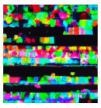


ZKSECURITY

Audit of zk-email

May 27, 2024

Introduction

On April 15, 2024, zkSecurity was engaged by the Ethereum Foundation to audit the zk-email project. The specific code to review was shared via GitHub as public repositories. The audit lasted 2 weeks with 3 consultants.

Scope

The scope of the audit included the following components:

zk-email-verify. The core circom library for email verification.

- https://github.com/zkemail/zk-email-verify
- The code was audited at commit `f2fb77c`.
- The audit covers all circom templates in the `packages/circuits` directory, including templates exposed as utilities and not directly used in the main `EmailVerifier` template. The TS helper code in `packages/helpers` is covered as well.
- The Solidity code in `packages/contracts` is excluded from the audit.

zk-regex. The regex-to-circom compiler, written in Rust.

- https://github.com/zkemail/zk-regex
- The code was audited at commit `9962a11`.
- The audit covers the core compiler in `packages/compiler`, the regex JSON inputs in `packages/circom/circuits/common` and the interaction between compiled regex circuits and zk-email circuits.

Summary and general comments

A number of issues were found in both zk-email-verify and zk-regex. You can find the full list near the end of this report.

All concrete high- and medium-severity issues were addressed by the zk-email team. For more information, see the "Client Response" section at the end of each finding. The commits at which all fixes are applied and reviewed by us are:

- 95cd90 for zk-email-verify
- 5396ec for zk-regex

We believe that in addition to these fixes, substantial changes are needed to ensure the security and robustness of the zk-regex library. We provided a finding with general recommendations for zk-regex, and we also recommend to re-audit this part of the project after all issues are addressed.

For existing users of zk-email, we want to emphasize that some of our findings pertain to using zk-regex or specific templates from zk-email-verify *as a library*. That means, they don't necessarily imply a critical vulnerability for end-to-end users of the `EmailVerifier` template. Specifically:

- Our finding on the complement regex does not affect `EmailVerifier`, as it doesn't use the complement regex operator.

- Our **finding on SHA256** does not affect `EmailVerifier` in a critical way, because exploiting it using malicious inputs is made infeasible by other constraints on those inputs in `EmailVerifier`.
- Our **recommendation to document implicit assumptions** does not affect `EmailVerifier` and is targeting usage of zk-email-verify as a library more generally.

Since the version of zk-regex that we audited is a complete rewrite of the one used in older versions of `EmailVerifier`, we can't comment on the security of previous versions of `EmailVerifier` more generally. We recommend to evaluate whether **the finding on start of string (`^`)** affects your use case.

A note on Circom compilation: During the audit, there was a concern that when compiling Circom with the `--O2` flag, some linear constraints are entirely removed from the constraint system. The zk-email team asked us to evaluate the security impact of this optimization, and whether `--O0` should be used instead for safety. In our assessment, `--O2` **can safely be used**, as it does not impact zk-email negatively in any way.

In general, while the effect of removing certain statements from the constraint system is surprising, we believe `--O2` shouldn't have a security impact for any project, since the optimization never changes the mathematical statement of the generated zk proof.

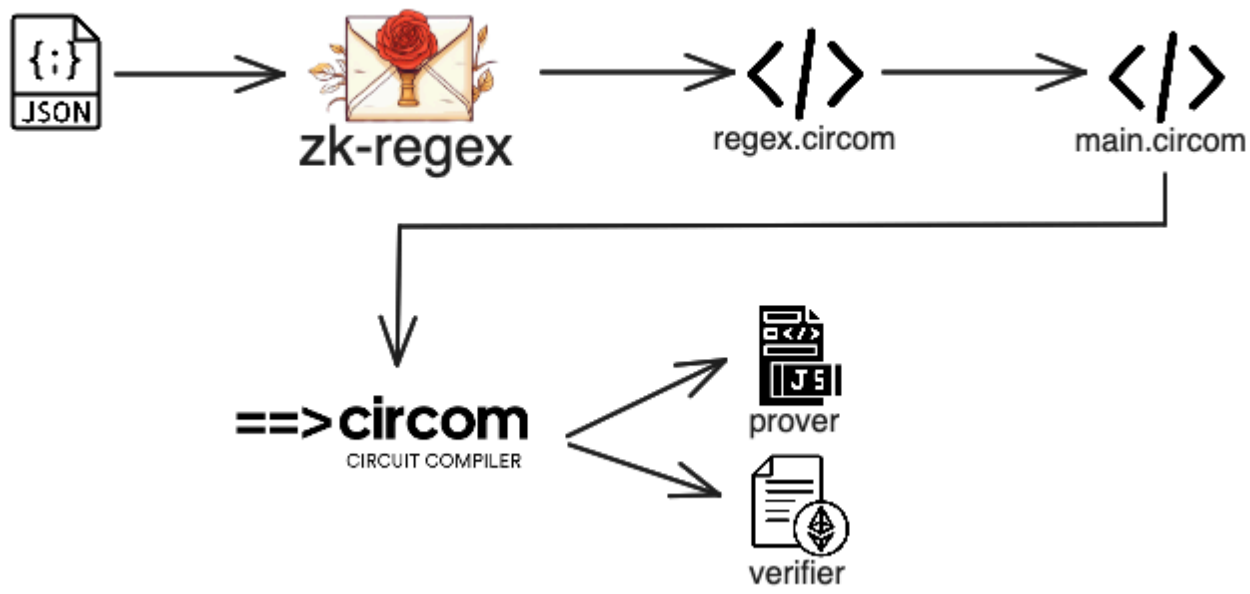
Overview of zk-regex

This section provides a simplified overview of the zk-regex compiler.

zk-regex is a tool that enables developers to create circuits that prove and verify matches of strings against a regex pattern and optionally reveal parts of the matching string. The zk-regex compiler takes a regex pattern as input and synthesizes a Circom template. Users can choose to decompose the regex into multiple parts and select specific parts to reveal. The compiler has two modes: the first, called **raw**, takes a regex and produces a Circom circuit; the second, called **decomposed**, takes a JSON with a decomposed regex and generates the appropriate Circom circuit. Hereafter, we will focus on the decomposed mode as it is commonly used in zk-email and is the most stable way to interact with zk-regex.

High-level Overview of zk-regex

The figure below describes the high-level interactions for using zk-regex. First, we define a regex for which we want to create a proof, then we use zk-regex to create the respective Circom code. In another Circom template, we can use our regex template as a component to feed data and get an output that represents whether the input satisfies the regex. Note that we can also produce a reveal array in the regex template to process a particular part of the input string that matches a specific regex. Afterward, we can use the circuit like any Circom circuit (e.g., perform preprocessing and create a JavaScript prover and a Solidity verifier). This allows us to perform regex checks on-chain using off-chain computations, without revealing the input string or only revealing part of it.



zk-regex overview.

