Zellic

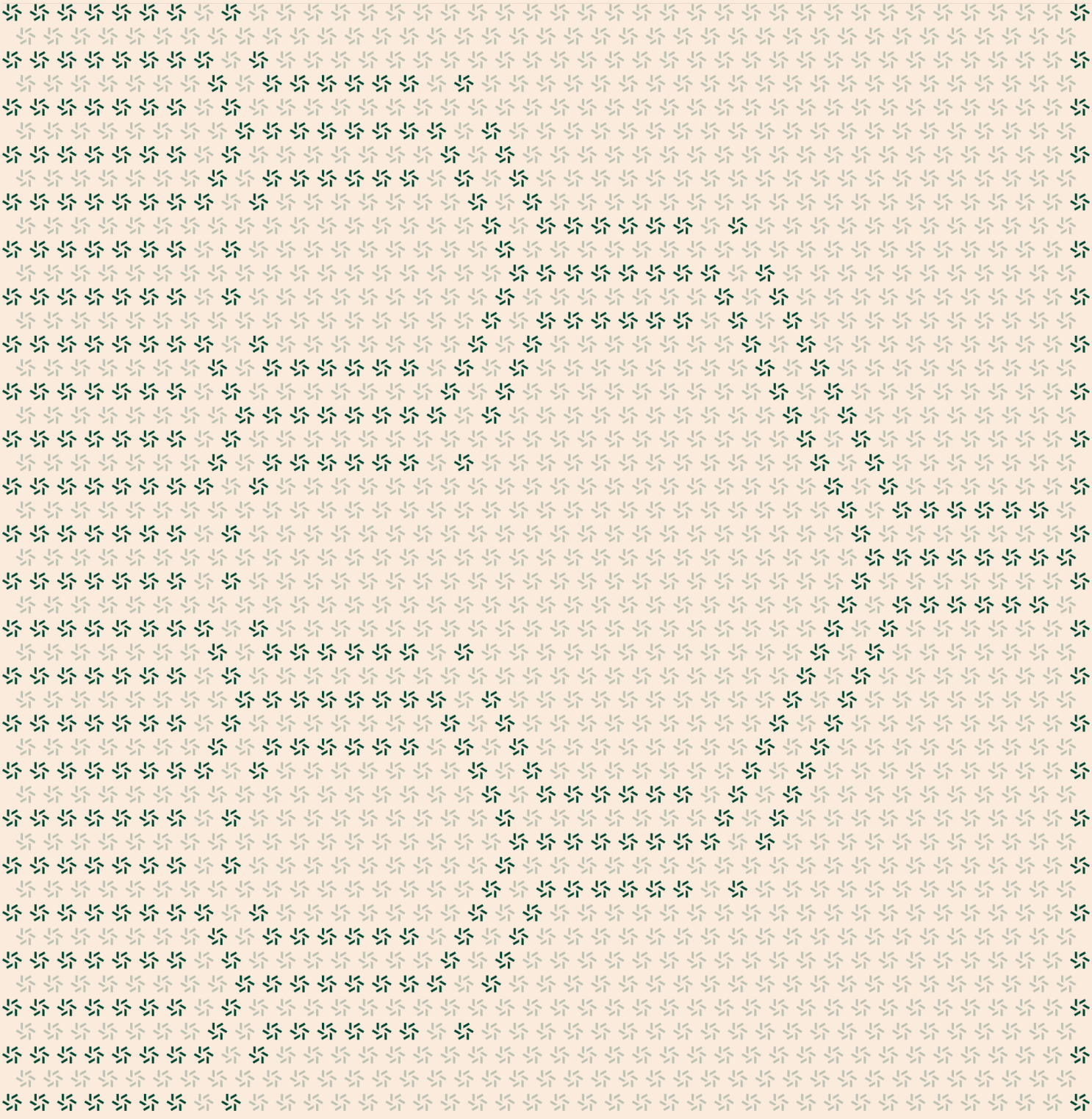
Prepared for
Haichen Shen
Scroll

Prepared by
Malte Leip
Mohit Sharma
Zellic

June 5, 2024

Scroll zkEVM

Proof Circuit 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 Scroll from May 9th to May 28th, 2024. During this engagement, Zellic reviewed Scroll zkEVM'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 regarding the changes made to the proof circuits in the pull requests implementing EIP-5656 and EIP-1153:

- Are the MCOPY, MSTORE, and MLOAD instructions implemented in a manner consistent with their specification?
- Do the changes preserve completeness and soundness of the circuits?

1.3. Non-goals and Limitations

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

- Correctness of code outside of the circuits
- Front-end components
- Infrastructure relating to the project

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 Scroll zkEVM circuits, we discovered seven findings. Four critical issues were found. One was of medium impact and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Scroll's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	0
Medium	1
Low	0
Informational	2

2. Introduction

2.1. About Scroll zkEVM

Scroll contributed the following description of Scroll zkEVM:

> Scroll seamlessly extends Ethereum's capabilities through zero knowledge tech and EVM compatibility. The L2 network built by Ethereum devs for Ethereum devs.

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:

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities and in some cases provide a proof of the fact.

Overconstrained circuits. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witness will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks and, in certain cases, provide a proof that the given set of range checks is sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices, guidelines, and code quality standards.

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 circuits itself. These observations — found in the Discussion (4. ↗) 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:

Scroll zkEVM Circuits

Type	Rust
Platform	halo2

Target: zkevm-circuits

Repository	https://github.com/scroll-tech/zkevm-circuits ↗
Pull requests	In scope were changes to the circuits introduced in the following pull requests: • EIP-5656 support as implemented in pull request 1209 ↗ and pull request 1255 ↗ • EIP-1153 support as implemented in pull request 1233 ↗

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-weeks. The assessment was conducted over the course of three 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 ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 15, 2024	Kick-off call
May 9, 2024	Start of primary review period
May 24, 2024	End of primary review period
May 24, 2024	Call to discuss some findings

3. Detailed Findings

3.1. Missing constraints in the copy circuit for MCOPY allow inserting illegitimate entries in the rw table

Target	Copy circuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This finding concerns constraints missing in the copy circuit regarding copy events associated to `MCOPY` instructions, which ultimately allow inserting illegitimate entries in the rw table. To describe this issue, we will start by describing the relevant aspects of the rw table and the intended way to ensure there are no illegitimate entries in the rw table, via the EVM circuit's counting of legitimate rw lookups. We will then discuss how, in the case of `MCOPY` instructions, the correctness of this count depends on the copy circuit correctly constraining a certain value. Finally, we can discuss how the copy circuit constrains that value, and we will see that it is insufficiently constrained.

Purpose of the rw table and state circuit

The rw table is used to keep track of consistency of read and write operations that write to or read from, among other locations, storage, memory, and stack. This table is constrained by the state circuit, which ensures that, for example, reads that follow a write to some location will read the value written, that reads do not change the value, and so on. The state circuit does not however check that the entries are actually legitimate, so from the perspective of the state circuit only, any writes may appear in the table. Non-`Start` entries of the rw table are intended to have a unique ID, given by the `counter`, which is to increment on each read or write. Entries of the `Start` type, which are used as padding at the start of the table, have separate counters, counting up on each row. By looking up a `Start` entry with `counter` being `1` and one `Start` entry with counter being a high number, an upper bound on the number of non-`Start` entries in the rw table can be enforced.

Ensuring that there are no malicious entries in the rw table

Other circuits make lookups to the rw table to perform reads and writes to storage, memory, and so on. In order to ensure that all non-`Start` entries in the rw table actually arise from such legitimate lookups, the EVM circuit keeps track of the current counter for the next entry of the rw table in `state.rw_counter` of each step. This is done by adding the number of rw lookups that were done when transitioning from one step to the next. At the very end, the `rw_counter` should reflect the total number of legitimate lookups done by the EVM circuit or by the copy circuit. The EVM circuit then, at the end of the block, looks up two entries in the rw table of the `Start` type to ensure that the remaining number of rows available (not of type `Start`) are at most as many as expected for legitimate

lookups:

```
// 3. Verify rw_counter counts to the same number of meaningful rows in
// rw_table to ensure there is no malicious insertion.
// Verify that there are at most total_rws meaningful entries in the rw_table
cb.rw_table_start_lookup(1.expr());
cb.rw_table_start_lookup(max_rws.expr() - total_rws.expr());
// Since every lookup done in the EVM circuit must succeed and uses
// a unique rw_counter, we know that at least there are
// total_rws meaningful entries in the rw_table.
// We conclude that the number of meaningful entries in the rw_table
// is total_rws.
```

This will ensure that there are indeed no malicious entries in the rw table as long as two assumptions hold:

1. The count of legitimate rw lookups done is accurate.
2. No two legitimate rw lookups are identical.

If either of these assumptions is violated, there will be additional rows available in the rw table in which an adversary may insert arbitrary malicious entries, such as writes to storage of their choosing.

This finding concerns the possibility of violating both assumptions for `MCOPY` instructions due to missing constraints in the copy circuit.

Constraints on `rw_counter` in the EVM circuit for `MCOPY`

In `zkevm-circuits/src/evm_circuit/execution/mcopy.rs`, the `MCopyGadget::configure` function adds the constraints for the `MCOPY` operation. Here is a relevant snippet:

```
// copy_rwc_inc used in copy circuit lookup.
let copy_rwc_inc = cb.query_cell();
cb.condition(memory_src_address.has_length(), |cb| {
    cb.copy_table_lookup(
        cb.curr.state.call_id.expr(),
        CopyDataType::Memory.expr(),
        cb.curr.state.call_id.expr(),
        CopyDataType::Memory.expr(),
        // src_addr
        memory_src_address.offset(),
        // src_addr_end
        memory_src_address.end_offset(),
        // dest_addr
```

```
        memory_dest_address.offset(),
        memory_dest_address.length(),
        // rlc_acc is 0 here.
        0.expr(),
        copy_rwc_inc.expr(),
    );
});

cb.condition(not::expr(memory_src_address.has_length()), |cb| {
    cb.require_zero(
        "if no bytes to copy, copy table rwc inc == 0",
        copy_rwc_inc.expr(),
    );
});
```

The crucial circuit variable to consider here is `copy_rwc_inc`, which is part of the arguments for `copy_table_lookup`, which will add the copy table lookup. Otherwise `copy_rwc_inc` is not constrained here as long as the length of the the copy operation is positive. In the case that the copy operation is for length zero, `copy_rwc_inc` is constrained to be zero as well. We will only consider the case for positive length in the following.

