Auditing Report

FOR

circom-bigint (circomlib)

0xPARC Community

Ethereum Foundation

Veridise Inc.

Nov 6, 2022

► **Prepared For:**

circom-bigint (circom-lib).
https://github.com/0xbok/circom-bigint
https://github.com/iden3/circomlib

► **Prepared By:**

♦ **0xPARC Community**
- Brian Gu
- Michael Chu
- Yi Sun
- Jonathan Wang

♦ **Ethereum Foundation**
- blockdev
- kcharbo

♦ **Veridise Inc.**
- Yanju Chen
- Junrui Liu
- Hanzhi Liu
- Yu Feng

► **Contact Us:**
hello@0xPARC.org
info@ethereum.org
contact@veridise.com

► **Version History:**

October 2, 2022 Draft

Contents

Executive Summary 1

From July 28 to November 15, 2022, through a joint effort among 0xPRAC Community, Ethereum Foundation, and Veridise, we reviewed the security of the `circom-bigint` curcuit implementation and its dependent library `circomlib`. The review covered all circuits implemented using circom in the `circom-bigint`'s repository (`https://github.com/0xbok/circom-bigint/tree/audit`) * and its dependant library (`https://github.com/iden3/circomlib`) †. We conducted this assessment over 24 person-weeks, with 4 engineers working on commit 7505e5c of the client's repository (cff5ab6 of its dependant repository). The auditing strategy involved both manual and tool-assisted analysis of the source code performed by engineers from 0xPARC Community, EF, and Veridise. The tools that were used in the audit include a combination of static analysis and interactive theorem prover using Coq. The outcome of the auditing includes 1) this auditing report, and 2) 20K+ lines of machine-checkable proof in Coq.

Summary of issues detected. The audit uncovered 14 issues in `circomlib`, including 9 issue of critical severity. The critical severity issues (V-BIGINT-COD-001, V-CIRCOMLIB-PIC-001 ~ V-CIRCOMLIB-PIC-008) correspond to underconstrained issues in circuits, which allow attackers to construct spurious proofs that violate the intended functional behavior yet bypass the validation checks, thus compromising potential functionality of the system that incorporates the target circuits. The rest of the issues include 5 low-severity issues (V-BIGINT-VUL-001 ~ V-BIGINT-VUL-003, V-CIRCOMLIB-PIC-009, V-CIRCOMLIB-PIC-010) that involves empty circuit templates that could potentially cause underconstrained errors when used, as well as optimizations on constraint size and documentations. In addition to the above-mentioned issues, we also formally verified (with machine-checkable proof in Coq) the functional correctness of the circuits with respect to their specifications written by 0xPARC Community and proved the absence of underconstrained issues.

Code assessment. The core `circom-bigint` library implements big integer operations in circom. The core logic of the library is split into the following 3 parts:

- `bigint.circom`: This contains the core logic and major templates for big integer operations.
- `bigint_4x64_mult.circom`: This contains pre-defined big integer functions for use in Secp256k1 elliptic curve.
- `bigint_func.circom`: This contains the core helper functions for the major templates.

The repository is intended to be used as a library to support scenarios that involve big integer operations, e.g. ECDSA operations as seen in `circom-ecdsa` library. However, as our investigation uncovered, there's insufficient documentation on the library templates provided. Specifically, documentations about some templates' input/output assumptions and functional properties are missing, which, as suggested in the detailed analysis report in the follow-up sections, could induce critical issues when misused by developers. We would strongly encourage

* commit:7505e5c
† commit:cff5ab6

the developers to improve the documentation of the project, especially on some indispensable security related assumptions for using the templates from the library.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall any of the parties and auditors be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
circom-bigint	2eceb9c	Circom	Native/Linux
circomlib	cff5ab6	Circom	Native/Linux

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
July 28-Nov 15, 2022	Manual & Tools	4	24 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed
High-Severity Issues	9	0
Medium-Severity Issues	0	0
Low-Severity Issues	5	0

Table 2.4: Category Breakdown.

Name	Number
Optimization	3
Underconstrained Error	11

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the `circom-bigint` and its dependent library `circomlib`. Specifically, we sought to answer the following questions:

- Are the circuits implemented according to their functional specification?
- Are the circuits secured against adversaries?
- Are all circuit (critical) components properly constrained?
- Does the protocol perform all necessary data validation and checks for its inputs?

3.2 Audit Methodology & Scope

Audit Methodology. Because this audit includes a wide range of goals, some of which are not amenable to automation, our audit methodology involved a combination of human experts and a variety of automated program analysis tools. In particular, during our audit, we leveraged the following technologies:

- *Static analysis*. To ensure that the `circom-bigint` circuits and their dependencies are free of any common defects, we used an open-source static analysis tool, PICUS, developed by Veridise. Specifically, PICUS aims to perform the following two tasks: 1. detect common buggy patterns in circom circuits and 2. determine whether all circuit templates are properly constrained, which is a crucial security property for ZK circuits. Additionally, when a buggy pattern or underconstrained signal is found, PICUS invokes a process based on automated theorem proving to reason about and compute a concrete counterexampmle, which is a set of witnesses that violate the security properties yet admitted by the constraint system.
- *Formal verification*. Notably, we formally verified the circuits in `circom-bigint` using Veridise's tool CODA. The formal verification includes functional specifications from the original authors as well as 20K+ machine-checkable proof in Coq. A brief summary of our formal verification using CODA can be found in Section 4.
- *Manual inspection*. We further performed manual inspection on the entire codebase in `circom-bigint` and suggest a couple of improvement regarding the performance of the circuits (Section 5).

Scope. The scope of this audit includes all `circom-bigint` circom circuits, as well as its dependant library `circomlib`. As such, Our security engineers first reviewed the provided documentation to understand the desired behavior of the protocol as a whole. They then inspected the provided tests to understand the desired behavior of the protocol's circuits as well as how users are expected to interact with them.

Limitations. Due to the scope of our audit, the recommendations provided in this report are limited to the functional specification provided by the `circom-bigint` developers. The overall security of the system can be compromised if:

1. the circuits are not deployed according to industry standards, i.e., following a secure trusted setup ceremony, the whole protocol can be at risk in case the common reference string (CRS) is leaked,
2. the original specification is not strong enough to cover the actual behavior, or
3. any component outside the scope of the audit is vulnerable.

3.3 Classification of Vulnerabilities

When our auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Moderate
Likely	Warning	Low	Moderate	High
Very Likely	Low	Moderate	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Formal Verification Results

As mentioned in Section 3, part of our audit efforts include formal verification using tools developed by Veridise, namely Coda and Picus. Coda is an open-source library that can be used to prove the functional correctness of zero-knowledge circuits by leveraging the Coq proof assistant. Picus is an open-source security analysis tool that can be used to automatically find safety bugs in zero-knowledge circuits.

In what follows, we elaborate on the verification results from both tools. Specifically:

- For proving functional correctness on `circom-bigint` using Coda, we first give an overview about necessary background, workflow along with additional assumptions. Then we give a high-level overview of the templates certified, and go with details about the findings.
- For findings of underconstrained bugs on `circom-bigint`'s dependent library `circomlib` using Picus, we list out the details of each of them found, with potential impacts and recommendations for patching.

Both Coda and Picus are open-source tools developed by Veridise. They are available on Github:

- Coda: `https://github.com/Veridise/Coda`
- Picus: `https://github.com/Veridise/Picus`

4.1 Formal Verification Using Coda

In this section we elaborate on the formal verification process in order to prove the functional correctness of a circuit from `circom-bigint` and its dependant library `circomlib` using Coda.

Overview Coda is an open-source Coq library for semi-automatically performing formal verification on zero-knowledge circuits. To improve the degree of automation, Coda provides a set of useful tactics for formally verifying functional correctness of ZK circuits.

Workflow To verify the functional correctness of a given template, Coda starts with its specification from the original repository (see here and here), where the following conditions are given:

- Pre-conditions, including inputs X and assumptions of relations between them $\phi(X)$;
- Post-conditions, including outputs Y and properties of relations between them $\psi(Y)$.