Consider the following regex example:

```
m[01]+-[ab]+;
```

To use zk-regex to obtain a Circom circuit that creates and verifies a proof for inputs that match this regex, we can use the following JSON:

```
{
  "parts": [
    {
      "regex_def": "m[01]+-[ab]+;",
      "is_public": false
    }
  ]
}
```

To produce the Circom circuit, we run the following command:

```
zk-regex decomposed -d ./simple_regex_decomposed.json -c ./simple_regex.circom -t
SimpleRegex -g true
```

This results in a Circom template with the following structure. Note that we must define the number of bytes our input message should be at compile time. This limits how large the message can be.

```
template SimpleRegex(msg_bytes) {
    signal input msg[msg_bytes];
    signal output out;
    ...
}
```

To call that template as a component in another template, we could use the following code:

```
signal simpleRegexMatch <== SimpleRegex(maxLength)(msg);
simpleRegexMatch === 1;
```

This will result in accepting the proof if the `msg` matches the regex; otherwise, the constraint `simpleRegexMatch === 1` will fail. Now let's consider the case where we want to reveal the part between `-` and `;`. In this case, we would use the following JSON:

```
{
  "parts": [
    {
      "regex_def": "m[01]+-",
      "is_public": false
    },
    {
      "regex_def": "[ab]+",
      "is_public": true
    },
    {
      "regex_def": ";",
      "is_public": false
    }
  ]
}
```

This will result in a Circom template with the following structure:

```
template SimpleRegex(msg_bytes) {
```

```
    signal input msg[msg_bytes];
    signal output out;
    ...
    signal output reveal0;
    ...
}
```

That can be used as follows:

```
signal (simpleRegexMatch, simpleReveal[maxLength]) <== SimpleRegex(maxLength)(msg);
simpleRegexMatch === 1;
```

The reveal part is an array that is full of zeros, and in the positions of the revealing part, it contains the input that matches that part of the regex. For example, for the input `m01-aab;` and length of the message set to `10`, the result of the reveal part would be `aab`. Note that in practice the template takes as input UTF-8 encoded messages in decimal format. Hence, the actual input array would be `["109", "48", "49", "45", "97", "97", "98", "59", "0", "0"]` (you can use **this** tool for the conversion), and the reveal array would be `["0", "0", "0", "0", "97", "97", "98", "0", "0", "0"]`.

Synthesizing Circom Circuits for Regex

zk-regex operates on top of Deterministic Finite Automatons (DFAs), proving that a string satisfies a DFA and optionally revealing parts of the transition steps involved.

Regex to DFA

To understand how that works, we should go a step backward. First of all, a **Deterministic Finite Automaton (DFA)** is a finite-state machine that accepts or rejects a given string of symbols by running through a state sequence uniquely determined by the string. A regular expression can be converted to a DFA using the following process:

1. **Parse the Regular Expression**: Initially, the regex string is parsed into a data structure, such as a parse tree or an abstract syntax tree (AST). This structure represents the hierarchical organization of the regex, including its characters, operators, and sub-expressions.
2. **Convert the Parse Tree to an NFA**: Using the parse tree, a Non-deterministic Finite Automaton (NFA) is constructed. An NFA is a finite-state machine that may include several possible transitions for a single state and a given input symbol.
3. **Convert the NFA to a DFA**: The NFA is then transformed into a DFA, which ensures a single, deterministic transition for each state and input symbol combination. This process can also include minimizing the DFA to reduce its complexity without changing the language it recognizes.

These steps are foundational in compiling regex into a format that can be processed efficiently. More information on this process can be found in various online resources.

DFA Definition

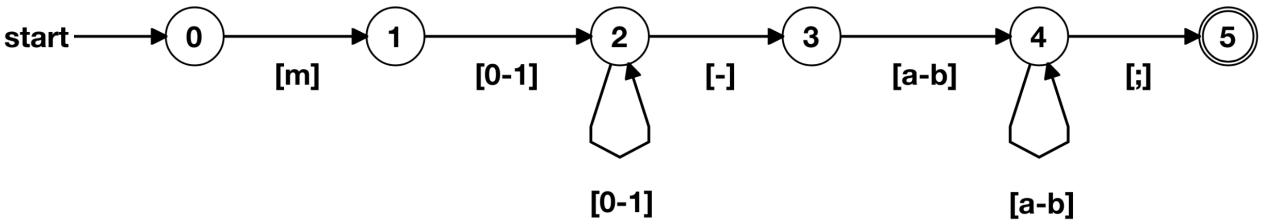
A deterministic finite automaton M is defined as a 5-tuple, $(\mathrm{Q}, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$, where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (the alphabet),
- δ is the transition function $\delta : \mathrm{Q} \times \Sigma \rightarrow \mathrm{Q}$,
- q_0 is the initial state, and

- F is the set of accept states, $\mathrm{F} \subseteq \mathrm{Q}$.

Example DFA Transition Matrix

Furthermore, typically, we have a transitions matrix for a DFA. Let's consider our example from above, i.e., `m[01]+-[ab]+;`.



DFA of m[01]+-[ab]+; (produced using https://zkregex.com/min_dfa).

The transition matrix will be:

DFA State	Type	`m`	`[0-1]`	`-`	`[a-b]`	`;`
0		1				
1			2			
2			2	3		
3					4	
4					4	5
5	Accept					

For a string (or part of the input string) to match the regex, it must match the DFA, or part of it should satisfy it.

Synthesizing Circom Code and Applying Constraints

Now, we can proceed to explain how zk-regex goes from the decomposed JSON input to the circom circuit and how it works. Initially, we will use an example where we have a single part and don't reveal anything. After that, we will see what happens when we have multiple parts and how we reveal some parts of the matched string.

The first step in the compiler, once all the inputs are read and parsed, is to convert the regex to a DFA. This is done using `DFAGraphInfo::regex_and_dfa`. If we have multiple regexes, their DFAs are computed separately and merged. Then, we compute which substrings should be revealed. The result is a structure called `RegexAndDFA`, which contains `regex_str, dfa_val, substrs_defs`. Let's first understand what happens when we have a single regex.

Let's consider the case where we have the following input.

```
{
  "parts": [
    {
      "regex_def": "m[01]+-[ab]+;",
      "is_public": false
    }
  ]
```

```
}
```

The resulting `RegexAndDFA` will be:

```
RegexAndDFA { regex_str: "m[01]+-[ab];", dfa_val: DFAGraph { states: [DFAState { type: "",
state: 0, edges: {1: {109}} }, DFAState { type: "", state: 1, edges: {2: {48, 49}} },
DFAState { type: "", state: 2, edges: {2: {48, 49}, 3: {45}} }, DFAState { type: "", state:
3, edges: {4: {97, 98}} }, DFAState { type: "", state: 4, edges: {5: {59}} }, DFAState {
type: "accept", state: 5, edges: {} }] }, substrs_defs: SubstrsDefs { substr_defs_array: [],
substr_endpoints_array: None } }
```

It transforms it by using the rust `DFA:Builder` after wrapping the regex into another regex with a specific anchored start `^` and end `$`, i.e.,`^Input_Regex$`. The output of the `DFA: Builder` is the following:

```
D 000000:
Q 000001:
 *000002:
  000003: m => 8
  000004: - => 5, 0-1 => 4
  000005: a-b => 6
  000006: ; => 7
  000007: EOI => 2
  000008: 0-1 => 4
```

This is transformed into a graph, and it is sorted. Note that `*000002` is the accepting state, and `EOI` is the end of the input. The parsing and conversion are happening in the function `parse_dfa_output`. Then, the graph is post-processed by `dfa_to_graph` to get the decimal representation of the UTF-8 characters, which is the alphabet of the provided regex. This function does the following:

- Define a custom key mapping for `"\\n", "\\r", "\\t", "\\v", "\\f", "\\0"` to their UTF-8-encoded decimals.
- It specially handles the space (` `) to transform it into a format that it can handle and translate to a UTF-8 encoding correctly.
- It handles ranges (e.g., `0-9`), by generating a set of all UTF-8 encoded values within that range.
- It also specially handles the custom key mappings
- Finally, it also specially handles hexadecimal representations (`\xNN`); it parses the string after `\x` as a hexadecimal number. Note that `DFA:Builder` translates non-ASCII characters to their hexadecimal UTF-8 encodings.

