spans\_three\_bytes" constraints Cryptography Informational 14 Missing a large number of test cases that should fail Cryptography Low HashEye 16 Scroll zstd Compression Security Assessment PUBLIC

Detailed Findings 1. Multiple missing Boolean constraints on Boolean advice columns Severity: High Difficulty: Medium Type: Data Validation Finding ID: TOB-SCROLLZSTD-1 Target: zkevm-circuits/aggregator/src/ Description The zstd decoder tables require the use of many Boolean type values, being either 0 or 1. Other values, such as 2 or -1, would cause many of the formulas used for Boolean logic to catastrophically fail, leading to potential constraint compromise. Below, we detail a non-exhaustive list of Boolean advice columns that appear to be unconstrained. • FseTable::is\_new\_symbol • FseTable::is\_skipped\_state • FseSortedStatesTable::is\_new\_symbol • FseSortedStatesTable::is\_skipped\_state • FseDecoder::is\_repeat\_bits\_loop • FseDecoder::is\_trailing\_bits • BitstreamDecoder::is\_nb0 • BitstreamDecoder::is\_nil • BlockConfig::compression\_modes • TagConfig::is\_reverse Exploit Scenario An attacker leverages the soundness issues of an unconstrained Boolean to disable other constraints that are derived from the unconstrained Boolean. This would allow an attacker to generate a valid proof with an invalid witness, compromising the correctness of the zstd decoder circuit. Recommendations Short term, perform the correct Boolean constraints on the necessary columns. Long term, consider annotating the Boolean columns with a wrapper type that checks if the constraint function has been called. Consider a set of checks that will ensure that all BooleanAdvice columns have been constrained. Perhaps an automated way to do this is HashEye 17 Scroll zstd Compression Security Assessment PUBLIC

a custom Drop implementation that will panic, alerting developers during testing. An alternative is to restrict access to the underlying column until the constraint has been added. struct BooleanAdvice { col: Column <Advice>, constrained: bool, } impl BooleanAdvice { fn column (& self ) → & Column <Advice> { if self .constrained { & self .col } else { panic! ( "unconstrained boolean advice" ) } } } Figure 1.1: An example Boolean advice wrapper and protected access HashEye 18 Scroll zstd Compression Security Assessment PUBLIC

2. Column annotations do not match lookup table columns Severity: Informational Difficulty: N/A Type: Data Validation Finding ID: TOB-SCROLLZSTD-2 Target: aggregator/src/aggregation/decoder/tables/{seqinst\_table.rs, literals\_header.rs}, zkevm-circuits/src/rlp\_circuit\_fsm.rs Description The column annotations for the SeqInstTable, RlpFsmDataTable, and LiteralsHeaderTable tables do not match the tables' columns: • In the SeqInstTable the n\_seq and block\_index annotations are out of order; • In the LiteralsHeaderTable, there is an additional annotation ( byte\_offset ) that would cause all subsequent annotations to refer to the wrong column; • The RlpFsmDataTable has an unannotated column ( gas\_cost\_acc ). fn columns (& self ) → Vec <Column<Any>> { vec! [ self .q\_enabled.into(), self .block\_index.into(), self .n\_seq.into(), self .s\_beginning.column.into(), self .seq\_index.into(), self .literal\_len.into(), self .match\_offset.into(), self .match\_len.into(), ] } fn annotations (& self ) → Vec <String > { vec! [ String ::from( "q\_enabled" ), String ::from( "n\_seq" ), String ::from( "block\_index" ), ] } Figure 2.1: aggregator/src/aggregation/decoder/tables/seqinst\_table.rs#L130-L147 fn columns (& self ) → Vec

```
<Column<Any>> { vec! [ self .block_idx.into(), self .byte0.into(), HashEye 19 Scroll zstd  
Compression Security Assessment PUBLIC
```

```
self .byte1.into(), self .byte2.into(), self .size_format_bit0.into(), self  
.size_format_bit1.into(), self .regen_size.into(), self .is_padding.column.into(), ] } fn  
annotations (& self ) → Vec < String > { vec! [ String ::from( "block_idx" ), String ::from(  
"byte_offset" ), String ::from( "byte0" ), String ::from( "byte1" ), String ::from( "byte2" ),  
String ::from( "size_format_bit0" ), String ::from( "size_format_bit1" ), String ::from(  
"regen_size" ), String ::from( "is_padding" ), ] } Figure 2.2:
```

```
aggregator/src/aggregation/decoder/tables/literals_header.rs#L274-L301 impl <F: Field >  
LookupTable<F> for RlpFsmDataTable { fn columns (& self ) → Vec <Column<Any>> { vec! [ self  
.tx_id.into(), self .format.into(), self .byte_idx.into(), self .byte_rev_idx.into(), self  
.byte_value.into(), self .bytes_rlc.into(), self .gas_cost_acc.into(), ] } fn annotations (& self )  
→ Vec < String > { vec! [ String ::from( "tx_id" ), String ::from( "format" ), String ::from(  
"byte_idx" ), String ::from( "byte_rev_idx" ), String ::from( "byte_value" ), String ::from(  
"bytes_rlc" ), ] } } Figure 2.3: zkevm-circuits/src/rlp_circuit_fsm.rs#L61-L84 HashEye 20 Scroll  
zstd Compression Security Assessment PUBLIC
```