Coda then incorporates a semi-automated transpilation process that converts the original circuit template into its corresponding Coq constraint representation $\theta(X, Y)$. Together with the pre-conditions and post-conditions from specification, Coda can certify the functional correctness of a given template, if the following holds:

$$\forall X, Y. (\phi(X) \wedge \theta(X, Y)) \Rightarrow \psi(Y).$$

Otherwise, we derive additional conditions to verify the template but mark it as "Not Certified".

Assumptions and background Sometimes, the provided specification is insufficient to prove circuit soundness. This is usually remedied via the following means:

- Adding/Strengthening pre-conditions:
  - It can be the case that the developer forgets to make explicit some assumptions about the parameters of a circuit, perhaps because the circuit has only been instantiated internally and hence all said assumptions are satisfied implicitly. However, if the circuit is to be made for public use, explicitly stating those assumptions becomes critical. In practice, as circuit verifiers, we try to add the weakest pre-conditions that can make the post-condition hold.
  - Since most circuits make liberal use of loops, we also need to supply loop invariants when constructing a correctness proof, and prove that those loop invariants hold. In practice, the loop invariants are either trivial, or can be easily derived from the post-condition (or back propagation of the post-condition).
- In order for Coq to be able to reason about circuit correctness, we need to encode, or embed, circuits as native Coq terms. The Coq embedding of circuits that we have chosen is straightforward, and can be coded into a mechanical procedure:
  - A circuit template is represented as a record type that has as its fields the public signals, and a special field `cons` that represents the circuit body as a relation over the public signals.

- We use existential quantifiers to represent private signals and circuit components (i.e. instantiation).
- Loops are represented using the high-order function:

  `iter: (nat -> A -> A) -> nat -> A -> A`.

  That is, we represent the loop body as an anonymous function `f` of type `nat -> A -> A`, where `nat` is the current loop index, and `A` is the type of the variables that are modified inside the loop (called states). Then, `iter` takes the number of iterations `n` and the initial state, and outputs the final state obtained by applying `f` to the initial state `n` times.

► We use the following external libraries in addition to Coq's standard library.

- We use the formalization of finite fields developed by fiat-crypto.
- We use coqprime to generate primality proof for the BabyJubjub prime.

Results Table 4.1 summarizes the verification results from Coda. In total, 30 templates from both `circom-bigint` and `circomlib` are verified by Coda, where one of them is found buggy* and we provide a fix for it (`380e5430fe3e4effbd62fdb5abb7ea93af686f97`). We elaborate in the following sections more details on the results. The full set of Coda proofs for `circom-bigint` and `circomlib` can be found here and here.

Table 4.1: Summary of Coda verification results.

Library	Template	Status
circomlib/circuits/bitify.circom	Num2Bits	Certified
circomlib/circuits/bitify.circom	Bits2Num	Certified
circomlib/circuits/comparators.circom	IsZero	Certified
circomlib/circuits/comparators.circom	IsEqual	Certified
circomlib/circuits/comparators.circom	LessThan	Certified
circomlib/circuits/gates.circom	AND	Certified
circomlib/circuits/gates.circom	OR	Certified
circomlib/circuits/gates.circom	XOR	Certified
circomlib/circuits/gates.circom	NAND	Certified
circomlib/circuits/gates.circom	NOR	Certified
circomlib/circuits/gates.circom	NOT	Certified
circomlib/circuits/multiplexer.circom	EscalarProduct	Certified
circuits/bigint.circom	BigIsEqual	Certified
circuits/bigint.circom	BigIsZero	Certified
circuits/bigint.circom	ModSubThree	Certified
circuits/bigint.circom	ModSumThree	Certified
circuits/bigint.circom	ModProd	Certified
circuits/bigint.circom	Split	Certified
circuits/bigint.circom	SplitThree	Certified
circuits/bigint.circom	BigAdd	Certified
circuits/bigint.circom	BigMultShortLongUnequal	In Progress
circuits/bigint.circom	LongToShortNoEndCarry	In Progress
circuits/bigint.circom	BigMult	In Progress
circuits/bigint.circom	BigLessThan	Certified
circuits/bigint.circom	BigMod	Fixed
circuits/bigint.circom	BigAddModP	Certified
circuits/bigint.circom	BigSub	Certified
circuits/bigint.circom	BigSubModP	Certified
circuits/bigint.circom	BigModInv	In Progress
circuits/bigint.circom	CheckCarryToZero	Certified

Properties [| x |] denotes the value of a big integer x.

Num2Bits(n)	Convert a signal vector to the number it represents in little-endian base-2 representation.
Pre-condition	$\top$
Post-condition	$\mathsf{in} = \sum_{i=0}^{n-1} 2^i \cdot \mathsf{out}[i] \wedge (\forall i < \mathsf{n}.\mathsf{out}[i] = 0 \vee \mathsf{out}[i] = 1)$
Property	-

* The bug only appeared in bigInt used by the circom-pairing library.

IsZero(n)	Check if given input is equal to zero.
Pre-condition	$\top$
Post-condition	$\mathsf{ite}(\mathsf{in} = 0, \mathsf{out} = 1, \mathsf{out} = 0)$
Property	-

IsEqual(n)	Check if the two given inputs are equal.
Pre-condition	$\top$
Post-condition	$\mathsf{ite}(\mathsf{in}[0] = \mathsf{in}[1], \mathsf{out} = 1, \mathsf{out} = 0)$
Property	-

LessThan(n)	Check if a given bigint is less than the other.
Pre-condition	$n \leq 252 \land \mathsf{in}[0] \leq 2^n - 1 \land \mathsf{in}[1] \leq 2^n - 1$
Post-condition	$\mathsf{ite}(\mathsf{toZ}(\mathsf{in}[0]) <_{\mathbb{Z}} \mathsf{toZ}(\mathsf{in}[1]), \mathsf{out} = 1, \mathsf{out} = 0)$
Property	-

BigIsEqual(k)	Check if two bigints are equal.
Pre-condition	$1 \leq k \leq 253$
Post-condition	$(a = b \Rightarrow \mathsf{out} = 1) \land (a \neq b \Rightarrow \mathsf{out} = 0)$
Property	if a equals b, out equals 1; otherwise, out equals 0.

BigIsZero(k)	Check if bigint a is equal to zero.
Pre-condition	$1 \leq k \leq 253$
Post-condition	$(\mathsf{in} = 0 \Rightarrow \mathsf{out} = 1) \land (\mathsf{in} \neq 0 \Rightarrow \mathsf{out} = 0)$
Property	If in equals 0, out equals 1; otherwise, out equals 0.

ModSubThree(n)	Compute $a - b - c$ with borrow bit.
Pre-condition	$n \leq 251 \land a \leq 2^n - 1 \land b \leq 2^n - 1 \land \mathsf{bin}(c)$
Post-condition	$\mathsf{out} = (a + \mathsf{borrow} \cdot 2^n) - b - c \land \mathsf{out} \leq 2^n - 1 \land \mathsf{bin}(\mathsf{borrow}) \land$ $(\mathsf{borrow} = 1 \Leftrightarrow a < b + c)$
Property 1	$\mathsf{out} = a - b - c + \mathsf{borrow} \cdot 2^n \land \mathsf{out} < 2^n - 1$
Property 2	borrow is binary and $\mathsf{borrow} = 1$ if and only if $a < b + c$

ModSumThree(n)	Compute addition mod 2^n with carry bit.
Pre-condition	$n \leq 252 \land a \leq 2^n - 1 \land b \leq 2^n - 1 \land \mathsf{bin}(s)$
Post-condition	$\mathsf{sum} + \mathsf{carry} \cdot 2^n = a + b + c \land \mathsf{sum} \leq 2^n - 1 \land \mathsf{bin}(c)$
Property 1	sum equals $a + b + c - \mathsf{carry} \cdot 2^n$ and is less than $2^n - 1$
Property 2	carry is binary