After having the `RegexAndDFA`, then it uses the `RegexAndDFA:gen_circom` to generate the Circom circuit, which in turn uses the `gen_circom_allstr` function to produce the Circom code for the DFA.

The `gen_circom_allstr` is a complicated function, and we believe it should be further documented and decoupled while also being extensively tested to reveal any subtle bugs. In a nutshell, it works as follows.

```
fn gen_circom_allstr(dfa_graph: &DFAGraph, template_name: &str, regex_str: &str) -> String
```

`dfa_graph` — a collection of states and, for each state, transitions to other states labeled by letters. `template_name` — the name of the generated Circom template. `regex_str` — the regex string implemented by the generated Circom template.

Important variables:

- `n: usize` — number of states in the DFA.
- `mut rev_graph: BTreeMap<usize, BTreeMap<usize, Vec<u8>>>` — reverse graph of the DFA, where the key is the state index and the value is a map of the states, that have transitions to the key state, and the letters of the transitions.
- `accept_node: usize` — the index of the accepting state of the DFA.
- `mut lines: Vec<String>` — the core generated Circom template code, each line in a separate element.
- `mut declarations: Vec<String>` — the initial declarations in the Circom code: pragma, import, template declaration, input and output signals, components defined at the top of the template.
- `mut init_code: Vec<String>` — the initialization code for `states[0][0..n-1]`.
- `mut accept_lines: Vec<String>` — the code for the computing the output from the accepting state.

High level flow:

1. Populate the reverse graph and `to_init_graph` based on the DFA. Also, identify the accepting states.
2. Check if there is only one accepting state, if not, panic.
3. Initialize counters for different types of checks.
4. Initialize data structures for different types of checks.
5. Generate the Circom code for each state in the DFA.
6. Generate the Circom code for the final state.
7. Combine all the generated Circom code and return.

The main signals and components used to determine if we have a match are:

- `signal in[num_bytes];`: This signal represents the user input prepended by `256`.
- `states[num_bytes+1][n];`: This signal represents the transitions of states based on the input. Every time we go from a valid state to an invalid one, we transition back to the initial state.
- `states_tmp[num_bytes+1][n];`: This is a helper signal to handle complicated conditions.
- `from_zero_enabled[num_bytes+1];`: This signal checks if we are at the zero state.
- `state_changed[num_bytes];`: This signal tracks the steps we took to change the state successfully (i.e., progress). The comparison components used to check if an input matches a character, a set of characters, or a range. These components are: `IsEqual`, `LessEqThan`, `MultiOR`, etc.
- `final_state_result`: This component is used to constrain `out` to `1` if we reach the acceptance state at some point.
- `is_consecutive[msg_bytes+1][n-1];`: Keeps track of whether the matches are consecutive across the input, ensuring that patterns requiring consecutive elements (like a*) are correctly validated.

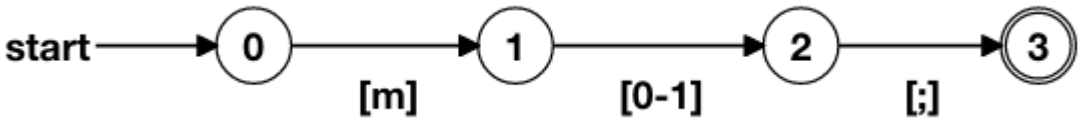
The steps in the Circom circuit are:

1. It has a main loop that goes over the inputs and mainly sets and constrains `states` and `state_changed`.
2. It sets and constrains `final_state_result`, which sets and constrains `out`.
3. It computes and appropriately constrains the `is_consecutive` signal.

For simplicity, we are going to describe how that works for the regex `m[ab];` and the input:

```
["109", "0", "109", "98", "59"]
```

So, our DFA is the following.



Visualisation for DFA of m[ab]+; (produced using https://zkregex.com/min_dfa).

The initialization Circom code would be:

```
signal input msg[msg_bytes];
signal output out;

var num_bytes = msg_bytes+1;
signal in[num_bytes];
in[0]<==255;
for (var i = 0; i < msg_bytes; i++) {
    in[i+1] <== msg[i];
}

component eq[4][num_bytes];
component and[3][num_bytes];
component multi_or[1][num_bytes];
signal states[num_bytes+1][4];
signal states_tmp[num_bytes+1][4];
signal from_zero_enabled[num_bytes+1];
from_zero_enabled[num_bytes] <== 0;
component state_changed[num_bytes];

for (var i = 1; i < 4; i++) {
    states[0][i] <== 0;
}
```

Note that the `in` has one more initial byte that is set to the invalid decimal `255`. `states` is two-dimensional. The first dimension is set to `num_bytes+1` because we want to have a dummy initial array in the beginning to process the initial byte, and it is used to keep the state for each byte. The second dimension represents the states of the DFA. The other components are used to make comparisons and evaluations. Next, we have the main loop, which is implemented as follows.

```
for (var i = 0; i < num_bytes; i++) {
    state_changed[i] = MultiOR(3);
    states[i][0] <== 1;
    eq[0][i] = IsEqual();
    eq[0][i].in[0] <== in[i];
    eq[0][i].in[1] <== 109;
    and[0][i] = AND();
    and[0][i].a <== states[i][0];
    and[0][i].b <== eq[0][i].out;
    states_tmp[i+1][1] <== 0;
    eq[1][i] = IsEqual();
    eq[1][i].in[0] <== in[i];
    eq[1][i].in[1] <== 97;
    eq[2][i] = IsEqual();
    eq[2][i].in[0] <== in[i];
    eq[2][i].in[1] <== 98;
    and[1][i] = AND();
```

```
    and[1][i].a <== states[i][1];
    multi_or[0][i] = MultiOR(2);
    multi_or[0][i].in[0] <== eq[1][i].out;
    multi_or[0][i].in[1] <== eq[2][i].out;
    and[1][i].b <== multi_or[0][i].out;
    states[i+1][2] <== and[1][i].out;
    eq[3][i] = IsEqual();
    eq[3][i].in[0] <== in[i];
    eq[3][i].in[1] <== 59;
    and[2][i] = AND();
    and[2][i].a <== states[i][2];
    and[2][i].b <== eq[3][i].out;
    states[i+1][3] <== and[2][i].out;
    from_zero_enabled[i] <== MultiNOR(3)([states_tmp[i+1][1], states[i+1][2], states[i+1]
[3]]);
    states[i+1][1] <== MultiOR(2)([states_tmp[i+1][1], from_zero_enabled[i] * and[0]
[i].out]);
    state_changed[i].in[0] <== states[i+1][1];
    state_changed[i].in[1] <== states[i+1][2];
    state_changed[i].in[2] <== states[i+1][3];
}
```

