Overview

Sources https://github.com/zkemail/zk-email-verify/
Initial Commit: https://github.com/zkemail/zk-email-verify/commit/1f28986e9f4dcf3693d3a0f3f6459c08d4cadf2f
Language: Circom

Files

`circuits/helpers/*.circom`

Issues

1. Missing constraint for illegal characters

Code file: `circuits/helpers/base64.circom`

Description: The `Base64Lookup()` circuit is designed to convert the ASCII codes of characters `(A~Z, a~z, 0~9, +, /)` to their corresponding base64 codes. According to the base64 encoding table, the output signal should be `000000` when the input signal `in` is set to 65, which represents the character `A`. However, the current implementation fails to properly handle input signals that correspond to illegal characters. For example, when `in` is set to 35, which represents the character `#`, the output signal is still `0`, which is incorrect. To ensure the proper functionality and security of the circuit, it is recommended to implement appropriate error handling mechanisms to detect and handle illegal input signals.

```
template Base64Lookup() {
    signal input in;
    signal output out;

    // ['A', 'Z']

    ...
    // '+'
    component equal_plus = IsZero();
    equal_plus.in <== in - 43;
    signal sum_plus <== sum_09 + equal_plus.out * (in + 19);

    // '/'
    component equal_slash = IsZero();
    equal_slash.in <== in - 47;
    signal sum_slash <== sum_plus + equal_slash.out * (in + 16);

    // @audit: sum_slash would be zero, if in isn't in (A~Z, a~z, 0~9, +, /).
    out <== sum_slash;
}
```

Recommendation: To enhance the security of the `Base64Lookup()` circuit template, it is recommended to add a constraint to validate that the input signal `in` is within the range of valid ASCII characters, i.e., `(A~Z, a~z, 0~9, +, /)`. Implementing appropriate constraints will ensure that only valid input values are processed by the circuit, mitigating the risk of potential attacks or errors due to invalid input signals.

```
template Base64Lookup() {
    signal input in;
    signal output out;

    // ['A', 'Z']

    ...
    // '+'
    component equal_plus = IsZero();
    equal_plus.in <== in - 43;
    signal sum_plus <== sum_09 + equal_plus.out * (in + 19);

    // '/'
    component equal_slash = IsZero();
    equal_slash.in <== in - 47;
    signal sum_slash <== sum_plus + equal_slash.out * (in + 16);

    // @audit: sum_slash would be zero, if in isn't in (A~Z, a~z, 0~9, +, /).
    // @audit: add the constaint below
    range_AZ + range_az + range_09 + equal_plus + equal_slash === 1

    out <== sum_slash;
}
```

2. Incorrect use of division operation

Code file: `circuits/helpers/rsa.circom`

Description: In the `RSAPad(n, k)`, the following code is used to calculate the idx value,

```
var idx = (i - (base_len + 8)) / 8;
```

In this section of the code, the division operator `/` is a modulo division, which is not suitable for the numerical division required by the context. Although `(i - (base_len + 8))` happens to be divisible by `8`, using `/` operator here can be risky as it may produce unexpected results if the operands are not guaranteed to be divisible. Therefore, it is recommended to use the `\` operator for numerical division instead of `/` to avoid any potential issues.

Recommendation: Use `\` instead of `/`.

```
var idx = (i - (base_len + 8)) \ 8;
```