Recommendations Short term, fix the reported column annotations. Long term, add debug assertions or tests that ensure that the number of columns and annotations is the same for a lookup table.

```
Consider using zip_eq instead of zip in the LookupTable::{:annotate_columns,  
annotate_columns_in_region} functions: /// Annotates a lookup table by passing annotations for each  
of it's /// columns. fn annotate_columns (& self , cs: & mut ConstraintSystem<F>) { self .columns()  
.iter() .zip( self .annotations().iter()) .for_each(|(&col, ann)|  
cs.annotate_lookup_any_column(col, || ann)) } /// Annotates columns of a table embedded within a  
circuit region. fn annotate_columns_in_region (& self , region: & mut Region<F>) { self .columns()  
.iter() .zip( self .annotations().iter()) .for_each(|(&col, ann)| region.name_column(|| ann, col))  
} Figure 2.4: zkevm-circuits/src/table.rs#87-102 Alternatively, consider adding a derive macro for
```

```
LookupTable . This way, columns made available for lookup can easily be annotated via derive macro  
helper attributes . The figure below shows an example: #[derive(Clone, Debug, LookupTable)] pub  
struct SeqInstTable <F: Field > { #[lookup] q_enabled: Column <Fixed>, #[lookup] block_index:  
Column <Advice>, #[lookup] n_seq: Column <Advice>, #[lookup] seq_index: Column <Advice>, #[lookup]  
s_beginning: Column <Advice>, // ... offset: Column <Advice>, acc_literal_len: Column <Advice>, //  
... } Figure 2.5: A derive macro-based lookup and annotation system HashEye 21 Scroll zstd  
Compression Security Assessment PUBLIC
```

```
3. Unexpected BlockType for LiteralsHeader reaches unreachable! macro Severity: Informational  
Difficulty: N/A Type: Denial of Service Finding ID: TOB-SCROLLZSTD-3 Target:  
aggregator/src/aggregation/decoder/tables/literals_header.rs Description The witness assignment  
code uses an unreachable! macro if an unexpected BlockType is found while parsing the header bytes.  
If handpicked values are chosen in byte0 , this will lead to a runtime panic during witness  
generation/assignment. let lh_bytes = [ byte0 as u8 , byte1 as u8 , byte2 as u8 ]; let  
literals_block_type = BlockType::from( lh_bytes[ 0 ] & 0x3 ); let size_format = (lh_bytes[ 0 ] >> 2  
) & 3 ; let [ n_bits_fmt, n_bits_regen, n_bytes_header]: [ usize ; 3 ] = match literals_block_type {  
BlockType::RawBlock ⇒ match size_format { 0b00 | 0b10 ⇒ [ 1 , 5 , 1 ], 0b01 ⇒ [ 2 , 12 , 2 ],  
0b11 ⇒ [ 2 , 20 , 3 ], _ ⇒ unreachable! ( "size_format out of bound" ), }, _ ⇒ unreachable! (  
"BlockType::* unexpected. Must be raw bytes for literals." ), }; Figure 3.1:
```

```
aggregator/src/aggregation/decoder/tables/literals_header.rs#206-221 Exploit Scenario The prover is  
passed a malformed compressed blob with an unexpected block type, causing the prover to halt.  
Recommendations Short term, return an Error instead of calling the unreachable! macro. Long term,  
investigate all calls to the unreachable! macro used during witness assignment. HashEye 22 Scroll  
zstd Compression Security Assessment PUBLIC
```

```
4. RomTagTransition table does not allow ZstdBlockSequenceHeader → BlockHeader transitions  
Severity: Undetermined Difficulty: Undetermined Type: Cryptography Finding ID: TOB-SCROLLZSTD-4  
Target: aggregator/src/aggregation/decoder/tables/rlp_circuit_fsm.rs,  
aggregator/src/aggregation/decoder.rs Description The RomTagTransition table does not allow  
transitioning from ZstdBlockSequenceHeader to BlockHeader . This causes a completeness issue where  
honestly generated compressed data does not satisfy the tag transition circuit. Figure 4.1 shows  
the allowed transitions and highlights the transitions that reach BlockHeader - none originate from  
ZstdBlockSequenceHeader : [ (FrameHeaderDescriptor, FrameContentSize), (FrameContentSize,  
BlockHeader ), (BlockHeader, ZstdBlockLiteralsHeader), (ZstdBlockLiteralsHeader,  
ZstdBlockLiteralsRawBytes), (ZstdBlockLiteralsRawBytes, ZstdBlockSequenceHeader),  
(ZstdBlockSequenceHeader, ZstdBlockSequenceFseCode), (ZstdBlockSequenceHeader,  
ZstdBlockSequenceData), (ZstdBlockSequenceFseCode, ZstdBlockSequenceFseCode),  
(ZstdBlockSequenceFseCode, ZstdBlockSequenceData), (ZstdBlockSequenceData, BlockHeader ), // multi-
```

block (ZstdBlockSequenceData, Null), (Null, Null), ] Figure 4.1: aggregator/src/aggregation/decoder/tables/fixd/tag\_transition.rs#L28-L41 However, in the specification, the ZstdBlockSequenceHeader to BlockHeader transition is described and has associated constraints: HashEye 23 Scroll zstd Compression Security Assessment PUBLIC

Figure 4.2: Circuit specification where a ZstdBlockSequenceHeader to BlockHeader transition is mentioned Figure 4.3 shows the implementation of the constraint from figure 4.2:

```
cb.condition(is_prev_sequence_header(meta), |cb| { cb.require_equal( "tag::prev=SeqHeader" , config .block_config .is_empty_sequences(meta, Rotation::prev()), 1. expr(), ); });
```

 Figure 4.3:

aggregator/src/aggregation/decoder.rs#1807-1815 Exploit Scenario A two-block encoded data, where the first block contains no sequences to decode, is honestly generated by the zstd compression algorithm. However, due to the missing transition, this compressed data does not satisfy the decoder circuit. HashEye 24 Scroll zstd Compression Security Assessment PUBLIC

Recommendations Short term, allow for the ZstdBlockSequenceHeader to BlockHeader transition to occur in the tag transition circuit, or document why this is not allowed. Long term, add tests with zstd compressed data that exercise all possible valid tag transitions. HashEye 25 Scroll zstd Compression Security Assessment PUBLIC

5. The back referencing phase is not properly constrained to a monotone behavior once activated Severity: Undetermined Difficulty: Low Type: Cryptography Finding ID: TOB-SCROLLZSTD-5 Target: aggregator/src/aggregation/decoder/seq\_exec.rs Description The column s\_back\_ref\_phase is not properly constrained to ensure the back-referencing phase is monotone once activated. The issue is likely due to a typo. Executing decoding sequences happens in phases: Literal copy and back-references. The two phases must not occur simultaneously. Furthermore, each phase must have a monotone behavior—i.e., the literal copy phase (the back-referencing phase, respectively) remains deactivated (activated, respectively). The constraint in figure 6.1 is meant to ensure the monotonicity of the back referencing phase once activated. However, likely due to a typo, s\_back\_ref\_phase\_prev is constrained to be equal to 1 instead of s\_back\_ref\_phase . cb.condition( and::expr([ not::expr(is\_inst\_begin.expr()), s\_back\_ref\_phase\_prev.expr(), ]), |cb| { cb.require\_equal( "inside a inst, backref phase keep 1 once it changed to 1" , s\_back\_ref\_phase\_prev.expr(), 1. expr(), ); }, );

 Figure 5.1:

aggregator/src/aggregation/decoder/seq\_exec.rs#L393-L405 However, other constraints enforce that either a phase is activated or the current rows correspond to padding. But exploiting this issue appears difficult due to adjacent constraints. The monotonicity of the literal copy phase once deactivated is guaranteed by appropriate constraints. Therefore, to use arbitrary values in the column s\_back\_ref\_phase , a malicious prover must produce a copy command that is compatible with the copied value. On the other hand, when no phase is activated, the current rows correspond to padding rows. An attacker could potentially abuse the ineffective constraints HashEye 26 Scroll zstd Compression Security Assessment PUBLIC

to provide a shorter witness. It is unclear whether this approach is feasible and what impact such an attack may ultimately have on the overall system. Exploit Scenario An attacker notices the defective constraints and produces a false witness by exploiting the missing constraints for monotonicity on s\_back\_ref\_phase . We could not fully determine the feasibility of producing malicious witnesses or whether monotonicity is fully guaranteed by adjacent constraints.