So, let's see how the transition signal `states` gets filled and how we constrain it to the correct value. The initial values for `states` are:

state	0
0	0
1	0
2	0
3	0

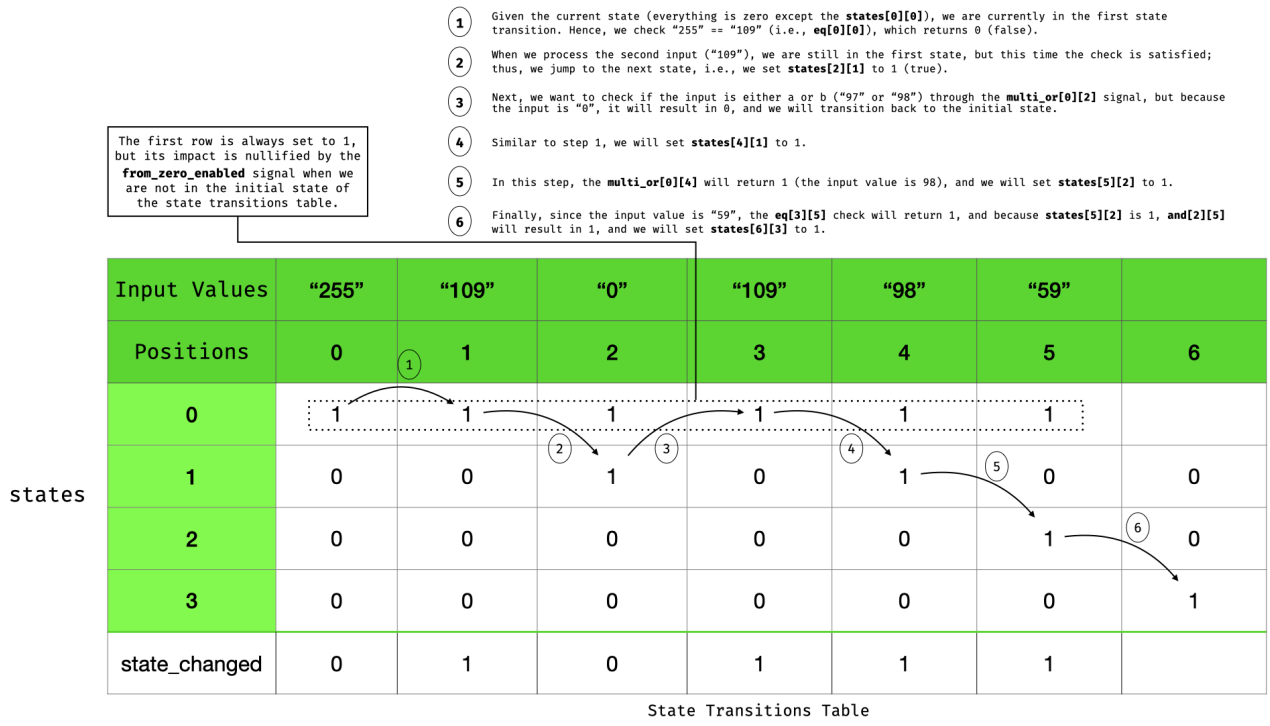
Where `state` column represent the DFA states, and `0` the current state when we processing `in[0]` (`255`). The first byte we process is `255` and the resulting state will be the following. We also keep track of the `state_changed` (`sc`) variable. Note, that we set the state from 1 to 3 for the next input.

state	0	1
0	1	
1	0	0
2	0	0
3	0	0
sc	0	

We set all three states to `0` because we are not making the correct transition from the first state (`0`) to state' 1', i.e., transitioning to `109` to go to state `1`. Next, we will be in state `0` and need `m` (`109`) to transition. Here, we should also note that we always set state `0` to `1`, but we handle that later with the `from_zero_enabled` component in case we are not actually in state `0`. Our next input is `109`. The resulting table will be:

state	0	1	2
0	1	1	
1	0	0	1
2	0	0	0
3	0	0	0
sc	0	1	

Note that we transition to state `1` because `and[0][i]` is `1` (meaning that state[i]0 is `1` and current input is `109`), and `from_zero_enabled` is `1`. Next, being in state `1` and processing the next input, we want to check if we can transition to state `2`, which means that we need input to be `97` or `98`. As it is not (`0`) we change all the states for the next input to `0` which means we return to state `0`. Following the same logic, we eventually will have the following state transition table.



State transitions table.

The following code verifies that we have at least a match:

```
component final_state_result = MultiOR(num_bytes+1);
for (var i = 0; i <= num_bytes; i++) {
    final_state_result.in[i] <== states[i][3];
}
out <== final_state_result.out;
```

This is because we check if we reach the state `3`, which is the accepted state, at least once. The final part of the code is the following:

```
signal is_consecutive[msg_bytes+1][3];
is_consecutive[msg_bytes][2] <== 1;
for (var i = 0; i < msg_bytes; i++) {
```

```
    is_consecutive[msg_bytes-1-i][0] <== states[num_bytes-i][3] * (1 -
is_consecutive[msg_bytes-i][2]) + is_consecutive[msg_bytes-i][2];
    is_consecutive[msg_bytes-1-i][1] <== state_changed[msg_bytes-i].out *
is_consecutive[msg_bytes-1-i][0];
    is_consecutive[msg_bytes-1-i][2] <== ORAnd()([(1 - from_zero_enabled[msg_bytes-i+1]),
states[num_bytes-i][3], is_consecutive[msg_bytes-1-i][1]]);
}
```

Which is a reverse loop to keep track of the state transitions using consecutive correct transitions.

Merging DFAs

Remember that we can split our regex into multiple parts. This functionality works by parsing each part in a separate DFA and then merging the DFAs by adding an anode from the first's accepted state to the second's first, etc.

Revealing sub-parts

When using the decomposed JSON and select to reveal a part of a regex, the `regex_and_dfa` code will identify from an accepted state which transitions lead to it and will add it to the `substr_defs_array` as a set of edges for the DFA of each public regex part. Then, in the Circom code, some additional code will be produced per public substring to be revealed that will be an output signal array of the size of the input message and will be filled with `0`'s and the input value for the part of the regex it matches. For example, for the following JSON.

```
{
  "parts": [
    {
      "regex_def": "m",
      "is_public": false
    },
    {
      "regex_def": "[ab]",
      "is_public": true
    },
    {
      "regex_def": ";",
      "is_public": false
    }
  ]
}
```

The following Circom code will be produced:

```
// substrings calculated: [{(1, 2)}]
signal is_substr0[msg_bytes];
signal is_reveal0[msg_bytes];
signal output reveal0[msg_bytes];
for (var i = 0; i < msg_bytes; i++) {
     // the 0-th substring transitions: [(1, 2)]
    is_substr0[i] <== MultiOR(1)([states[i+1][1] * states[i+2][2]]);
    is_reveal0[i] <== is_substr0[i] * is_consecutive[i][2];
    reveal0[i] <== in[i+1] * is_reveal0[i];
}
```

That checks if the proper state transitions are enabled and if they are part of a consecutive string. Then, it reveals the input; otherwise, it sets the value to `0`. Note that this code has some issues, as demonstrated in finding "In accepting input, reveal array reveals more values than the matching ones". Also, the user should be careful if the `0` value is supposed to be part of the revealed match.

