SALUS SECURITY

AUG 2024

CODE SECURITY ASSESSMENT

POLYHEDRA

Overview

Project Summary

- Name: Polyhedra - ecdsa
- Language: Go
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Polyhedra - ecdsa
Version	v2
Type	Go
Dates	Aug 09 2024
Logs	May 23 2024; Aug 09 2024

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	0
Total Low-Severity issues	3
Total informational issues	16
Total	20

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue can lead to substantial financial, reputation, availability, or privacy damage.
Medium Risk	The issue can lead to moderate financial, reputation, availability, or privacy damage. Or the issue can lead to substantial damage under extreme and unlikely circumstances.
Low Risk	The issue does not pose an immediate security threat, but may be a lack of following best practices or more easily lead to the future introductions of bugs.
Informational	Information not relevant to security, but may be helpful for efficiency, costs, etc.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high caliber, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Architectural Design
- Logic Error
- Underconstrained Variable
- Unused Circuit/Variable
- Information Leakage
- Access Control
- Data Validation
- Overflow/Underflow
- Denial of Service
- Redundancy
- Improving readability

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s), circuit(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Constraint setting error for `AND` in BigLessThan	High	Logic Error	Resolved
2	Missing checks for extreme cases in mod_inv	Low	Architectural Design	Resolved
3	Missing check for `n` in myIsLess	Low	Architectural Design	Resolved
4	String parsing error in secp256k1_func.go	Low	Overflow/Underflow	Resolved
5	Unused circuits and variables in ECDSAVerifyNoPubkeyCheck	Info	Unused Circuit/Variable	Resolved
6	Unnecessary constraints `qVarTemp` in CheckCubicModPIsZero and CheckQuadraticModPIsZero	Info	Constraint Optimization	Acknowledged
7	Unnecessary constraints on `temp` in CheckCubicModPIsZero and CheckQuadraticModPIsZero	Info	Redundancy	Acknowledged
8	Redundant constraints in Secp256k1ScalarMult	Info	Redundancy	Acknowledged
9	Complex constraints in Decoder	Info	Improving readability	Acknowledged
10	Unnecessary constraints in multiplexer.go	Info	Redundancy	Acknowledged
11	The value of the dummy point (G * 2 * * 255) is calculated incorrectly in get_dummy_point	Info	Data Validation	Acknowledged
12	Redundant constraints in CheckCarryToZero	Info	Redundancy	Acknowledged
13	Incorrect values of `LogCeilK` in LongToShortNoEndCarry	Info	Logic Error	Acknowledged
14	Unnecessary assignments in secp256k1_utils.go	Info	Unused Circuit/Variable	Acknowledged

15	Unnecessary constraints in ModProd and Split	Info	Redundancy	Acknowledged
16	Redundant constraints in Split	Info	Redundancy	Acknowledged
17	Unnecessary constraints in shr_no_check	Info	Redundancy	Acknowledged
18	Using deprecated functions in Test_secp256k1_hint	Info	Architectural Design	Acknowledged
19	Assertion missing in ModSubThree	Info	Architectural Design	Acknowledged
20	Unnecessary constraints in BigMultNoCarry	Info	Redundancy	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Constraint setting error for `AND` in BigLessThan	
Severity: High	Category: Logic Error
Target: - ecdsa_circuit/bigint.go - ecdsa_circuit/bigint_test.go	

Description

The **BigLessThan** circuit compares two `k` bit numbers `a` and `b` to determine if `a` is less than `b`. It uses `myIsLess` and `IsEqual` circuits to compare each bit of `a` and `b` from the most significant to the least significant bit. The following snippet attempts to construct `AND` gates for `ands[i]` and `eq_ands[i]`. Because the `AND` gates are used to construct conditions that ensure the bits compared so far are equal and that a specific bit of `a` is less than the corresponding bit of `b`.

bigint.go:L276-L286

```
func BigLessThan(api frontend.API, n int, k int, a []frontend.Variable, b
[]frontend.Variable) frontend.Variable {
...
        for i := k - 2; i >= 0; i-- {
                if i == k-2 {
                        ands[i] = api.Add(eq[k-1], lt[k-2])
                        eq_ands[i] = api.Add(eq[k-1], eq[k-2])
                        ors[i] = or(api, lt[k-1], ands[i])
                } else {
                        ands[i] = api.Add(eq_ands[i+1], lt[i])
                        eq_ands[i] = api.Add(eq_ands[i+1], eq[i])
                        ors[i] = or(api, ors[i+1], ands[i])
                }
        }
...
}
```

However, an error occurred in the construction of the `AND` gate in the snippet, where `api.Mul()` was mistakenly implemented as `api.Add()`. This could cause the circuit's results to deviate from the expected outcome.

Additionally, in the test case for this circuit, the data `A`, `B`, and `C` used happen to yield the correct conclusion, i.e., `B` > `A`, even with the current erroneous implementation. Therefore, the data in this test case should also be modified to avoid misleading the users.

bigint_test.go:L355-L360

```
func Test_BigLessThan(t *testing.T) {
	assert := test.NewAssert(t)
	var circuit circuit_BigLessThan
	A := [4]frontend.Variable{uint64(14179718031704721038),
uint64(15556385072872112361), uint64(7043740883455610960), uint64(740596429572950603)}
	B := [4]frontend.Variable{uint64(13822214165235122497),
uint64(13451932020343611451), uint64(18446744073709551614),
uint64(18446744073709551615)}
	C := frontend.Variable(1)
```

Impact: Using `api.Add()` instead of `api.Mul()` for the `AND` gate introduces significant logical errors. The addition operation misrepresents binary logic, resulting in incorrect outcomes for the `AND` operation. This causes logical inconsistencies, leading to incorrect evaluations of `ors[i]`, `ands[i]`, and `eq_ands[i]`. Consequently, the overall function fails to correctly determine if one number is less than the other, making the circuit unreliable. In cryptographic applications, this could introduce security vulnerabilities by compromising the correctness of proofs or verifications.