Recommendations Short term, amend the code so that it constrains s\_back\_ref\_phase to be equal to 1 instead of s\_back\_ref\_phase\_prev . cb.condition( and::expr([ not::expr(is\_inst\_begin.expr()), s\_back\_ref\_phase\_prev.expr(), ]), |cb| { cb.require\_equal( "inside a inst, backref phase keep 1 once it changed to 1" , s\_back\_ref\_phase.expr(), 1. expr(), ); }, );

 Figure 5.2: An example fix of the aforementioned issue Long term, review the codebase for potential variable typos and patterns that could render constraints void. Some examples of patterns to investigate include: comparing a variable with itself, subtracting a variable from itself, or naming conventions such as: (variable\_name, variable\_name\_prev) . HashEye 27 Scroll zstd Compression Security Assessment PUBLIC

6. The blob-based public input commitment scheme is poorly documented Severity: Informational Difficulty: N/A Type: Cryptography Finding ID: TOB-SCROLLZSTD-6 Target: aggregator/ Description To reduce the gas cost of verification, the Scroll team has moved some of the ZkEVM's public input data into the EIP-4844 "blob" structure, whereby the underlying data is stored only temporarily, at a lower gas cost. The verifier contract has access only to a polynomial commitment to the blob data and evaluation proofs. Additionally, with the addition of the zstd circuits reviewed in this report, the data in the blob is instead in zstd-compressed form. When the ZkEVM verifier is deployed in the Scroll rollup contract, a batch, consisting of both L1 and L2 transactions and associated metadata, is "committed" in the rollup contract. Committed batches can then either be reverted or finalized. To finalize a batch, a ZkEVM proof is checked against the metadata provided in the commit stage, and if the proof succeeds, the batch is finalized and can no longer be reverted. Since the ZkEVM prover should be untrusted in the system, the metadata in the commit

stage must uniquely identify the transactions in the underlying batch; otherwise, the prover may be able to finalize a different sequence of transactions than was intended. Prior to EIP-4844 integration, the sequence of transactions in a chunk was fully determined by the "public input hash" of the ZKEVM circuit, which would then be combined together into an overall public input hash by the aggregation circuit. In this scheme, the unique identification is a straightforward consequence of the collision resistance of the hash used. However, the current scheme is more complex. The transactions are now split between this public input hash and the blob structure. The PI subcircuit of the ZKEVM splits the underlying sequence of transactions into "L1" and "L2" transactions. The L1 transactions are included in the public input hash, and the L2 transactions are included in the "chunk tx hash." Unless there is some as-yet-unknown flaw in the PI subcircuit, this guarantees the uniqueness of the L1 transactions as before. To commit to the L2 transactions, the overall public inputs of the aggregation circuit include a tuple  $(\text{versionedHash}, z, y)$ , representing the polynomial commitment to the blob and the evaluation that must be checked by the verifier. To conclude that this uniquely identifies a particular sequence of L2 transactions, we must determine that: HashEye 28 Scroll zstd Compression Security Assessment PUBLIC

1.  $q$ , where  $q$  is a unique polynomial corresponding to  $\text{versionedHash}$ ;  $q(x) = \sum_{i=0}^{4096} q_i x^i$ . 2.  $z$ , where  $z$  is a unique polynomial corresponding to the L2 transaction sequence;  $z(x) = \sum_{i=0}^{4096} z_i x^i$ . 3.  $y$  is a pseudorandom challenge derived from a transcript including commitments to both the L2 transaction sequence and  $\text{versionedHash}$ . These three requirements suffice because blobs represent degree-4096 polynomials over a 254-bit finite field. The chance that  $q(x) = z(x)$  for a randomly sampled  $x$  is negligible, and requirement (3) allows us to treat  $x$  as randomly sampled for the purpose of checking that by the Fiat-Shamir heuristic. Requirement (1) is ensured because the verifier contract checks a point evaluation proof, and  $\text{versionedHash}$  is a hash of a KZG commitment, which uniquely determines  $q(x)$ . The evaluation in requirement (2) is checked by the `BarycentricEvaluationConfig` circuit. However, the uniqueness of  $z$  is more complicated, since the underlying transaction data is not stored in the blob. Instead, the `BatchDataConfig` subcircuit of the aggregation circuit includes the whole L2 transaction sequence data as private witness values. The aggregation circuit checks those hashes against the "chunk tx hash" public inputs of the ZkEVM proofs being aggregated, and checks that the L2 transaction sequence is the result of zstd-decompressing the data used in the `BarycentricEvaluationConfig` circuit. To then establish that  $z$  is unique, we must assume (a) that the zstd decompression circuit is deterministic; and (b) that the serialization of the L2 transactions when computing the chunk tx hash is unique. Finally, requirement (3) is ensured by checking that the challenge can be computed as a hash of data that includes the  $\text{versionedHash}$  and the chunk tx hashes, via an internal lookup in the `BatchDataConfig` table, shown below in figure 6.1. // lookup challenge digest in keccak table. meta.lookup\_any( "BatchDataConfig (metadata/chunk\_data/challenge digests in keccak table)", |meta| { let is\_hash = meta.query\_selector(config.hash\_selector); let is\_boundary = meta.query\_advice(config.is\_boundary, Rotation::cur()); // when is\_boundary is set in the "digest RLC" section. // this is also the last row of the "digest RLC" section. let cond = is\_hash \* is\_boundary; // - metadata\_digest: 32 bytes // - chunk[i].chunk\_data\_digest: 32 bytes each // - versioned\_hash: 32 bytes let preimage\_len = 32. expr() \* (N\_SNARKS + 1 + 1).expr(); HashEye 29 Scroll zstd Compression Security Assessment PUBLIC

```
[ 1. expr(), // q_enable 1. expr(), // is final meta.query_advice(config.preimage_rlc,
Rotation::cur()), // input rlc preimage_len, // input len meta.query_advice(config.digest_rlc,
Rotation::cur()), // output rlc ].into_iter() .zip_eq(keccak_table.table_exprs(meta)) .map(|
(value, table)| (cond.expr() * value, table)) .collect() }, );
```

Figure 6.1: Lookups from the `chunk_data` section of the table to the challenge section of the table ( `zkevm-circuits/aggregator/src/aggregation/batch_data.rs#334-362` ) All of these properties appear to hold; however, they are neither explicitly stated nor explicitly justified in Scroll's documentation. If any of them fails, it would allow a malicious prover to finalize a different batch of transactions than was committed, causing many potential issues such as denial of service or state divergence. Recommendations Short term, document this commitment scheme and specify what properties of different components it relies upon (e.g., deterministic decompression). Long term, explicitly document all intended security properties of the Scroll rollup, and what is required of each system component to ensure those properties. HashEye 30 Scroll zstd Compression Security Assessment PUBLIC

7. Left shift leads to undefined behavior Severity: Low Difficulty: Low Type: Undefined Behavior Finding ID: TOB-SCROLLZSTD-7 Target: `aggregator/src/aggregation/decoder/witgen/util.rs` Description The `read_variable_bit_packing` function accepts both an offset parameter and a maximum value, `r`, indicating the maximum value to be returned. The `r` parameter is specified as a `u64`. The number of bits required to store the decoded value is computed using a left shift with a variable shift size (see figure 7.1) from the function `bit_length`. // number of bits required to fit a value in the range `0..=r`. let size = bit\_length(r) as u32; let max = 1 << size; Figure 7.1: Calculating bit storage requirements from the range value `r` ( `zkevm-`

circuits/aggregator/src/aggregation/decoder/witgen/util.rs#23-24 ) If the top bit of `r` is set, the `bit_length` function will return 64. This shift is equal to the bit length of the max variable, leading to undefined behavior. When compiled in debug mode, this should lead to a panic, but in release mode, the behavior is unspecified and can vary from platform to platform. On x86-64 processors, shift lengths are bit masked against `0x3f`, meaning that 64 will reduce to zero, so `max` will be equal to 1. Other platforms may shift the one "off the end," causing `max` to be 0. Additionally, on a platform where `1 == 1 << 64`, using the value `u64::MAX` for `r` will result in the check for non-variable bit packing to fail, leading to potential decoding errors. Recommendations Short term, add checks to specifically handle the case where `r` is 64 bits long, whether through expanded handling or by explicitly rejecting values of `r` that can trigger this behavior. Long term, develop tests that integrate edge cases and validate correct handling. If `r` is restricted to values that will not trigger this edge case, ensure that this is clearly and conspicuously documented. HashEye 31 Scroll zstd Compression Security Assessment PUBLIC