UTF-8 encoding

zk-regex works on any UTF-8 encoded Unicode character in its decimal representation. This means that `.` should match any valid UTF-8 encoded beyond the line terminator. More specifically, every character that is described using the following format and represented in its decimal form is accepted (note that it should broken into bytes), and a single character could be broken into multiple bytes, resulting in multiple consecutive nodes in the DFA. Still, there needs to be more specific documentation on what is accepted and what is not. It is up to the user to define a verifier by either using a very strict regex that will be sure to accept only the strings that he wants or defining and checking how a message should look.

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 conversion table (ref: https://en.wikipedia.org/wiki/UTF-8).

Attack Methodologies

This section describes the various attack methodologies employed during the audit of the zk-regex compiler.

Underconstrained Produced Circom Code

Upon understanding the algorithm used to transform the regex into a Circom template, we manually checked the produced examples to identify potential underconstrained vulnerabilities in the produced code. Our focus was not on the logic of the translation but rather on the application of constraints in the circuit.

Incorrect Compilation and Handling of Regexes

We examined some common patterns (also used in the examples) to check if the produced code behaved as expected. This led to the discovery of high-severity bugs, such as findings "Complement regex could lead to soundness issues" and "Start ('^') and end ('$') of line match does not work".

Unexpected Leakage of Private Information

We also evaluated whether there were instances where the reveal arrays disclosed more inputs than those matched, or revealed parts of the input when there was no match. This investigation led to findings "The reveal array leaks inputs in a non-match" and "In accepting input, reveal array reveals more values than the matching ones".

Unexpected Behaviors of the Compiler

Finally, we tested whether the compiler could effectively process well-defined user inputs. This revealed findings, such as "**Regex using | leads to a compiler crash**".

Findings

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

ID	COMPONENT	NAME	RISK
00	zk-regex/packages/compiler/regex.rs	Complement regex leads to unsound template	High
01	zk-regex/packages/compiler	The zk-regex compiler lacks specifications and is not mature	High
02	zk-email-verify/packages/circuits/lib/sha.circom	SHA256 templates can be made return 0 on arbitrary inputs	High
03	zk-regex/packages/compiler/regex.rs	Start ('^') and end ('$') of line match does not work.	High
04	zk-regex/packages/compiler	The reveal array leaks inputs in a non-match	Medium
05	zk-regex/packages/compiler	In accepting input, reveal array reveals more values than the matching ones	Medium

ID	COMPONENT	NAME	RISK
06	zk-email-verify/packages/circuits/*	Templates do not document assumptions on input signals	Medium
07	zk-email-verify/packages/circuits/lib/base64.circom	Base64Decode does not distinguish between 'A' and '='	Low
08	zk-regex/packages/compiler	Compiler error when processing a valid regex	Low
09	zk-regex/packages/compiler/regex.rs	The matching all pattern leads to an error when compiling the produced circom template	Low
0a	zk-regex/packages/compiler/circom.rs	Regex using | leads to a compiler crash	Low
0b	zk-email-verifier/packages/circuits/lib/*	Unused templates	Low
0c	zk-regex/packages/compiler	Empty regex generates invalid Circom	Informational
0d	zk-regex/packages/compiler	Panics instead of returning error results	Informational

00 - Complement regex leads to unsound template

zk-regex/packages/compiler/regex.rs

High

Description

The complement regex (e.g., `[^0]` which matches any character that is not `0`) is not properly enforced in the current implementation. When a regex utilizing such a pattern, the resulting circuit may incorrectly accept inputs that do not match the intended pattern. For example, the following JSON defines a pattern that should reject strings containing `"a"` or `"b"`:

```
{
  "parts": [
    {
      "is_public": true,
      "regex_def": "[^ab]"
    }
  ]
}
```

However, when instantiated in a circuit with the input corresponding to `abb`:

```
component main { public [ msg ] } = SimpleRegex(3);

/* INPUT = {
 * "msg": ["97", "98", "98"]
 * }
 */
```

The circuit erroneously accepts this input (`out` is `1`), indicating a pass, even though the input clearly includes a non-matching string. Interestingly, the `reveal0` array does not reveal the non-matching pattern as it should (since it matches it).

Impact

This flaw could lead to significant soundness vulnerabilities in systems relying on these circuits for validation or verification processes.

Recommendation

Fix the compiler to properly handling complement regexes or throw an error to prevent soundness issues.

Client response

The issue was fixed in commit 6f2a5d0. We suggest adding further tests to catch any potential errors in corner cases.

01 - The zk-regex compiler lacks specifications and is not mature

zk-regex/packages/compiler

High

Description

The zk-regex compiler, which translates regular expressions into Circom circuits, has multiple significant issues affecting its reliability and functionality. The compiler crashes or produces errors on multiple occasions, indicating a robustness problem. Furthermore, in some cases it produces circuits with soundness issues. Some of the uncovered issues are very simple to trigger, meaning that they might cover other significant bugs. A lack of documentation and specifications adds to the problem by forcing developers into a trial-and-error approach, leading to usability issues and potential misconceptions about the zk-regex compiler. Additionally, the absence of formal documentation on the Regex features supported by the compiler leads to ambiguity in its functionality. This, combined with an unclear description of the internal process and algorithm of the compiler, results in a lack of clarity on how regex patterns are processed and converted into circuits. Finally, the compiler's test coverage is insufficient -- lacking both positive and negative tests and automated testing techniques -- raising concerns about the robustness of the compiler.

More specifically, both the generated Circom code and the zk-regex library itself lack specifications and documentation. It is extremely hard to understand the generated Circom code:

- What are the formats of input and outputs of the generated code? What assumptions are made about the input?
- The generated code is not structured well, nor commented, so it's very challenging to understand what it does, and more importantly, why.

The zk-regex library implementation code is also hard to understand. All public functions have clear signatures, but their implementations are poorly structured and completely undocumented. This applies both to internal data structure preparation and Circom code generation. On the data structure preparation side: there are two functions that are always called together, `parse_dfa_output` and `dfa_to_graph`.

```
    let mut graph = dfa_to_graph(&parse_dfa_output(&re_str));
```

These should be combined into one function (even if it'll end up as short as the one-liner above). More importantly, `dfa_to_graph` seems to be re-doing some of the work of `parse_dfa_output` (also, `regex_automata::dfa` API could be used directly instead of parsing its textual printout). Notably, these two are not enough, because `gen_circom_allstr` still needs more pre-processing before it can get to its actual purpose: it first prepares `rev_graph`, which could easily be passed to it (could be a part of `RegexAndDFA` struct). On the Circom code generation side: `gen_circom_allstr` is a function of 420+ lines of code, with 12 total loops and conditionals on the top level alone, occasionally very deeply nested (up to 6 levels). There's no documentation or comments.