3. Missing range checks for output signals

Code file: `circuits/helpers/bigint.circom`

Description: In the circuit template for `BigMod(n, k)`, the `long_div()` function is utilized to compute two arrays of signals, `div` and `mod`. It is important to note that the bit length of each signal should be less than `n`. However, the code only checks the range of `div` and fails to verify the bit length of `mod`. This oversight could potentially lead to unexpected behavior or errors in the circuit. Therefore, we recommend adding additional checks to ensure that the bit length of both `div` and `mod` are within the required range.

```
template BigMod(n, k) {
    assert(n <= 126);
    signal input a[2 * k];
    signal input b[k];

    signal output div[k + 1];
    signal output mod[k];

    var longdiv[2][100] = long_div(n, k, k, a, b);
    for (var i = 0; i < k; i++) {
        div[i] <-- longdiv[0][i];
        mod[i] <-- longdiv[1][i];
    }
    div[k] <-- longdiv[0][k];
    component range_checks[k + 1];
    for (var i = 0; i <= k; i++) {
        range_checks[i] = Num2Bits(n);
        range_checks[i].in <== div[i];
    }

// @audit: no check for signal mod[i]

    component mul = BigMult(n, k + 1);
    for (var i = 0; i < k; i++) {
        mul.a[i] <== div[i];
        mul.b[i] <== b[i];
    }
    mul.a[k] <== div[k];
    mul.b[k] <== 0;

    component add = BigAdd(n, 2 * k + 2);
    for (var i = 0; i < 2 * k; i++) {
        add.a[i] <== mul.out[i];
        if (i < k) {
            add.b[i] <== mod[i];
        } else {
            add.b[i] <== 0;
        }
    }
    add.a[2 * k] <== mul.out[2 * k];
    add.a[2 * k + 1] <== mul.out[2 * k + 1];
```

```
        add.b[2 * k] <== 0;
        add.b[2 * k + 1] <== 0;

        for (var i = 0; i < 2 * k; i++) {
            add.out[i] === a[i];
        }
        add.out[2 * k] === 0;
        add.out[2 * k + 1] === 0;

        component lt = BigLessThan(n, k);
        for (var i = 0; i < k; i++) {
            lt.a[i] <== mod[i];
            lt.b[i] <== b[i];
        }
        lt.out === 1;
    }
```

Recommendation: To improve the security and reliability of the circuit, it is recommended to add a range check for the `mod` bits.

Discussion: We notice that there was a commit which has fixed this issue in another repo. However zkemail repo omits the fix. Here is the commit:

- `https://github.com/0xPARC/circom-ecdsa/pull/10/commits/d3edd7503f48f98a71b6013c248ef3ad55e19703`

4. Missing constraints for a signal input

Code file: `helpers/utils.circom`

Description: In the circuit `Modulo(divisor_bits)`, a `MultiRangeProof` template component called `rp` is needed. The `MultiRangeProof` template requires `n+1` signal inputs, including `max_abs_value`. However, the code is missing the necessary provisions to pass a value or set constraints for this input. Although the original code included a constraint `rp.max_abs_value <== SQRT_P`, it has been commented out. To ensure proper functionality and security of the circuit, it is recommended that appropriate constraints be set for the `max_abs_value` input to prevent potential errors or vulnerabilities.

```
    template Modulo(divisor_bits) {
        signal input dividend; // -8
        signal input divisor; // 5
        signal output remainder; // 2
        signal output quotient; // -2

        ...
        component rp = MultiRangeProof(3, 128);
        rp.in[0] <== divisor;
        rp.in[1] <== quotient;
        rp.in[2] <== dividend;
    // @audit: the following code cannot be commented out.
        //rp.max_abs_value <== SQRT_P;

```

```
        // check that 0 <= remainder < divisor
        component remainderUpper = LessThan(divisor_bits);
        remainderUpper.in[0] <== remainder;
        remainderUpper.in[1] <== divisor;
        remainderUpper.out === 1;
    }

    template MultiRangeProof(n, bits) {
        signal input in[n];
        signal input max_abs_value;
        component rangeProofs[n];

        for (var i = 0; i < n; i++) {
          rangeProofs[i] = RangeProof(bits);
          rangeProofs[i].in <== in[i];
          rangeProofs[i].max_abs_value <== max_abs_value;
        }
    }
```

Recommendation: To ensure the proper functionality and security of the `Modulo(divisor_bits)` circuit, it is recommended to uncomment the constraint `rp.max_abs_value <== SQRT_P`. This constraint sets an appropriate upper bound for the `max_abs_value` input of the `MultiRangeProof` template, which helps to prevent potential vulnerabilities or errors in the circuit.