8. Missing constraints for `Block_Maximum_Size` Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-SCROLLZSTD-8 Target: aggregator/src/aggregation/decoder.rs Description The zstd specification document states that the `Block_Size` value should be bounded by `Block_Maximum_Size`, which is the smallest of `Window_Size`, or 128 KB. In Scroll's zstd encoded blobs, the `Single_Segment_flag` is set, so `Window_Size` should equal `Frame_Content_Size` according to the specification. Figure 8.1: Zstd specification defining how the `Block_Size` should be validated HashEye 32 Scroll zstd Compression Security Assessment PUBLIC

However, the constraints named `DecoderConfig: tag BlockHeader (Block_Size)` do not check that the `Block_Size` value is limited by the smallest value between `Frame_Content_Size` and 128KB. 

```
// block_size = block_header >> 3 // // i.e. block_header - (block_size * (2^3)) < 8 let block_header_lc = meta.query_advice(config.byte, Rotation( 2 )) * 65536. expr() + meta.query_advice(config.byte, Rotation( 1 )) * 256. expr() + meta.query_advice(config.byte, Rotation( 0 )); let block_size = meta.query_advice(config.block_config.block_len, Rotation::cur()); let diff = block_header_lc - (block_size * 8. expr()); vec! [(condition * diff, config.range8.into())] }
```

 Figure 8.1: aggregator/src/aggregation/decoder.rs#L1833-L1843 Exploit Scenario A malicious prover generates a proof for a zstd blob that does not follow the specification. Due to the missing constraint, the verifier still accepts the proof as valid. Recommendations Short term, add the necessary constraint that ensures the correct validation of the `BlockSize` value. Long term, add positive and negative tests that parse and validate the `BlockHeader` according to the specification. HashEye 33 Scroll zstd Compression Security Assessment PUBLIC

9. Apparent discrepancy between `bitwise-op-table` configuration and code comment Severity: Informational Difficulty: N/A Type: Cryptography Finding ID: TOB-SCROLLZSTD-9 Target: aggregator/src/aggregation/decoder.rs, zkevm-circuits/src/table.rs Description The implementation and configuration of the `bitwise_op_table` are misleading, as the table's generic parameter does not match the code comment. The `BitwiseOpTable` structure uses generic arguments to configure which operation it implements. 

```
/// Bitwise operation table ( AND only) bitwise_op_table: BitwiseOpTable < 1, L, R>, Figure 9.1: aggregator/src/aggregation/decoder.rs#L88-L89
```

 However, the code comment states that the table should be for the bitwise-and operation, but the generic parameter of 1 corresponds to the bitwise-or operation in the `BitwiseOp` structure: 

```
#[derive(Clone, Copy, Debug)] /// Bitwise operation types. pub enum BitwiseOp { /// AND AND = 0, /// OR OR, /// XOR XOR, } impl_expr!(BitwiseOp); /// Lookup table for bitwise AND/OR/XOR operations. #[derive(Clone, Copy, Debug)] pub struct BitwiseOpTable < const OP_CHOICE: usize, const RANGE_L: usize, const RANGE_R: usize > { /// Denotes op: AND == 0, OR == 1, XOR == 2. pub op: Column <Fixed>, /// Denotes the left operand.
```

 HashEye 34 Scroll zstd Compression Security Assessment PUBLIC

```
pub lhs: Column <Fixed>, /// Denotes the right operand. pub rhs: Column <Fixed>, /// Denotes the bitwise operation on lhs and rhs. pub output: Column <Fixed>, Figure 9.2: zkevm-circuits/src/table.rs#L3270-L3294
```

 Despite this, the actual implementation is correct as an `OP_CHOICE` of 1 corresponds to the `BitwiseOp::AND` variant: 

```
/// Assign values to the BitwiseOp table. pub fn load <F: Field >(& self, layouter: & mut impl Layouter<F>) -> Result <(), Error> { layouter.assign_region( || "BitwiseOp table", | mut region| { let mut offset = 0; let chosen_ops = match OP_CHOICE { 1 => vec! [BitwiseOp::AND], 2 => vec! [BitwiseOp::OR], 3 => vec! [BitwiseOp::XOR],
```

 Figure 9.3: zkevm-circuits/src/table.rs#L3310-L3319 Recommendations Short term, unify the representations and use enum variants in the generic arguments to clarify the implementation. HashEye 35 Scroll zstd Compression Security Assessment PUBLIC

10. The compression mode reserved field is not enforced to equal zero Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-SCROLLZSTD-10 Target: aggregator/src/aggregation/decoder.rs Description The symbol compression mode specification states that the first two bits, corresponding to the `Reserved` field, "must be all-zeroes." However, the implementation does not constrain these witness values to be zero. Figure 10.1: Zstd specification stating that the `Reserved` field should

be zero Figure 10.2 shows how the bits at positions 7 to 2 are constrained, but no constraints exist for the bits at positions 1 and 0. let comp\_mode\_bit0\_ll = select::expr( byte0\_lt\_0x80.expr(), meta.query\_advice(bits[ 6 ], Rotation( 1 )), select::expr( byte0\_lt\_0xff.expr(), meta.query\_advice(bits[ 6 ], Rotation( 2 )), meta.query\_advice(bits[ 6 ], Rotation( 3 )), ), ); let comp\_mode\_bit1\_ll = select::expr( byte0\_lt\_0x80.expr(), meta.query\_advice(bits[ 7 ], Rotation( 1 )), select::expr( byte0\_lt\_0xff.expr(), meta.query\_advice(bits[ 7 ], Rotation( 2 )), meta.query\_advice(bits[ 7 ], Rotation( 3 )), ), ); HashEye 36 Scroll zstd Compression Security Assessment PUBLIC

```
let comp_mode_bit0_om = select::expr( byte0_lt_0x80.expr(), meta.query_advice(bits[ 4 ], Rotation( 1 )), select::expr( byte0_lt_0xff.expr(), meta.query_advice(bits[ 4 ], Rotation( 2 )), meta.query_advice(bits[ 4 ], Rotation( 3 )), ), ); let comp_mode_bit1_om = select::expr( byte0_lt_0x80.expr(), meta.query_advice(bits[ 5 ], Rotation( 1 )), select::expr( byte0_lt_0xff.expr(), meta.query_advice(bits[ 5 ], Rotation( 2 )), meta.query_advice(bits[ 5 ], Rotation( 3 )), ), ); let comp_mode_bit0_ml = select::expr( byte0_lt_0x80.expr(), meta.query_advice(bits[ 2 ], Rotation( 1 )), select::expr( byte0_lt_0xff.expr(), meta.query_advice(bits[ 2 ], Rotation( 2 )), meta.query_advice(bits[ 2 ], Rotation( 3 )), ), ); let comp_mode_bit1_ml = select::expr( byte0_lt_0x80.expr(), meta.query_advice(bits[ 3 ], Rotation( 1 )), select::expr( byte0_lt_0xff.expr(), meta.query_advice(bits[ 3 ], Rotation( 2 )), meta.query_advice(bits[ 3 ], Rotation( 3 )), ), );
```