The circuit builder's `EVMConstraintBuilder::copy_table_lookup` function is implemented in `zkevm-circuits/src/evm_circuit/util/constraint_builder.rs` as follows:

```
pub(crate) fn copy_table_lookup(
    &mut self,
    src_id: Expression<F>,
    src_tag: Expression<F>,
    dst_id: Expression<F>,
    dst_tag: Expression<F>,
    src_addr: Expression<F>,
    src_addr_end: Expression<F>,
    dst_addr: Expression<F>,
    length: Expression<F>,
    rlc_acc: Expression<F>,
    rwc_inc: Expression<F>,
) {
    self.add_lookup(
        "copy lookup",
        Lookup::CopyTable {
            is_first: 1.expr(), // is_first
            src_id,
            src_tag,
            dst_id,
            dst_tag,
            src_addr,
```

```
            src_addr_end,
            dst_addr,
            length,
            rlc_acc,
            rw_counter: self.curr.state.rw_counter.expr() +
    self.rw_counter_offset(),
            rwc_inc: rwc_inc.clone(),
        },
    );

    self.rw_counter_offset = self.rw_counter_offset.clone() +
    self.condition_expr() * rwc_inc;
}
```

The `copy_rwc_inc` circuit variable from `MCopyGadget::configure` is used here as `rwc_inc` in the actual copy table lookup. It is also used to increase the read write counter offset accordingly. This means that if the operation currently executing actually is `MCOPY` (so the condition expression is `1`), then the circuit builder's `rw_counter_offset` will increase by `copy_rwc_inc`.

Back in `MCopyGadget::configure` in `zkevm-circuits/src/evm_circuit/execution/mcopy.rs`, the state transition is constrained by the following snippet:

```
let step_state_transition = StepStateTransition {
    rw_counter: Transition::Delta(cb.rw_counter_offset()),
    program_counter: Transition::Delta(1.expr()),
    stack_pointer: Transition::Delta(3.expr()),
    memory_word_size:
    Transition::To(memory_expansion.next_memory_word_size()),

    gas_left: Transition::Delta(-gas_cost),
    ..Default::default()
};
let same_context = SameContextGadget::construct(cb, opcode,
    step_state_transition);
```

So the `rw_counter` will increase by `cb.rw_counter_offset` for the next execution step. In effect this means that `rw_counter` will increase by `copy_rwc_inc`, and the only other way this value is constrained in the evm circuit is via the copy table lookup.

The copy circuit

In the copy circuit, the column containing `copy_rwc_inc` is called `rwc_inc_left`. The copy circuit is thus responsible to constrain the value of the `rwc_inc_left` column in the first row of the copy event to be the number of distinct rw table lookups that are performed by it.

However, this is not the case. The top-level function introducing constraints for the copy circuit is `CopyCircuitConfig::new` in `zkevm-circuits/src/copy_circuit.rs`. The only place `rwc_inc_left` is used there is in a call to `constrain_rw_counter` in `zkevm-circuits/src/copy_circuit/copy_gadgets.rs`.

That function introduces two different types of constraints, depending on whether or not the copy operation is a memory-to-memory copy.

The case of copies that are not memory-to-memory

The copy table contains a row for each byte read or written during the copy operation. However, reads and writes actually happen per word, which is 32 bytes of length. Multiple bytes will thus be read or written from the same lookup in the rw table. Some relevant columns (or expressions formed from columns) for the discussion here are as follows:

`is_row_end`	A boolean that is 1 if and only if this is the last byte (index 31) of the word that is being read or written.
`rw_counter`	The counter for the rw table lookup underlying the read or written byte of the current row.
`is_last`	A boolean that is true on the last row of of reading and writing a word. This will be enabled on the write row only.

Copy events have rows with read and write rows alternating. For copies that are not memory-to-memory, the first row will read the first byte, the second row will write the first byte, and so on. There is some additional padding logic if the copy operation begins or ends in the middle of a word, but we will ignore this in our exposition here, as it is not relevant to the finding.

We depict below an example for a copy operation that involves copying two words. The `r/w` column indicates whether the row is a read or write row, the `word` column indicates which word is being copied, and `word_index` is the index of the byte currently copied within the current word. In this example, we have not yet taken into account the constraints imposed by `constrain_rw_counter`.

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	Y
0	0	w	0	0	?	?
0	1	r	0	0	?	?
0	1	w	0	0	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	?	?
0	31	w	1	0	?	?
1	0	r	0	0	?	?
1	0	w	0	0	?	?
1	1	r	0	0	?	?
1	1	w	0	0	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	?	?
1	31	w	1	1	?	?

Here, we indicated the `rw_counter` and `rwc_inc_left` entries in the first row with X and Y to indicate that these are (as we have not yet imposed the constraints from `constrain_rw_counter`) arbitrary values — but looked up by the EVM circuit. Entries marked with a question mark indicate that these are fully unconstrained cells (apart from `rw_counter` being used in lookups into the rw table).

We are now ready to consider the constraints imposed by `constrain_rw_counter` in the case where `is_memory_copy` is 0 (indicating we are not doing a memory-to-memory copy).

```
// Decrement rwc_inc_left for the next row, when an RWrwc_inc_left operation
    happens.
let rwc_diff = is_rw_type.expr() * is_row_end.expr();
let new_value = meta.query_advice(rwc_inc_left, CURRENT) - rwc_diff;
let is_last = meta.query_advice(is_last_col, CURRENT);

// ...

// At the end, it must reach 0.
let update_or_finish = select::expr(
    not::expr(is_last.clone()),
    meta.query_advice(rwc_inc_left, NEXT_ROW),
    0.expr(),
);
```

```
// ...

cb.condition(not::expr(is_memory_copy.clone()), |cb| {
    cb.require_equal(
        "rwc_inc_left[1] == rwc_inc_left[0] - rwc_diff, or 0 at the end",
        new_value.clone(),
        update_or_finish,
    );
});

// ...

// Maintain rw_counter based on rwc_inc_left. Their sum remains constant in
    all cases.
cb.condition(not::expr(is_last.expr()), |cb| {
    cb.require_equal(
        "rw_counter[0] + rwc_inc_left[0] == rw_counter[1] + rwc_inc_left[1]",
        meta.query_advice(rw_counter, CURRENT) +
    meta.query_advice(rwc_inc_left, CURRENT),
        meta.query_advice(rw_counter, NEXT_ROW) +
    meta.query_advice(rwc_inc_left, NEXT_ROW),
    );
});
```

As `is_rw_type` will be 1, `rwc_diff` will be the same as `is_row_end`. Thus, `new_value` will be `rwc_inc_left` decremented by one at rows for the last byte of a word, otherwise it will be `rwc_inc_left` unchanged.

There are then two cases to distinguish depending on whether `is_last` is 1 or 0. If it is 0, then `update_or_finish` will be `rwc_inc_left` of the next row. Thus, `rwc_inc_left` of the next row will be constrained to be the same as for the current row or decremented by 1 in the case of the last byte in a word. We can fill out some parts of the table with this information:

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	Y
0	0	w	0	0	?	Y
0	1	r	0	0	?	Y
0	1	w	0	0	?	Y
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	?	Y
0	31	w	1	0	?	$Y - 1$
1	0	r	0	0	?	$Y - 2$
1	0	w	0	0	?	$Y - 2$
1	1	r	0	0	?	$Y - 2$
1	1	w	0	0	?	$Y - 2$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	?	$Y - 2$
1	31	w	1	1	?	$Y - 3$

When we are on the very last row, where `is_last = 1`, we will have `update_or_finish = 0` instead. As on that row `is_row_end = 1` as well, we have `new_value = rwc_inc_left - 1`. Thus, the constraint `new_value = update_or_finish` enforces `rwc_inc_left = 1`. We can see that we already filled in that cell with $Y - 3$. This implies that we must have $Y = 4$ in this example:

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	4
0	0	w	0	0	?	4
0	1	r	0	0	?	4
0	1	w	0	0	?	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	?	4
0	31	w	1	0	?	3
1	0	r	0	0	?	2
1	0	w	0	0	?	2
1	1	r	0	0	?	2
1	1	w	0	0	?	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	?	2
1	31	w	1	1	?	1

Finally, we consider the remaining constraint:

```
// Maintain rw_counter based on rwc_inc_left. Their sum remains constant in
    all cases.
cb.condition(not::expr(is_last.expr()), |cb| {
    cb.require_equal(
        "rw_counter[0] + rwc_inc_left[0] == rw_counter[1] + rwc_inc_left[1]",
        meta.query_advice(rw_counter, CURRENT) +
    meta.query_advice(rwc_inc_left, CURRENT),
        meta.query_advice(rw_counter, NEXT_ROW) +
    meta.query_advice(rwc_inc_left, NEXT_ROW),
    );
});
```

This ensures that the sum of `rw_counter` and `rwc_inc_left` cannot change within a copy event. As the sum in the first row of our example is $X + 4$, it must thus be in the remaining rows as well, so we can fill in the remainder of the table:

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	4
0	0	w	0	0	X	4
0	1	r	0	0	X	4
0	1	w	0	0	X	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	X	4
0	31	w	1	0	$X+1$	3
1	0	r	0	0	$X+2$	2
1	0	w	0	0	$X+2$	2
1	1	r	0	0	$X+2$	2
1	1	w	0	0	$X+2$	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	$X+2$	2
1	31	w	1	1	$X+3$	1

We would like to emphasize in particular here that the constraints enforce that the value of `rwc_inc_left` in the first row accurately reflect the number of words read or written. There is a single countdown for read and write rows together.