Impact

Due to the aforementioned issues, the compiler's usability is greatly affected, and there's a high risk of unexpected behaviours leading to security vulnerability. Furthermore, it's hard to maintain this code. It is very hard

to understand the generated Circom code, which makes it hard to trust. Due to the lack of specifications, the complexity of the code, and the lack of testing, we believe that there are potentially more bugs in the code.

Recommendation

To address these concerns, the following steps are recommended:

- **Refactoring:** All mentioned functions could use refactoring, ideally such that they could be read in a single glance, calling to meaningfully named helper functions, each doing one thing. Finite state machines are a simple, well-known concept, and the generated code could reflect that: essentially, the generated circuit is a single pass through the FSM, feeding it the input string one byte at a time. The transition function could be factored out into a separate template, which would make the main template much simpler. Both the main template and the transition template should be either well-structured or well-commented (or both), explaining what they do and why.
- **Specification and Documentation**: Clearly define and document the supported regex features and syntax. Define which are the expected behaviors to enable users to securely use the compiler.
- **Process and Algorithm Documentation**: Describe the internal algorithm and steps the compiler takes in converting regex patterns to Circom circuits, to help auditors an developers alike.
- **Test Suite Enhancement**: Develop a comprehensive test suite, including both correct and incorrect cases, to validate all supported features and any additional functionalities, like revealing sub-regexes.
- **Automated Testing**: Implement automated testing techniques to systematically uncover subtle bugs and regressions, ensuring the compiler's reliability.

Client response

The issues described here were acknowledged by the client.

02 - SHA256 templates can be made return 0 on arbitrary inputs

zk-email-verify/packages/circuits/lib/sha.circom

High

Description

Both custom SHA256 templates `Sha256General` and `Sha256Partial`, as well as their bytes variants `Sha256Bytes` and `Sha256BytesPartial`, are vulnerable to an attack where the prover passes in a maliciously crafted `paddedInLength` which causes the returned hash result to be the all-zeros bit array.

In `Sha256General` and `Sha256Partial`, `paddedInLength` represents the bit length of the input message. It is connected to a newly created signal `inBlockIndex` by the constraint

```
inBlockIndex <-- (paddedInLength >> 9);
paddedInLength === inBlockIndex * 512;
```

SHA2 operates on the input in chunks, and `inBlockIndex` represents the number of chunks to be hashed. To support inputs of dynamic length, the template stores the intermediate hash result after each chunk, and at the end selects which one to return, using the `ItemAtIndex` template. This is done for each output bit:

```
component arraySelectors[256];
for (k=0; k<256; k++) {
    arraySelectors[k] = ItemAtIndex(maxBlocks);
    for (j=0; j<maxBlocks; j++) {
        arraySelectors[k].in[j] <== sha256compression[j].out[k];
    }
    arraySelectors[k].index <== inBlockIndex - 1; // The index is 0 indexed and the block
numbers are 1 indexed.
    out[k] <== arraySelectors[k].out;
}
```

The exploit we found manages to set `inBlockIndex - 1` to a large number close to the native modulus. In particular, the index is larger than the array size `maxBlocks`. On an index that exceeds the array bounds, the output of `ItemAtIndex` is zero. Therefore, the hash result ends up being an array of 0 bits -- regardless of the input value.

The issue is that `ItemAtIndex` does not properly constrain the index to lie in its intended range:

```
template ItemAtIndex(maxArrayLen) {
    // ...
    signal input index;
    // ...

    // Ensure that index < maxArrayLen
    component lessThan = LessThan(bitLength);
    lessThan.in[0] <== index;
    lessThan.in[1] <== maxArrayLen;
    lessThan.out === 1;
```

As we can see in this code excerpt, `index` is passed to `LessThan(bitLength)`. For `LessThan` to be sound, both inputs have to fit in `bitLength` bits. However, this property is never checked on the `index`.

Adding to the misfortune, the SHA2 input bit length `paddedInLength` is passed to an instance of `LessThan` as well, with the *same* mistake of not constraining its bit range (in `Sha256General` and `Sha256Partial`).

Generally speaking, when using `LessThan` without a bit constraint, small "negatives" modulo the field size p, like $-1 \equiv \mathrm{p} - 1$, are treated as being *less* than a small positive value. A detailed analysis of the SHA256 templates reveals that no constraints prevent `inBlockIndex` to be one of the following values:

$$-5, \ldots, -1, 0$$

These values are able to trick both instances of `LessThan` (the one for `paddedInLength` and the one for `index`) into returning 1, which is incorrect.

To exploit this in an isolated instance of `Sha256Bytes`, the prover just has to:

- pass in one of $-5 \cdot 64, \ldots, -64, 0$ as `paddedInLength` (note: this is multiplied by 8 in `Sha256Bytes`)
- change the `inBlockIndex` witness generation code to work for the negative case:

```
- inBlockIndex <-- (paddedInLength >> 9);
+ inBlockIndex <-- (paddedInLength / 512);
```

With this simple attack, we are able to create a valid proof that the SHA256 output consists of all zeros, regardless of its input.

Impact

Meaningfully exploiting this issue in `EmailVerifer` is infeasible, because `AssertZeroPadding` is also used on the two SHA256 input lengths (`emailHeaderLength` and `emailBodyLength`), and given small negative lengths implies that the email header or body consists of all zeros. However, consumers of the zk-email-verify library which use SHA256 templates in a different context may be vulnerable to the attack.

Recommendation

- In `ItemAtIndex`, remove `LessThan` and instead add an assertion that the sum of all `eqs[i]` equals 1. This proves that the index to select is one of the array indices.
- In all SHA256 templates, **document the assumption** that `paddedInLength` is constrained to a certain number of bits:
  - `ceil(log2(maxBitLength))` bits in the case of `Sha256General` and `Sha256Partial`
  - `ceil(log2(8 * maxByteLength))` bits in the case of `Sha256Bytes` and `Sha256BytesPartial`
- In `EmailVerifier`, use `Num2Bits` to constrain both `emailHeaderLength` and `emailBodyLength` to appropriate bit lengths, right after those signals are introduced.

Client response

The issue was fixed in commits **ad0ad6**, **e14e88** and **d667a1**.

03 - Start ('^') and end ('$') of line match does not work.

zk-regex/packages/compiler/regex.rs

High

Description

The start and end of line regexes, `^` and `$` respectively, do not function as expected. In zk-regex, both the regexes `a` and `^a` produce a DFA that accepts the string `"ba"`, despite the expectation that `^a` should not. This issue might arise because **each regex is wrapped in a "^...$"** pattern before generating the DFA. The following DFAs are produced the compiler.

```
RegexAndDFA { regex_str: "a", dfa_val: DFAGraph { states: [DFAState { type: "", state: 0,
edges: {1: {97}} }, DFAState { type: "accept", state: 1, edges: {} }] }, substrs_defs:
SubstrsDefs { substr_defs_array: [], substr_endpoints_array: None } }
```

```
RegexAndDFA { regex_str: "^a", dfa_val: DFAGraph { states: [DFAState { type: "", state: 0,
edges: {1: {97}} }, DFAState { type: "accept", state: 1, edges: {} }] }, substrs_defs:
SubstrsDefs { substr_defs_array: [], substr_endpoints_array: None } }
```

which are identical.