Recommendation

Change `api.Add()` to `api.Mul()` to construct the correct AND gate. We recommend using `api.Mul()` instead of `api.And()` because `api.And()` adds constraints to enforce that the inputs are binary, this is unnecessary since these inputs have already been constrained to be binary.

bigint.go

```
func BigLessThan(api frontend.API, n int, k int, a []frontend.Variable, b
[]frontend.Variable) frontend.Variable {
...
	for i := k - 2; i >= 0; i-- {
		if i == k-2 {
			ands[i] = api.Mul(eq[k-1], lt[k-2])
			eq_ands[i] = api.Mul(eq[k-1], eq[k-2])
			ors[i] = or(api, lt[k-1], ands[i])
		} else {
			ands[i] = api.Mul(eq_ands[i+1], lt[i])
			eq_ands[i] = api.Mul(eq_ands[i+1], eq[i])
			ors[i] = or(api, ors[i+1], ands[i])
		}
	}
...
}
```

Next, change the test data of **BigLessThan** to the following (just an example):

bigint_test.go

```
func Test_BigLessThan(t *testing.T) {
	assert := test.NewAssert(t)
	var circuit circuit_BigLessThan
	A := [4]frontend.Variable{uint64(1), uint64(2), uint64(3), uint64(5)}
	B := [4]frontend.Variable{uint64(1), uint64(2), uint64(4), uint64(4)}
	C := frontend.Variable(1)
```

In this case, we make sure `A[2] < B[2]` and `A[3] > B[3]`, then `ands[2]` will be `1` (but `0` is correct). Therefore, when `api.Mul()` is incorrectly written as `api.Add()`, the result of this test will fail.

Status

The team has resolved this issue in the latest version.

2. Missing checks for extreme cases in mod_inv

Severity: Low	Category: Architectural Design
Target: - ecdsa_circuit/bigint_func.go	

Description

The **mod_inv** hint function calculates the modular inverse of a large integer `a` modulo a prime `p` using multiple registers, then splits the result back into `k` registers and stores it in the `outputs` array.

bigint_func.go:L297-L314

```
func mod_inv(curveID ecc.ID, inputs []*big.Int, outputs []*big.Int) error {
	if !inputs[0].IsUint64() || !inputs[1].IsUint64() {
		panic("mod_inv: inputs[0] and inputs[1] must be uint64")
	}
	n := int(inputs[0].Int64())
	k := int(inputs[1].Int64())
	a := inputs[2 : 2+k]
	p := inputs[2+k : 2+2*k]

	a_bigint := big.NewInt(0)
	p_bigint := big.NewInt(0)
	for i := k - 1; i >= 0; i-- {
		a_bigint.Lsh(a_bigint, uint(n))
		p_bigint.Lsh(p_bigint, uint(n))
		a_bigint.Add(a_bigint, a[i])
		p_bigint.Add(p_bigint, p[i])
	}
	a_bigint.ModInverse(a_bigint, p_bigint)
...
}
```

However, the code does not enforce the condition noted in the comment `// if a == 0 mod p, returns 0`. This causes `a_bigint.ModInverse(a_bigint, p_bigint)` to return `nil` when `a_bigint` is `big.NewInt(0)`.

Impact: No impact to the current code base. Because **mod_inv** hint function is only called in the **ECDSAVerifyNoPubkeyCheck** circuit, and the `s` from signature `(r, s)` is unlikely to be `0`. However, if we consider this hint function independently, it does not fully implement the expected functionality. If it is called independently by the user in the future, it may result in unexpected errors.

Recommendation

Perform a conditional check on `a_bigint` before calling `ModInverse`. If it equals `big.NewInt(0)`, assign `0` to each `outputs[i]`.

Status

The team has resolved this issue in the latest version.

3. Missing check for `n` in myIsLess

Severity: Low	Category: Architectural Design
Target: - ecdsa_circuit/bigint.go	

Description

The **myIsLess** circuit determines whether variable `a` is less than variable `b`. It does this by computing `a + 2^n - b`, converting the result to binary representation, and then checking the most significant bit of this binary representation. If this bit is 1, it indicates that `a` is less than `b`.

bigint.go:L24-L28

```
func myIsLess(api frontend.API, n int, a frontend.Variable, b frontend.Variable)
frontend.Variable {
        neg_b := api.Neg(b)
        bi1 := api.ToBinary(api.Add(a, frontend.Variable(LeftShift(1, uint64(n))),
neg_b), n+1)
        return api.Sub(frontend.Variable(1), bi1[n])
}
```

However, the circuit does not restrict the size of `n`. When `2 ^ (n+1)` exceeds the prime order of the current curve's finite field (e.g., using BN254 or BLS12-381), there's a risk of overflow in the field.

Impact: If `n` is too large, converting the numbers to binary `a + (1 << n) - b` might exceed the maximum value of the finite field, resulting in incorrect binary outputs and consequently erroneous circuit results. In addition, circuit operations might fail to execute correctly, leading to the failure of the entire proof generation process or the generation of invalid proofs.

Recommendation

Limit the size of `n` to be less than `253`.

bigint.go

```
func myIsLess(api frontend.API, n int, a frontend.Variable, b frontend.Variable)
frontend.Variable {
        if n > 252 {
                panic("myIsLess: n is too large")
        }
        neg_b := api.Neg(b)
...
}
```

Status

The team has resolved this issue in the latest version.

4. String parsing error in secp256k1_func.go

Severity: Low	Category: Overflow/Underflow
Target: - ecdsa_circuit\secp256k1_func.go	

Description

In **secp256k1_func.go**, many functions use `strconv.ParseUint()` to handle the branch where `n == 86 && k == 3`, such as `get_gx`, `get_gy`, `get_secp256k1_prime`, `get_secp256k1_order`, and `get_dummy_point`. However, this parsing method has serious issues. For example, in one of the functions, `get_secp256k1_prime`:

secp256k1_func.go:L38-L44

```
func get_secp256k1_prime(n int, k int) []uint64 {
        if n == 86 && k == 3 {
                outputs := make([]uint64, 3)
                outputs[0], _ = strconv.ParseUint("77371252455336262886226991", 10, 64)
                outputs[1], _ = strconv.ParseUint("77371252455336267181195263", 10, 64)
                outputs[2], _ = strconv.ParseUint("19342813113834066795298815", 10, 64)
                return outputs
...
}
```

When attempting to parse the string "77371252455336262886226991" using `strconv.ParseUint`, the program encounters an overflow error. This occurs because the value exceeds the maximum limit that can be represented by `uint64`.

Impact: No impact to the current code base. Currently, none of the functions called in this audit project have entered the branch where `n == 86 && k == 3`, so the above problematic functions will not impact the project logic for now. However, if the problematic functions are called in future development or actual applications, it could lead to significant issues. This overflow error prevents correct parsing of extremely large numbers, which could result in the circuit failing to compile, or the circuit output not matching the expected results.

Recommendation

To handle large integers beyond the range of standard `64` bit types, use the `math/big` package, which supports arbitrary-precision arithmetic.

For example, for `outputs[0]`, it can be modified as follows:

secp256k1_func.go

```
func get_secp256k1_prime(n int, k int) []uint64 {
        if n == 86 && k == 3 {
                outputs := make([]uint64, 3)
                num := new(big.Int)
                outputs[0], ok := num.SetString("77371252455336262886226991", 10)
                if !ok {
                        panic("Error parsing string to big.Int")
                }
...
}
```

Status

The team has resolved this issue in the latest version.