The case of memory-to-memory copies

In the case of memory-to-memory copies, the table we obtained above would not be adequate. To see why that is, imagine that the copy operation is to copy words 0 and 1 to words 1 and 2. With the above table, the reads and writes happening would be as follows:

rw_counter	
X	Read word 0
$X+1$	Write the word just read to word 1
$X+2$	Read word 1
$X+3$	Write the word just read to word 2

Note that with this access order, the first write will overwrite the original value at word 1. However,

MCOPY should operate as if an intermediate buffer were used. Thus this access order will not work for MCOPY. It is instead necessary to first carry out all reads and only then perform the writes.

The table that is intended in the case of our example above, but for MCOPY, would be as follows:

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	4
0	0	w	0	0	$X+2$	2
0	1	r	0	0	X	4
0	1	w	0	0	$X+2$	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	X	4
0	31	w	1	0	$X+2$	2
1	0	r	0	0	$X+1$	3
1	0	w	0	0	$X+3$	1
1	1	r	0	0	$X+1$	3
1	1	w	0	0	$X+3$	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	$X+1$	3
1	31	w	1	1	$X+3$	1

Let us now consider the constraints for this case:

```
// Decrement rwc_inc_left for the next row, when an RWrwc_inc_left operation
    happens.
let rwc_diff = is_rw_type.expr() * is_row_end.expr();
let new_value = meta.query_advice(rwc_inc_left, CURRENT) - rwc_diff;
let is_last = meta.query_advice(is_last_col, CURRENT);
let is_last_two = meta.query_advice(is_last_col, NEXT_ROW);

// ...

let update_or_finish_mcopy = meta.query_advice(rwc_inc_left, NEXT_STEP);

// ...

// handle is_memory_copy case: for all read steps, `rwc_inc_left` decrease by
    1 or 0,
// for all write steps, `rwc_inc_left` decrease by 1 or 0 as well. this is not
    the same as
```

```
// normal case( `rwc_inc_left` decrease by 1 or 0 for consecutive read steps-->
    write step
// --> read step -->write step ...).
cb.condition(
    is_memory_copy * not::expr(is_last_two) * not::expr(is_last.clone()),
    |cb| {
        cb.require_equal(
            "rwc_inc_left[2] == rwc_inc_left[0] - rwc_diff, or 0 at the end",
            new_value,
            update_or_finish_mcopy,
        );
    },
);

// Maintain rw_counter based on rwc_inc_left. Their sum remains constant in
    all cases.
cb.condition(not::expr(is_last.expr()), |cb| {
    cb.require_equal(
        "rw_counter[0] + rwc_inc_left[0] == rw_counter[1] + rwc_inc_left[1]",
        meta.query_advice(rw_counter, CURRENT) +
    meta.query_advice(rwc_inc_left, CURRENT),
        meta.query_advice(rw_counter, NEXT_ROW) +
    meta.query_advice(rwc_inc_left, NEXT_ROW),
    );
});
```

For rows except the last two, `rwc_inc_left` of two rows down (`NEXT_STEP` is `2`, whereas `NEXT_ROW` is `1`) is constrained to be `new_value`, which is the possibly decremented value of `rwc_inc_left` of the current row, as before. For the last two rows, there is no constraint imposed. We can fill in the table with this information again. The constraint that the sum of `rw_counter` and `rwc_inc_left` stays constant is the same as before, so we fill that in as well. Here we gave the value of `rw_counter` in the second row the value `Z`, which is arbitrary and not a constraint.

word	word_index	r/w	is_row_end	is_last	rw_counter	rwc_inc_left
0	0	r	0	0	X	Y
0	0	w	0	0	$X+Y-Z$	Z
0	1	r	0	0	X	Y
0	1	w	0	0	$X+Y-Z$	Z
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	31	r	1	0	X	Y
0	31	w	1	0	$X+Y-Z$	Z
1	0	r	0	0	$X+1$	$Y-1$
1	0	w	0	0	$X+Y-Z+1$	$Z-1$
1	1	r	0	0	$X+1$	$Y-1$
1	1	w	0	0	$X+Y-Z+1$	$Z-1$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	31	r	1	0	$X+1$	$Y-1$
1	31	w	1	1	$X+Y-Z+1$	$Z-1$

Implications for the EVM circuit

Using the notation from the above example, the EVM circuit fixes X, as this is the current `rw_counter` at the start of the copy operation. The value of Y is `copy_rwc_inc`, which is used in the state transition to obtain the correct `rw_counter` for the next operation and ultimately constrain the number of nonpadding entries available in the rw table.

As we could see, this value can be chosen arbitrarily. By choosing a value that is larger than the actual number of read and write lookups done for the copy event, an adversary will cause an insufficient number of padding rows to be enforced in the rw table. They will then be able to insert additional rows between the reads and writes done by the copy event and the next legitimate reads and writes. This could include arbitrary writes to storage, including in other contexts.

Another value that can be chosen arbitrarily, and independently of Y, is Z, the `rw_counter` of the first write performed by the copy event. This enables another variant with which an adversary could insert malicious entries into the rw table, even if Y is set to the intended value. They could repeat the same `MCOPY` operation twice, and in such a way that during writes done during the copy operation, the previous values are also the same. In that case, the writes looked up by the two `MCOPY`s will only differ in possibly the `rw_counter`. The adversary can then actually use the *same* `rw_counter` (by setting Z to the same value), looking up the same entry in the rw table twice. Thus, the number of distinct write lookups done by the two copy events will be half of the intended and counted number. This frees up rows in the rw table that can be used for malicious writes.

Note that choosability of Z also means that writes can potentially happen in a different order than intended; for example by having two MCOPYs but the writes of the first are performed at the time of the second MCOPY and vice versa.

Impact

For transactions involving execution of MCOPY instructions of nonzero length, and adversary can insert arbitrary malicious writes to the rw table. The correct ordering of reads and writes is also not ensured for MCOPY.

We tested this issue with the variant in which `copy_rwc_inc` is set to a larger than intended value, allowing to insert a malicious write to the rw table. To do so, we made the following changes to the bus mapping.

```
--- a/bus-mapping/src/evm/opcodes/mcopy.rs
+++ b/bus-mapping/src/evm/opcodes/mcopy.rs
@@ -66,6 +66,9 @@ fn gen_copy_event(
     let (read_steps, write_steps, prev_bytes) =
         state.gen_copy_steps_for_memory_to_memory(exec_step, src_addr,
    dst_addr, length)?;

+ // Adding additional rw operation
+ state.memory_write_word(exec_step, 0x20u64.into(), Word::from(4247u64))?;
+
     Ok(CopyEvent {
         src_type: CopyDataType::Memory,
         // use call_id as src id for memory --> memory type.
```

This above first change to `gen_copy_event` adds an additional memory write when generating the data for the MCOPY execution step. This is the malicious write we insert. To be able to pass the constraints on the number of padding start rows, we must also make `copy_rwc_inc` too large. For this, we changed `CopyEvent::rw_counter_delta` to increase the value of `copy_rwc_inc` by one as well.

```
--- a/bus-mapping/src/circuit_input_builder/execution.rs
+++ b/bus-mapping/src/circuit_input_builder/execution.rs
@@ -522,8 +522,7 @@ impl CopyEvent {
             // operations.
             return self.full_length();
         }
-
- (self.is_source_rw() as u64 + self.is_destination_rw() as u64) *
    (self.full_length() / 32)
+ (self.is_source_rw() as u64 + self.is_destination_rw() as u64) *
    (self.full_length() / 32) + 1
     }
```

```
}
```

The code in `bus-mapping` is used to generate witnesses, so these changes only impact assignments to the circuit, not the constraints (and thus also not the verification key).

We then added the following test case to `zkevm-circuits/src/evm_circuit/execution/mcopy.rs`:

```
fn zellic_mcopy_rwc_inc() {
    let src_offset = Word::from(0x20);
    let dest_offset = Word::from(0x40);
    let length = Word::from(0x20);
    let mut code = Bytecode::default();
    code.append(&bytecode! {
        PUSH32(word!("123456789"))
        PUSH2(0x20)
        MSTORE
        PUSH32(length)
        PUSH32(src_offset)
        PUSH32(dest_offset)
        #[start]
        MCOPY
        PUSH2(0x20)
        MLOAD
        PUSH2(0x40)
        MLOAD
        STOP
    });

    let ctx = TestContext::<3, 1>::new(
        None,
        |accs| {
            accs[0]

    .address(address!("0x0000000000000000000000000000000000000cafe"))
                .code(code);
            accs[1]

    .address(address!("0x0000000000000000000000000000000000000010"))
                .balance(Word::from(1u64 << 20));
        },
        |mut txs, accs| {
            txs[0]
                .to(accs[0].address)
                .from(accs[1].address)
                .gas(1_000_000.into());
        },
```

```
        |block, _tx| block.number(0x1111111),
    )
    .unwrap();

    CircuitTestBuilder::new_from_test_ctx(ctx)
        .params(CircuitsParams {
            max_copy_rows: 1750,
            ..Default::default()
        })
        .run();
}
```

Note that an `MCOPY` from `0x20` to `0x40` happens, copying the value `0x123456789 = 4886718345`. Then the contents of `0x20` and `0x40` are loaded to the stack, which should both be `0x123456789 = 4886718345`.