A pattern for testing this issue can be defined as follows:

```
{
  "parts": [
    {
      "is_public": false,
      "regex_def": "^a"
    }
  ]
}
```

The test can be instantiated in a circuit with:

```
component main { public [ msg ] } = SimpleRegex(3);

/* INPUT = {
 *  "msg": ["98", "97", "0"]
 * }
 */
```

which will return `out=1`.

Impact

This malfunction can lead to severe consequences for circuits that rely on accurate string boundary matching. A verifier could incorrectly accept strings that do not meet specified criteria, leading to potential security and

functionality flaws. This bug impacts various circuits within the `circuits/common/` directory, which are utilized by zk-email circuits.

Recommendation

It is critical to address this issue either by correctly supporting `^` and `$` patterns in regexes or by generating an error when these patterns are used. A refactoring might be required to support this feature that seems critical in some of the example circuits.

Client response

The issue was partially fixed in commit **6aac440**. Furthermore, the README was updated to outline some limitations. We suggest adding further tests to catch any potential errors in corner cases.

04 - The reveal array leaks inputs in a non-match

zk-regex/packages/compiler

Medium

Description

When a non-matching input is processed against a defined regex pattern, the reveal array erroneously exposes part of the input data. For instance, in a pattern meant to match `"aba"`, an input of `"aca"` should not match. Hence, the circuit should not reveal any characters. However, the reveal array incorrectly exposes the character `a` in the last position.

E.g.,

```
{
  "parts": [
    {
      "is_public": true,
      "regex_def": "aba"
    }
  ]
}
```

Input:

```
component main { public [ msg ] } = SimpleRegex(3);

/* INPUT = {
    "msg": ["97", "99", "97"]
} */
```

Output:

```
Output:
out = 0
reveal0[0] = 0
reveal0[1] = 0
reveal0[2] = 97
```

Impact

This behavior could lead to unintended behavior leaking private information.

Recommendation

The issue should be corrected to ensure the reveal array remains empty when the pattern does not match the input.

Client response

The issue was fixed in commit **0a5943a**. We suggest adding further tests to catch any potential errors in corner cases.

05 - In accepting input, reveal array reveals more values than the matching ones

zk-regex/packages/compiler

Medium

Description

When the regex pattern partially matches the input, the reveal array unexpectedly exposes more characters than those involved in the match. For example, when matching `` `"a[ab]"` `` against `` `"aba"` ``, the third character should not be revealed as it is not part of the accepted match sequence.

Example:

```
{
  "parts": [
    {
      "is_public": true,
      "regex_def": "a[ab]"
    }
  ]
}
```

Input:

```
component main { public [ msg ] } = SimpleRegex(3);

/* INPUT = {
    "msg": ["97", "98", "97"]
} */
```

Result:

```
Output:
out = 1
reveal0[0] = 97
reveal0[1] = 98
reveal0[2] = 97
```

Impact

This could lead to erroneous behaviors where more data than necessary is revealed.

Recommendation

Ensure that only the characters from the matching sequence are revealed.

Client response

The issue was fixed in commit **0a5943a**. We suggest adding further tests to catch any potential errors in corner cases.

06 - Templates do not document assumptions on input signals

zk-email-verify/packages/circuits/*

Medium

Description

We found a number of templates throughout the zk-email-verify codebase which implicitly assume inputs to be constrained in a certain way, but don't document that assumption anywhere.

What follows is a list of undocumented assumptions:

- `Sha256General`
  - `paddedIn` consists of bits
- `Sha256Partial`
  - `preHash` consists of bits
  - `paddedIn` consists of bits
- `Sha256General`, `Sha256Partial`, `Sha256Bytes`, `Sha256BytesPartial`
  - `paddedInLength` fits in a certain number of bits, see the SHA256 finding
- `PackBytes`
  - `in` elements fit in 8 bits
- `PackBytesSubArray`
  - `in` elements fit in 8 bits
  - `length` fits in `ceil(log2(maxSubArrayLen))` bits
- `PackRegexReveal`
  - `in` elements fit in 8 bits
- `SelectSubArray`
  - `length` fits in `ceil(log2(maxSubArrayLen))` bits
- `CalculateTotal`
  - `nums[n]` are small enough that their sum does not overflow the field
- `AssertZeroPadding`
  - `startIndex - 1` fits in `ceil(log2(maxArrayLen))` bits
  - This one is not exploitable, because the only way it can behave incorrectly is to constrain a larger number of array elements to be 0 than expected

- `DigitBytesToInt`
  - inputs are between 48 and 57
- `FpPow65537Mod`
  - `base` and `modulus` consist of `k` limbs, each of which must fit in `n` bits
- `FpMul`
  - `a`, `b` and `p` consist of `k` limbs, each of which must fit in `n` bits
- `BigLessThan`
  - `a` and `b` consist of `k` limbs, each of which must fit in `n` bits
- `PoseidonLarge`
  - `in` elements must fit in `bytesPerChunk` bits to avoid hash collisions
  - Note: `bytesPerChunk` refers to the number of *bits* per chunk (!), should be renamed
- `EmailVerifier`
  - `emailHeaderLength` fits in `ceil(log2(maxHeadersLength))` bits
  - `emailBodyLength` fits in `ceil(log2(maxBodyLength))` bits

This list attempts to be exhaustive, but excludes **unused templates that we recommend to remove**.

We also want mention a few counter-examples: cases where on the surface it seems that a template makes an implicit assumption, but when looking closer, it turns out that the input already gets fully constrained.

- `Base64Lookup`
  - fully constrains `in` to consist of valid base64 characters, even though `LessThan` and `GreaterThan` are used without bit range checks.
  - by extension, the same is true for `Base64Decode`
- `RSAVerifier65537`
  - fully constrains `message`, `signature` and `modulus` to consist of `n`-bit limbs. Note that the bit decompositions of `message` and `modulus` are buried inside `RSAPad`.
- `SelectRegexReveal`
  - is sound for all practically relevant values of the `maxRevealLen` static parameter, even though it uses `GreaterThan` on a value (`startIndex + maxRevealLen - 1`) that can overflow the target bit length.
  - Note that `startIndex` is bit-constrained by `VarShiftLeft`

Impact

As **our SHA256 finding** demonstrates, it is easy to forget to implicit assumptions. All of the templates listed above are potentially used by developers that consume zk-email-verify as a library. It seems likely that developers will skip undocumented constraints, and might end up with critical bugs in their applications.

Recommendation

A safe way to defend against unsatisfied assumptions is to remove them: constrain inputs *within* every single template that makes assumptions on them.

The downside of constraining inputs is that as the same input gets passed to multiple templates, it will be constrained multiple times, which is inefficient.