2.3 Informational Findings

5. Unused circuits and variables in ECDSAVerifyNoPubkeyCheck	
Severity: Info	Category: Unused Circuit/Variable
Target: - ecdsa_circuit/ecdsa.go	

Description

This circuit implements ECDSA signature verification without checking the validity of the public key. It computes the multiplicative inverse of `s`, verifies its correctness, and then calculates the sum of `$h \cdot s$^-$1 \cdot G$` and `$r{\cdot}s$^-$1 \cdot pubkey$`. Finally, it checks if the x-coordinate of the result equals `$r$`.

ecdsa.go:L168-L176

```
func ECDSAVerifyNoPubkeyCheck(api frontend.API, n int, k int, r []frontend.Variable, s
[]frontend.Variable, msghash []frontend.Variable, pubkey [][]frontend.Variable)
frontend.Variable {
...
        p_out := get_secp256k1_prime(n, k)
        order_out := get_secp256k1_order(n, k)

        p := make([]frontend.Variable, k)
        order := make([]frontend.Variable, k)
        for i := range p {
                p[i] = p_out[i]
                order[i] = order_out[i]
        }
...
}
```

However, in this circuit, it doesn't utilize the outputs `p_out` and the array `p` from the `get_secp256k1_prime` circuit in subsequent operations, leading to many meaningless constraints.

Impact: No impact to the current code base. But the effects of unused circuit calls include performance degradation due to wasted resources and time, increased resource consumption such as memory and storage, and heightened complexity in maintenance and modification. Furthermore, unused circuit components may introduce potential errors, leading to data inconsistencies or logical inaccuracies.

Recommendation

Remove the circuit `get_secp256k1_prime` from **ECDSAVerifyNoPubkeyCheck**, and delete the unused variable `p[]`.

Status

The team has resolved this issue in the latest version.

6. Unnecessary constraints `qVarTemp` in CheckCubicModPIsZero and CheckQuadraticModPIsZero

Severity: Info	Category: Constraint Optimization
Target: - ecdsa_circuit/secp256k1_utils.go	

Description

CheckCubicModPIsZero checks if a cubic polynomial modulo a prime number P is zero. In this snippet, `qVarTemp` is used to receive the quotient and remainder returned by the hint function `long_div`.

secp256k1_utils.go:L142-L153

```
func CheckCubicModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        qVarTemp, err := api.Compiler().NewHint(long_div, 10, inputLD...)
        if err != nil {
                panic(err)
        }
        //fmt.Println("qVarTemp: ")
        for i := 0; i < 3; i++ {
                q[i] = qVarTemp[i]
                //      api.Println(q[i])
        }
        for i := 3; i < 10; i++ {
                api.AssertIsDifferent(qVarTemp[i], LeftShift(1, 66))
        }
...
}
```

Because the **CheckCubicModPIsZero** circuit only requires the quotient stored in the first three elements of `qVarTemp`, the constraints on the remaining seven elements of `qVarTemp` in the snippet are unnecessary. However, due to the nature of hint functions in gnark, the code has to take such measures to ensure proper compilation. The same situation also occurs in circuit **CheckQuadraticModPIsZero**.

Impact: No impact to the current code base. However, unnecessary constraints in a cryptographic circuit lead to increased computational and verification costs, making the system less efficient. Additionally, they can complicate debugging and maintenance.

Recommendation

Keep the original `long_div` and create a new hint function `long_div_first3` that only outputs the quotient to optimize the number of constraints in this logic.

Status

This issue has been acknowledged by the team.

7. Unnecessary constraints on `temp` in CheckCubicModPIsZero and CheckQuadraticModPIsZero

Severity: Info	Category: Redundancy

Target:
- ecdsa_circuit/secp256k1_utils.go

Description

The hint function **getProperRepresentation** converts an array of potentially negative, overflowed registers into an array of properly sized, non-negative registers. It ensures that values are properly represented within the bit constraints specified.

secp256k1_utils.go: L117-L138

```go
func CheckCubicModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        temp, err := api.Compiler().NewHint(getProperRepresentation, 8, inputPR...)
        //fmt.Println("temp: ")
        for i := 0; i < 8; i++ {
                //      api.Println(temp[i])
        }
        if err != nil {
                panic(err)
        }
        for i := range temp {
                api.AssertIsDifferent(temp[i], LeftShift(1, 66))
        }

        inputLD := make([]frontend.Variable, 15)
        for i := 0; i < 8; i++ {
                inputLD[i+3] = temp[i]
        }
        for i := 0; i < 4; i++ {
                inputLD[i+11] = p[i]
        }
        for i := 0; i < 8; i++ {
                api.AssertIsDifferent(temp[i], LeftShift(1, 66))
        }
...
}
```

In this code snippet, it retains the `for` loop from the development testing (the same issue also occurs in lines 97-99 of this file) and applies redundant constraint checks on the `temp` array twice. Furthermore, the purpose of `temp` is to create the circuit variable `qVarTemp` mentioned later in the code. However, to ensure proper compilation in gnark, the circuit will have to apply constraints on all `outputs[]` of the hint functions **long_div** and **getProperRepresentation**, resulting in a large number of unnecessary constraints about `temp`.

Impact: No impact to the current code base. However, unnecessary constraints in a cryptographic circuit lead to increased computational and verification costs, making the system less efficient. Additionally, they can complicate debugging and maintenance.

Recommendation

One way to address this issue is to create a new hint function that combines the functionality of both **getProperRepresentation** and **long_div**. This would eliminate the need to fully constrain `temp`. By removing the duplicate loop and `temp` in this manner, the circuit can save approximately 248 constraints.

Status

This issue has been acknowledged by the team.

8. Redundant constraints in Secp256k1ScalarMult

Severity: Info	Category: Redundancy

Target:
- ecdsa_circuit/secp256k1.go

Description

This circuit **Secp256k1ScalarMult** performs scalar multiplication on elliptic curve points for the Secp256k1 curve. The following snippet shows where the issue lies in:

secp256k1.go:L258-L266

```
func Secp256k1ScalarMult(api frontend.API, n int, k int, scalar []frontend.Variable,
point [][]frontend.Variable) [][]frontend.Variable {
...
    has_prev_non_zero := make([]frontend.Variable, k*n)
    for i := k - 1; i >= 0; i-- {
        for j := n - 1; j >= 0; j-- {
            if i == k-1 && j == n-1 {
                has_prev_non_zero[n*i+j] = api.Or(0, n2b[i][j])
            } else {
                has_prev_non_zero[n*i+j] = api.Or(has_prev_non_zero[n*i+j+1], n2b[i][j])
            }
        }
    }
...
}
```

This section of the code initializes the `has_prev_non_zero` array, which tracks whether there has been any non-zero bit encountered up to each position in the binary representation of the scalar. Because `api.Or()` adds constraints to ensure that the input is binary, but the values of `has_prev_non_zero[n*i+j+1]` and `n2b[i][j]` have already been constrained to binary, this may result in redundant constraints.