We ran this test with `RUST_BACKTRACE=1 RUST_LOG=trace cargo test zellic_mcopy_rwc_inc --nocapture`. The test passes, indicating that all constraints are satisfied. Below is an excerpt of the output produced by the state circuit, with the lines reordered so that `rw_counter` is increasing:

```
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65479 row:Memory { rw_counter: 41, is_write: true, call_id:
    1, memory_address: 32, value: 4886718345, value_prev: 0 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65483 row:Memory { rw_counter: 42, is_write: true, call_id:
    1, memory_address: 64, value: 0, value_prev: 0 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65480 row:Memory { rw_counter: 49, is_write: false, call_id:
    1, memory_address: 32, value: 4886718345, value_prev: 4886718345 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65484 row:Memory { rw_counter: 50, is_write: true, call_id:
    1, memory_address: 64, value: 4886718345, value_prev: 0 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65481 row:Memory { rw_counter: 51, is_write: true, call_id:
    1, memory_address: 32, value: 4247, value_prev: 4886718345 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65482 row:Memory { rw_counter: 55, is_write: false, call_id:
    1, memory_address: 32, value: 4247, value_prev: 4247 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65485 row:Memory { rw_counter: 56, is_write: false, call_id:
    1, memory_address: 64, value: 4886718345, value_prev: 4886718345 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65486 row:Memory { rw_counter: 60, is_write: false, call_id:
    1, memory_address: 64, value: 4886718345, value_prev: 4886718345 }
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65487 row:Memory { rw_counter: 61, is_write: false, call_id:
```

```
    1, memory_address: 96, value: 0, value_prev: 0 }
```

For `rw_counter` being 49 and 50, we can see the read and write happening due to the copy event. At 51, we then see the illegitimate write that we inserted; a value of 4247 is written, which should not happen according to the bytecode we are executing. Then at counter 55, this smuggled-in value is read again, as part of the MLOAD. There is also a corresponding line with this value being written to the stack, confirming this:

```
[2024-05-23T13:59:00Z TRACE zkevm_circuits::state_circuit] state circuit
    assign offset:65478 row:Stack { rw_counter: 54, is_write: true, call_id:
    1, stack_pointer: 1023, value: 4247 }
```

Recommendations

It should be ensured that the read and write `rwc_inc_left` countdowns match up and end with 1. Concretely, `rwc_inc_left` for the last read must be one bigger than `rwc_inc_left` for the first write, and `rwc_inc_left` for the last write must be 1.

Currently, there are essentially two constraints: `rwc_inc_left[i+2] = rwc_inc_left - (1 if last byte of a word, else 0)` and `rw_counter[i+1] + rwc_inc_left[i+1] = rw_counter[i] + rwc_inc_left[i]`. In this notation, `rwc_inc_left[i]` refers to the value of the `rwc_inc_counter` column in the `i`-th row of the copy event. The latter constraint can be rewritten and split up in three cases by plugging in the first one:

1. For the first read and write, `rw_counter[1] - rw_counter[0] = rwc_inc_left[0] - rwc_inc_left[1]`.
2. When at the last byte of a word, `rw_counter[i+2] = rw_counter[i] + 1`.
3. When not at the last byte of a word, `rw_counter[i+2] = rw_counter[i]`.

To fix this issue, we suggest to add for the memory-to-memory case the constraints:

A. Constrain `rwc_inc_left[0] = 2 * rwc_inc_left[1]`. In other words, conditional on being on the first row of a copy event that is memory-to-memory, constrain `rwc_inc_left` to be twice the value of `rwc_inc_left` in the next row.

B. Constrain the last row's `rwc_inc_left` to be 1.

We will now discuss why this fixes the issues. Let `n` be the number of writes and reads (i.e., `n` writes and also `n` reads). Then constraint B implies that `rwc_inc_left` must be 1 for the last write. Together with the constraint `rwc_inc_left[i+2] = rwc_inc_left - (1 if last byte of a word, else 0)`, this implies that for the first write we must have `rwc_inc_left[1] = n`. Now constraint A implies `rwc_inc_left[0] = 2*n`. This means the `i`-th read has `rwc_inc_left = 2*n - i` and the `i`-th write has `rwc_inc_left = n - i`. Now constraint 1 implies that `rw_counter[1] = rw_counter[0] + n`. Constraints 2 and 3 then imply that the `i`-th read has `rw_counter = rw_counter[0] + i` and the `i`-th write has `rw_counter = rw_counter[0] + n + i`. That `rwc_inc_left[0] = 2*n` means that the rw

counter delta used back in the EVM circuit will be `2*n`, and we just showed that the actual reads and writes looked up by the copy circuit are using rw counters `rw_counter[0]` through `rw_counter[0] + 2*n - 1`. There is thus no way anymore to look up a different write or to obtain an incorrect rw counter delta.

See 4.1. ↗ for a discussion of a proposal to increase robustness against such vulnerabilities more long-term.

Remediation

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

- `9f34b5e5` ↗
- `ac610772` ↗
- `6a50fd6a` ↗
- `44472527` ↗
- `77f8c2ae` ↗
- `460b2f78` ↗
- `f07abbd7` ↗

3.2. Lack of constraints specific to transient storage and transaction receipts in the state circuit

Target	State circuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the state circuit, constraints specific to entries of the rw table with tag `RwTableTag::AccountTransientStorage` and `RwTableTag::TxReceipt` are missing. The two most important constraints missing due to this were the following:

1. Constraining the `initial_value` column to be zero in the case of `RwTableTag::AccountTransientStorage`
2. Constraining the `state_root` column not to have changed from the previous row in both cases

Even though constraints specific to `AccountTransientStorage` and `TxReceipt` are missing, the general constraints still apply. These include the following:

- Constraining `not_first_access`
- Constraining the `value` at a first access read to be `initial_value`
- Constraining the `value` at other reads to be the `value` in the previous row

Impact

The missing constraint for `initial_value` to be zero for `AccountTransientStorage` means that an adversary may set any value. This will then result in the first read of that slot to read this value chosen by the adversary. In some applications, smart contracts may rely critically on transient storage slots being initialized as zero.

Lack of constraints for the `state_root` column implies an adversary can arbitrarily change the value of `state_root` in rows of type `AccountTransientStorage` or `TxReceipt`. Such a change could then correspond to updating any data stored in the state root, such as the storage of any smart contract address or any account state data.

Recommendations

Implement the missing constraints for `AccountTransientStorage` and `TxReceipt` rows in the state circuit.

Remediation

We brought up the lack of constraints specific to `RwTableTag::AccountTransientStorage` rows in the kick-off call, and the missing constraints were quickly implemented. We then reviewed them as part of our audit. The case of `RwTableTag::TxReceipt` was out of scope for this audit.

This issue has been acknowledged by Scroll, and a fix was implemented in commit `92d34c51` ↗.

3.3. Source address is not constrained for `ErrorOOGMemoryCopyGadget`, allowing illegitimate reverts on `MCOPY`

Target	EVM circuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The EVM circuits `ErrorOOGMemoryCopyGadget` gadget is used to constrain reverts due to running out of gas on a memory copy operation. Let us consider this snippet from `ErrorOOGMemoryCopyGadget::configure` in `zkevm-circuits/src/evm_circuit/execution/error_oog_memory_copy.rs`:

```
let dst_memory_addr = MemoryExpandedAddressGadget::construct_self(cb);
// src can also be possible to overflow for mcopy.
let src_memory_addr = MemoryExpandedAddressGadget::construct_self(cb);

cb.stack_pop(dst_memory_addr.offset_rlc());
cb.stack_pop(src_memory_addr.offset_rlc());
cb.stack_pop(dst_memory_addr.length_rlc());

// ...

cb.require_equal(
    // for mcopy, both dst_memory_addr and dst_memory_addr likely overflow.
    "Memory address is overflow or gas left is less than cost",
    or::expr([
        dst_memory_addr.overflow(),
        src_memory_addr.overflow(),
        insufficient_gas.expr(),
    ]),
    1.expr(),
);
```

The length for `src_memory_addr` is never constrained, so an attacker can set it to anything. The implementation for `MemoryExpandedAddressGadget::overflow` and the relevant constraints are as follows.

```
/// Check if overflow.
pub(crate) fn overflow(&self) -> Expression<F> {
    not::expr(self.within_range())
```

```
}

/// Check if within range.
pub(crate) fn within_range(&self) -> Expression<F> {
    or::expr([
        self.length_is_zero.expr(),
        and::expr([
            self.sum_lt_cap.expr(),
            self.sum_within_u64.expr(),
            not::expr(self.offset_length_sum.carry().as_ref().unwrap()),
        ]),
    ])
}

fn construct_self(cb: &mut EVMConstraintBuilder<F>) -> Self {
    let offset = cb.query_word_rlc();
    let length = cb.query_word_rlc();
    let sum = cb.query_word_rlc();

    let sum_lt_cap = LtGadget::construct(
        cb,
        from_bytes::expr(&sum.cells[..N_BYTES_U64]),
        (MAX_EXPANDED_MEMORY_ADDRESS + 1).expr(),
    );

    let sum_overflow_hi = sum::expr(&sum.cells[N_BYTES_U64..]);
    let sum_within_u64 = IsZeroGadget::construct(cb, sum_overflow_hi);

    let length_is_zero = IsZeroGadget::construct(cb,
    sum::expr(&length.cells));
    let offset_length_sum = AddWordsGadget::construct(cb, [offset, length],
    sum);

    Self {
        length_is_zero,
        offset_length_sum,
        sum_lt_cap,
        sum_within_u64,
    }
}
```

When `length` can be set to an arbitrary value, an arbitrary value for `sum = offset + length` can be arranged, in particular a value larger than $2^{64} - 1$. Thus, it can be arranged that `sum_within_u64` is false, making `within_range` return false, and thus making `overflow` return true.

This means an attacker can prove an execution where arbitrary `MCOPY`s of their choice incorrectly fail due to being out of gas.

Impact

An attacker can prove execution of a transaction where `MCOPY`s of their choice might incorrectly revert. There are some scenarios in which an attacker can profit from such reverts. One scenario would be in cases where a caller wishes to use a feature the callee may or may not support, so the caller wraps the call in a try block, falling back to some default behavior in the case that the call reverted. Some examples of this are here ↗ and here ↗. Another scenario would be cases in which something negative for user A is caused by a transaction submitted by a third party B, for example in the case of liquidation of collaterals for a loan or opening of a commitment that shows that user A lost a bet.

Recommendations

Constrain the `length` used in `src_memory_addr` to be equal to the `length` in `dst_memory_addr` (which is in turn already constrained to be the length popped from the stack).

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `929589e2` ↗.