ModProd(n)	Compute product mod 2^n with carry.
Pre-condition	$2 \cdot n <= k$
Post-condition	$\mathsf{carry} \cdot 2^n + \mathsf{prod} = a \cdot b \land \mathsf{prod} \leq 2^n - 1$
Property 1	$\mathsf{carry} \cdot 2^n + \mathsf{prod} = a \cdot b$
Property 2	$\mathsf{prod} \leq 2^n - 1$

Split(n, m)	Split a $(n + m)$ bit input into two outputs.
Pre-condition	$\top$
Post-condition	$\mathsf{small} \leq 2^n - 1 \land \mathsf{big} \leq 2^m - 1 \land \mathsf{in} = \mathsf{small} + \mathsf{big} \cdot 2^n$
Property	-

SplitThree(n, m, k)	Split a $(n+m+k)$ bit input into three outputs.
Pre-condition	$\top$
Post-condition	$\mathsf{small} \leq 2^{n} - 1 \wedge \mathsf{medium} \leq 2^{m} - 1 \wedge \mathsf{big} \leq 2^{k} - 1 \wedge$ $\mathsf{in} = \mathsf{small} + \mathsf{medium} \cdot 2^{n} + \mathsf{big} \cdot 2^{(n+m)}$
Property	-

BigAdd(n, k)	Add two bigints.
Pre-condition	$n > 0 \wedge k > 0 \wedge n \leq 252 \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b)$
Post-condition	$[\lvert out\rvert] = [\lvert a\rvert] + [\lvert b\rvert] \wedge \mathsf{bigint}(out)$
Property	-

BigLessThan(n, k)	Check which of two bigints is larger.
Pre-condition	$n \leq 252 \wedge 2 \leq k \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b)$
Post-condition	$\mathsf{binary}(out) \wedge (out = 1 \Leftrightarrow [\lvert a\rvert] < [\lvert b\rvert])$
Property	-

BigMod(n, k)	Division with remainder of two bigints.
Pre-condition	$n > 0 \wedge k > 0 \wedge n \leq 251 \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b)$
Post-condition	$\mathsf{bigint}(div) \wedge \mathsf{bigint}(mod) \wedge [\lvert a\rvert] = [\lvert div\rvert] \cdot [\lvert b\rvert] + [\lvert mod\rvert]$
Property	-

BigAddModP(n, k)	Add two bigints.
Pre-condition	$n > 0 \wedge k > 0 \wedge n <= 251 \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b) \wedge \mathsf{bigint}(p) \wedge$ $[\lvert a\rvert] < [\lvert p\rvert] \wedge [\lvert b\rvert] < [\lvert p\rvert]$
Post-condition	$[\lvert out\rvert] = ([\lvert a\rvert] + [\lvert b\rvert]) \bmod [\lvert p\rvert] \wedge \mathsf{bigint}(out)$
Property	-

BigSub	Subtract two bigints.
Pre-condition	$n > 0 \wedge k > 0 \wedge n \leq 251 \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b)$
Post-condition	$\mathsf{bigint}(out) \wedge \mathsf{bin}(underflow) \wedge ([\lvert a\rvert] \geq [\lvert b\rvert] \Longrightarrow underflow = 0 \wedge$ $[\lvert out\rvert] = [\lvert a\rvert] - [\lvert b\rvert]) \wedge ([\lvert a\rvert] < [\lvert b\rvert] \Longrightarrow underflow = 1 \wedge$ $[\lvert out\rvert] = 2^{(n \cdot k)} + [\lvert a\rvert] - [\lvert b\rvert])$
Property 1	$out = a - b$
Property 2	underflow equals how much is borrowed at the highest register of subtraction; only nonzero if $a < b$

BigSubModP	Subtract two bigints.
Pre-condition	$n > 0 \wedge k > 0 \wedge n \leq 251 \wedge \mathsf{bigint}(a) \wedge \mathsf{bigint}(b) \wedge \mathsf{bigint}(p) \wedge$ $[\lvert a\rvert] < [\lvert p\rvert] \wedge [\lvert b\rvert] < [\lvert p\rvert]$
Post-condition	$\mathsf{bigint}(out) \wedge [\lvert out\rvert] = ([\lvert a\rvert] - [\lvert b\rvert]) \bmod [\lvert p\rvert]$
Property	-

CheckCarryToZero(n, m, k)	Constrain that in[] (signed overflow representation) evaluated at $X = 2^{n}$ as a big integer equals zero.
Pre-condition	$1 \leq n \leq m \wedge k \geq 2 \wedge m \leq 251 \wedge \forall i < k.\ \mathsf{in}[i] \in (-2^{m-1}, 2^{m-1})$
Post-condition	$[\lvert in\rvert] = 0$
Property	-

4.1.1 Example: BigIsEqual(k)

`BigIsEqual` is an important circuit in the library since it is used by both circom-pairing and circom-ecdsa. Essentially, this circuit checks if k-register variables a, b are equal everywhere.

```
template BigIsEqual(k) {
    signal input a[k];
    signal input b[k];
    signal output out;

    component isEquals[k];
    var total = k;
    for (var i = 0; i < k; i ++) {
        isEquals[i] = IsEqual();
        isEquals[i].in[0] <== a[i];
        isEquals[i].in[1] <== b[i];
        total -= isEquals[i].out;
    }
    component checkZero = IsZero();
    checkZero.in <== total;
    out <== checkZero.out;
}
```

To prove correctness of this circuit, we have to first translate the above specification into a program written in Coda's *specification language*. This corresponds to the following code snippet taken from the Coda repository:

```
Definition spec (c: t) : Prop :=
  (* pre-condition *)
  1 <= k <= 253 →
  (* post-condition *)
  if (forallb (fun x ⇒  (fst x = snd x)? ) (ListUtil.map2 pair (' c.(a)) (' c.(b))))
  then
    c.(out) = 1
  else
    c.(out) = 0
```

Assuming that $1 <= k <= 253$, this specification ensures the following property:

- if k-register variables a, b are equal everywhere, out is equal to 1. if not, out is equal to 0

The property of this circuit can be verified by manually providing the following loop invariant in Coda:

```
pose (Inv := fun (i:nat) '((total, _cons): (F * Prop)) ⇒ _cons →
      total = (fold_left
                  (fun x y ⇒  if (fst y = snd y)? then x - 1 else x)
                  (ListUtil.map2 pair (' a [:i]) (' b [:i]))
                  (F.of_nat q k))).
```

4.1.2 Example: BigAdd(n, k)

`BigAdd` is a crucial component of the library as it is used by real-world applications such as circom-pairing. Essentially, this circuit performs addition on big integers.

```
template BigAdd(n, k) {
    assert(n <= 252);
    signal input a[k];
    signal input b[k];
    signal output out[k + 1];

    component unit0 = ModSum(n);
    unit0.a <== a[0];
    unit0.b <== b[0];
    out[0] <== unit0.sum;

    component unit[k - 1];
    for (var i = 1; i < k; i++) {
        unit[i - 1] = ModSumThree(n);
        unit[i - 1].a <== a[i];
        unit[i - 1].b <== b[i];
        if (i == 1) {
            unit[i - 1].c <== unit0.carry;
        } else {
            unit[i - 1].c <== unit[i - 2].carry;
        }
        out[i] <== unit[i - 1].sum;
    }
    out[k] <== unit[k - 2].carry;
}
```

The functional correctness of `BigAdd` corresponds to the following code snippet taken from the Coda repository:

```
Definition spec (w: t) : Prop :=
  (* pre-condition *)
  n > 0 →
  k > 0 →
  (n <= 252)%Z →
  (* a and b are proper big int *)
  'w.(a) |: (n) →
  'w.(b) |: (n) →
  (* post-condition *)
  ([|| w.(out) ||] = [|| w.(a) ||] + [|| w.(b) ||])%Z ∧
  'w.(out) |: (n).
```

Assuming that $0 < n <= 252$ and $k > 0$, this specification ensures the following properties for `BigAdd` if a and b are in proper big integer representation:

- out is in proper big integer representation
- `[| out |] = [| a |] + [| b |]`

4.1.3 V-BIGINT-COD-001: Missing range checks in `BigMod`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circuits/bigint.circom		
Functions	template BigMod(n, k)		

The big integer remainders `mod[i]` is not properly constrained (missing additional constraints) as opposed to `div[i]`, as shown below:

```
...
template BigMod(n, k) {
...
    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];
    }
...
```

It is insufficient to guarantee that `mod[i]` is in proper big integer representation. Specifically, in big integer base 2^n is used, so templates need to maintain the invariant that every digit is less than 2^n. However, in this template, it only enforce this invariant for `div[i]` and not for `mod[i]`, which makes it possible for a malicious prover to supply illegal values that break the invariant.