Figure 10.2: aggregator/src/aggregation/decoder.rs#L378-L433 Exploit Scenario A malicious prover generates a proof for a zstd blob that does not follow the specification. Due to the missing constraint, the verifier still accepts the proof as valid. Recommendations Short term, add the necessary require\_zero constraints to ensure that decoding is compliant with the specification. Long term, add positive and negative tests for all edge cases in the specification. HashEye 37 Scroll zstd Compression Security Assessment PUBLIC

11. The tag\_config.is\_change witness is partially unconstrained Severity: Informational Difficulty: N/A Type: Cryptography Finding ID: TOB-SCROLLZSTD-11 Target: aggregator/src/aggregation/decoder.rs Description The constraints for the tag\_config.is\_change witness are not immediately obvious, making it difficult for the reader to know if it is correctly constrained. The tag\_config.is\_change witness is constrained to be true whenever byte\_idx\_delta && tag\_idx\_eq\_tag\_len\_prev holds. However, this implies only that these two conditions are sufficient for is\_change to be true, where if the conditions are met, then is\_change = true . cb.condition(and::expr([byte\_idx\_delta, tag\_idx\_eq\_tag\_len\_prev]), |cb| { cb.require\_equal( "is\_change is set" , meta.query\_advice(config.tag\_config.is\_change, Rotation::cur()), 1. expr(), ); }); Figure 11.1: aggregator/src/aggregation/decoder.rs#L1322-L1328 The constraint for the necessary part is constrained only later, where is\_change = true implies the two conditions. meta.create\_gate( "DecoderConfig: new tag" , |meta| { let condition = and::expr([ meta.query\_fixed(config.q\_enable, Rotation::cur()), meta.query\_advice(config.tag\_config.is\_change, Rotation::cur()), ]); let mut cb = BaseConstraintBuilder::default(); // The previous tag was processed completely. cb.require\_equal( "tag\_idx::prev = tag\_len::prev" , meta.query\_advice(config.tag\_config.tag\_idx, Rotation::prev()), meta.query\_advice(config.tag\_config.tag\_len, Rotation::prev()), ); // Tag change also implies that the byte\_idx transition did happen. HashEye 38 Scroll zstd Compression Security Assessment PUBLIC

```
cb.require_equal( "byte_idx::prev + 1 = byte_idx::cur" , meta.query_advice(config.byte_idx, Rotation::prev()) + 1. expr(), meta.query_advice(config.byte_idx, Rotation::cur()), );
```

Figure 11.2: aggregator/src/aggregation/decoder.rs#L1358-L1378 Exploit Scenario A malicious prover can control the is\_change witness when its value should be zero. By setting it to one when it should be zero, a malicious prover could bypass constraints related to ZstdBlockSequenceFseCode because they are constrained by not(tag\_config.is\_change) : meta.create\_gate( "DecoderConfig: tag ZstdBlockSequenceFseCode (other rows)" , |meta| { let condition = and::expr([ meta.query\_fixed(q\_enable, Rotation::cur()), meta.query\_advice(config.tag\_config.is\_fse\_code, Rotation::cur()), not::expr(meta.query\_advice(config.tag\_config.is\_change, Rotation::cur())) , not::expr( meta.query\_advice(config.fse\_decoder.is\_trailing\_bits, Rotation::cur()), ), ]); Figure 11.3: aggregator/src/aggregation/decoder.rs#2230-2240 A malicious prover could also bypass lookups into the VariableBitPacking table: meta.lookup\_any( "DecoderConfig: tag ZstdBlockSequenceFseCode (variable bit-packing)" , |meta| { // At every row where a non-nil bitstring is read: // - except the AL bits (is\_change=true) // - except when we are in repeat-bits loop // - except the trailing bits (if they exist) let condition = and::expr([ meta.query\_fixed(config.q\_enable, Rotation::cur()), meta.query\_advice(config.tag\_config.is\_fse\_code, Rotation::cur()), config.bitstream\_decoder.is\_not\_nil(meta, Rotation::cur()), not::expr(meta.query\_advice(config.tag\_config.is\_change, Rotation::cur())) , not::expr( meta.query\_advice(config.fse\_decoder.is\_repeat\_bits\_loop, Rotation::cur()), ), not::expr( meta.query\_advice(config.fse\_decoder.is\_trailing\_bits, HashEye 39 Scroll zstd Compression Security Assessment PUBLIC

```
rotation::cur(), ), ); [ ... ] let range = table_size - probability_acc + 1. expr(); [
FixedLookupTag::VariableBitPacking.expr(), range, value_read, value_decoded, num_bits, 0. expr(),
0. expr(), ] .into_iter() .zip_eq(config.fixed_table.table_exprs(meta)) .map(|(arg, table)|
(condition.expr() * arg, table)) .collect() }, );
```

Figure 11.4: aggregator/src/aggregation/decoder.rs#2552-2597 There are several other variables whose constraint soundness depends on the soundness of tag\_config.is\_change, meaning an accidental change in these two necessary and sufficient conditions may undermine the security of many circuit components. Recommendations Short term, document where the necessary and sufficient checks are performed, and therefore, constraints for is\_change = false are not needed. HashEye 40 Scroll zstd Compression Security Assessment PUBLIC

12. The is\_llt/is\_mot/is\_mlt constraints are only valid if self.table\_kind is in {1, 2, 3} Severity: High Difficulty: High Type: Data Validation Finding ID: TOB-SCROLLZSTD-12 Target: aggregator/src/aggregation/decoder.rs Description The implementation of the is\_llt, is\_mot and is\_mlt functions relies on the table\_kind witness being in the set {1, 2, 3}. It is not immediately clear from the implementation that the table\_kind variable is constrained to that set. Upon reporting this finding, the Scroll team identified that the table\_kind witness is unconstrained on a trailing bits row. Without this constraint, a malicious prover can set an incorrect table\_kind and then set the next table\_kind value incorrectly (by essentially skipping one step). impl FseDecoder { fn is\_llt (& self, meta: & mut VirtualCells<Fr>, rotation: Rotation) → Expression <Fr> { let table\_kind = meta.query\_advice( self .table\_kind, rotation); let invert\_of\_2 = Fr::from( 2 ).invert().expect( "infallible" ); (FseTableKind::MLT.expr() - table\_kind.expr()) \* (FseTableKind::MOT.expr() - table\_kind.expr()) \* invert\_of\_2 } fn is\_mot (& self, meta: & mut VirtualCells<Fr>, rotation: Rotation) → Expression <Fr> { let table\_kind = meta.query\_advice( self .table\_kind, rotation); (table\_kind.expr() - FseTableKind::LLT.expr()) \* (FseTableKind::MLT.expr() - table\_kind.expr()) } fn is\_mlt (& self, meta: & mut VirtualCells<Fr>, rotation: Rotation) → Expression <Fr> { let table\_kind = meta.query\_advice( self .table\_kind, rotation); let invert\_of\_2 = Fr::from( 2 ).invert().expect( "infallible" ); (table\_kind.expr() - FseTableKind::LLT.expr()) \* (table\_kind.expr() - FseTableKind::MOT.expr()) \* invert\_of\_2 } Figure 12.1: aggregator/src/aggregation/decoder.rs#L747-L768 HashEye 41 Scroll zstd Compression Security Assessment PUBLIC