3.4. Step transition for `end_tx` not constrained

Target	EVM circuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The EVM circuit, which constrains the actual execution of instruction in the EVM, is organized around steps. For each step, a `StepState` is stored, containing the following data:

```
pub(crate) struct StepState<F> {
    /// The execution state selector for the step
    pub(crate) execution_state: DynamicSelectorHalf<F>,
    /// The Read/Write counter
    pub(crate) rw_counter: Cell<F>,
    /// The unique identifier of call in the whole proof, using the
    /// `rw_counter` at the call step.
    pub(crate) call_id: Cell<F>,
    /// The transaction id of this transaction within the block.
    pub(crate) tx_id: Cell<F>,
    /// Whether the call is root call
    pub(crate) is_root: Cell<F>,
    /// Whether the call is a create call
    pub(crate) is_create: Cell<F>,
    /// The block number the state currently is in. This is particularly
    /// important as multiple blocks can be assigned and proven in a single
    /// circuit instance.
    pub(crate) block_number: Cell<F>,
    /// Denotes the hash of the bytecode for the current call.
    /// In the case of a contract creation root call, this denotes the hash of
    /// the tx calldata.
    /// In the case of a contract creation internal call, this denotes the hash
    /// of the chunk of bytes from caller's memory that represent the
    /// contract init code.
    pub(crate) code_hash: Cell<F>,
    /// The program counter
    pub(crate) program_counter: Cell<F>,
    /// The stack pointer
    pub(crate) stack_pointer: Cell<F>,
    /// The amount of gas left
    pub(crate) gas_left: Cell<F>,
```

```
    /// Memory size in words (32 bytes)
    pub(crate) memory_word_size: Cell<F>,
    /// The counter for reversible writes
    pub(crate) reversible_write_counter: Cell<F>,
    /// The counter for log index
    pub(crate) log_id: Cell<F>,
    /// Whether this is end_tx. Boolean.
    pub(crate) end_tx: Cell<F>,
}
```

To constrain these values from one step to the next, the individual gadgets for EVM instructions (as well as, e.g., block end) create a `StepStateTransition`, which stores the transition for most of the `StepState` fields. A transition could be a change to an absolute value or a delta to be added to the previous value.

```
pub(crate) struct StepStateTransition<F: Field> {
    pub(crate) rw_counter: Transition<Expression<F>>,
    pub(crate) call_id: Transition<Expression<F>>,
    pub(crate) is_root: Transition<Expression<F>>,
    pub(crate) is_create: Transition<Expression<F>>,
    pub(crate) code_hash: Transition<Expression<F>>,
    pub(crate) program_counter: Transition<Expression<F>>,
    pub(crate) stack_pointer: Transition<Expression<F>>,
    pub(crate) gas_left: Transition<Expression<F>>,
    pub(crate) memory_word_size: Transition<Expression<F>>,
    pub(crate) reversible_write_counter: Transition<Expression<F>>,
    pub(crate) log_id: Transition<Expression<F>>,
    pub(crate) end_tx: Transition<Expression<F>>,
}

pub(crate) enum Transition<T> {
    Same,
    Delta(T),
    To(T),
    Any,
}
```

They then call the `require_step_state_transition` function of the `EVMConstraintBuilder`, defined in `zkevm-circuits/src/evm_circuit/util/constraint_builder.rs`, which constrains the `StepState` to transition to the next step as given by the `StepStateTransition`:

```
pub(crate) fn require_step_state_transition(
    &mut self,
    step_state_transition: StepStateTransition<F>,
) {
```

```
    macro_rules! constrain {
        ($name:tt) => {
            match step_state_transition.$name {
                Transition::Same => self.require_equal(
                    concat!("State transition (same) constraint of ",
    stringify!($name)),
                    self.next.state.$name.expr(),
                    self.curr.state.$name.expr(),
                ),
                Transition::Delta(delta) => self.require_equal(
                    concat!("State transition (delta) constraint of ",
    stringify!($name)),
                    self.next.state.$name.expr(),
                    self.curr.state.$name.expr() + delta,
                ),
                Transition::To(to) => self.require_equal(
                    concat!("State transition (to) constraint of ",
    stringify!($name)),
                    self.next.state.$name.expr(),
                    to,
                ),
                _ => {}
            }
        };
    }

    constrain!(rw_counter);
    constrain!(call_id);
    constrain!(is_root);
    constrain!(is_create);
    constrain!(code_hash);
    constrain!(program_counter);
    constrain!(stack_pointer);
    constrain!(gas_left);
    constrain!(memory_word_size);
    constrain!(reversible_write_counter);
    constrain!(log_id);
}
```

However, `constrain!(end_tx)` is missing here. There are thus no constraints added that constrain the next steps's `end_tx` value based on the transition and the current steps `end_tx`.

Impact

As this part of the code was not in scope for our assessment, we did not fully explore the impact of this finding.

Unless otherwise prevented, it appears likely that the missing constraints allow ending transactions at any step by setting the `end_tx` field to true and the `execution_state` dynamic selector to `EndTx`. In practice, this could be used for example by taking a flash loan and then ending the transaction without paying it back. In the other direction, it may be possible to continue execution past instructions such as `STOP` that should end the transaction.

Recommendations

Add the constraints for `end_tx`:

```
    constrain!(rw_counter);
    constrain!(call_id);
    constrain!(is_root);
    constrain!(is_create);
    constrain!(code_hash);
    constrain!(program_counter);
    constrain!(stack_pointer);
    constrain!(gas_left);
    constrain!(memory_word_size);
    constrain!(reversible_write_counter);
    constrain!(log_id);
    constrain!(end_tx);
}
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 0c7a5602 ↗.

3.5. Completeness issue for some out-of-gas cases for MCOPY

Target	EVM circuit		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Reverts due to being out of gas during memory copy operations are handled by the EVM circuit's `ErrorOOGMemoryCopyGadget`. However, the gas costs of `MCOPY` instructions are not calculated correctly.

In the `MCopyGadget`, the gas cost is calculated as follows:

```
let memory_expansion = MemoryExpansionGadget::construct(
    cb,
    [
        memory_src_address.end_offset(),
        memory_dest_address.end_offset(),
    ],
);
let memory_copier_gas = MemoryCopierGasGadget::construct(
    cb,
    memory_src_address.length(),
    memory_expansion.gas_cost(),
);

// dynamic cost + constant cost
let gas_cost = memory_copier_gas.gas_cost() +
    OpcodeId::MCOPY.constant_gas_cost().expr();
```

The above gas-cost calculation corresponds to the Ethereum specification. However, in the `ErrorOOGMemoryCopyGadget`, we have this:

```
let memory_expansion = MemoryExpansionGadget::construct(cb, [dst_memory_
    addr.end_offset()]);
let memory_copier_gas = MemoryCopierGasGadget::construct(
    cb,
    dst_memory_addr.length(),
    memory_expansion.gas_cost(),
);
```

```
let constant_gas_cost = select::expr(
    is_extcodecopy.expr(),
    // According to EIP-2929, EXTCODECOPY constant gas cost is different for
    cold and warm
    // accounts.
    select::expr(
        is_warm.expr(),
        GasCost::WARM_ACCESS.expr(),
        GasCost::COLD_ACCOUNT_ACCESS.expr(),
    ),
    // Constant gas cost is same for CALLDATACOPY, CODECOPY and RETURNDATACOPY.
    OpcodeId::CALLDATACOPY.constant_gas_cost().expr(),
);

let insufficient_gas = LtGadget::construct(
    cb,
    cb.curr.state.gas_left.expr(),
    constant_gas_cost + memory_copier_gas.gas_cost(),
);
cb.require_equal(
    // for mcopy, both dst_memory_addr and dst_memory_addr likely overflow.
    "Memory address is overflow or gas left is less than cost",
    or::expr([
        dst_memory_addr.overflow(),
        src_memory_addr.overflow(),
        insufficient_gas.expr(),
    ]),
    1.expr(),
);
```

Note that memory expansion only takes into account the destination, not the source. According to the Ethereum specification, not only writes but also reads cause memory expansion. An `MCOPY` operation of nonzero length from a very high memory address to a low memory address will thus cause memory expansion, causing an out-of-gas error (as long as the source address was high enough). However, the `ErrorOOGMemoryCopyGadget` will not consider this memory expansion and thereby calculate a too small `memory_copier_gas.gas_cost()`, which then makes `insufficient_gas` false, making it impossible to satisfy the constraint at the end of the snippet above, unless the addresses are so large to also overflow.

We confirmed this issue with the following test:

```
#[test]
fn zellic_mcopy_out_of_gas() {
    let src_offset = Word::from(0x1000000);
    let dest_offset = Word::from(0x0);
```

```
    let length = Word::from(0x20);
    let mut code = Bytecode::default();
    code.append(&bytecode! {
        PUSH32(length)
        PUSH32(src_offset)
        PUSH32(dest_offset)
        MCOPY
        STOP
    });

    let ctx = TestContext::<3, 1>::new(
        None,
        |accs| {
            accs[0]

    .address(address!("0x0000000000000000000000000000000000000cafe"))
                .code(code);
            accs[1]

    .address(address!("0x0000000000000000000000000000000000000010"))
                .balance(Word::from(1u64 << 20));
        },
        |mut txs, accs| {
            txs[0]
                .to(accs[0].address)
                .from(accs[1].address)
                .gas(1_000_000.into());
        },
        |block, _tx| block.number(0x1111111),
    )
    .unwrap();

    CircuitTestBuilder::new_from_test_ctx(ctx)
        .params(CircuitsParams {
            max_copy_rows: 1750,
            ..Default::default()
        })
        .run();
}
```

As expected, trying to run this test fails because of the `Memory address is overflow or gas left is less than cost` constraint:

```
[2024-05-27T20:49:00Z INFO halo2_proofs::dev] MockProver synthesize took
    245.328931ms
[2024-05-27T20:49:00Z DEBUG halo2_proofs::dev] regions.len() = 18
```