Coda requires proper post-conditions of `mod[i]` to finish the proof. In order to use the property of BigAdd when proving the soundness of BigMod, `add.b[i]` needs to be in proper big integer representation (the pre-condition of BigAdd), but since no range check is performed on `mod[i]`, this condition cannot be obtained, as shown below (line 8):

```
1  ...
2  template BigMod(n, k) {
3  ...
4      component add = BigAdd(n, 2 * k + 2);
5      for (var i = 0; i < 2 * k; i++) {
6          add.a[i] <== mul.out[i];
7          if (i < k) {
8              add.b[i] <== mod[i];
9          } else {
10             add.b[i] <== 0;
11         }
12     }
13 ...
```

Impact Attackers can bypass checking for the results by constructing a counterexample, thus potentially breaking down the protocol.

Recommendation Add additional range checking constraints for `mod[i]`. An example fix would be:

```
...
template BigMod(n, k) {
...
    component div_range_checks[k + 1];
    for (var i = 0; i <= k; i++) {
        div_range_checks[i] = Num2Bits(n);
        div_range_checks[i].in <== div[i];
    }
    component mod_range_checks[k];
    for (var i = 0; i < k; i++) {
        mod_range_checks[i] = Num2Bits(n);
        mod_range_checks[i].in <== mod[i];
    }
...
```

4.2 Formal Verification Using Picus

In this section we elaborate on an extended process we followed to verify `circom-bigint`'s dependent library: `circomlib`, using Picus, an open-source tool developed by Veridise.

Overview Picus leverages the power of static analysis and SMT solver to perform security analysis over zero-knowledge circuits. As shown in Figure 4.1, given a ZK circuit, Picus analyzes its security by invoking an interaction loop between its two components: the analyzer and SMT solver, where for each signal of the circuit, the analyzer performs light-weight inference and SMT solver performs in-depth semantic reasoning. Picus proves a circuit unsafe by finding an underconstrained signal from it with automatically synthesized concrete exploit/counterexample.

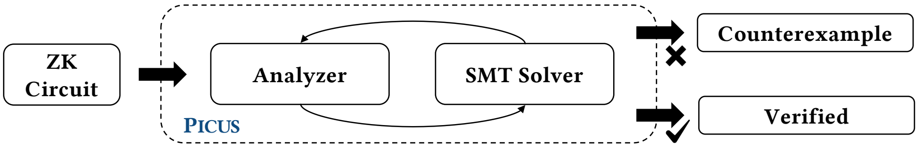

Figure 4.1: Framework overview of Picus.

Workflow Picus identifies an underconstrained bug by finding a counterexample that violates the *uniqueness* property: a circuit is unsafe (breaks the uniqueness property) if there exists two sets of signals that share the same input signals but differ on output signals. We refer to such two sets of signals as models, and they form a counterexample that attacker can use for conducting potential exploits. Thus, a counterexample is a crucial indicator for the safety of a circuit.

Since Picus works on circuit level, we perform the security analysis on a set of 163 circuits that are instantiated from `circomlib` with carefully picked arguments to ensure the coverage of the analysis. The set of instantiated circuits can be found here and here.

Results Table 4.2 shows a summary of the verification results of the security analysis. While majority of the circuits are properly constrained, Picus is able to identify 10 vulnerability issues, with 8 of them being critical underconstrained bugs. Picus attaches with each bug a concrete counterexample that demonstrates how an exploit should be performed by a potential attacker.

Table 4.2: Summary of Picus verification results.

ID	Description	Severity	Status
V-CIRCOMLIB-PIC-001	`Decoder` accepting bogus output signal	Critical	Open
V-CIRCOMLIB-PIC-002	Underconstrained: `Edwards2Montgomery`	Critical	Open
V-CIRCOMLIB-PIC-003	Underconstrained: `Montgomery2Edwards`	Critical	Open
V-CIRCOMLIB-PIC-004	Underconstrained: `MontgomeryAdd`	Critical	Open
V-CIRCOMLIB-PIC-005	Underconstrained: `MontgomeryDouble`	Critical	Open
V-CIRCOMLIB-PIC-006	Underconstrained: `BitElementMulAny`	Critical	Open
V-CIRCOMLIB-PIC-007	Underconstrained: `Window4`	Critical	Open
V-CIRCOMLIB-PIC-008	Underconstrained: `WindowMulFix`	Critical	Open
V-CIRCOMLIB-PIC-009	Underconstrained: `Bits2Point`	Warning	Open
V-CIRCOMLIB-PIC-010	Underconstrained: `Point2Bits`	Warning	Open

We elaborate the findings in the sections followed.

4.2.1 V-CIRCOMLIB-VUL-001: `Decoder` accepting bogus output signal

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/multiplexer.circom		
Functions	template Decoder(w)		

The `Decoder` template from `multiplexer` can be instantiated into a circuit that attempts to convert a number `inp` into its "one-hot" representation, which is an array `out` with corresponding index set to `1` while others remaining `0`. When the specified number is larger than the size of the array, the `Decoder` instantiated circuit is expected to return `0` (indicating failure of the process), otherwise `1` (indicating success of the process).

While the output representation array `out` is not properly constrained, this allows attackers to construct exploits that cause inconsistency between the decoded representation array `out` and the state indicator `success`, when the target template is not properly called. The root cause is shown as below:

```
...
for (var i=0; i<w; i++) {
    out[i] <-- (inp==i) ? 1 : 0;
    out[i] * (inp-i) === 0
    lc = lc + out[i];
}
lc ==> success;
...
```

Here, even though the usage of `<--` states the relations between `inp`, `i` and `out[i]` on signal computation phase, such a relation does not propagate into the constraint generation phase. As a result, as long as `out[i]=0`, the constraints generated will always be satisfied, no matter what value `inp` gets. For instance, in Table 4.3 we show the following counterexample (for the instantiated circuit with `w=2`) that demonstrates the underconstrained bug described above, where given the same input signal, at least two sets of outputs are allowed by the generated constraints with contradictory `success` signals.

Table 4.3: A counterexample for `Decoder` template instantiated with `w=2`. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

	model 1	model 2
`inp`	1	1
`out[0]`	0	0
`out[1]`	1	0
`success`	1	0

Impact Attackers can bypass checking for the decoding results by constructing a counterexample as shown above, if the `Decoder` template is not used in a proper way.