Impact: No impact to the current code base. However, redundant constraints can lead to increased computational resources and circuit size, thus increasing the cost of proving and verifying.

Recommendation

Change `api.Or()` to either `or` (from **bigint.go**) or `correctAPIOr` (from **ecdsa.go**). For the first `has_prev_non_zero[n*i+j]`, we can change its `correctAPIOr` to just `n2b[i][k]` since `OR` with `0` doesn't change the output.

For example:

```
func Secp256k1ScalarMult(api frontend.API, n int, k int, scalar []frontend.Variable,
point [][]frontend.Variable) [][]frontend.Variable {
...
    has_prev_non_zero := make([]frontend.Variable, k*n)
    for i := k - 1; i >= 0; i-- {
        for j := n - 1; j >= 0; j-- {
            if i == k-1 && j == n-1 {
                has_prev_non_zero[n*i+j] = n2b[i][j]
            } else {
                has_prev_non_zero[n*i+j] = correctAPIOr(api, has_prev_non_zero[n*i+j+1],
n2b[i][j])
            }
        }
    }
...
}
```

Status

This issue has been acknowledged by the team.

9. Complex constraints in Decoder

Severity: Info	Category: Improving readability

Target:

- ecdsa_circuit/multiplexer.go

Description

The **Decoder** circuit is used to assist the execution of the `Multiplexer` circuit, determining whether the dummy point `G * 2 ** 255` should be involved in the computation. In the code, `success` is used to store the state of the **Decoder** circuit: if `inp > w`, `success` is set to `0`; otherwise, it is set to `1`.

multiplexer.go:L42

```
func Decoder(api frontend.API, w frontend.Variable, inp frontend.Variable)
([]frontend.Variable, frontend.Variable) {
...
      api.AssertIsEqual(api.Mul(success, api.Sub(success, 1)), 0)
...
}
```

However, the way `success` is constrained to binary in the code snippet is somewhat convoluted.

Impact: No impact to the current code base.

Recommendation

The suggestion is to modify the constraint on `success` in the code snippet to `api.AssertIsBoolean()` to enhance readability. The modified effect is as follows:

multiplexer.go

```
func Decoder(api frontend.API, w frontend.Variable, inp frontend.Variable)
([]frontend.Variable, frontend.Variable) {
...
      api.AssertIsBoolean(success)
...
}
```

Status

This issue has been acknowledged by the team.

10. Unnecessary constraints in multiplexer.go

Severity: Info	Category: Redundancy
Target: - ecdsa_circuit/multiplexer.go	

Description

In the following three circuits, they respectively pass single or multiple `frontend.Variable` parameters and call `api.ConstantValue`, `*big.Int.Int64()`, and `int()` to convert them to `int` type.

multiplexer.go:L8-L14

```go
func EscalarProduct(api frontend.API, w frontend.Variable, in1 []frontend.Variable, in2
[]frontend.Variable) frontend.Variable {
        var w_int int
        if w_big, ok := api.ConstantValue(w); ok {
                w_int = int(w_big.Int64())
        } else {
                panic("EscalarProduct: w is not a constant")
        }
...
}
```

multiplexer.go:L24-L30

```go
func Decoder(api frontend.API, w frontend.Variable, inp frontend.Variable)
([]frontend.Variable, frontend.Variable) {
        var w_int int
        if w_big, ok := api.ConstantValue(w); ok {
                w_int = int(w_big.Int64())
        } else {
                panic("Decoder: w is not a constant")
        }
...
}
```

multiplexer.go:L47-L58

```go
func Multiplexer(api frontend.API, wIn frontend.Variable, nIn frontend.Variable, inp
[][]frontend.Variable, sel frontend.Variable) []frontend.Variable {
        var w_int, n_int int
        if w_big, ok := api.ConstantValue(wIn); ok {
                w_int = int(w_big.Int64())
        } else {
                panic("Multiplexer: wIn is not a constant")
        }
        if n_big, ok := api.ConstantValue(nIn); ok {
                n_int = int(n_big.Int64())
        } else {
                panic("Multiplexer: nIn is not a constant")
        }
...
}
```

However, this approach introduces many meaningless constraints. It might be a better choice to pass the variables expected to be modified directly as `int` type in the circuit parameters.

Impact: No impact to the current code base. However, this approach leads to a waste of many unnecessary constraints, potentially impacting the computational cost and time consumption of the circuit.

Recommendation

Change some of the `frontend.Variable` types in the circuit parameters to be passed as `int`.

The effect is as follows:

multiplexer.go

```
func EscalarProduct(api frontend.API, w_int int, in1 []frontend.Variable, in2
[]frontend.Variable) frontend.Variable {
...
}

func Decoder(api frontend.API, w_int int, inp frontend.Variable) ([]frontend.Variable,
frontend.Variable) {
...
}

func Multiplexer(api frontend.API, w_int int, nIn int, inp [][]frontend.Variable, sel
frontend.Variable) []frontend.Variable {
        out := make([]frontend.Variable, w_int)

        dec, success := Decoder(api, nIn, sel)

        for i := 0; i < w_int; i++ {
                in1 := make([]frontend.Variable, nIn)
                for j := 0; j < nIn; j++ {
...
}
```

This approach reduces unnecessary constraints and optimizes the use of computational resources and time.

Status

This issue has been acknowledged by the team.

11. The value of the dummy point (G * 2 * * 255) is calculated incorrectly in get_dummy_point

Severity: Info	Category: Data Validation
Target: - ecdsa_circuit/secp256k1_func.go	

Description

The **get_dummy_point** function is designed to return a specific elliptic curve point, which is used as a dummy or placeholder in the `ECDSAPrivToPub` function. The dummy point is `G * 2 ** 255`, where `G` is a generator point on the elliptic curve.

secp256k1_func.go: L91-L105

```
func get_dummy_point(n int, k int) [][]uint64 {
...
        } else if n == 64 && k == 4 {
                for i := range outputs {
                        outputs[i] = make([]uint64, 4)
                }
->              outputs[0][0], _ = strconv.ParseUint("10184385086782357888", 10, 64)
->              outputs[0][1], _ = strconv.ParseUint("16068507144229249874", 10, 64)
->              outputs[0][2], _ = strconv.ParseUint("17097072337414981695", 10, 64)
->              outputs[0][3], _ = strconv.ParseUint("1961476217642676500", 10, 64)

->              outputs[1][0], _ = strconv.ParseUint("15220267994978715897", 10, 64)
->              outputs[1][1], _ = strconv.ParseUint("2812694141792528170", 10, 64)
->              outputs[1][2], _ = strconv.ParseUint("9886878341545582154", 10, 64)
->              outputs[1][3], _ = strconv.ParseUint("4627147115546938088", 10, 64)
                return outputs
        } else {
...
}
```

However, due to some errors in calculating the values of the dummy point in the branch where `n == 64 && k == 4` in this function, the return value `outputs` may not equal `G * 2 * * 255`. The branch for `n == 86 && k == 3` has a similar error.