```
thread 'evm_circuit::execution::mcopy::test::zellic_mcopy_out_of_gas' panicked
    at zkevm-circuits/src/test_util.rs:99:17:
assertion `left == right` failed
  left: Err([ConstraintCaseDebug {
    constraint: Constraint {
        gate: Gate {
            index: 381,
            name: "ErrorOutOfGasMemoryCopy",
        },
        index: 51,
        name: "ErrorOutOfGasMemoryCopy: Memory address is overflow or gas left
    is less than cost",
    },
    location: InRegion {
        region: Region 15 ('Execution step region1_0'),
        offset: 22,
    },
    cell_values: [
        (
            DebugVirtualCell {
                name: "",
                column: DebugColumn {
                    column_type: Advice,
                    index: 71,
                    annotation: "EVM_q_step",
                },
                rotation: 0,
            },
            "1",
        ),
        (
            DebugVirtualCell {
                name: "",
                column: DebugColumn {
                    column_type: Fixed,
                    index: 20,
                    annotation: "",
                },
                rotation: 0,
            },
            "1",
        ),
    ],
}])
 right: Ok(())
```

Impact

It will not be possible to prove blocks in which an `MCOPY` reverts due to an out of gas error that was solely caused by memory expansion caused by the reads from the source. While it appears unlikely that such reverts appear in benign transactions, it cannot be ruled out that such a completeness issue could be used as a denial-of-service vector.

Recommendations

In the `MCOPY` case, make the calculation of `memory_expansion` in `ErrorOOGMemoryCopyGadget::configure` correspond to the calculation done in `MCopyGadget::configure` by extending the list passed as the `addresses` argument to `MemoryExpansionGadget::construct` with `src_memory_addr.end_offset()`.

Remediation

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

- `929589e2` ↗
- `bc49717d` ↗
- `f3433f43` ↗
- `f4f920d2` ↗
- `6deef485` ↗
- `2e30e31c` ↗

3.6. Incorrect gas-cost calculation for `MCOPY` out-of-gas test cases

Target	EVM circuit tests		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `zkevm-circuits/src/evm_circuit/execution/error_oog_memory_copy.rs`, the `tests::TestingData::new_for_mcopy` function is used to generate test cases for `MCOPY`, generating some bytecode to perform an `MCOPY` and calculating the gas costs.

```
pub fn new_for_mcopy(
    src_offset: u64,
    dst_offset: u64,
    copy_size: u64,
    gas_cost: Option<u64>,
) -> Self {
    let bytecode = bytecode! {
        PUSH32(copy_size)
        PUSH32(src_offset)
        PUSH32(dst_offset)
        MCOPY
    };

    let gas_cost = gas_cost.unwrap_or_else(|| {
        let cur_memory_word_size = (src_offset + 31) / 32;
        let memory_word_size = (dst_offset + copy_size + 31) / 32;

        OpcodeId::PUSH32.constant_gas_cost().0 * 3
            + memory_copier_gas_cost(
                cur_memory_word_size,
                memory_word_size,
                copy_size,
                GasCost::COPY.as_u64(),
            )
    });

    Self { bytecode, gas_cost }
}
```

The function `memory_copier_gas_cost` expects as arguments the current memory size in words, the memory size in words after expansion, the number of bytes to copy, and the amount of gas used per word copied. The first two arguments are incorrect, however.

Impact

Test cases using this function can behave in unintended ways (for example, panic) in some instances due to the incorrect gas calculation.

Recommendations

For the shown bytecode, no writes to memory have been performed before `MCOPY`, so the current memory size should be zero words. The new memory size will then be the maximum of `(src_offset + copy_size + 31) / 32` and `(dst_offset + copy_size + 31) / 32` if `copy_size` is nonzero and zero if `copy_size` is zero.

Remediation

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

- `e88d53a0` ↗
- `695afd31` ↗

3.7. Improvements possible for testing out-of-gas on MCOPY

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

Description

The EVM circuit has some tests to test out-of-gas errors being handled by the `ErrorOOGMemoryCopyGadget` on MCOPY execution, given by the following test function:

```
#[test]
fn test_oog_memory_copy_for_mcopy() {
    for (src_offset, dest_offset, copy_size) in TESTING_MCOPY_PARIS {
        let testing_data =
            TestingData::new_for_mcopy(*src_offset, *dest_offset, *copy_size,
    None);

        test_root(&testing_data);
        test_internal(&testing_data);
    }
}
```

This function first uses `TestingData::new_for_mcopy` to obtain a `TestingData` struct, containing bytecode executing an MCOPY instruction as well as the calculated gas cost. Then `test_root` and `test_internal` are called, which perform the actual test. In the former case, the bytecode including the MCOPY will be executed directly (at the top level of the transaction), and in the latter case, it will be executed one level lower, with a wrapper contract calling the bytecode to be tested. In both cases, the amount of gas is adjusted so that it should be just one gas too little to execute the MCOPY instruction:

```
fn test_root(testing_data: &TestingData) {
    let gas_cost = GasCost::TX
        .0
        // Decrease expected gas cost (by 1) to trigger out of gas error.
        .checked_add(testing_data.gas_cost - 1)
        .unwrap_or(MOCK_BLOCK_GAS_LIMIT);
    let gas_cost = if gas_cost > MOCK_BLOCK_GAS_LIMIT {
        MOCK_BLOCK_GAS_LIMIT
    } else {
```

```
        gas_cost
    };

    let ctx = TestContext::<2, 1>::new(
        None,
        account_0_code_account_1_no_code(testing_data.bytecode.clone()),
        |mut txs, accs| {
            txs[0]
                .from(accs[1].address)
                .to(accs[0].address)
                .gas(gas_cost.into());
        },
        |block, _tx| block.number(0xcafe_u64),
    )
    .unwrap();

    CircuitTestBuilder::new_from_test_ctx(ctx)
        .params(CircuitsParams {
            max_copy_rows: 1750,
            ..Default::default()
        })
        .run();
}

fn test_internal(testing_data: &TestingData) {
    // ...

    // Code A calls code B.
    let code_a = bytecode! {
        // populate memory in A's context.
        PUSH8(U256::from_big_endian(&rand_bytes(8)))
        PUSH1(0x00) // offset
        MSTORE
        // call ADDR_B.
        PUSH1(0x00) // retLength
        PUSH1(0x00) // retOffset
        PUSH32(0x00) // argsLength
        PUSH32(0x20) // argsOffset
        PUSH1(0x00) // value
        PUSH32(addr_b.to_word()) // addr
        // Decrease expected gas cost (by 1) to trigger out of gas error.
        PUSH32(gas_cost_b - 1) // gas
        CALL
        STOP
    };
```

```
    // ...
}
```

If we run this test with `RUST_BACKTRACE=1 RUST_LOG=trace cargo test test_oog_memory_copy_for_mcopy -- --nocapture`, we can also see in the debug output that an out-of-memory error happens, as seen here for the very last test case:

```
[2024-06-04T18:40:55Z TRACE zkevm_circuits::evm_circuit::execution]
    assign_exec_step offset: 48 state ErrorOutOfGasMemoryCopy step: ExecStep {
    call_index: 1, rw_indices: [(Stack, 22), (Stack, 23), (Stack, 24),
    (CallContext, 49), (CallContext, 50), (CallContext, 51), (CallContext, 52),
    (CallContext, 53), (CallContext, 54), (CallContext, 55), (CallContext, 56),
    (CallContext, 57), (CallContext, 58), (CallContext, 59), (CallContext, 60),
    (CallContext, 61), (CallContext, 62)], copy_rw_counter_delta: 0,
    execution_state: ErrorOutOfGasMemoryCopy, rw_counter: 95, program_counter:
    99, stack_pointer: 1021, gas_left: 14, gas_cost: 42, memory_size: 0,
    reversible_write_counter: 0, reversible_write_counter_delta: 0, log_id:
    0, opcode: Some(MCOPY), block_num: 0, aux_data: None } call: Call { id:
    50, is_root: false, is_create: false, code_hash:
    690884794144076031747796054244630245235459425678395572163820579066864808141 79,
    rw_counter_end_of_reversion: 111, caller_id: 1, depth: 2, caller_address:
    0x000000000000000000000000000000000cafe111, callee_address:
    0x000000000000000000000000000000000cafe222, code_address:
    Some(0x000000000000000000000000000000000cafe222), call_data_offset: 0,
    call_data_length: 0, return_data_offset: 0, return_data_length: 0,
    last_callee_id: 0, last_callee_return_data_offset: 0,
    last_callee_return_data_length: 0, value: 0, is_success: false,
    is_persistent: false, is_static: false }
[2024-06-04T18:40:55Z DEBUG
    zkevm_circuits::evm_circuit::execution::error_oog_memory_copy]
    ErrorOutOfGasMemoryCopy: opcode = MCOPY, gas_left = 14, gas_cost = 42
```

However, the way these tests are run does not check that an out-of-gas error actually happens on the `MCOPY`.

We verified this by changing the amount of gas available to be sufficient:

```
fn test_root(testing_data: &TestingData) {
    let gas_cost = GasCost::TX
        .0
        // Decrease expected gas cost (by 1) to trigger out of gas error.
        .checked_add(testing_data.gas_cost - 1)
        .checked_add(testing_data.gas_cost + 100000)
        .unwrap_or(MOCK_BLOCK_GAS_LIMIT);
    let gas_cost = if gas_cost > MOCK_BLOCK_GAS_LIMIT {
```

```
            MOCK_BLOCK_GAS_LIMIT
        } else {
            gas_cost
        };

        // ...
}

fn test_internal(testing_data: &TestingData) {
        // ...