Recommendation Based on the nature and design of `circomlib` and the semantics of the `Decoder` template, we recommend one of the following fixes:

- Clarify the proper usage of the `Decoder` template, where an assertion about the valuation of its output `success` should be explicitly added when called;
- Properly constrain all the output signals from within the `Decoder` template itself, using `IsZero` template from `circomlib`. We provide an example fixed version as below:

```
include "comparators.circom";
template Decoder(w) {
    signal input inp;
    signal output out[w];
    signal output success;
    var lc = 0;

    component checkZero[w];
    for (var i=0; i<w; i++) {
        checkZero[i] = IsZero();
        checkZero[i].in <== inp - i;
        checkZero[i].out ==> out[i];
        lc = lc + out[i];
    }
    lc ==> success;
}
```

4.2.2 V-CIRCOMLIB-VUL-002: Underconstrained points in `Edwards2Montgomery`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/montgomery.circom		
Functions	template Edwards2Montgomery()		

The `Edwards2Montgomery` converts a point (`in[0]` , `in[1]`) from Edwards curve to its equivalent point (`out[0]` , `out[1]`) on Montgomery curve, which is given by:

$$\texttt{out[0]} = \frac{1 + \texttt{in[1]}}{1 - \texttt{in[1]}}, \quad \texttt{out[1]} = \frac{1 + \texttt{in[1]}}{(1 - \texttt{in[1]}) \cdot \texttt{in[0]}}$$

The `Edwards2Montgomery` template places additional *implicit* restrictions over `in[0]` and `in[1]` in signal computation phase, as shown by the code snippet below (line 4-5):

```
1 template Edwards2Montgomery() {
2     signal input in[2];
3     signal output out[2];
4     out[0] <-- (1 + in[1]) / (1 - in[1]);
5     out[1] <-- out[0] / in[0];
6     out[0] * (1-in[1]) === (1 + in[1]);
7     out[1] * in[0] === out[0];
8 }
```

where $1 -$ `in[1]` $\neq 0$ and `in[0]` $\neq 0$. However, such restrictions are not properly propagated to constraint generation phase (line 6-7), where when $1 -$ `in[1]` or `in[0]` is set to 0, their corresponding multipliers in the same terms, namely `out[0]` and `out[1]` become under-constrained. Attackers can construct an exploit to bypass the restrictions on circuit outputs. For instance, in Table 4.4 we show the following counterexample for the instantiated circuit of `Edwards2Montgomery` that demonstrates the underconstrained bug described above, where given the same inputs (`in[0]` and `in[1]`), there exist two satisfying sets of outputs, which contradicts with the semantics of signal computation phase.

Table 4.4: A counterexample for `Edwards2Montgomery` template. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

	model 1	model 2
`in[0]`	0	0
`in[1]`	-1	-1
`out[0]`	0	0
`out[1]`	0	1

Impact Attackers can bypass restrictions for the outputs (`out[0]` and `out[1]`) by setting the inputs (`in[0]` and `in[1]`) with carefully designed values, thus could potentially exploit the application circuit, if the `Edwards2Montgomery` template is not used in a proper way.

Recommendation Based on the nature and design of `circomlib` and the semantics of the `Edwards2Montgomery` template, we recommend one of the following fixes:

- Clarify the proper usage of the `Edwards2Montgomery` template, where assertions about the valuation of its inputs (pre-conditions) should be satisfied when calling the template. In particular, the following pre-conditions should be enforced by the caller:

$$\texttt{in[1]} \neq 1 \land \texttt{in[0]} \neq 0$$

- Properly and explicitly constrain all the input signals from within the `Edwards2Montgomery` template itself, using `IsZero` template from `circomlib`. We provide an example fixed version as below:

```
include "comparators.circom";
template Edwards2Montgomery() {
    signal input in[2];
    signal output out[2];
    component checkZero0 = IsZero();
    component checkZero1 = IsZero();

    checkZero0.in <== in[0];
    checkZero0.out === 0;

    checkZero1.in <== 1 - in[1];
    checkZero1.out === 0;

    out[0] <-- (1 + in[1]) / (1 - in[1]);
    out[1] <-- out[0] / in[0];
    out[0] * (1-in[1]) === (1 + in[1]);
    out[1] * in[0] === out[0];
}
```

4.2.3 V-CIRCOMLIB-VUL-003: Underconstrained points in `Montgomery2Edwards`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/montgomery.circom		
Functions	template Montgomery2Edwards()		

The `Montgomery2Edwards` converts a point (`in[0]` , `in[1]`) from Montgomery curve to its equivalent point (`out[0]` , `out[1]`) on Edwards curve, which is given by:

$$\texttt{out[0]} = \frac{\texttt{in[0]}}{\texttt{in[1]}}, \quad \texttt{out[1]} = \frac{\texttt{in[0]} - 1}{\texttt{in[0]} + 1}$$

The `Edwards2Montgomery` template places additional *implicit* restrictions over `in[0]` and `in[1]` in signal computation phase, as shown by the code snippet below (line 4-5):

```
1 template Montgomery2Edwards() {
2     signal input in[2];
3     signal output out[2];
4     out[0] <-- in[0] / in[1];
5     out[1] <-- (in[0] - 1) / (in[0] + 1);
6     out[0] * in[1] === in[0];
7     out[1] * (in[0] + 1) === in[0] - 1;
8 }
```

where $1 +$ `in[0]` $\neq 0$ and `in[1]` $\neq 0$. However, such restrictions are not properly propagated to constraint generation phase (line 6-7), where when $1 +$ `in[0]` or `in[1]` is set to 0, their corresponding multipliers in the same terms, namely `out[1]` and `out[0]` become underconstrained. Attackers can construct an exploit to bypass the restrictions on circuit outputs. For instance, in Table 4.5 we show the following counterexample for the instantiated circuit of `Montgomery2Edwards` that demonstrates the underconstrained bug described above, where given the same inputs (`in[0]` and `in[1]`), there exist two satisfying sets of outputs, which contradicts with the semantics of signal computation phase.

Table 4.5: A counterexample for `Montgomery2Edwards` template. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

	model 1	model 2
`in[0]`	0	0
`in[1]`	0	0
`out[0]`	0	1
`out[1]`	-1	-1

Impact Attackers can bypass restrictions for the outputs (`out[0]` and `out[1]`) by setting the inputs (`in[0]` and `in[1]`) with carefully designed values, thus could potentially exploit the application circuit, if the `Montgomery2Edwards` template is not used in a proper way.

Recommendation Based on the nature and design of `circomlib` and the semantics of the `Montgomery2Edwards` template, we recommend one of the following fixes:

- Clarify the proper usage of the `Montgomery2Edwards` template, where assertions about the valuation of its inputs (pre-conditions) should be satisfied when calling the template. In particular, the following pre-conditions should be enforced by the caller:

$$\texttt{in[1]} \neq 0 \land \texttt{in[0]} \neq -1$$

- Properly and explicitly constrain all the input signals from within the `Montgomery2Edwards` template itself, using `IsZero` template from `circomlib`. We provide an example fixed version as below:

```
include "comparators.circom";
template Montgomery2Edwards() {
    signal input in[2];
    signal output out[2];
    component checkZero0 = IsZero();
    component checkZero1 = IsZero();

    checkZero0.in <== in[1];
    checkZero0.out === 0;

    checkZero1.in <== in[0] + 1;
    checkZero1.out === 0;

    out[0] <-- in[0] / in[1];
    out[1] <-- (in[0] - 1) / (in[0] + 1);
    out[0] * in[1] === in[0];
    out[1] * (in[0] + 1) === in[0] - 1;
}
```

4.2.4 V-CIRCOMLIB-VUL-004: Underconstrained points in `MontgomeryAdd`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/montgomery.circom		
Functions	template MontgomeryAdd()		

The `MontgomeryAdd` performs addition operation over two points on Montgomery curve. Given two points (`in1[0]` , `in1[1]`) and (`in2[0]` , `in2[1]`), the addition operation is defined as below:

$$\begin{aligned} \texttt{out[0]} &= B \cdot \texttt{lambda}^2 - A - \texttt{in1[0]} - \texttt{in2[0]} \\ \texttt{out[1]} &= \texttt{lambda} \cdot (\texttt{in1[0]} - \texttt{out[0]}) - \texttt{in1[1]}, \end{aligned}$$

where A and B are constants, and `lambda` is given by:

$$\texttt{lambda} = \frac{\texttt{in2[1]} - \texttt{in1[1]}}{\texttt{in2[0]} - \texttt{in1[0]}}$$

The `MontgomeryAdd` template places additional *implicit* restrictions over `in2[0]` and `in1[0]` in signal computation phase, as shown by the code snippet below (line 5):

```
1  template MontgomeryAdd() {
2      ...
3      signal lamda;
4  
5      lamda <-- (in2[1] - in1[1]) / (in2[0] - in1[0]);
6      lamda * (in2[0] - in1[0]) === (in2[1] - in1[1]);
7  
8      out[0] <== B*lamda*lamda - A - in1[0] -in2[0];
9      out[1] <== lamda * (in1[0] - out[0]) - in1[1];
10 }
```

where `in2[0]` $-$ `in1[0]` $\neq 0$. However, such restrictions are not properly propagated to constraint generation phase (line 6), where when `in2[0]` $-$ `in1[0]` is set to 0, its corresponding multiplier in the same term, `lambda` becomes underconstrained, which further affects the corresponding terms `out[0]` and `out[1]` from line 8-9. Attackers can construct an exploit to bypass the restrictions on circuit outputs. For instance, in Table 4.6 we show a counterexample for the instantiated circuit of `MontgomeryAdd` that demonstrates the underconstrained bug described above, where p corresponds to the prime of the field, and given the same inputs (`in1[0]` , `in1[1]` , `in2[0]` , `in2[1]`), there exist two satisfying sets of outputs, which contradicts with the semantics of signal computation phase.