We can verify the calculated result through the following Sage script, which reveals that the computed dummy point value in **get_dummy_point** is incorrect.

```
# Define the secp256k1 elliptic curve parameters
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
a = 0
b = 7
E = EllipticCurve(GF(p), [a, b])

# The generator point (Gx, Gy) and its order
Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
G = E(Gx, Gy)

# Compute 2^255 * G
result = (2^255) * G
```

```
# Output the result
print("The result of 2^255 * G is:")
print(result.xy())

print("The result of `get_dummy_point` when n == 64 && k == 4:")
print(10184385086782357888 + (16068507144229249874 * 2^64) + (17097072337414981695 *
2^64 * 2^64) + (1961476217642676500 * 2^64 * 2^64 * 2^64))
print(15220267994978715897 + (2812694141792528170 * 2^64) + (9886878341545582154 * 2^64
* 2^64) + (4627147115546938088 * 2^64 * 2^64 * 2^64))

print("The result of `get_dummy_point` when n == 86 && k == 3:")
print(34318960048412842733519232 + (3417427387252098100392906 * 2^86) +
(2056756886390887154635856 * 2^86 * 2^86))
print(35848273954982567597050105 + (74802087901123421621824957 * 2^86) +
(4851915413831746153124691 * 2^86 * 2^86))
```

Impact: No impact to the current code base. Because in this function, the dummy point actually doesn't participate in the final calculation of the public key. However, we still recommend modifying it to prevent potential issues that may arise from the use of this function in future development.

Recommendation

Modify the values in the **get_dummy_point** function so that `outputs[0]` and `outputs[1]` have the same x and y coordinates as `G * 2 * * 255`. For example, when `n == 64 && k == 4`:

secp256k1_func.go

```
func get_dummy_point(n int, k int) [][]uint64 {
...
        } else if n == 64 && k == 4 {
                for i := range outputs {
                        outputs[i] = make([]uint64, 4)
                }
                outputs[0][0], _ = strconv.ParseUint("16770615581224985476", 10, 64)
                outputs[0][1], _ = strconv.ParseUint("8208947961671825091", 10, 64)
                outputs[0][2], _ = strconv.ParseUint("2673685488914591858", 10, 64)
                outputs[0][3], _ = strconv.ParseUint("12841891897255804443", 10, 64)

                outputs[1][0], _ = strconv.ParseUint("15062930234956941326", 10, 64)
                outputs[1][1], _ = strconv.ParseUint("1724884103647382360", 10, 64)
                outputs[1][2], _ = strconv.ParseUint("16777333066489264453", 10, 64)
                outputs[1][3], _ = strconv.ParseUint("18188747282752823003", 10, 64)
                return outputs
        } else {
...
}
```

Status

This issue has been acknowledged by the team.

12. Redundant constraints in CheckCarryToZero

Severity: Info	Category: Redundancy

Target:
- ecdsa_circuit/bigint.go

Description

The **CheckCarryToZero** circuit ensures that the sum of a sequence of integers, represented by the array `in`, is zero. The integers are encoded as `k` limbs (slices) of `n` bits each, with values ranging from `-2^(m-1)` to `2^(m-1)`.

bigint.go:L507-L516

```
func CheckCarryToZero(api frontend.API, n int, m int, k int, in []frontend.Variable) {
...
        if i == 0 {
            //carry[i] = shr_no_check(api, in[i], n, m)
            carry[i] = api.Mul(in[i], api.Inverse(LeftShift(1, uint64(n))))
            api.AssertIsEqual(in[i], api.Mul(carry[i], LeftShift(1, uint64(n))))
        } else {
            this_in := api.Add(in[i], carry[i-1])
            //carry[i] = shr_no_check(api, this_in, n, m)
            carry[i] = api.Mul(this_in, api.Inverse(LeftShift(1, uint64(n))))
            api.AssertIsEqual(this_in, api.Mul(carry[i], LeftShift(1, uint64(n))))
        }
...
}
```

When `i` is equal to `0`, the `carry[i]` array is calculated by right-shifting `in[i]` by `n` bits, done by multiplying with the inverse of `2^n`. However, the constraint composed of `api.AssertIsEqual` on the next line is unnecessary because `carry[i]` has already been constrained.
Similarly, when `i` is greater than `0`, it adds the previous `carry[i-1]` to `in[i]` to get `this_in`, then computes the `carry[i]` for this_in similarly by right-shifting it. However, it is unnecessary to ensure that `this_in` is equal to `carry[i]` multiplied by `2^n`, as `carry[i]` has already been constrained earlier.

Impact: No impact to the current code base. But in zero-knowledge proof circuits, repeated constraints can pose significant risks. These redundancies increase the complexity and size of the circuit, leading to higher computational costs and slower verification times.

Recommendation

Remove the two occurrences of `api.AssertIsEqual` in the code. Because they duplicate the constraints on `carry[i]` from the previous line.

Status

This issue has been acknowledged by the team.

13. Incorrect values of `LogCeilK` in LongToShortNoEndCarry

Severity: Info	Category: Logic Error
Target: - ecdsa_circuit/bigint.go	

Description

In **LongToShortNoEndCarry**, this code segment calculates `LogCeilK`, which determines the maximum bit width required for carrying. It is used to compute the ceiling of the base-2 logarithm of `k`, which effectively gives the maximum bit width required to represent `k` in binary. This value is then utilized to ensure that all carry values and intermediate results are properly handled within the required bit width.

bigint.go:L228-L233

```
func LongToShortNoEndCarry(api frontend.API, n int, k int, in []frontend.Variable)
[]frontend.Variable {
...
        LogCeilK := 0
        for i := 0; (1 << i) < k; i++ {
                if ((k >> i) & 1) == 1 {
                        LogCeilK = i + 1
                }
        }
...
}
```

However, after running the `for` loop, `LogCeilK` always remains `0` because the if condition is never satisfied during this process.