Therefore, our general recommendation for a mature circuit library is to **constrain outputs, but not inputs**. Signals that carry a range-check or other assumption should be constrained in the most top-level template where they originate, right next to their definition -- and nowhere else.

When following this style of not constraining inputs, it is of highest importance that input assumptions are always clearly documented as part of the comment preceding a template. We recommend the following:

- Add documentation in comments for all the assumptions listed above.
- In addition, mention the policy of not constraining inputs in the high-level documentation of the library -- for example, as a section in `packages/circuits/README.md`.

Exceptions to the "never constrain inputs" rule make sense when a template should be usable directly as the `main` component. If that is desired, inputs should be constrained within the template. See `EmailVerifier`, where we recommend to constrain `emailHeaderLength` and `emailBodyLength`.

Client response

The issue was addressed in commits f4d0eba, ea2226, 4a0572 and 4b17ae.

07 - Base64Decode does not distinguish between 'A' and '='

zk-email-verify/packages/circuits/lib/base64.circom

Low

Description

`Base64Lookup` returns 0 when passing into it 'A' (i.e., `65`) -- as expected --, but also when passing '=' (i.e., `61` which is used for padding). That means that when we use `Base64Decode`, the result will be the same for aS== and aSA=. Furthermore, someone could replace occurrences of A with =.

Impact

This is a low-severity issue, as it does not seem to be easily exploited in a practical scenario. Nevertheless, as this template is supposed to be used as a library, we propose to fix it.

Recommendation

You should specially handle the padding character (`=`) in the `Base64Lookup` template.

Client response

The client chose to document the behaviour of the template instead of changing it (commit 27ccac). We think this is appropriate.

08 - Compiler error when processing a valid regex

zk-regex/packages/compiler

Low

Description

Certain valid regex patterns, such as those specifying a range of repetitions like `{2,3}`, are being rejected by the compiler with an error message about the size of accept nodes. Oddly, similar patterns with quantifiers such as `*` or `+` are accepted. Note that we have also observed this behavior with regexes like: `(^|h)$`.

Impact

This could lead to developers not being able to describe their regexes.

09 - The matching all pattern leads to an error when compiling the produced circom template

zk-regex/packages/compiler/regex.rs

Low

Description

The compiler produces a circuit that cannot be compiled due to a signal initialization error when processing the universal matching pattern `.*`.

```
stderr:
error[T3001]: Exception caused by invalid access: trying to access to a signal that is not
initialized
    ┌── "untitled3.circom":299:32
    │
299 │         final_state_result.in[i] <== states[i][0];
    │                                    ^^^^^^^^^^^^ found here
    │
    = call trace:
      ->SimpleRegex

previous errors were found
```

Impact

Users cannot use a valid regex.

Client response

The client acknowledged the issue. The compiler currently does not support it, and the README has been updated.

0a - Regex using | leads to a compiler crash

zk-regex/packages/compiler/circom.rs

Low

Description

The compiler crashes when the regex pattern includes the alternation operator (`|`). Patterns that should be valid, such as `"a|b"`, are causing compiler crashes.

Impact

The crash affects the compiler's usability, preventing users from utilizing certain valid regex patterns.

0b - Unused templates

zk-email-verifier/packages/circuits/lib/*

Low

Description

The files `fp.circom`, `bigint.circom` and `bigint-func.circom`, which originally were taken from other Circom projects, contain a large number of templates and functions which are neither currently used in zk-email-verify nor likely to be useful to consumers of the library.

Furthermore, some of these templates have since received security patches as part of other projects, for example:

- **`ModSumThree()` accepts unexpected solutions**, from a circom-ecdsa PR
- **`ModSumFour()` uses an incorrect assertion**, from the Veridise Succinct audit
- **`BigMod()` and `BigModInv()` do not work for k >= 50**, from the same Veridise audit
- **Templates `Split()` and `SplitThree()` have non-deterministic instantiations**, from the same Veridise audit

No template in this list is actually used, so it seems to be an unnecessary burden to continue maintaining them and keeping up to date with fixes.

Recommendation

We recommend to remove the following templates from the project:

- `fp.circom`: Remove everything except `FpMul`
- `bigint.circom` : Remove everything except `BigLessThan`and `CheckCarryToZero`
- `bigint-func.circom`: Remove `isNegative`, `SplitFn`, `SplitThreeFn`, `prod`, `mod_exp`, `mod_inv`, `long_sub_mod_p`, `prod_mod_p`

Client response

The recommended changes were made in commit 5e7a5f.

0c - Empty regex generates invalid Circom

zk-regex/packages/compiler

Informational

Description

Given an empty regex (or `^$`, `^` or `$`), the Circom generation succeeds, but generates invalid code.

To reproduce, run

```
zk-regex decomposed -d ./empty.json -c ./empty.circom -t EmptyRegex -g true
```

with the following `empty.json`:

```
{
  "parts": [
    {
      "is_public": false,
      "regex_def": ""
    }
  ]
}
```

The resulting `empty.circom` will contain the following snippet:

```
    for (var i = 0; i < num_bytes; i++) {
        state_changed[i] = MultiOR(0);
        states[i][0] <== 1;
        from_zero_enabled[i] <== MultiNOR(0)([]);
    }
```

Circom compiler will print out the following error:

```
error[P1012]: illegal expression
    ┌── "./empty.circom":30:41
    │
30  │         from_zero_enabled[i] <== MultiNOR(0)([]);
    │                                                 ^ here
```

Impact

Probably not a high severity issue, as empty regexes are not used a lot. However, an empty regex is still a valid regex, without the caveats mentioned in the project README.md (certain characters, lookaheads, lookbehinds), so it's expected to produce valid Circom. With programmatic use of the zk-regex library, this could lead to unexpected behavior.

Recommendation

The Circom generation should be fixed to handle empty regexes correctly.

0d - Panics instead of returning error results

zk-regex/packages/compiler

Informational

Description

zk-regex public functions throw panics instead of returning compile errors.

For example, run

```
RUST_BACKTRACE=1 zk-regex decomposed -d ./panic.json -c ./oh-no.circom -t NoWayRegex -g true
```

with the following `panic.json`:

```
{
  "parts": [
    {
      "is_public": false,
      "regex_def": "^(hey|there$"
    }
  ]
}
```

The regex_def above is invalid, as the group is not closed.

This will result in panic:

```
thread 'main' panicked at packages/compiler/src/regex.rs:265:14:
called `Result::unwrap()` on an `Err` value: BuildError { kind: NFA(BuildError { kind:
Syntax(Parse(Error { kind: GroupUnclosed, pattern: "^(hey|there$", span: Span(Position(o: 1,
l: 1, c: 2), Position(o: 2, l: 1, c: 3)) })) }) }
stack backtrace:
   0: _rust_begin_unwind
   1: core::panicking::panic_fmt
   2: core::result::unwrap_failed
   3: zk_regex_compiler::regex::regex_and_dfa
   4: zk_regex_compiler::gen_from_decomposed
   5: zk_regex::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Impact

This will lead to crashes of programs using zk-regex as a library.

Recommendation

Consider replacing panics with returning error `Result`s.