Impact Attackers can bypass restrictions for the outputs (`out[0]` and `out[1]`) by setting the inputs with carefully designed values, thus could potentially exploit the application circuit, if the `MontgomeryAdd` template is not used in a proper way.

Table 4.6: A counterexample for `MontgomeryAdd` template. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

	model 1	model 2
`in1[0]`	0	0
`in1[1]`	0	0
`in2[0]`	0	0
`in2[0]`	0	0
`out[0]`	p-168698	p-168697
`out[1]`	0	168697
`lambda`	0	1

Recommendation Based on the nature and design of `circomlib` and the semantics of the `MontgomeryAdd` template, we recommend one of the following fixes:

- Clarify the proper usage of the `MontgomeryAdd` template, where assertions about the valuation of its inputs (pre-conditions) should be satisfied when calling the template. In particular, the following pre-conditions should be enforced by the caller:

$$\texttt{in2[0]} \neq \texttt{in1[0]}$$

- Properly and explicitly constrain all the input signals from within the `MontgomeryAdd` template itself, using `IsZero` template from `circomlib`. We provide an example fixed version as below:

```
include "comparators.circom";
template MontgomeryAdd() {
    signal input in1[2];
    signal input in2[2];
    signal output out[2];

    var a = 168700;
    var d = 168696;

    var A = (2 * (a + d)) / (a - d);
    var B = 4 / (a - d);

    component checkZero = IsZero();
    checkZero.in <== in2[0] - in1[0];
    checkZero.out === 0;

    signal lamda;

    lamda <-- (in2[1] - in1[1]) / (in2[0] - in1[0]);
    lamda * (in2[0] - in1[0]) === (in2[1] - in1[1]);

    out[0] <== B*lamda*lamda - A - in1[0] -in2[0];
    out[1] <== lamda * (in1[0] - out[0]) - in1[1];
}
```

4.2.5 V-CIRCOMLIB-VUL-005: Underconstrained points in `MontgomeryDouble`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/montgomery.circom		
Functions	template MontgomeryDouble()		

The `MontgomeryDouble` performs doubling operation over a given point on Montgomery curve. Given a point (`in[0]` , `in[1]`), the doubling operation is defined as below:

$$\begin{aligned} \texttt{out[0]} &= B \cdot \texttt{lambda}^2 - A - 2 \cdot \texttt{in[0]} \\ \texttt{out[1]} &= \texttt{lambda} \cdot (\texttt{in[0]} - \texttt{out[0]}) - \texttt{in[1]}, \end{aligned}$$

where A and B are constants, and `lambda` is given by:

$$\texttt{lambda} = \frac{3 \cdot \texttt{in[0]}^2 + 2 \cdot A \cdot \texttt{in[0]} + 1}{2 \cdot B \cdot \texttt{in[1]}}$$

The `MontgomeryDouble` template places additional *implicit* restrictions over `in[1]` in signal computation phase, as shown by the code snippet below (line 6):

```
1  template MontgomeryDouble() {
2      ...
3      signal lamda;
4      ...
5
6      lamda <-- (3*x1_2 + 2*A*in[0] + 1 ) / (2*B*in[1]);
7      lamda * (2*B*in[1]) === (3*x1_2 + 2*A*in[0] + 1 );
8
9      out[0] <== B*lamda*lamda - A - 2*in[0];
10     out[1] <== lamda * (in[0] - out[0]) - in[1];
11 }
```

where `in[1]` $\neq 0$. However, such restrictions are not properly propagated to constraint generation phase (line 7), where when `in[1]` is set to 0, its corresponding multiplier in the same term, `lambda` becomes underconstrained, which further affects the corresponding terms `out[0]` and `out[1]` from line 9-10. Attackers can construct an exploit to bypass the restrictions on circuit outputs. For instance, we show the following counterexample in Table 4.7 for the instantiated circuit of `MontgomeryDouble` that demonstrates the underconstrained bug described above, where p corresponds to the prime of the field, and given the same inputs, there exist two satisfying sets of outputs, which contradicts with the semantics of signal computation phase.

Impact Attackers can bypass restrictions for the outputs (`out[0]` and `out[1]`) by setting the inputs with carefully designed values, thus could potentially exploit the application circuit, if the `MontgomeryDouble` template is not used in a proper way.

Table 4.7: A counterexample for `MontgomeryDouble` template. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 1	
`main.in[0]`	19192010538876120388543940170329655827361864530218831473775418363317877843\u200b50
`main.in[1]`	0
`main.lambda`	0
`main.x1_2`	`in[0]`^{2}
`main.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.out[1]`	0

model 2	
`main.in[0]`	1919201053887612038854394017032965582736186453021883147377541836331787784350
`main.in[1]`	0
`main.lambda`	1919201053887612038854394017032965582736186453021883147377541836331787784350
`main.x1_2`	`in[0]`^{2}
`main.out[0]`	0
`main.out[1]`	11395287471962378606215025428882238971762841540906324053591198862844560648166

Recommendation Based on the nature and design of `circomlib` and the semantics of the `MontgomeryDouble` template, we recommend one of the following fixes:

- Clarify the proper usage of the `MontgomeryDouble` template, where assertions about the valuation of its inputs (pre-conditions) should be satisfied when calling the template. In particular, the following pre-conditions should be enforced by the caller:

  `in[1]` $\neq 0$

- Properly and explicitly constrain all the input signals from within the `MontgomeryDouble` template itself, using `IsZero` template from `circomlib`. We provide an example fixed version as below:

```
include "comparators.circom";
template MontgomeryDouble() {
    signal input in[2];
    signal output out[2];
    var a = 168700;
    var d = 168696;
    var A = (2 * (a + d)) / (a - d);
    var B = 4 / (a - d);

    component checkZero = IsZero();
    checkZero.in <== in[1];
    checkZero.out === 0;

    signal lamda;
    signal x1_2;
    x1_2 <== in[0] * in[0];
    lamda <-- (3*x1_2 + 2*A*in[0] + 1 ) / (2*B*in[1]);
    lamda * (2*B*in[1]) === (3*x1_2 + 2*A*in[0] + 1 );
    out[0] <== B*lamda*lamda - A - 2*in[0];
    out[1] <== lamda * (in[0] - out[0]) - in[1];
}
```

4.2.6 V-CIRCOMLIB-VUL-006: Underconstrained outputs in `BitElementMulAny`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/escalarmulany.circom		
Functions	template BitElementMulAny()		

The `BitElementMulAny` directly utilizes the `MontgomeryAdd` and `MontgomeryDouble` template in its computation without explicit range checks for their inputs/outputs. As discussed in previous sections about `MontgomeryAdd` and `MontgomeryDouble`, an attacker could construct a counterexample that bypass the restrictions of the instantiated circuit and perform potential exploits. We show a concrete counterexample in Table 4.8 (model 1) and Table 4.9 (model 2), where given same set of inputs, the outputs are not properly constrained.