        // Code A calls code B.
        let code_a = bytecode! {
            // populate memory in A's context.
            PUSH8(U256::from_big_endian(&rand_bytes(8)))
            PUSH1(0x00) // offset
            MSTORE
            // call ADDR_B.
            PUSH1(0x00) // retLength
            PUSH1(0x00) // retOffset
            PUSH32(0x00) // argsLength
            PUSH32(0x20) // argsOffset
            PUSH1(0x00) // value
            PUSH32(addr_b.to_word()) // addr
            // Decrease expected gas cost (by 1) to trigger out of gas error.
            PUSH32(gas_cost_b - 1) // gas
            PUSH32(gas_cost_b + 100000) // gas
            CALL
            STOP
        };

        // ...
}
```

The tests still pass. However, no out-of-gas error occurs:

```
[2024-06-04T18:46:28Z TRACE zkevm_circuits::evm_circuit::execution]
    assign_exec_step offset: 48 state MCOPY step: ExecStep { call_index: 1,
    rw_indices: [(Stack, 22), (Stack, 23), (Stack, 24), (Memory, 2), (Memory,
    3), (Memory, 4), (Memory, 5), (Memory, 6), (Memory, 7), (Memory, 8),
    (Memory, 9)], copy_rw_counter_delta: 8, execution_state: MCOPY,
    rw_counter: 95, program_counter: 99, stack_pointer: 1021, gas_left:
    100015, gas_cost: 42, memory_size: 0, reversible_write_counter: 0,
    reversible_write_counter_delta: 0, log_id: 0, opcode: Some(MCOPY),
    block_num: 0, aux_data: None } call: Call { id: 50, is_root: false,
    is_create: false, code_hash:
```

```
    690884794144076031747796054244630245235459425678395572163820579066864808141 79,
    rw_counter_end_of_reversion: 0, caller_id: 1, depth: 2, caller_address:
    0x0000000000000000000000000000000000cafe111, callee_address:
    0x0000000000000000000000000000000000cafe222, code_address:
    Some(0x0000000000000000000000000000000000cafe222), call_data_offset: 0,
    call_data_length: 0, return_data_offset: 0, return_data_length: 0,
    last_callee_id: 0, last_callee_return_data_offset: 0,
    last_callee_return_data_length: 0, value: 0, is_success: true,
    is_persistent: true, is_static: false }
[2024-06-04T18:46:28Z TRACE zkevm_circuits::evm_circuit::execution]
    assign_exec_step offset: 50 state STOP step: ExecStep { call_index: 1,
    rw_indices: [(CallContext, 49), (CallContext, 50), (CallContext, 51),
    (CallContext, 52), (CallContext, 53), (CallContext, 54), (CallContext, 55),
    (CallContext, 56), (CallContext, 57), (CallContext, 58), (CallContext, 59),
    (CallContext, 60), (CallContext, 61)], copy_rw_counter_delta: 0,
    execution_state: STOP, rw_counter: 106, program_counter: 100,
    stack_pointer: 1024, gas_left: 99973, gas_cost: 0, memory_size: 288,
    reversible_write_counter: 0, reversible_write_counter_delta: 0, log_id:
    0, opcode: Some(STOP), block_num: 0, aux_data: None } call: Call { id:
    50, is_root: false, is_create: false, code_hash:
    690884794144076031747796054244630245235459425678395572163820579066864808141 79,
    rw_counter_end_of_reversion: 0, caller_id: 1, depth: 2, caller_address:
    0x0000000000000000000000000000000000cafe111, callee_address:
    0x0000000000000000000000000000000000cafe222, code_address:
    Some(0x0000000000000000000000000000000000cafe222), call_data_offset: 0,
    call_data_length: 0, return_data_offset: 0, return_data_length: 0,
    last_callee_id: 0, last_callee_return_data_offset: 0,
    last_callee_return_data_length: 0, value: 0, is_success: true,
    is_persistent: true, is_static: false }
```

Impact

Should the gas-cost calculation done by `TestingData::new_for_mcopy` be correct, these tests will test that the assignments to the `ErrorOOGMemoryCopyGadget` satisfy the constraints. However, should `TestingData::new_for_mcopy` return an incorrect gas cost that is too high, then these tests will not actually use the `ErrorOOGMemoryCopyGadget`. Those constraints will thus not be tested. Furthermore, the current tests do not ensure that the gas costs calculated inside the circuit match those calculated by `TestingData::new_for_mcopy`.

Note that Finding 3.6. ↗ precisely discusses a bug in the gas-cost calculation by `TestingData::new_for_mcopy`.

Recommendations

We recommend to enhance the testing code in the following way. Firstly, the bytecode is run twice. Once with available gas set to `testing_data.gas_cost - 1` and once with `testing_data.gas_cost`. Secondly, the testing code checks whether the test actually did reach an out-of-gas error on the `MCOPY` instruction. The test should then only succeed if all of the following are true:

1. Both runs succeeded, as in the constraints were satisfied.
2. The first run reached an out-of-gas error on the `MCOPY` instruction.
3. The second run did not reach an out-of-gas error on the `MCOPY` instruction.

This will ensure that the gas-cost calculation done by `TestingData::new_for_mcopy` matches the calculation done in circuit in the cases tested and that the `ErrorOOGMemoryCopyGadget` is indeed tested.

Remediation

This issue has been acknowledged by Scroll. Scroll informed us that they are considering to add the discussed tests in the future.

4. 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.

4.1. More robust constraints ensuring that all rw table entries are legitimate

Here we discuss a possible way to make the circuits more robust against the kind of issue discussed in Finding 3.1. ↗.

We recall that the rw table is used to keep track of the consistency of read and write operations to, for example, storage, memory, and stack. This table is constrained by the state circuit, which ensures that, for example, reads that follow a write to some location will read the value written, that reads do not change the value, and so on. The state circuit does not however check that the entries are actually legitimate, so from the perspective of the state circuit only, any writes may appear in the table. Non-`Start` entries of the rw table are intended to have a unique ID, given by the `counter`, which is to increment on each read or write. Entries of the `Start` type have separate counters, counting up on each row. The number of rows that the state circuit has is fixed, and they are lexicographically ordered, first by the type, and those of type `Start` come first. The table is padded by adding `Start` rows, and `rw_counter` is constrained to increase by one between any two `Start` rows.

The end block gadget in the EVM circuit is performing lookups of `Start` rows in the rw table as follows (`EndBlockGadget::configure` function in `zkevm-circuits/src/evm_circuit/execution/end_block.rs`):

```
let max_rws = cb.query_copy_cell();

// ...

// If the block is empty, we do 0 rw_table lookups
// If the block is not empty, we will do 1 call_context lookup
// and add 1 withdraw_root lookup
let total_rws = not::expr(is_empty_block.expr())
    * (cb.curr.state.rw_counter.clone().expr() - 1.expr() + 1.expr())
    + 1.expr();

// ...

// 3. Verify rw_counter counts to the same number of meaningful rows in
// rw_table to ensure there is no malicious insertion.
// Verify that there are at most total_rws meaningful entries in the rw_table
cb.rw_table_start_lookup(1.expr());
cb.rw_table_start_lookup(max_rws.expr() - total_rws.expr());
// Since every lookup done in the EVM circuit must succeed and uses
// a unique rw_counter, we know that at least there are
```

```
// total_rws meaningful entries in the rw_table.
// We conclude that the number of meaningful entries in the rw_table
// is total_rws.
```

Here, `max_rws` is intended to be the fixed total number of rows in the rw table when including padding and `total_rws` the number of rw rows that the EVM circuit tracked as having looked up (perhaps through the copy circuit). It is looked up that a `Start` row with `rw_counter = 1` and a `Start` row with `rw_counter = max_rws - total_rws` exist. As `max_rws - total_rws` should be positive and small enough that no wraparound can happen, there thus must be at least `max_rws - total_rws` many `Start` rows. This leaves at most `total_rws` many rows for everything else. Thus, if there really were `total_rws` many different legitimate rw lookups, then this ensures that there are no additional illegitimate ones in the rw table.

However, this hinges on `cb.curr.state.rw_counter` actually accurately reflecting the number of legitimate rw lookups done. But there are many different places in which rw lookups are done, including not in the EVM circuit itself, as in the case of copies where the rw lookups are done by the copy circuit. This means there are many places in which a mistake could mean that the `rw_counter` is not correctly tracked in a state transition, thereby allowing malicious writes to be inserted into the rw table. Finding 3.1. ↗ is an example of this, where `cb.curr.state.rw_counter` does not track the number of legitimate rw lookups correctly. When there is a mistake such that the tracked counter is increased too much in one of the EVM circuit gadgets, then this will always be a critical issue due to allowing to add arbitrary illegitimate rw entries. For defense in depth, it could be considered to have a more robust catch-all check rather than relying on the `rw_counter` being tracked correctly in the many different gadgets.

What would prevent illegitimate rw rows from being inserted without relying on `rw_counter` tracking being fully correct for every state transition would be to directly ensure that there is no non-`Start` row in the rw table that has not been looked up legitimately. There can be multiple ways to implement this. Here we discuss one style of idea of how this could be done.

Let H be a hash function acting as a random oracle. What is described below will need to be done in a phase after the rw table and all lookups in it have been assigned, and it needs to use a challenge c that depends on those.

In the `EVMConstraintBuilder::rw_lookup` function (the function all lookups to the rw table in the EVM circuit go through), calculate a hash of the lookup being done together with the challenge, so $H\left(c||L\right)$ where L is all the data associated to the lookup, and add this to an accumulator. At the end of execution, the EVM circuit will then have calculated the sum of hashes for all rw lookups done by the EVM circuit (i.e., $\sum_{L\in\mathcal{L}_{\text{EVM}}} H\left(c||L\right)$ where $\mathcal{L}_{\text{EVM}}$ is the set of all rw lookups done by the EVM circuit). Similarly in the copy circuit, also calculate the sum of hashes for rw lookups being done; $\sum_{L\in\mathcal{L}_{\text{copy}}} H\left(c||L\right)$ where $\mathcal{L}_{\text{copy}}$ is the set of all rw lookups done by the copy circuit. Both circuits expose this value to the super circuit, which adds them up and thereby obtains the sum over the hash of each actually looked-up rw row, $\sum_{L\in\mathcal{L}_{\text{EVM}}\cup\mathcal{L}_{\text{copy}}} H\left(c||L\right)$. The state circuit should then similarly calculate such a sum over all non-`Start` rows and expose it, so $\sum_{L\in\mathcal{L}_{\text{State}},L_{\text{type}\neq\text{START}}} H\left(c||L\right)$, where $\mathcal{L}_{\text{State}}$ is the set of rows in the rw table. Finally, the super circuit checks that these two values agree, $\sum_{L\in\mathcal{L}_{\text{EVM}}\cup\mathcal{L}_{\text{copy}}} H\left(c||L\right) = \sum_{L\in\mathcal{L}_{\text{State}},L_{\text{type}\neq\text{START}}} H\left(c||L\right)$.

The first attacker model we consider here is where an attacker cannot modify which legitimate

rw lookups are being done by the EVM and copy circuit, but they can cause the counter tracking to be off, so they can add additional rw entries. As the legitimate lookups must succeed, passing the above new check would then amount to finding illegitimate rw entries $\mathcal{L}_{\text{attack}}$ such that $\sum_{L \in \mathcal{L}_{\text{attack}}} H(c||L) = 0$. But c is a hash of, among other inputs, the rw table, and that includes these illegitimate entries. All summands of the left-hand side are thus unpredictable random values until all $\mathcal{L}_{\text{attack}}$ have been fixed. This should imply that the attacker can do no better than using brute force to search for a positive number illegitimate entries where the left-hand side sums to zero, which should then be computationally intractable. Thus, the only feasible solution the prover could have used should be the one where there are zero summands (i.e., no illegitimate entries).

The second attacker model would be if the attacker can make two legitimate lookups actually look up the same entry in the table, as in the second variant discussed in Finding 3.1. ↗. This should also be stopped by the above check; this time, the attacker would need to satisfy $\sum_{L \in \mathcal{L}_{\text{attack}}} H(c||L) = \sum_{L \in \mathcal{L}_{\text{duplicate}}} H(c||L)$, where $\mathcal{L}_{\text{duplicate}}$ is the multiset of duplicate lookups (if the same row is looked up $n > 1$ times, then the entry should occur $n - 1$ times in $\mathcal{L}_{\text{duplicate}}$). This should again be infeasible for the same reasons as in the first scenario. Note that this kind of attack is one that will not be caught by counter tracking as it is implemented even if the rw counter deltas are all correct, because the attacker "saves" one lookup they can then use for a smuggled entry.

The above suggestion is a starting point for the possibility of a more robust way of ensuring that no illegitimate entries occur in the rw table. This particular suggestion might not be implementable in practice due to, for example, performance requirements, as this would require a large number of in-circuit hashes.

4.2. Unused `tx_id` column in `StepState`

The `StepState` for each step of execution in the EVM circuit holds data as follows:

```
#[derive(Clone, Debug)]
pub(crate) struct StepState<F> {
    /// The execution state selector for the step
    pub(crate) execution_state: DynamicSelectorHalf<F>,
    /// The Read/Write counter
    pub(crate) rw_counter: Cell<F>,
    /// The unique identifier of call in the whole proof, using the
    /// `rw_counter` at the call step.
    pub(crate) call_id: Cell<F>,
    /// The transaction id of this transaction within the block.
    pub(crate) tx_id: Cell<F>,
    /// Whether the call is root call
    pub(crate) is_root: Cell<F>,
    /// Whether the call is a create call
    pub(crate) is_create: Cell<F>,
    /// The block number the state currently is in. This is particularly
```

```
    /// important as multiple blocks can be assigned and proven in a single
    /// circuit instance.
    pub(crate) block_number: Cell<F>,
    /// Denotes the hash of the bytecode for the current call.
    /// In the case of a contract creation root call, this denotes the hash of
    /// the tx calldata.
    /// In the case of a contract creation internal call, this denotes the hash
    /// of the chunk of bytes from caller's memory that represent the
    /// contract init code.
    pub(crate) code_hash: Cell<F>,
    /// The program counter
    pub(crate) program_counter: Cell<F>,
    /// The stack pointer
    pub(crate) stack_pointer: Cell<F>,
    /// The amount of gas left
    pub(crate) gas_left: Cell<F>,
    /// Memory size in words (32 bytes)
    pub(crate) memory_word_size: Cell<F>,
    /// The counter for reversible writes
    pub(crate) reversible_write_counter: Cell<F>,
    /// The counter for log index
    pub(crate) log_id: Cell<F>,
    /// Whether this is end_tx. Boolean.
    pub(crate) end_tx: Cell<F>,
}
```

However, the field `tx_id` is never used, only assigned to. While in various places of the EVM circuit the transaction ID is needed, it is always taken from a lookup in the call context rather than from the `StepState`, like so (line taken from `SstoreGadget::configure`):

```
let tx_id = cb.call_context(None, CallContextFieldTag::TxId);
```

The `tx_id` field of `StepState` could thus be removed to save some cells and avoid confusion.

4.3. Asserting assumptions about equal gas costs

The `ErrorOOGMemoryCopyGadget` handles out-of-gas errors for different instructions that copy to memory, namely `CALLDATACOPY`, `CODECOPY`, `EXTCODECOPY`, `RETURNDATACOPY`, and `MCOPY`. For the case of instructions that are not `EXTCODECOPY`, the constant gas cost is obtained as in the following snippet from `ErrorOOGMemoryCopyGadget::configure` in `zkevm-circuits/src/evm_circuit/execution/error_oog_memory_copy.rs`:

```
// Constant gas cost is same for CALLDATACOPY, CODECOPY and RETURNDATACOPY.
OpcodeId::CALLDATACOPY.constant_gas_cost().expr(),
```

As the comment points out, a single constant can be used here as the relevant instructions (`MCOPY` should be added to the list in the comment) all have constant gas costs of 3. To ensure that the code here is changed should this change in the future, we recommend adding asserts to check this (e.g., `assert_eq!(OpcodeId::CALLDATACOPY.constant_gas_cost(), OpcodeId::MCOPY.constant_gas_cost())`). Then, should there at some point be an update where the constant gas costs become different, the assert will fail on tests, ensuring that it will not be forgotten to add the necessary logic distinguishing the opcodes here, thereby preventing an issue similar to Finding 3.3. ↗.

4.4. Comment explaining gas-cost calculations for `MCOPY` is incorrect

The `MCopyGadget::configure` function in `zkevm-circuits/src/evm_circuit/execution/mcopy.rs` contains code to calculate the gas cost of an `MCOPY` operation. This includes gas costs due to possible memory expansion. Memory is expanded both for reads and writes but only for bytes actually read or written. Thus, an `MCOPY` operation will not cause memory expansion when no bytes are copied (due to the length being zero), even for large addresses for source and destination. The relevant part of the source code in `MCopyGadget::configure` is the following:

```
let memory_src_address =
    MemoryAddressGadget::construct(cb, src_offset.clone(), length.clone());
let memory_dest_address =
    MemoryAddressGadget::construct(cb, dest_offset.clone(), length.clone());

// if no acutal copy happens, memory_word_size doesn't change.
    MemoryExpansionGadget handle
// it internally
let memory_expansion = MemoryExpansionGadget::construct(
    cb,
    [
        memory_src_address.end_offset(),
        memory_dest_address.end_offset(),
    ],
);
```

The comment here suggests that `MemoryExpansionGadget` handles the case of length zero. However, it does not actually contain any special logic for this case. In fact, from the above snippet it seems that it would be impossible for `MemoryExpansionGadget` to do so, at it is only passed the end offsets of the two addresses (so the start address plus the length).

The memory-expansion calculation is still correct, but it is due to a special behavior of the `MemoryAd-`

`dressGadget` instead. The `end_offset()` function is implemented as follows:

```
fn end_offset(&self) -> Expression<F> {
    self.offset() + self.length()
}

fn length(&self) -> Expression<F> {
    from_bytes::expr(&self.memory_length.cells)
}

pub(crate) fn offset(&self) -> Expression<F> {
    self.has_length() * from_bytes::expr(&self.memory_offset_bytes.cells)
}

pub(crate) fn has_length(&self) -> Expression<F> {
    1.expr() - self.memory_length_is_zero.expr()
}
```

The upshot is that `MemoryAddressGadget` will report an offset of zero whenever length is zero, independent of the original offset, and thus the reported end offset will in this case also be zero. This is what ensures that the memory-expansion calculation is correct.

We recommend to update the comment to clarify the reason the memory-expansion calculation is correct.

The comments were corrected in commit `8c1e8b4a` ↗.

4.5. Mistakes in the example table in the EVM circuit documentation

The file `docs/EVM_Circuit.md` documents the functioning of the EVM circuit. Under the section "Circuit Layout", an example table is provided. There are some differences between this table and the code.

In the table, `q_step_first` is enabled for all rows of the first step, but in the code it is only enabled on the first row of the first step, analogously for `q_step_last`. For `num_rows_until_next_step`, the table shows this being zero in the last row of a step and then counting down until the next zero. However, in the code, it is zero in the first row of each step. Thus, the column should be 0, 5, 4, 3, 2, 1, 0, 6, 5, … instead of 5, 4, 3, 2, 1, 0, 6, 5, ….

The `q_step_first` and `q_step_last` columns were fixed in commit `18c5444b` ↗.

4.6. Incorrect comments for memory gadgets

In `zkevm-circuits/src/evm_circuit/util/memory_gadget.rs`, there are some comments that appear to refer to constants that have since been renamed. Concretely, in `MemoryExpansionGadget::construct`, there are several comments referring to `MAX_QUAD_COST_IN_BYTES` and `MAX_MEMORY_SIZE_IN_BYTES` that should be `N_BYTES_GAS` and `N_BYTES_MEMORY_WORD_SIZE` instead. There are also other functions referring to the old names.

The comment for `MemoryCopierGasGadget::construct` seems copied from `MemoryExpansionGadget::construct` and does not fit the function it is documenting.

The incorrect constant names for `MemoryExpansionGadget::construct` were fixed in commit `9789bf33` ↗

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Scroll zkEVM circuits, we discovered seven findings. Four critical issues were found. One was of medium impact and the remaining findings were informational in nature.

5.1. 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.