There are lookups into tables that correctly constrain table\_kind, such as the lookup into the PredefinedFse table: // For predefined FSE tables, we must validate against the ROM predefined table fields for // every state in the FSE table. meta.lookup\_any( "FseTable: predefined table validation", |meta| { let condition = and::expr([ meta.query\_fixed(q\_enable, Rotation::cur()), meta.query\_advice(config.sorted\_table.is\_predefined, Rotation::cur()), not::expr(meta.query\_advice(config.is\_skipped\_state, Rotation::cur())), not::expr(meta.query\_advice(config.is\_padding, Rotation::cur())), ]); let (table\_kind, table\_size, state, symbol, baseline, nb) = ( meta.query\_advice(config.sorted\_table.table\_kind, Rotation::cur()), meta.query\_advice(config.sorted\_table.table\_size, Rotation::cur()), meta.query\_advice(config.state, Rotation::cur()), meta.query\_advice(config.symbol, Rotation::cur()), meta.query\_advice(config.baseline, Rotation::cur()), meta.query\_advice(config.nb, Rotation::cur()), ); [ FixedLookupTag::PredefinedFse.expr(), table\_kind, table\_size, state, symbol, baseline, nb, ] .into\_iter() .zip\_eq(fixed\_table.table\_exprs(meta)) .map(|(arg, table)| (condition.expr() \* arg, table)) .collect() }); Figure 12.2:

aggregator/src/aggregation/decoder/tables/fse.rs#L590-L622 However, the set of conditions under which these lookups occur differs from other sets of conditions where the is\_llt, is\_mot and is\_mlt functions are used (either explicitly or implicitly), which could lead to soundness issues. Recommendations Short term, add explicit constraints to the table\_kind witness, and ensure that the witness is constrained in every case. Long term, add negative tests to ensure that incorrect or unexpected values of table\_kind do not satisfy the constraints of the decoder circuit. HashEye 42 Scroll zstd Compression Security Assessment PUBLIC

13. Values larger than 23 satisfy the "spans\_three\_bytes" constraints Severity: Informational Difficulty: N/A Type: Cryptography Finding ID: TOB-SCROLLZSTD-13 Target: aggregator/src/aggregation/decoder.rs Description The spans\_three\_bytes function should constrain the bit\_index\_end witness to lie in the [16, 23] interval. However, it accepts a value as long as bit\_index\_end > 15. In particular, values larger than 23 will satisfy the constraint. /// A bitstring spans 3 bytes if the bit\_index at which it ends is such that: /// - 16 ≤ bit\_index\_end ≤ 23. fn spans\_three\_bytes (& self, meta: & mut VirtualCells<Fr>, at: Rotation) → Expression <Fr> { let lhs = meta.query\_advice( self .bit\_index\_end, at); let (lt2, eq2) = self .bit\_index\_end\_cmp\_15.expr\_at(meta, at, lhs, 15. expr()); not::expr(lt2 + eq2) } Figure 13.1: aggregator/src/aggregation/decoder.rs#L637-L643 In practice, this issue is unexploitable—thus the informational severity—because the current BitstringTable implementation does not support bitstrings spanning more than three bytes. However, we still highly recommend correctly

constraining the bit\_index\_end witness: if the bitstream decoder starts supporting more than three bytes in the future, the missing constraint would cause soundness issues. Recommendations Short term, add a constraint enforcing that the bit\_index\_end witness is smaller than 24. HashEye 43 Scroll zstd Compression Security Assessment PUBLIC

14. Missing a large number of test cases that should fail Severity: Low Difficulty: N/A Type: Cryptography Finding ID: TOB-SCROLLZSTD-14 Target: aggregator/src/aggregation/\* Description Overall, we found that there is a severe lack of testing, with a total of only 17 tests that do not comprehensively test the functionality of the decoder circuit. Not only should tests cover the expected success cases, but they should also ensure that well-defined failure modes are not possible within the confines of the system. We have compiled a non-exhaustive list of such test cases below: • Boolean witnesses should not satisfy the witness generator if they are assigned non-Boolean values. This type of test is most valuable before implementing the custom wrapper type described in TOB-SCROLLZSTD-1, but tests similar in nature are still recommended for similarly constrained values. • Add tests for all valid RomTagTransition pairs, ensuring compatibility with the zstd specification (TOB-SCROLLZSTD-4). • Add tests for various BlockHeader configurations, ensuring configurations that fall outside the specification are rejected (TOB-SCROLLZSTD-8). • Add tests for the reserved compression mode bits (TOB-SCROLLZSTD-10). • Add tests that try to insert an invalid table\_kind value (TOB-SCROLLZSTD-12). • Add randomized round trip encoder tests. There is currently a single test that checks the satisfiability of the encoding of a known string. However, this test is also for a singular string and is not easily generalized to other input sizes. Randomized round trip testing can better guarantee the correctness of the encoding and decoding steps by easily generating larger inputs. We also recommend the addition of fuzz testing. In large systems like these, it is hard to systematically test every edge case of the system. Fuzz testing enables the user to automate the randomization of inputs, leading to potentially higher constraint coverage. Recommendations Short term, add additional unit tests for the failure modes listed in the above issues. HashEye 44 Scroll zstd Compression Security Assessment PUBLIC

Long term, add support for fuzzing of the witness generator, where a valid witness can have some cells perturbed. If any of these small perturbations leads to another valid witness assignment, there is a high likelihood of a missing constraint somewhere. HashEye 45 Scroll zstd Compression Security Assessment PUBLIC

A. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories Category Description Access Controls Insufficient authorization or assessment of rights Auditing and Logging Insufficient auditing of actions or logging of problems Authentication Improper identification of users Configuration Misconfigured servers, devices, or software components Cryptography A breach of system confidentiality or integrity Data Exposure Exposure of sensitive information Data Validation Improper reliance on the structure or values of data Denial of Service A system failure with an availability impact Error Reporting Insecure or insufficient reporting of error conditions Patching Use of an outdated software package or library Session Management Improper identification of authenticated users Testing Insufficient test methodology or test coverage Timing Race conditions or other order-of-operations flaws Undefined Behavior Undefined behavior triggered within the system HashEye 46 Scroll zstd Compression Security Assessment PUBLIC

Severity Levels Severity Description Informational The issue does not pose an immediate risk but is relevant to security best practices. Undetermined The extent of the risk was not determined during this engagement. Low The risk is small or is not one the client has indicated is important. Medium User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. High The flaw could affect numerous users and have serious reputational, legal, or financial implications. Difficulty Levels Difficulty Description Undetermined The difficulty of exploitation was not determined during this engagement. Low The flaw is well known; public tools for its exploitation exist or can be scripted. Medium An attacker must write an exploit or will need in-depth knowledge of the system. High An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue. HashEye 47 Scroll zstd Compression Security Assessment PUBLIC