Table 4.8: A counterexample for `BitElementMulAny` template: Model 1. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 1	
`main.dblOut[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.dblOut[1]`	0
`main.addOut[0]`	0
`main.addOut[1]`	0
`main.sel`	0
`main.dblIn[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.dblIn[1]`	0
`main.addIn[0]`	0
`main.addIn[1]`	0
`main.adder.out[0]`	2661034181063526690380968414130598626815208015128985754079958221158257086763
`main.adder.out[1]`	0
`main.adder.in1[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.adder.in1[1]`	0
`main.adder.in2[0]`	0
`main.adder.in2[1]`	0
`main.adder.lamda`	0
`main.doubler.out[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.doubler.out[1]`	0
`main.doubler.in[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.doubler.in[1]`	0
`main.doubler.lamda`	17227713526953394140741591647523112279518733089659685263306152777950041868941
`main.doubler.x1_2`	18039640916646237372880335511686420348069741665070947478680356439334569097561
`main.selector.out[0]`	0
`main.selector.out[1]`	0
`main.selector.sel`	0
`main.selector.in[0][0]`	0
`main.selector.in[0][1]`	0
`main.selector.in[1][0]`	2661034181063526690380968414130598626815208015128985754079958221158257086763
`main.selector.in[1][1]`	0

Table 4.9: A counterexample for `BitElementMulAny` template: Model 2. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 2	
`main.dblOut[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.dblOut[1]`	0
`main.addOut[0]`	0
`main.addOut[1]`	0
`main.sel`	0
`main.dblIn[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.dblIn[1]`	0
`main.addIn[0]`	0
`main.addIn[1]`	0
`main.adder.out[0]`	16566174509712221841484468916996077834917948370158062835538287744259293984695
`main.adder.out[1]`	0
`main.adder.in1[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adder.in1[1]`	0
`main.adder.in2[0]`	0
`main.adder.in2[1]`	0
`main.adder.lamda`	0
`main.doubler.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.doubler.out[1]`	0
`main.doubler.in[0]`	19227208690775748531865437331126676461733156385287048589618245965417551240156
`main.doubler.in[1]`	0
`main.doubler.lamda`	0
`main.doubler.x1_2`	18039640916646237372880335511686420348069741665070947478680356439334569097561
`main.selector.out[0]`	0
`main.selector.out[1]`	0
`main.selector.sel`	0
`main.selector.in[0][0]`	0
`main.selector.in[0][1]`	0
`main.selector.in[1][0]`	16566174509712221841484468916996077834917948370158062835538287744259293984695
`main.selector.in[1][1]`	0

Impact Since this vulnerability is caused by `MontgomeryAdd` and `MontgomeryDouble`, please check Section 4.2.4 and Section 4.2.5 for more details about potential impact.

Recommendation Since this vulnerability is caused by `MontgomeryAdd` and `MontgomeryDouble`, please check Section 4.2.4 and Section 4.2.5 for more details about potential recommendations.

4.2.7 V-CIRCOMLIB-VUL-007: Underconstrained outputs in `Window4`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/pedersen.circom		
Functions	template Window4()		

The `Window4` directly utilizes the `MontgomeryAdd` and `MontgomeryDouble` template in its computation without explicit range checks for their inputs/outputs. As discussed in previous sections about `MontgomeryAdd` and `MontgomeryDouble`, an attacker could construct a counterexample that bypass the restrictions of the instantiated circuit and perform potential exploits. We show a concrete counterexample in Table 4.10, Table 4.11, Table 4.12 and Table 4.13, where given same set of inputs, the outputs are not properly constrained.

4.2.8 V-CIRCOMLIB-VUL-008: Underconstrained outputs in `WindowMulFix`

Severity	Critical	**Commit**	cff5ab6
Type	Underconstrained Error	**Status**	Open
Files	circomlib/circuits/escalarmulfix.circom		
Functions	template WindowMulFix()		

The `WindowMulFix` directly utilizes the `MontgomeryAdd` and `MontgomeryDouble` template in its computation without explicit range checks for their inputs/outputs. As discussed in previous sections about `MontgomeryAdd` and `MontgomeryDouble`, an attacker could construct a counterexample that bypass the restrictions of the instantiated circuit and perform potential exploits. We show the counterexample in Section 4.2.7 as a reference for constructing a counterexample for `WindowMulFix`.

Impact Since this vulnerability is caused by `MontgomeryAdd` and `MontgomeryDouble`, please check Section 4.2.4 and Section 4.2.5 for more details about potential impact.

Recommendation Since this vulnerability is caused by `MontgomeryAdd` and `MontgomeryDouble`, please check Section 4.2.4 and Section 4.2.5 for more details about potential recommendations.

Table 4.10: A counterexample for `Window4` template: Model 1. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 1	
`main.out[0]`	0
`main.out[1]`	0
`main.out8[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.out8[1]`	0
`main.in[0]`	0
`main.in[1]`	0
`main.in[2]`	14371310148765874102861111177432241573644682305993849736623517215016714561
`main.in[3]`	0
`main.base[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.base[1]`	0
`main.adr3.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr3.out[1]`	0
`main.adr3.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr3.in1[1]`	0
`main.adr3.in2[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr3.in2[1]`	0
`main.adr3.lamda`	0
`main.adr4.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr4.out[1]`	0
`main.adr4.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr4.in1[1]`	0
`main.adr4.in2[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr4.in2[1]`	0
`main.adr4.lamda`	0
`main.adr5.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr5.out[1]`	0
`main.adr5.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr5.in1[1]`	0
`main.adr5.in2[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr5.in2[1]`	0
`main.adr5.lamda`	0
`main.adr6.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr6.out[1]`	0
`main.adr6.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr6.in1[1]`	0
`main.adr6.in2[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr6.in2[1]`	0
`main.adr6.lamda`	0
`main.adr7.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr7.out[1]`	0
`main.adr7.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr7.in1[1]`	0
`main.adr7.in2[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr7.in2[1]`	0
`main.adr7.lamda`	0
`main.adr8.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.adr8.out[1]`	0
`main.adr8.in1[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461

Table 4.11: A counterexample for `Window4` template: Model 1 (Cont'd). `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 1 (cont'd)	
`main.adr8.in1[1]`	0
`main.adr8.in2[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr8.in2[1]`	0
`main.adr8.lamda`	0
`main.dbl2.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.dbl2.out[1]`	0
`main.dbl2.in[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.dbl2.in[1]`	0
`main.dbl2.lamda`	-4660529344885881081504814097734162809029631310756349080392051408625766626676
`main.dbl2.x1_2`	-3848601955193037849366070233570854740478622735345086865017847747241239398056
`main.mux.out[0]`	0
`main.mux.out[1]`	0
`main.mux.c[0][0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][1]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][2]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.mux.c[0][3]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][4]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.mux.c[0][5]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][6]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.mux.c[0][7]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[1][0]`	0
`main.mux.c[1][1]`	0
`main.mux.c[1][2]`	0
`main.mux.c[1][3]`	0
`main.mux.c[1][4]`	0
`main.mux.c[1][5]`	0
`main.mux.c[1][6]`	0
`main.mux.c[1][7]`	0
`main.mux.s[0]`	0
`main.mux.s[1]`	0
`main.mux.s[2]`	14371310148765874102861111177432241573644682305993849736623517215016714561
`main.mux.a210[0]`	0
`main.mux.a210[1]`	0
`main.mux.a21[0]`	0
`main.mux.a21[1]`	0
`main.mux.a20[0]`	0
`main.mux.a20[1]`	0
`main.mux.a2[0]`	7983102543190580071142905242391795880445624045386957262239874663474771597685
`main.mux.a2[1]`	0
`main.mux.a10[0]`	0
`main.mux.a10[1]`	0
`main.mux.a1[0]`	0
`main.mux.a1[1]`	0
`main.mux.a0[0]`	0
`main.mux.a0[1]`	0
`main.mux.a[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.a[1]`	0
`main.mux.s10`	0