Impact: No impact to the current code base. Because `LogCeilK` only appears in the subsequent `shr_no_check` and `api.ToBinary()` in the circuit, and it does not affect carry range checks or circuit output results. However, the issue still needs to be addressed because the calculated value of `LogCeilK` does not match the developer's expectations.

Recommendation

Changing the condition statement of the `for` loop from `(1 << i) < k` to `(1 << i) <= k` will resolve the issue.

bigint.go

```
func LongToShortNoEndCarry(...) []frontend.Variable {
...
        LogCeilK := 0
        for i := 0; (1 << i) <= k; i++ {
                if ((k >> i) & 1) == 1 {
                        LogCeilK = i + 1
                }
        }
...
}
```

Status

This issue has been acknowledged by the team.

14. Unnecessary assignments in secp256k1_utils.go

Severity: Info	Category: Unused Circuit/Variable

Target:
- ecdsa_circuit/secp256k1_utils.go

Description

In the following three circuits in **secp256k1_utils.go**, they each construct a `frontend.Variable` slice to store the binary values of `Variable`.

secp256k1_utils.go:L62-L64

```
func CheckInRangeSecp256k1(api frontend.API, in []frontend.Variable) {
...
         for i := 0; i < 4; i++ {
                range64[i] = api.ToBinary(in[i], 64)
        }
...
}
```

secp256k1_utils.go:L155-L158

```
func CheckCubicModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        qRangeChecks := make([]frontend.Variable, 3)
        for i := 0; i < 3; i++ {
                qRangeChecks[i] = api.ToBinary(q[i], 64)
        }
...
}
```

secp256k1_utils.go:L249-L252

```
func CheckQuadraticModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        qRangeChecks := make([]frontend.Variable, 2)
        for i := 0; i < 2; i++ {
                qRangeChecks[i] = api.ToBinary(q[i], 64)
        }
...
}
```

However, the issue is that `range64[i]` and `qRangeChecks[i]` shown in the code are not used in the subsequent circuits. If a variable is not used, there's generally no point in storing the binary representation of `in[i]` and `q[i]`. Therefore, these assignment operations can be removed to improve the adherence of the code to standards and its readability.

Impact: No impact to the current code base.

Recommendation

Remove the unnecessary assignments in the three locations:

secp256k1_utils.go

```
func CheckInRangeSecp256k1(api frontend.API, in []frontend.Variable) {
...
         for i := 0; i < 4; i++ {
                 api.ToBinary(in[i], 64)
        }
...
}
```

secp256k1_utils.go

```
func CheckCubicModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        for i := 0; i < 3; i++ {
                api.ToBinary(q[i], 64)
        }
...
}
```

secp256k1_utils.go

```
func CheckQuadraticModPIsZero(api frontend.API, m int, in []frontend.Variable) {
...
        for i := 0; i < 2; i++ {
                api.ToBinary(q[i], 64)
        }
...
}
```

Status

This issue has been acknowledged by the team.

15. Unnecessary constraints in ModProd and Split

Severity: Info	Category: Redundancy

Target:

- ecdsa_circuit/bigint.go
- ecdsa_circuit/utils.go

Description

ModProd multiplies two input variables, `a` and `b`, producing a result `aMulb`. It then calculates the carry by right-shifting `aMulb` by `n` bits using the **shr** circuit. The `prod` is obtained by subtracting the shifted carry value from `aMulb`, effectively yielding the product modulo `2^n`.

bigint.go:L78-L86

```
func ModProd(api frontend.API, n int, a frontend.Variable, b frontend.Variable)
(frontend.Variable, frontend.Variable) {
        if n > 126 {
                panic("ModProd: n is too large")
        }
        aMulb := api.Mul(a, b)
        carry := shr(api, aMulb, n, 2*n)
        prod := api.Sub(aMulb, api.Mul(carry, LeftShift(1, uint64(n))))
        return prod, carry
}
```

The **shr** circuit uses the **shr_bigint** to decompose the input variable `n` into high `n` bits (`shr_res[0]`) and low `n` bits (`shr_res[1]`), then asserts this relationship: `n == shr_res[1] + (shr_res[0] << m)`. The calculation of `prod` is based on subtracting the shifted high bits from `aMulb`, which should yield the low `n` bits.

utils.go:L59-L73

```
func shr_bigint(curveID ecc.ID, inputs []*big.Int, outputs []*big.Int) error {
        n := inputs[1].Uint64()
        nmask := new(big.Int).Lsh(big.NewInt(1), uint(n))
        nmask = nmask.Sub(nmask, big.NewInt(1))
        outputs[0] = new(big.Int).Rsh(inputs[0], uint(n))
        outputs[1] = new(big.Int).And(inputs[0], nmask)
        return nil
}

func shr(api frontend.API, n frontend.Variable, m int, maxBits int) frontend.Variable {
        shr_res, _ := api.Compiler().NewHint(shr_bigint, 2, n, m)
        api.ToBinary(shr_res[1], m)
        api.AssertIsEqual(api.Add(shr_res[1], api.Mul(shr_res[0], LeftShift(1,
uint64(m)))), n)
        return shr_res[0]
}
```

Therefore, the value stored in the variable `prod` in the **ModProd** circuit is identical to the value of `shr_res[1]` in the **shr** circuit.

The same issue also occurs in the **Split** circuit, where the variable `small` should be directly assigned the value of `shr_res[1]` instead of being calculated using unnecessary constraints.

bigint.go:L89-L98

```
func Split(api frontend.API, n int, m int, in frontend.Variable) (frontend.Variable,
frontend.Variable) {
	if n > 126 {
		panic("Split: n is too large")
	}

	big := shr(api, in, n, n+m)
	small := api.Sub(in, api.Mul(big, LeftShift(1, uint64(n))))
	api.AssertIsEqual(in, api.Add(small, api.Mul(big, LeftShift(1, uint64(n)))))
	return small, big
}
```

Impact: No impact to the current code base. But this problem will lead to unnecessary constraint addition and waste of computing resources

Recommendation

Make the **shr** circuit return the complete `shr_res` instead of just `shr_res[0]`, or write another circuit `shr_re_1` that can return `shr_res[1]`. Then simply assigning `shr_res[1]` to `prod` and `small` will avoid adding unnecessary constraints.

Status

This issue has been acknowledged by the team.