B. Code Maturity Categories The following tables describe the code maturity categories and rating criteria used in this document. Code Maturity Categories Category Description Arithmetic The proper use of mathematical operations and semantics Auditing The use of event auditing and logging to support monitoring Authentication / Access Controls The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system Complexity Management The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions Cryptography and Key Management The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution Documentation The presence of comprehensive and readable codebase

documentation Memory Safety and Error Handling The presence of memory safety and robust error-handling mechanisms Testing and Verification The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage Rating Criteria Rating Description Strong No issues were found, and the system exceeds industry standards. Satisfactory Minor issues were found, but the system is compliant with best practices. Moderate Some issues that may affect system safety were found. Weak Many issues that affect system safety were found. Missing A required component is missing, significantly affecting system safety. Not Applicable The category is not applicable to this review. Not Considered The category was not considered in this review. HashEye 48 Scroll zstd Compression Security Assessment PUBLIC

Further Investigation Required Further investigation is required to reach a meaningful conclusion. HashEye 49 Scroll zstd Compression Security Assessment PUBLIC

C. Code Quality Findings We identified the following code quality issues through manual and automated code review. • Panic in functions that returns Result . The synthesizer function in compression/circuit.rs returns Result<(), Error> , but line 104 is an expect call, which will result in a panic. Consider translating this to an error that is returned to higher-level logic for possible recovery or improved error reporting. • Duplicated byte reconstruction logic in BitstringTable . The BitstringTable::configure function creates several constraints that validate that bits 0..7, 8..15, and 16..23 are equal to byte1 , byte2 , and byte3 , respectively. However, these 8 bit-to-byte calculations are done manually, and it may make more sense to refactor the LE/BE byte reconstruction into a reusable subroutine to decrease the chances of copy-paste errors when the table is expanded to more bytes in the future. • Special casing of 1 and 2 byte BitstringTable s. The BitstringTable has a const generic parameter that expects a value of 1, 2, or 3, with 3 being the case we are most interested in. This results in additional special casing requiring several checks against N\_BYTES . Making this struct more generic by consolidating byte1 , byte2 , and byte3 into byte: [Column<Advice>; N\_BYTES] , as well as writing more generic code, could make further expansion easier and less likely to introduce a bug. • Typo on constraint label. The word "kee" should be "keep": cb.condition(not::expr(is\_inst\_begin.expr()), |cb| { cb.require\_equal( "backref offset kee the same in one inst" , backref\_offset.expr(), backref\_offset\_prev.expr(), ) }); Figure C.1: aggregator/src/aggregation/decoder/seq\_exec.rs#507-513 • Redundant iteration over large vector. The implementation of the RomVariableBitPacking::values function first constructs a large vector of RomVariableBitPacking elements, and then iterates over it to construct the result vector of 7-tuples. Instead, to prevent having to iterate over the vector twice, define an auxiliary function that returns the 7-tuple of values. Then, use that auxiliary function to construct the intended result in the first iteration. HashEye 50 Scroll zstd Compression Security Assessment PUBLIC

```
impl FixedLookupValues for RomVariableBitPacking { fn values () → Vec <[Value<Fr>; 7 ]> { // The maximum range R we ever have is 512 (1 << 9) as the maximum possible accuracy log is // 9. So we only need to support a range up to R + 1, i.e. 513. let rows = ( 1 ..= 513 ) .flat_map(|range| { // Get the number of bits required to represent the highest number in this range. let size = bit_length(range) as u32 ; let max = 1 << size; // Whether ``range`` is a power of 2 minus 1, i.e. 2^k - 1. In these cases, we // don't need variable bit-packing as all values in the range can be represented by // the same number of bits. let is_no_var = range & (range + 1) == 0 ; // The value read is in fact the value decoded. if is_no_var { return ( 0 ..=range) .map(|value_read| RomVariableBitPacking { range, value_read, value_decoded: value_read , num_bits: size as u64 , } ) .collect::<Vec<_>>(); } let n_total = range + 1 ; let lo_pin = max - n_total; let n_remaining = n_total - lo_pin; let hi_pin_1 = lo_pin + (n_remaining / 2); let hi_pin_2 = max - (n_remaining / 2); ( 0 ..max) .map(|value_read| { // the value denoted by the low (size - 1)-bits. let lo_value = value_read & (( 1 << (size - 1) ) - 1 ); let (num_bits, value_decoded) = if ( 0 ..lo_pin).contains(&lo_value) { (size - 1 , lo_value) } else if (lo_pin..hi_pin_1).contains(&value_read) { (size, value_read) } else if (hi_pin_1..hi_pin_2).contains(&value_read) { (size - 1 , value_read - hi_pin_1) } else { assert! ((hi_pin_2..max).contains(&value_read)); (size, value_read - lo_pin) }; HashEye 51 Scroll zstd Compression Security Assessment PUBLIC
```

```
RomVariableBitPacking { range, value_read, value_decoded, num_bits: num_bits .into(), } }) .collect::< Vec <_>>() }) .collect::< Vec <_>>(); rows.iter() .map(|row| { [ Value::known(Fr::from(FixedLookupTag::VariableBitPacking as u64 )), Value::known(Fr::from(row.range)), Value::known(Fr::from(row.value_read)), Value::known(Fr::from(row.value_decoded)), Value::known(Fr::from(row.num_bits)), Value::known(Fr::zero()), Value::known(Fr::zero()), ] }) .collect() } Figure C.2: aggregator/src/aggregation/decoder/tables/variable_bit_packing.rs#15-86 • Excessive function length. The process_sequences function is over 1,100 lines. This hinders maintainability and review. Related portions of the function should be broken into their own routines. • Vacuous
```

constraint on `is_inst_begin` . The witness `is_inst_begin` must be constrained as a `Boolean`. However, the implementation includes a redundant constraint `is_inst_begin` . The constraint is ineffective since it is applied to a variable unrelated to the `SeqExecConfig` . Furthermore, an effective constraint is defined in the subsequent lines of code. // boolean constraint that index is increment `cb.require_boolean( "instruction border is boolean" , is_inst_begin.expr()); [...]`  
`is_inst_begin = select::expr( is_block_begin.expr(), 1. expr(), meta.query_advice(seq_index, Rotation::cur()))` HashEye 52 Scroll zstd Compression Security Assessment PUBLIC

- `meta.query_advice(seq_index, Rotation::prev()), ); cb.require_boolean( "inst border is boolean" , is_inst_begin.expr());` Figure C.3: aggregator/src/aggregation/decoder/seq\_exec.rs#339-354 • Redundant initialization of witness values . The first row of the table associated with `SeqExecConfig` is initialized by assigning constants to all columns of the table. The columns `decoded_len` , `decoded_rlc` , and `block_index` are assigned the constant `F::zero()` twice; furthermore, the assignments are performed with two different methods: `assign_advice_from_constant` and `assign_advice` . The `assign_advice_from_constant` method should be preferred for initializing the first row, it additionally increases the readability of the codebase. `for col in [ self .decoded_byte, self .decoded_len, self .decoded_rlc, [...] self .backref_progress, ] {`