Discussion: We notice that this line of code isn't commented out in the file from the URL (provided also in the docs) . `https://sourcegraph.com/github.com/darkforest-eth/circuits/-/blob/perlin/perlin.circom?L89`

5. Missing constraints for output signals

Code file: `helpers/utils.circom`

Description: The `Modulo(divisor_bits)` circuit includes two output signals, `raw_remainder` and `neg_remainder`. However, the current implementation lacks constraints to validate their values, leaving the circuit vulnerable to potential attacks. Malicious actors could potentially replace the values of `raw_remainder` and `neg_remainder` with arbitrary values, compromising the integrity and security of the circuit.

```
    template Modulo(divisor_bits) {
        signal input dividend; // -8
        signal input divisor; // 5
        signal output remainder; // 2
        signal output quotient; // -2

        ...

    // @audit: no constraint for raw_remainder
        signal output raw_remainder;
        raw_remainder <-- abs_dividend % divisor;

    // @audit: no constraint for neg_remainder
        signal output neg_remainder;
```

```
15        neg_remainder <-- divisor - raw_remainder;
16
17        if (is_dividend_negative == 1 && raw_remainder != 0) {
18            remainder <-- neg_remainder;
19        } else {
20            remainder <-- raw_remainder;
21        }
22
23        quotient <-- (dividend - remainder) / divisor; // (-8 - 2) / 5 = -2.
24
25    }
```

Recommendation: It is suggested to either remove the `raw_remainder` and `neg_remainder` signals or implement appropriate constraints to validate their values.

6. Issue with value retrieval in the LongToShortNoEndCarry

Code file: `helpers/bigint.circom`

Description: The circuit template `LongToShortNoEndCarry(n, k)` utilizes the `SplitThreeFn` function to split each input into three parts. However, it has been observed that the third value obtained from splitting the last input, `split[k-1][2]` , is not included in the calculation of the `out` array. It is unclear whether this value can be optimized or if its exclusion from the calculation is intentional.

```
1   template LongToShortNoEndCarry(n, k) {
2       assert(n <= 126);
3       signal input in[k];
4       signal output out[k+1];
5
6       var split[k][3];
7       for (var i = 0; i < k; i++) {
8           split[i] = SplitThreeFn(in[i], n, n, n);
9       }
10
11      var carry[k];
12      carry[0] = 0;
13      out[0] <-- split[0][0];
14      if (k == 1) {
15    out[1] <-- split[0][1];
16      }
17      if (k > 1) {
18          var sumAndCarry[2] = SplitFn(split[0][1] + split[1][0], n, n);
19          out[1] <-- sumAndCarry[0];
20          carry[1] = sumAndCarry[1];
21      }
22      if (k == 2) {
23    out[2] <-- split[1][1] + split[0][2] + carry[1];
24      }
25      if (k > 2) {
26          for (var i = 2; i < k; i++) {
```

```
            var sumAndCarry[2] = SplitFn(split[i][0] + split[i-1][1] + split[i-2][2] +
carry[i-1], n, n);
            out[i] <-- sumAndCarry[0];
            carry[i] = sumAndCarry[1];
        }
        out[k] <-- split[k-1][1] + split[k-2][2] + carry[k-1];
    }

    component outRangeChecks[k+1];
    for (var i = 0; i < k+1; i++) {
        outRangeChecks[i] = Num2Bits(n);
        outRangeChecks[i].in <== out[i];
    }

    signal runningCarry[k];
    component runningCarryRangeChecks[k];
    runningCarry[0] <-- (in[0] - out[0]) / (1 << n);
    runningCarryRangeChecks[0] = Num2Bits(n + log_ceil(k));
    runningCarryRangeChecks[0].in <== runningCarry[0];
    runningCarry[0] * (1 << n) === in[0] - out[0];
    for (var i = 1; i < k; i++) {
        runningCarry[i] <-- (in[i] - out[i] + runningCarry[i-1]) / (1 << n);
        runningCarryRangeChecks[i] = Num2Bits(n + log_ceil(k));
        runningCarryRangeChecks[i].in <== runningCarry[i];
        runningCarry[i] * (1 << n) === in[i] - out[i] + runningCarry[i-1];
    }
    runningCarry[k-1] === out[k];
}
```

Discussion: Further explanation is needed to determine whether `split[k-1][2]` can be safely optimized or if it should be included in the calculation of the `out` array. Additionally, it is unclear whether this value is always zero or if its exclusion from the calculation is intentional. Providing more detailed documentation or comments in the code may help to clarify these questions and ensure the proper functionality and security of the circuit.