16. Redundant constraints in Split

Severity: Info	Category: Redundancy
Target: - ecdsa_circuit/bigint.go	

Description

The **Split** circuit splits an input variable into `small` and `big` parts based on specified bit lengths `n` and `m`. It right-shifts the input to extract the `big` part and calculates the `small` part by subtracting the `big` part shifted back, then ensures the sum of these parts equals the original input.

bigint.go: L89-L98

```
func Split(api frontend.API, n int, m int, in frontend.Variable) (frontend.Variable,
frontend.Variable) {
	if n > 126 {
		panic("Split: n is too large")
	}

	big := shr(api, in, n, n+m)
	small := api.Sub(in, api.Mul(big, LeftShift(1, uint64(n))))
	api.AssertIsEqual(in, api.Add(small, api.Mul(big, LeftShift(1, uint64(n)))))
	return small, big
}
```

However, the constraint for `small` has already been enforced in `api.Sub()`, so the series of constraints wrapped by `api.AssertIsEqual()` in the next line are redundant.

Impact: No impact to the current code base.

Recommendation

Remove the line `api.AssertIsEqual(in, api.Add(small, api.Mul(big, LeftShift(1, uint64(n)))))` in **Split**.

Status

This issue has been acknowledged by the team.

17. Unnecessary constraints in shr_no_check

Severity: Info	Category: Redundancy
Target: - ecdsa_circuit/utils.go	

Description

The **shr_no_check** circuit performs a right shift operation on a variable n by m bits without overflow checks. It uses a hint function to compute the result, ensuring that the high part of the result (`shr_res[1]`) is zero and verifies that reconstructing the original value from the shifted result matches n.

utils.go:L74-L79

```
func shr_no_check(api frontend.API, n frontend.Variable, m int, maxBits int)
frontend.Variable {
	shr_res, _ := api.Compiler().NewHint(shr_bigint, 2, n, m)
	api.AssertIsEqual(shr_res[1], 0)
	api.AssertIsEqual(api.Add(shr_res[1], api.Mul(shr_res[0], LeftShift(1,
uint64(m)))), n)
	return shr_res[0]
}
```

Since `shr_res[1]` is never used for any other purpose, we can skip its constraint because the prover always assigns it to `0`, and it is not used anywhere. Unfortunately, due to the characteristics of the hint function in gnark, this circuit needs to constrain the entire return value of `shr_bigint`. This forces the code to constrain `0 + shr_res[0] * 2^m === n` instead of `shr_res[0] * 2^m === n`, which introduces some unnecessary constraints and reduces the readability of the code.

Impact: No impact to the current code base.

Recommendation

In the circuits using **shr_no_check**, replace `a := shr_no_check(api, x, n, m)` with `a = api.Mul(x, api.Inverse(LeftShift(1, uint64(n))))`. This approach has already been used in circuits such as `CheckCarryToZero`.

Status

This issue has been acknowledged by the team.

18. Using deprecated functions in Test_secp256k1_hint

Severity: Info	Category: Architectural Design

Target:
- ecdsa_circuit/secp256k1_test.go

Description

In this test function, it uses `elliptic.GenerateKey()` to generate a public-private key pair on the respective elliptic curve. The private key is generated using the given reader, which must return random data.

secp256k1_test.go:L112-L119

```
func Test_secp256k1_hint(t *testing.T) {
        assert := test.NewAssert(t)
        _, x1, y1, _ := elliptic.GenerateKey(crypto.S256(), rand.Reader)
        x_32, y_32 := Bigint_to_32_bytes(x1), Bigint_to_32_bytes(y1)

        x1_input, y1_input := convert_32_bytes_to_4_big_int(x_32),
convert_32_bytes_to_4_big_int(y_32)

        _, x2, y2, _ := elliptic.GenerateKey(crypto.S256(), rand.Reader)
...
}
```

However, this function has been officially deprecated. More details about it can be found at this link.

Impact: No impact to the current code base.

Recommendation

For ECDSA test, use the `GenerateKey` function of the `crypto/ecdsa` package instead.

Status

This issue has been acknowledged by the team.

19. Assertion missing in ModSubThree

Severity: Info	Category: Architectural Design

Target:
- ecdsa_circuit/bigint.go

Description

ModSubThree computes the result of `a - b - c` in a modular arithmetic context, ensuring non-negativity by adding `2^n` if necessary. It first checks that `n` is less than `251` to prevent overflow, then calculates `b + c` and `a + 2^n`, determines if borrowing is needed by comparing `a` and `b + c`, selects the appropriate value of `a` based on the `borrow`, and finally performs the subtraction.

bigint.go:L40-L53

```go
// a - b - c
// assume a - b - c + 2**n >= 0
func ModSubThree(api frontend.API, n int, a frontend.Variable, b frontend.Variable, c
frontend.Variable) (frontend.Variable, frontend.Variable) {
        //api.AssertIsLessOrEqual(n, 251)
        if n >= 251 {
                panic("ModSubThree: n is too large")
        }
        bPlusc := api.Add(b, c)
        borrowed_a := api.Add(a, LeftShift(1, uint64(n)))
        borrow := myIsLess(api, n+1, a, bPlusc)
        this_a := api.Select(borrow, borrowed_a, a)
        out := api.Sub(this_a, bPlusc)
        return borrow, out
}
```

However, the circuit does not ensure the condition `a - b - c + 2**n >= 0` mentioned in the comment, and it imposes unnecessary restrictions on the case when `n` equals `251`.

Impact: No impact to the current code base.

Recommendation

Add an `if` condition check in the circuit to ensure `a - b - c + 2**n >= 0` instead of a constraint. Also, modify the check for the value of `n` so that it can be equal to `251`.

bigint.go

```go
// a - b - c
// assume a - b - c + 2**n >= 0
func ModSubThree(api frontend.API, n int, a frontend.Variable, b frontend.Variable, c
frontend.Variable) (frontend.Variable, frontend.Variable) {
        //api.AssertIsLessOrEqual(n, 251)
        if n > 251 {
                panic("ModSubThree: n is too large")
        }
        if a - b - c + (1 << n) < 0 {
                panic("ModSubThree: a - b - c + 2**n < 0")
        }
        bPlusc := api.Add(b, c)
        borrowed_a := api.Add(a, LeftShift(1, uint64(n)))
        borrow := myIsLess(api, n+1, a, bPlusc)
        this_a := api.Select(borrow, borrowed_a, a)
        out := api.Sub(this_a, bPlusc)
        return borrow, out
}
```

Status

This issue has been acknowledged by the team.