Table 4.12: A counterexample for `Window4` template: Model 2. `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 2	
`main.out[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.out[1]`	0
`main.out8[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.out8[1]`	0
`main.in[0]`	0
`main.in[1]`	0
`main.in[2]`	143713101487658741028611111177432241573644682305993849736623517215016714561
`main.in[3]`	0
`main.base[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.base[1]`	0
`main.adr3.out[0]`	0
`main.adr3.out[1]`	-10492955399876896616031380316375036116785522859509710290107005323731247847451
`main.adr3.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr3.in1[1]`	0
`main.adr3.in2[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr3.in2[1]`	0
`main.adr3.lamda`	19192010538876120388543940170329655827361864530218831473775418363317787784350
`main.adr4.out[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr4.out[1]`	0
`main.adr4.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr4.in1[1]`	0
`main.adr4.in2[0]`	0
`main.adr4.in2[1]`	-10492955399876896616031380316375036116785522859509710290107005323731247847451
`main.adr4.lamda`	-19192010538876120388543940170329655827361864530218831473775418363317787784350
`main.adr5.out[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr5.out[1]`	0
`main.adr5.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr5.in1[1]`	0
`main.adr5.in2[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr5.in2[1]`	0
`main.adr5.lamda`	-4660529344885881081504814097734162809029631310756349080392051408625766626676
`main.adr6.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr6.out[1]`	0
`main.adr6.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr6.in1[1]`	0
`main.adr6.in2[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr6.in2[1]`	0
`main.adr6.lamda`	0
`main.adr7.out[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr7.out[1]`	0
`main.adr7.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461
`main.adr7.in1[1]`	0
`main.adr7.in2[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr7.in2[1]`	0
`main.adr7.lamda`	0
`main.adr8.out[0]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.adr8.out[1]`	0
`main.adr8.in1[0]`	-26610341810635266903809684141305986268152080151289857540799582211158257255461

Table 4.13: A counterexample for `Window4` template: Model 2 (Cont'd). `sig` indicates input signal of main component, `sig` indicates output signal of main component.

model 2 (cont'd)	
`main.adr8.in1[1]`	0
`main.adr8.in2[0]`	-26610341810635266903809684141305986268152080151289857540799582211582 57255461
`main.adr8.in2[1]`	0
`main.adr8.lamda`	0
`main.dbl2.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.dbl2.out[1]`	0
`main.dbl2.in[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.dbl2.in[1]`	0
`main.dbl2.lamda`	-4660529344885881081504814097734162809029631310756349080392051408625766626676
`main.dbl2.x1_2`	-3848601955193037849366070233570854740478622735345086865017847747241239398056
`main.mux.out[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.out[1]`	0
`main.mux.c[0][0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][1]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][2]`	0
`main.mux.c[0][3]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][4]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][5]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.mux.c[0][6]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.c[0][7]`	5322068362127053380761936828261197253630416030257971508159916442316514342224
`main.mux.c[1][0]`	0
`main.mux.c[1][1]`	0
`main.mux.c[1][2]`	-10492955399876896616031380316375036116785522859509710290107005323731247847451
`main.mux.c[1][3]`	0
`main.mux.c[1][4]`	0
`main.mux.c[1][5]`	0
`main.mux.c[1][6]`	0
`main.mux.c[1][7]`	0
`main.mux.s[0]`	0
`main.mux.s[1]`	0
`main.mux.s[2]`	14371310148765874102861111117743224157364468230599384973662351721501 6714561
`main.mux.a210[0]`	0
`main.mux.a210[1]`	0
`main.mux.a21[0]`	0
`main.mux.a21[1]`	0
`main.mux.a20[0]`	0
`main.mux.a20[1]`	0
`main.mux.a2[0]`	0
`main.mux.a2[1]`	0
`main.mux.a10[0]`	0
`main.mux.a10[1]`	0
`main.mux.a1[0]`	0
`main.mux.a1[1]`	0
`main.mux.a0[0]`	0
`main.mux.a0[1]`	0
`main.mux.a[0]`	-2661034181063526690380968414130598626815208015128985754079958221158257255461
`main.mux.a[1]`	0
`main.mux.s10`	0

4.2.9 V-CIRCOMLIB-VUL-009: Underconstrained outputs in `Bits2Point`

Severity	Warning	Commit	cff5ab6
Type	Underconstrained Error	Status	Open
Files	circomlib/circuits/pointbits.circom		
Functions	template Bits2Point()		

The `Bits2Point` does not have concrete signal computation code, nor constraint generation code.

Impact Use of this template may lead to potential exploits due to underconstrained output signals or failure to fulfill its functional correctness.

Recommendation Switch to `Bits2Point_Strict` or append proper constraints to the output signals.

4.2.10 V-CIRCOMLIB-VUL-010: Underconstrained outputs in `Point2Bits`

Severity	Warning	Commit	cff5ab6
Type	Underconstrained Error	Status	Open
Files	circomlib/circuits/pointbits.circom		
Functions	template Point2Bits()		

The `Point2Bits` does not have concrete signal computation code, nor constraint generation code.

Impact Use of this template may lead to potential exploits due to underconstrained output signals or failure to fulfill its functional correctness.

Recommendation Switch to `Point2Bits_Strict` or append proper constraints to the output signals.

Miscellaneous 5

To ensure additional edge cases are covered properly, we also performed manual inspection on circuits from `circom-bigint`. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 5.1 summarizes the issues found by our security engineers.

Table 5.1: Summary of Picus verification results.

ID	Description	Severity	Status
V-BIGINT-VUL-001	Unnecessary computation and constraints in `BigSubModP`	Warning	Open
V-BIGINT-VUL-002	`BigModInv` can use `BigMultModP` instead of `BigMult`	Warning	Open
V-BIGINT-VUL-003	Comment assumptions on input signals	Warning	Open

5.1 Background

In this section, we briefly summarize the `circom-bigint` library and several assumptions of the library, as follows:

- `circom-bigint` provides several useful arithmetic operations on Big Integers.
- In circom and in general cryptography, all operations are defined over the field $\mathbb{F}_p$ where p is a prime.
- All numbers are integers in $[0, p)$, called signals in circom.
- We need the capability to work with bigger numbers, hence bigint library.
- A "bigint" number is represented as an array of k signals, each of which has n bits.
- Basically, a k digit number is in base 2^n.

5.2 Detailed Description of Bugs

In this section, we provide a detailed description of each vulnerability.

5.2.1 V-BIGINT-VUL-001: Unnecessary computation and constraints in `BigSubModP`

Severity	Warning	**Commit**	7505e5c
Type	Optimization	**Status**	Open
Files	circuits/bigint.circom		
Functions	template BigSubModP(n, k)		

Background `BigSubModP(n,k)` takes three inputs `a[k]`, `b[k]` and `p[k]` representing big integers, and produces an output `out[k]`, which is constrained to `(a-b)%p`.

Description Internally `BigSubModP` calls `BigAdd` to compute `(a-b)+p`:

```
...
component add = BigAdd(n,k);
...
```

`BigAdd` also returns a carry register but it isn't used for computation nor for constraints downstream.

Recommendation Create a new template `BigAddNoCarry` and call it instead to have a optimized version of generated circuits.

5.2.2 V-BIGINT-VUL-002: `BigModInv` can use `BigMulModP` instead of `BigMult`

Severity	Warning	**Commit**	7505e5c
Type	Optimization	**Status**	Open
Files	circuits/bigint.circom		
Functions	template BigModInv(n, k)		

Background `BigModInv(n,k)` takes two inputs `in[k]`, `p[k]` representing big integers and produces an output `out[k]`, which is constrained to `(out*in)%p=1`.

Description `BigModInv` calculates multiplicative inverse of a big integer `a` modular big integer `p`. For constraint checking, it first multiplies the inverse with a through `BigMult`, then computes its remainder modular `p`. This can all be done through `BigMultModP` template, which improves readability.

Recommendation Use `BigMultModP` instead to computer the remainder to improve readability.

5.2.3 V-BIGINT-VUL-003: Comment assumptions on input signals

Severity	Warning	**Commit**	7505e5c
Type	Optimization	**Status**	Open
Files	circuits/bigint.circom, circuits/bigint_4x64_mult.circom		
Functions	*		

Description Several templates assume the inputs to be of size `n`-bits. However, this is not explicitly stated to warn the developers.

Recommendation Create specification of templates documenting input assumptions and add comments for corresponding circom templates.