`region.assign_advice(|| "top row flush" , col, offset, || Value::known(F::zero()))?; } for (col, val) in [ ( self .decoded_len, F::zero()), ( self .decoded_rlc, F::zero()), ( self .block_index, F::zero()), ] { region.assign_advice_from_constant(|| "top row constraint" , col, offset, val)? ; }` Figure C.4: aggregator/src/aggregation/decoder/seq\_exec.rs#912-934 • Potentially unused fixed column. Despite the comment stating that the column was used in `SeqExecConfig` , we did not find any use for it. `pub struct DecoderConfig < const L: usize , const R: usize > { /// constant column required by SeqExecConfig. _const_col: Column <Fixed>,` Figure C.5:

aggregator/src/aggregation/decoder.rs#45-47 • Duplicate code comment prevents documentation rendering. Remove the initial // from the code comment. // /// Helper table in the "output" region for accumulating the result of HashEye 53 Scroll zstd Compression Security Assessment PUBLIC

executing sequences. `sequence_execution_config: SequenceExecutionConfig <Fr>,` Figure C.6:

aggregator/src/aggregation/decoder.rs#104-105 • Usage of `IsEqualConfig` could be replaced by `IsZeroConfig` . We found three locations where the `IsEqualConfig` gadget is used to compare against 0. Instead, use the `IsZeroConfig` gadget. `is_empty_sequences: IsEqualConfig <Fr>, /// For sequence decoding, the tag=ZstdBlockSequenceHeader bytes tell us the Compression_Mode /// utilised for Literals Lengths, Match Offsets and Match Lengths. We expect only 2 /// possibilities: /// 1. Predefined_Mode (value=0) /// 2. Fse_Compressed_Mode (value=2) /// /// Which means a single boolean flag is sufficient to take note of which compression mode is /// utilised for each of the above purposes. The boolean flag will be set if we utilise the /// Fse_Compressed_Mode.`

`compression_modes: [Column<Advice>; 3 ], } impl BlockConfig { fn configure (meta: & mut ConstraintSystem<Fr>, q_enable: Column <Fixed>) → Self { let num_sequences = meta.advice_column(); Self { block_len: meta .advice_column(), block_idx: meta .advice_column(), is_last_block: meta .advice_column(), is_block: meta .advice_column(), num_sequences, is_empty_sequences: IsEqualChip ::configure( meta, |meta| meta.query_fixed(q_enable, Rotation::cur()), |meta| meta.query_advice(num_sequences, Rotation::cur()), |_| 0. expr(), ),` Figure C.7:

aggregator/src/aggregation/decoder.rs#206-233 `value_decoded_eq_0: IsEqualChip ::configure( meta, |meta| meta.query_fixed(q_enable, Rotation::cur()), |meta| meta.query_advice(value_decoded, Rotation::cur()), |_| 0. expr(), ),` HashEye 54 Scroll zstd Compression Security Assessment PUBLIC

Figure C.8: aggregator/src/aggregation/decoder.rs#731-736 `baseline_0x00: IsEqualChip ::configure( meta, |meta| meta.query_fixed(q_enable, Rotation::cur()), |meta| meta.query_advice(baseline, Rotation::cur()), |_| 0. expr(), ),` Figure C.9:

aggregator/src/aggregation/decoder/tables/fse.rs#1376-1381 • Typo in code comment. The word "bittsring" should be "bitstring." `/// The bit-index where the bittsring begins. 0 ≤ bit_index_start < 8. bit_index_start: Column <Advice>,` Figure C.10:

aggregator/src/aggregation/decoder.rs#453-454 • Unused local variables. `for i in 0 ..N_SNARKS { for j in 0 ..DIGEST_LEN { let mut t1 = Fr::default(); let mut t2 = Fr::default(); chunk_pi_hash_digests[i][j].value().map(|x| t1 = *x); snark_inputs[i * DIGEST_LEN + j].value().map(|x| t2 = *x);` Figure C.11: aggregator/src/aggregation/circuit.rs#372-377 HashEye 55 Scroll zstd Compression Security Assessment PUBLIC

D. Automated Analysis Tool Configuration We used the following tools to perform automated testing of the codebase. D.1. Semgrep We used the static analyzer Semgrep to search for dangerous API patterns and weaknesses in the source code repository. `semgrep --metrics=off --sarif --config custom_rule_path.yml` Figure D.1: The invocation command used to run Semgrep for each custom rule `semgrep --metrics=off --sarif --config "p/hashey"` Figure D.2: The invocation command used to run Semgrep with HashEye' public rules `Expression is reconstrained rule` We wrote this rule after identifying finding TOB-SCROLLZSTD-5 , trying to identify variants of the same issue. `rules : - id`

```
: expression-reconstrained message : "An expression appears in the condition and again in a
constraint. This could indicate a copy - paste error" languages : [ rust ] severity : ERROR
patterns : - pattern : cb.condition(<... $X.expr() ...>, |cb| { <... $X.expr() ...>}); -
metavariable-pattern : metavariable : $X patterns : - pattern-not : "1" Figure D.3: Custom Semgrep
rule that finds TOB-SCROLLZSTD-5 Besides the instance from TOB-SCROLLZSTD-5 , this rule finds two
other cases. Although neither is underconstrained, the value is reused in both condition and
constraint, meaning that the value could be replaced by 1 in the constraint. let is_padding_next =
1. expr() - s_lit_cp_phase_next.expr() - s_back_ref_phase_next.expr();
cb.condition(is_padding.expr(), |cb| { cb.require_equal( "padding never change once activated" ,
is_padding_next.expr(), is_padding.expr(), ); HashEye 56 Scroll zstd Compression Security
Assessment PUBLIC
```

```
}); Figure D.4: aggregator/src/aggregation/decoder/seq_exec.rs#407-415 cb.condition( and::expr([
s_last_lit_cp_phase_prev.expr(), not::expr(is_block_begin.expr()), ]), |cb| { cb.require_equal(
"phase must keep activated until block end" , s_last_lit_cp_phase_prev.expr(),
s_last_lit_cp_phase.expr(), ); }, ); Figure D.5:
```

```
aggregator/src/aggregation/decoder/seq_exec.rs#442-454 IsEqualChip rule After finding a couple of
instances where the IsEqualChip gadget could be replaced by the IsZeroConfig gadget, we wrote a
custom Semgrep rule to find other instances. rules : - id : is-equal-chip message : "The
IsEqualChip is used to compare a value to zero, when the IsZeroChip could be used instead."
```

```
languages : [ rust ] severity : ERROR patterns : - pattern : IsEqualChip::configure( $X, $Y, $Z,
|_| 0.expr(), ) Figure D.6: Custom Semgrep rule that finds the three instances described in the
Code Quality appendix D.2. Clippy The Rust linter Clippy can be installed using rustup by running
the command rustup component add clippy . Invoking cargo clippy -- -W clippy::pedantic in the root
directory of the project runs the tool with the pedantic ruleset. cargo clippy -- -W
```

```
clippy::pedantic Figure D.7: The invocation command used to run Clippy in the codebase HashEye 57
Scroll zstd Compression Security Assessment PUBLIC
```

Converting the output to the SARIF file format (e.g., with `clippy-sarif` ) allows easy inspection of the results within an IDE (e.g., using VSCode's SARIF Explorer extension). HashEye 58 Scroll zstd Compression Security Assessment PUBLIC