20. Unnecessary constraints in BigMultNoCarry

Severity: Info	Category: Redundancy

Target:
- ecdsa_circuit/bigint.go

Description

The **BigMultNoCarry** circuit performs a polynomial multiplication of two large integers represented as arrays of variables. It ensures that the multiplication does not exceed a specified bit length (`ma + mb <= 253`), and the multiplication is conducted in such a way that no carrying occurs.

bigint.go: L134-L174

```
func BigMultNoCarry(api frontend.API, n int, ma int, mb int, ka int, kb int, a
[]frontend.Variable, b []frontend.Variable) []frontend.Variable {
        if ma+mb > 253 {
                panic("BigMultNoCarry: ma + mb is too large")
        }

        out := make([]frontend.Variable, ka+kb-1)
->      a_poly := make([]frontend.Variable, ka+kb-1)
->      b_poly := make([]frontend.Variable, ka+kb-1)
->      out_poly := make([]frontend.Variable, ka+kb-1)

        for i := 0; i < ka+kb-1; i++ {
                out[i] = 0
        }
        for i := 0; i < ka; i++ {
                for j := 0; j < kb; j++ {
                        out[i+j] = api.Add(out[i+j], api.Mul(a[i], b[j]))
                }
        }
->      for i := 0; i < ka+kb-1; i++ {
->              a_poly[i] = 0
->              b_poly[i] = 0
->              out_poly[i] = 0
->              powers := make([]int, ka+kb-1)
->              for j := 0; j < ka+kb-1; j++ {
->                      powers[j] = PowInts(i, j)
->              }
->              for j := 0; j < ka+kb-1; j++ {
->                      out_poly[i] = api.Add(out_poly[i], api.Mul(out[j], powers[j]))
->              }
->              for j := 0; j < ka; j++ {
->                      a_poly[i] = api.Add(a_poly[i], api.Mul(a[j], powers[j]))
->              }
->              for j := 0; j < kb; j++ {
->                      b_poly[i] = api.Add(b_poly[i], api.Mul(b[j], powers[j]))
->              }
->      }
->      for i := 0; i < ka+kb-1; i++ {
->              api.AssertIsEqual(out_poly[i], api.Mul(a_poly[i], b_poly[i]))
->      }
        return out
}
```

The circuit iterates through each term of `a` and `b` using nested loops, performs polynomial multiplication, and accumulates the product of each term into the `out[]` array. Then, the circuit converts the coefficients into polynomial form by calculating the power of each term and multiplying it by the corresponding coefficient, summing them to form a polynomial. Finally, it uses `api.AssertIsEqual` to verify the result of the polynomial multiplication, and performs secondary validation and constraint on `out[]`. However, since the output `out[]` of this circuit has already been properly constrained within the `for` loop, the series of validations in the yellow part are unnecessary and introduce extra constraints.

Impact: No impact to the current code base. However, adding unnecessary constraints to the current circuit consumes additional computational resources and more time.

Recommendation

Remove the `->` section from the code segment.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
ecdsa_circuit/bigint.go	fc33a88644673c1d30424a1ea62410b16547ed81
ecdsa_circuit/bigint_4x64_mult.go	3f09a4570e4f85ecdc338531b40698390a171314
ecdsa_circuit/bigint_4x64_mult_test.go	0ab0bc9f10567b1662fca4cc285a8c7635f7d9a6
ecdsa_circuit/bigint_func.go	f8397288a335e31cbe8a6187f1fd801668136689
ecdsa_circuit/bigint_func_test.go	8d4ca392800827725dc53887ef84912acdcd274f
ecdsa_circuit/bigint_test.go	080ebeabe24c3f93e4ab4bc8b12b682ca7eddc14
ecdsa_circuit/circuit.go	0f6339a756dfc157da0bf17b0be41d2188d5938b
ecdsa_circuit/ecdsa.go	071ec4bf35a21ec7b119bb2096103f4863757b76
ecdsa_circuit/ecdsa_func.go	9941137cdb17555fb75d9d41d4af3c939a1777ae
ecdsa_circuit/ecdsa_test.go	52024b12908c523600f16760c4210bf14ae81f86
ecdsa_circuit/hint_idiv.go	811d60d21fdddaf71953dd896282ef13a9d0fe9a
ecdsa_circuit/multiplexer.go	c0a1a12659c0a80afb5ccbaa81edb4d81a3b6051
ecdsa_circuit/secp256k1.go	b3803dd2f34961c33763ec1e8afcff61b29a7756
ecdsa_circuit/secp256k1_func.go	b3880eca6ccc84fc737d69595a6e733ccfecc41c
ecdsa_circuit/secp256k1_test.go	740637d51859c6abf38b5da7dff28833f2dc8539
ecdsa_circuit/secp256k1_utils.go	7fae3e604daea23230c00a3d79e32e2a1ba68ea6
ecdsa_circuit/secp256k1_utils_test.go	6ee7af1a0c14215b84a81c5adf9b6019503b95c1
ecdsa_circuit/utils.go	65906849019949a387c8eb01d31b1f90b5bfbb0b

And we audited the following files after the team's fix:

File	SHA-1 hash
ecdsa_circuit/bigint.go	18543341f442b07d63db00f2f863ac84f17054bc
ecdsa_circuit/bigint_4x64_mult.go	ca9d639d69790cba4756e580b3533120debfc9e3
ecdsa_circuit/bigint_4x64_mult_test.go	ee6200987da7f2923398819144d7eb9086f47f9a
ecdsa_circuit/bigint_func.go	cefc083333c03b87562f18521374ed0c87b1da99
ecdsa_circuit/bigint_func_test.go	07c30f87b544aa8e8b312c171114b5a7137486ad
ecdsa_circuit/bigint_test.go	154900de95447eff3901a59228ccd861f4a2b931
ecdsa_circuit/circuit.go	1d3529aeb0f46499be8e62758b0c66d986e8ae0e
ecdsa_circuit/ecdsa.go	828d1daa58366903ecfa287c55a7876e4c198331
ecdsa_circuit/ecdsa_func.go	719e163963286b21aec76d689cb9a5f0b68e3208

ecdsa_circuit/ecdsa_test.go	50cefa685fbc0091c5a60afec76a22f31f447eba
ecdsa_circuit/hint_idiv.go	cb489a2ec8d0030d196e4ecbf13109cda741d2fe
ecdsa_circuit/multiplexer.go	0f1b98df939bc3fe77ffc147e2032070539cab81
ecdsa_circuit/secp256k1.go	db1f58540c73014c9dc7ba799f0e14a29d30c74b
ecdsa_circuit/secp256k1_func.go	b6e9cb233a77b06a391b08eae29eff96c5138f9e
ecdsa_circuit/secp256k1_test.go	932167d82a00bdef751f79a541b25d489f50b057
ecdsa_circuit/secp256k1_utils.go	a0351eb8b8c79b4e2c4bf4bcd0ba9c12f7271b1b
ecdsa_circuit/secp256k1_utils_test.go	7e015024e0ce99d15be8fe12d68b0fb6368fca50
ecdsa_circuit/utils.go	434dbdfae06d0cce05036233d3a875ad80165527